Ensembles of Gradient Boosting Regressors in Housing Price Error Prediction

Quan Nguyen
DePauw University
Dept. of Computer Science
Greencastle IN, 46135
quannguyen_2019@depauw.edu

Mason Seeger
DePauw University
Dept. of Computer Science
Greencastle IN, 46135
masonseeger_2019@depauw.edu

Steven Bogaerts*
DePauw University
Dept. of Computer Science
Greencastle IN, 46135
stevenbogaerts@depauw.edu

ABSTRACT

Gradient boosting regression is a common approach to prediction. This paper explores the use of an ensemble of gradient boosting regressors for prediction of housing prices, based on the contributions of many attributes. Using a dataset of houses sold in 2016 in three counties of California, this paper describes several data cleaning steps and three gradient boosting regression algorithms. The algorithms are considered individually, and then together in an ensemble. Optimal parameterization of the ensemble is determined experimentally.

Keywords: gradient boosting tree, ensemble, housing prices, data cleaning

1. INTRODUCTION

With a home being the largest purchase most people will ever make, it is not surprising that housing price prediction is a frequently-studied task. Highly reliable price predictions could bring many economic benefits. Sellers and buyers could be certain of a fair deal. Realtors could set prices perfectly according to the seller's goals, whether to sell quickly at a low price or more slowly at a higher price. Tax assessors could make perfect valuations. Perhaps most intriguingly, civic organizations and government officials could run models describing the effect of policies on housing prices, enabling more effective building regulations, zoning laws, and city planning. This paper explores strategies for making housing price predictions using a full list of real estate properties sold in three counties in California in 2016. Technical background is provided, followed by an explanation of the data cleaning steps, the application of regression algorithms and ensembles, a series of experiments, and results.

2. TECHNICAL BACKGROUND

Regression is a statistical process to determine the relationship between several independent variables and a single dependent variable. The result of regression is called a *regressor* or, more generally, a *predictor*. The intent is that after analyzing a dataset with both the independent variables and the dependent variable, the predictor can predict the value of the dependent variable when provided with a previously unseen combination of values for the independent variables.

A predictor is constructed using a *training set* of houses in which the actual sale price is known, attempting to establish a relationship between the sale prices and various attributes of the houses. The predictor can then be applied to a *testing set* of previously-unseen houses, with unknown sale prices. In the predicting process, the predictor looks at the attributes of a house in the testing set, and uses the model that it has learned from the training set to determine an estimated sale price.

For a given house we denote $\vec{x}$ as the vector of the attribute values of a house – the values of the independent variables. We use $\vec{x}_i$ to denote the value of the i^{th} attribute. We also denote y as the dependent variable value for the house. When discussing multiple houses, we use $\vec{x}^{(j)}$ and $y^{(j)}$ to indicate the vector of attribute values and the dependent variable value, respectively, of the j^{th} house in the dataset.

An *ensemble* is a collection of predictors that together make an aggregate prediction of a value. One simple implementation of an ensemble is an *additive* model in which the final prediction is a weighted sum of individual predictions:

$$h(\vec{x}) = \sum_i \rho_i h_i(\vec{x})$$

*Faculty advisor

Proceedings of the 2017 Midstates Conference on Undergraduate Research in Computer Science and Mathematics

where h_i denotes prediction of the i^{th} predictor, with weight ρ_i. Ensembles are well-established as a more effective predicting method than individual predictors [1].

While the above assumes the prediction of housing prices themselves, this work actually focuses on a related problem: the prediction of the *error* in another predictor's prediction of housing prices. This other predictor is the "Zestimate" from real estate database company Zillow Group, Inc ("Zillow"). The error in the Zestimate is based on its difference from the actual sale price when the home is sold. Thus, our predictors aim to determine the relationship between the attributes of a house and the error in the Zestimate. This task is also the subject of a Kaggle, Inc. competition [2].

3. ALGORITHMS AND METHODOLOGY

3.1 The Data

The dataset used in this work is drawn from houses sold in three counties in California. It includes 2,985,217 elements, each representing a house for sale at some point from January 1 to December 30, 2016. A small number of these houses appear multiple times in the dataset, indicating multiple sales of the house in the timespan under consideration.

The dataset contains 57 attributes, primarily focused on the physical characteristics of the house itself. Five types of attributes are present in the dataset:

- **Binary (6 attributes)**: Boolean attributes indicating whether a specific feature is applicable to the house. For example, *fireplaceflag* indicates whether or not a house has a fireplace.
- **Nominal (18 attributes)**: categorical data with no meaningful ordering of values. For example, *buildingclasstypeid* denotes different building framing types (wood, steel, concrete, etc.), but there is no clear ordering such as wood < steel < concrete.
- **Ordinal (1 attribute)**: categorical data with a meaningful ordering of values. The only ordinal attribute in the dataset is *buildingqualitytypeid*, with values ranging from 1 (best quality) to 12 (worst quality).
- **Discrete (13 attributes)**: integer-valued numerical data, such as number of rooms and year built.
- **Continuous (19 attributes)**: real-valued numerical data, such as total area in square feet, and the amount of tax assessed for the house.

Out of the nearly three million houses, we are given two additional attributes for 90,275 houses: *logerror* and *transactiondate*. These attributes represent the error in the Zestimate compared to the actual selling price, and the date in which the house was sold, respectively. *logerror* is calculated as follows:

$$logerror = \ln(z) - \ln(a)$$

where $\ln(\cdot)$ denotes the natural log, z is the Zestimate, and a is the actual sale price.

The 90,275 houses with the revealed *logerror* values make up the training set, while the nearly three million houses together form the testing set. The goal of this work is to predict *logerror* values for the testing set. It is important to note that while the testing set's actual *logerror* values are not revealed to us, they are known by Zillow. A comparison of *logerror* predictions to Zillow's unrevealed actual *logerror* values can be made by uploading predictions to kaggle.com.

3.2 Data Cleaning

Several steps must be taken to prepare the raw dataset for analysis. For example, it contains many attributes with large percentages of missing data, extreme outliers, and skewed data. Below we detail what steps are taken to correct these and other problems in the data in order to improve the model.

In each step of data cleaning, the full dataset (both training and testing) is used in any underlying calculations. This was done because the full dataset provides a far greater number of samples than the training set alone, and so basic statistics such as means and modes are more accurate when computed with the full dataset. All transformations are applied consistently to both the training and testing sets.

3.2.1 Dropping Outliers

The *logerror* attribute in the training set contains some values that are such significant outliers as to suggest they are erroneous. For example, a value over 4.5 suggests that price was mis-estimated by a factor of approximately 100 [3]. Here we describe a procedure for eliminating these outliers.

We use the *interquartile* method as described in [4], in which a range is computed based on some multiple of the first quartile $Q1$ and the third quartile $Q3$ of the dataset, and any values outside of that range are considered outliers to be eliminated from the dataset.

The bounds of the interquartile range are:

$$[Q1 - m(Q3 - Q1), Q3 + m(Q3 - Q1)]$$

for some constant m controlling the size of the range. Specifically, a lower m results in a smaller interquartile range, meaning more values are eliminated from the dataset. Similarly, a higher m keeps more data in the dataset. For example, using the common values of $m = 1.5$ and $m = 3.0$ results in 11.608% and 5.318% of houses being dropped, respectively. Preliminary experiments suggested that the model would learn better with fewer outliers dropped, and so we adopt a value of $m = 5.75$, resulting in 2.103% of houses being dropped. A more complete study of the ideal m value is proposed in Section 5.

3.2.2 Filling Binary Data

Some of the attributes in the dataset have extremely high counts of missing values, with the only non-missing values being "True" or "Y". In such cases, we interpret the missing values as "False" and fill them in such that any value that signals the presence of an attribute on a house is set to "1" and any value that represents absence is set to "0".

For example, the attributes *fireplaceflag*, *hashottubandspa*, and *taxdelinquencyflag* all have over 97% of their data missing. The only values present are "True" and "Y". This suggests that the missing values actually indicate "False" or "No". We transform these attributes into a consistent integer format of 1 and 0 for "True" and "False" respectively – no Boolean, String, or missing values remain within the binary attributes.

3.2.3 Dropping Missing Values

A common approach for handling missing values is to fill them in using the mean or mode of that attribute. If an attribute has a high percentage of values missing, however, then this can be an insufficient approach. It can result in many houses having the same value for a given attribute, therefore making the attribute less useful as a distinguishing characteristic between houses. Figure 1 shows the percentage of missing values for each attribute. We choose to drop any attributes having more than 50% missing values, resulting in 20 attributes being dropped from the dataset.

3.2.4 Filling Missing Values

The remaining attributes, with fewer than 50% missing values, must have their missing values filled. We apply the simple approach of filling in each missing value with the mean or mode for that attribute. Specifically, missing values in continuous attributes are filled with the mean, while nominal, ordinal, and discrete attributes use the mode.

3.2.5 Dropping Highly Correlated Values

Often, attributes are strongly correlated with each other when they carry similar information on a particular feature. For example, *bathroomcnt* and *fullbathcnt* have a correlation of over 0.95, as they indicate the same number when a house has only full bathrooms (no half or three-quarter bathrooms). Similarly, *taxamount*, *taxvaluedollarcnt*, and *structuretaxvaluedollarcnt* are all highly correlated to one another, as they are all related to the assessed tax value for the houses.

The use of strongly correlated attributes can cause machine learning algorithms to be excessively influenced by the corresponding underlying features. Therefore, it is often beneficial to remove all but one of each group of strongly correlated attributes. We single out such groups of attributes, using ± 0.9 as the correlation threshold. Within each group, we keep the attribute that is the most strongly correlated to *logerror* in the training set, hypothesizing that it is the most relevant of the group in predicting *logerror* in the testing set. The other attributes in each group are dropped.

3.2.6 Changing Transaction Date

Each house in the training set has a *transactiondate* attribute, indicating the day, month, and year of sale. All houses in this dataset were sold in 2016, and so the year value is dropped. The month may assist in capturing seasonal trends, so it is kept. The day is likely too fine-grained a measure, so it is dropped. Thus we transform this *transactiondate* attribute into merely a transaction month.

3.2.7 Transformation for Skewed Attributes

A number of numerical attributes are highly skewed, with only a few outlying points containing higher values. To address this, we first determine the skewness of each attribute according to [5]. For each highly skewed attribute a, we apply the Box-Cox transformation [6], as given by the formula:

$$a_i' = \begin{cases} \frac{a_i^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \ln(a_i) & \lambda = 0 \end{cases}$$

where a_i is an individual value for attribute a, and a_i' is the transformed value. λ is a real number calculated by the Box-Cox implementation from the SciPy module in Python, so that the transformed data is as close as possible to a normal distribution. This transformation results in a new dataset with greatly reduced outliers that is much closer to the normal distribution.

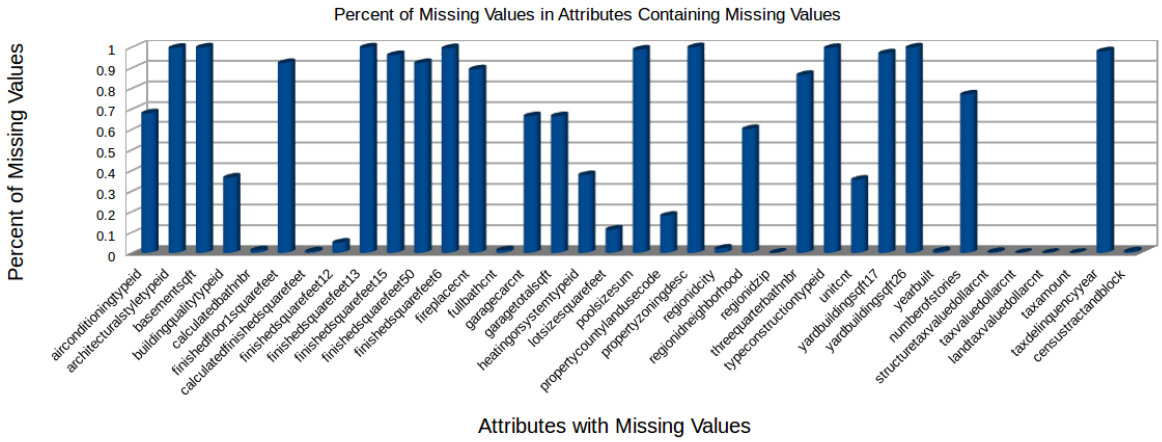

Figure 1: Graph of attributes that contain missing values and the percentage of values that are missing.

3.2.8 *Normalization of Numerical Attributes*

To normalize the numerical attributes to the range $[0, 1]$, we use *Min Max scaling*. For each numerical attribute a we transform each value a_i into a'_i using:

$$a'_i = \frac{a_i - min(a)}{max(a) - min(a)}$$

3.2.9 *Using Dummy Values for Nominal Attributes*

While a number of nominal attributes might appear numerical at first glance, in reality some such attributes have no clear numerical ordering or distance defined between them. For example, the attribute *buildingclass-typeid* contains numerical labels, each denoting a specific building framing type: 1 for steel frame, 2 for wood frame, 3 for concrete, etc. If left untouched, these labels would be interpreted by regression analysis models as values that have a meaningful numerical relationship to each other. For example, the models might assume that having a steel frame is in some way half as much as having wood frame, or that a steel frame and wood frame have the same "difference" as a wood frame and concrete.

To address this, we use *dummy variables* – a set of mutually exclusive binary attributes, one for each possible value of a given nominal attribute. For example, if attribute a has three possible values r, s, and t, then a is replaced by three mutually-exclusive Boolean dummy attributes a_r, a_s, and a_t. If a given house has $a = s$, for example, then we set $a_s = True$, while $a_r = a_t = False$.

3.2.10 *Further Handling of Nominal Attributes*

Seven of the nominal attributes in the dataset have many distinct values, making conversion to dummy variables unsuitable. For example, *propertyzoningdesc* has 5,368 distinct values, while *censustractandblock* has 96,771 distinct values. Dummy variables for these attributes would not only greatly increase the dimensionality of the dataset, but also create large areas of sparse data (data with a large number of 0 values). Sparse data is generally not desirable, as a number of machine learning algorithms tend to fail to generalize when the data is significantly sparse [7].

Thus for these attributes we use a modification of *sum encoding* [8]. In standard sum encoding, for a given nominal attribute q, the training set is split into subsets, where every element in a given subset has the same value for q. The mean of the (numerical) dependent variable is calculated for each subset. Each mean serves as the replacement value for each corresponding nominal value. In this way, nominal values are replaced with numerical values with ordering and differences determined by the dependent variable, as opposed to arbitrary ordering and differences.

More formally, let i be a particular value of q, and let S be the training set. Therefore, we denote $S_{q=i}$ as the subset of the training set for which attribute q has value i. Let $\overline{logerror}_{q=i}$ be the average *logerror* of $S_{q=i}$. Standard sum encoding defines the numerical replacement value $r_{q=i}$ for nominal value i as:

$$r_{q=i} = \overline{logerror}_{q=i}$$

Note again that standard sum encoding begins on the training set alone, since it requires that the dependent variable values are known. The resulting replacement values are then applied to both the training and test-

ing sets. This process can be effective as long as the training set and testing set have the same set of values for each nominal attribute under consideration. Suppose, however, that some nominal attribute q has a set of values q_{tr} in the training set and q_{te} in the testing set, where $q_{te} \nsubseteq q_{tr}$. In such a case, standard sum encoding, in which replacement values are based on the training set, will not provide replacement values for nominal values in the set difference $q_{te} - q_{tr}$.

Such a situation does in fact occur with six out of the seven nominal attributes under consideration here. Thus, instead of using replacement values based on *logerror*, we compute replacement values as a linear combination of certain other attributes.

More specifically, let A be the set of the n *numerical* attributes most strongly correlated with *logerror*. In this work, we use $n = 3$, though other values could be considered, as discussed in Section 5. For each numerical attribute $a \in A$, let w_a be the *correlation weight*: a weight based on the relative strength of correlation between a and *logerror* in the training set. Formally:

$$w_a = \frac{corr(a, logerror)}{\sum_{c \in A} corr(c, logerror)} \quad (1)$$

where $corr(\cdot, \cdot)$ is the correlation between the two specified attributes. Note that in this particular dataset, the attributes in A all have positive correlations with *logerror*, and so we do not consider here how to handle negative correlations.

Again, let q be the nominal attribute to be transformed, and i be a particular value of q. Also let D be the *entire* dataset – both training and testing sets combined. Therefore, we denote $D_{q=i}$ as the subset of D containing the houses where nominal attribute q has value i. For each numerical attribute $a \in A$, we denote the mean value of a across $D_{q=i}$ as $\bar{a}_{q=i}$.

With these quantities defined, replacement values are a linear combination of correlation weights w_a and means $\bar{a}_{q=i}$. Formally, the numerical replacement value $r_{q=i}$ for nominal value i is:

$$r_{q=i} = \sum_{a \in A} w_a \bar{a}_{q=i}$$

These $r_{q=i}$, for each value i of q, are used to replace every nominal value of q in the training and testing sets.

As an illustrative example of this process, suppose we would like to encode attribute *label* in the following dataset:

parcelid	logerror	s	t	label
1	0.0276	0.15	0.008	2
2	-0.0182	0.12	0.005	3
3	-0.1009	0.12	0.004	1
4	-0.0121	0.16	0.005	2
5	-0.0481	0.15	0.006	3
6	0.2897	0.17	0.006	2

Suppose further that we have already identified attributes *s* and *t* as those most strongly correlated to *logerror*, and that we will use just these $n = 2$ attributes in this conversion. Therefore, we define $A = \{s, t\}$.

We have the correlations of 0.6971 between *logerror* and *s*, and 0.3549 between *logerror* and *t*. From here, we compute the *correlation weights* w_s and w_t for the two numerical attributes:

$$w_s = \frac{0.6971}{0.6971 + 0.3549} = 0.6626$$

$$w_t = \frac{0.3549}{0.6971 + 0.3549} = 0.3374$$

Next we compute the $\bar{a}_{q=i}$ for each $a \in A$ and each subset $D_{q=i}$:

$D_{label=1}$:

parcelid	logerror	s	t	label
3	-0.1009	0.12	0.004	1

$$\bar{s}_{label=1} = 0.12 \quad \bar{t}_{label=1} = 0.004$$

$D_{label=2}$:

parcelid	logerror	s	t	label
1	0.0276	0.15	0.008	2
4	-0.0121	0.16	0.005	2
6	0.2897	0.17	0.006	2

$$\bar{s}_{label=2} = 0.16 \quad \bar{t}_{label=2} = 0.0063$$

$D_{label=3}$:

parcelid	logerror	s	t	label
2	-0.0182	0.12	0.005	3
5	-0.0481	0.15	0.006	3

$$\bar{s}_{label=3} = 0.135 \quad \bar{t}_{label=3} = 0.0055$$

Finally, each replacement value in each group is computed, using Formula 1:

$$r_{label=1} = w_s\ \bar{s}_{label=1} + w_t\ \bar{t}_{label=1} = 0.080867$$

$$r_{label=2} = w_s\ \bar{s}_{label=2} + w_t\ \bar{t}_{label=2} = 0.108159$$

$$r_{label=3} = w_s\ \bar{s}_{label=3} + w_t\ \bar{t}_{label=3} = 0.091312$$

These values are used to replace the nominal values in the original dataset, resulting in:

parcelid	logerror	s	t	label
1	0.0276	0.15	0.008	0.108159
2	-0.0182	0.12	0.005	0.091312
3	-0.1009	0.12	0.004	0.080867
4	-0.0121	0.16	0.005	0.108159
5	-0.0481	0.15	0.006	0.091312
6	0.2897	0.17	0.006	0.108159

3.3 Algorithms

Having established the data cleaning strategies above, we now describe the three algorithms considered in this work, followed by an ensemble approach combining all three. It is interesting to note, however, that each of the three algorithms described below is itself an ensemble, and so our final approach is more accurately described as an "ensemble of ensembles".

3.3.1 Gradient Tree Boosting

Gradient tree boosting is an ensemble method of prediction [9]. We can define a *strong learner* as a regressor with low error (by an arbitrary definition of "low"), while a *weak learner* is a regressor with higher error. *Boosting*, then, is one strategy of ensemble development, in which weak learners are iteratively added to an ensemble, with the intention of creating a strong learner in the aggregate [10].

In gradient tree boosting, the weak learners making up the ensemble are regression trees. A *regression tree* is a form of decision tree used to predict a continuous value, as opposed to the classification prediction of ordinary decision trees. Most commonly, regression trees predict some constant value at each leaf. For example, it may predict the mean of the dependent variable across the subset of the training set that lies at that leaf. A more complex approach would be to apply, for example, a local linear regression at each leaf.

In its simplest form, gradient tree boosting uses an additive ensemble with each member having equal weight. Thus the ensemble prediction $h(\vec{x})$ for independent values $\vec{x}$ is the sum of the predictions of the ensemble members:

$$h(\vec{x}) = \sum_i h_i(\vec{x})$$

Each regression tree may not be particularly effective on its own, but trees are iteratively added to the ensemble in order to address portions of the problem space with highest prediction error.

The following algorithm discussion is based on [11]. Suppose we have some initial "ensemble" $h^{(0)}(\vec{x})$ consisting of just a single predictor $h_0(\vec{x})$. This can be any simple approach, such as the mean of the dependent variable across the dataset: $h_0(\vec{x}) = \sum_j y^{(j)}/n$. This model likely exhibits significant prediction error, and so we wish to add a new regression tree $h_1(\vec{x})$ to obtain $h^{(1)}(\vec{x}) = h^{(0)}(\vec{x}) + h_1(\vec{x})$. More generally, this iterative process can be summarized with:

$$\begin{aligned} h^{(0)}(\vec{x}) &= h_o(\vec{x}) = \sum_j y^{(j)}/n \\ h^{(t)}(\vec{x}) &= h^{(t-1)}(\vec{x}) + h_t(\vec{x}) \\ &= \sum_{i=0}^{t} h_i(\vec{x}) \end{aligned}$$

where each h_i for $i > 0$ is a regression tree.

The question remains, then, how to define $h_t(\cdot)$ at each step in the iteration. Of course we hope that $h^{(t-1)}(\vec{x}) + h_t(\vec{x})$ will give a perfect prediction. That is, we hope that:

$$\forall j, h^{(t-1)}(\vec{x}^{(j)}) + h_t(\vec{x}^{(j)}) = y^{(j)}$$

Rearranging the terms, we can define as our goal:

$$\forall j, h_t(\vec{x}^{(j)}) = y^{(j)} - h^{(t-1)}(\vec{x}^{(j)})$$

That is, we want to define a regression tree h_t over the examples $\forall j, (\vec{x}^{(j)}, y^{(j)} - h^{(t-1)}(\vec{x}^{(j)}))$ as closely as possible. These $y^{(j)} - h^{(t-1)}(\vec{x}^{(j)})$ terms are called *residuals*. It can be shown that this regression problem is equivalent to gradient descent on a loss function, hence the name *gradient* tree boosting. Many possible termination criteria exist, including observing when the change in error on the training set is below some threshold or when the error converges to a particular value.

We use the Gradient Boosting Regressor (GBR) implementation provided in the scikit-learn module of

Python [12], performing all the data cleaning steps of Section 3.2 and passing the cleaned data to the algorithm.

3.3.2 LightGBM

Light Gradient Boosting Machine (LightGBM, or LGB) is an open-source gradient tree boosting algorithm released by Microsoft in December 2016 [13]. Thus the fundamental ideas of the algorithm remain the same as discussed in Section 3.3.1 above. While a full discussion of LGB is beyond the scope of this paper, we provide here a sketch of some key features.

First, the implementation of LGB is optimized for parallel execution, with efforts made to minimize required interprocess communication [14, 15]. This optimization is done in such a way that no additional effort in parallelism is required by the user.

A key insight of LGB is in the order of construction of the tree. Each regression tree is built in a best-first, rather than breadth-first, manner. That is, while the tree is under construction, the prediction error in each leaf is measured. The leaf with the highest error is chosen for expansion – this is the "best" leaf in terms of where the greatest prediction improvements are likely possible. While this can lead to over-fitting in small datasets, this is often a beneficial approach [16].

Another feature of LGB is the use of histograms to discretize each continuous attribute [17]. The discrete bins of a histogram dictate the subsets into which the data is split on that attribute when constructing the tree. This also determines what branch will be followed when a tree makes a prediction for a given house. The use of histograms for discretization also allows smaller data types to be used, thus reducing memory usage [18]. Because LGB handles this discretization via histograms itself, some of the data cleaning steps described in Section 3.2 are not necessary, and in fact, counter-productive, for LGB. Specifically, we apply only the transaction month, dropping outliers, and missing-value handling methods before providing the data to LGB.

3.3.3 XGBoost

Another gradient tree boosting algorithm with an open-source implementation is *Extreme Gradient Boosting* (XGBoost, or XGB). Again, a full discussion is beyond the scope of this paper, but we summarize here a few key ideas based on a more complete discussion in [19].

XGB utilizes a novel algorithm to handle sparse data, which is extremely common in real-world datasets. Sparse data can be due to many missing values, many zeros, or some feature engineering methods such as creating dummy attributes, as described in Section 3.2.9. The XGB implementation also manages memory usage through cache-aware access, data compression, and out-of-core computation. With these features, XGB can achieve scalable learning, using not only processors and memory but also disk space for big data. For this reason, XGB scales to larger datasets more effectively than many other algorithms.

Minimal data cleaning is needed for XGB, because most "unclean" data is handled directly in the XGB implementation. We simply eliminate the outliers in *logerror*, use transaction month, and fill the missing values in the dataset with the value 0, so that XGB can handle those values itself.

3.3.4 Ensemble of the Three Boosting Algorithms

Using the previous definition of an ensemble, we make an additive ensemble of the three algorithms described above:

$$h(\vec{x}) = \rho_l h_l(\vec{x}) + \rho_x h_x(\vec{x}) + \rho_g h_g(\vec{x})$$

with weights:

$$\begin{cases} \rho_l,\ \rho_x,\ \rho_g \geq 0 \\ \rho_l + \rho_x + \rho_g = 1 \end{cases}$$

where l, x, and g denote LGB, XGB, and GBR, respectively. Recall that $h(\cdot)$ represents the prediction of the ensemble, and ρ represents the individual weights of the ensemble members.

4. EXPERIMENTS

4.1 Experimental Setup

Experiments were conducted using two 44 CPU Virtual Machines, with 120GB of memory each, provided by XSEDE allocation TG-CIE170032 [20]. We used Python version 3.6.1 with the modules pandas 0.20.1, numpy 1.13.0, scikit-learn 0.18.1, scipy 0.19.0, xgboost 0.6a2, and lightgbm 2.0.2.

An ensemble of the three algorithms was built as described in Section 3.3.4. Various weight combinations were tested, with ρ_x and ρ_l both ranging from 0 to 1 with a step size of 0.1, while $\rho_g = 1 - (\rho_x + \rho_l)$. All experiments were done using the full training and testing sets provided by Zillow. Data cleaning techniques were applied selectively as described in Section 3.3.

Recall that the goal of this work is to predict the *logerror* of each house in the dataset. Specifically, a mean

absolute error (MAE) is computed on the Kaggle competition website:

$$MAE = \frac{1}{n}\sum_{i}^{n}|h(\vec{x}^{(i)}) - y^{*(i)}|$$

where y^* is the actual *logerror* values hidden from us, and $y^{*(i)}$ is the *logerror* of the i^{th} house.

It is important to note that in computing the MAE of *logerror* as above, even seemingly small changes in the MAE can indicate a significant improvement. For example, a naive model that predicts a *logerror* of 0 for every house earns an MAE of 0.0663010. At time of writing (August 1, 2017), the top score on the Kaggle website is 0.0641376. Given that the competition includes prizes for the top few teams totaling 1.2 million USD, it is likely that many teams of professional data scientists are expending considerable resources in the competition. This suggests that the top score, while only 0.0021634 lower than the naive model, represents a significant improvement in performance.

This small range of MAE scores, however, makes intuition more difficult. To assist in this, we introduce a *relative score* to transform MAE scores, where 0 corresponds to the baseline score of 0.0663010, and 100 corresponds to the current top score of 0.0641376. Intuitively, relative score is similar to a percentile, except that it is possible to have a relative score less than 0 (for performance worse than the baseline) or greater than 100 (better than the top score). Formally:

$$relative = 100 \cdot \frac{naiveMAE - MAE}{naiveMAE - topMAE}$$

4.2 Results and Discussion

We first consider results for each algorithm in isolation. That is, we consider an "ensemble" with only one nonzero weight. In order from best to worst, XGB on its own had a relative score of 82.16 (MAE = 0.0645236), LGB had 76.33 (MAE = 0.0646496), and GBR had 68.03 (MAE = 0.0648293).

Better performance, however, is obtained by various ensembles. Table 1 shows MAE scores for the top six ensemble weights. Note that even the best individual algorithm, XGB, scores worse than the top six ensembles. The best-scoring ensemble has weights 0.0, 0.7, and 0.3 for LGB, XGB, and GBR, respectively.

Before analyzing these results further, we also consider similarities and differences of the three algorithms. We first define the mean squared error (MSE) between the predictions of two algorithms as:

$$MSE = \frac{1}{n}\sum_{i}^{n}(h_1(\vec{x}) - h_2(\vec{x}))^2$$

where h_1 and h_2 are two specific algorithms.

Table 1: Weights (LGB, XGB, GBR), MAE, and Relative Scores for the Top Six Ensembles

Weights	MAE	Relative
0.0, 0.7, 0.3	.0644390	86.07
0.1, 0.7, 0.2	.0644406	85.99
0.2, 0.7, 0.1	.0644441	85.83
0.0, 0.6, 0.4	.0644446	85.81
0.1, 0.6, 0.3	.0644447	85.80
0.2, 0.6, 0.2	.0644469	85.70

Table 2: Mean Squared Errors and Descriptive Statistics of LGB, XGB, and GBR

	MSE		Statistics	
	LGB	XGB	μ	σ
LGB	-	-	.010264	.004250
XGB	.00015340	-	.013911	.014599
GBR	.00001528	.0001747	.010199	.004227

Table 2 shows that the distributions of LGB and GBR predictions have very similar means and standard deviations. The MSE between the predictions of the two algorithms is also smaller than any other pair of algorithms, at 0.00001528. In contrast, the mean, standard deviation, and MSE values for XGB suggest that that algorithm is more distinct from the other two.

Thus XGB is both the highest-scoring individual predictor, and the most distinct predictor according to Table 2. It is not surprising, therefore, that it has the highest weight, 0.7, in the optimal ensemble. What may be more surprising at first glance is that the second-best individual algorithm, LGB, has a weight of 0 in the optimal ensemble, while by far the worst individual performer, GBR, apparently makes a slightly more valuable contribution to the ensemble than LGB with the remaining 0.3 of weight. In fact, this pattern continues very clearly in the top three ensembles listed in Table 1. The top three all weight XGB at 0.7; the greater portion of the remaining 0.3 that goes to GBR, the better the resulting score. Continuing to places four through six in Table 1, again there is a clear pattern: 0.6 for XGB, while the greater portion of the remaining 0.4 that goes to GBR, the better the resulting score.

This pattern may seem counterintuitive since LGB and GBR appear quite similar – perhaps interchangeable – according to Table 2. We hypothesize that GBR is preferred over LGB in the ensemble because GBR takes a significantly different approach to data preparation. Only GBR uses the full range of data cleaning steps as described in Section 3.2. It is likely that some of these data cleaning steps – particularly the attribute engineering steps like Sections 3.2.9 and 3.2.10 – enable GBR to capitalize on the data in new ways. Some of this work is also customized for this particular dataset, as opposed to the more general one-size-fits-all approach of the core XGB and LGB algorithms. This work appears to lead GBR astray at times, as may be surmised from GBR's worst individual performance. But it also captures new information; in the spirit of ensembles merging multiple *distinct* weak learners for better aggregate performance, the weaker yet distinct approach of GBR with the data cleaning described makes it a valuable addition to the ensemble.

5. FUTURE WORK

A number of parameters in the data cleaning process have not been tested extensively for optimization:

- $m = 5.75$ to compute the interquartile range that classifies which values in the dataset are outliers, as discussed in Section 3.2.1.
- 50% as the threshold to determine which attributes have too much of their data missing, and should be dropped from the dataset, as discussed in Section 3.2.3.
- ± 0.9 as the correlation threshold to decide which groups of attributes carry the same information, as discussed in Section 3.2.5.
- $n = 3$ as the number of attributes that are most strongly correlated to *logerror* to encode nominal attributes in a numerically meaningful way, as discussed in Section 3.2.10.

While preliminary experiments suggest these values are reasonable, this work does not include comprehensive experiments to choose ideal values. In fact, optimal values for each parameter above, as well as the ensemble weights introduced in Section 3.3.4, may be dependent on each other.

For the value of n in Section 3.2.10 in particular, a problem might arise as the value for n increases. Recall that we use the $n = 3$ numerical attributes most strongly correlated to *logerror*. For this dataset, these happen to all have positive correlations. Their corresponding *correlation weights* can therefore be calculated according to Formula 1. If we were to increase the value of n, however, then we would need to be able to handle negative correlations as well, which Formula 1 does not do. We did not address this issue since it was not necessary for our setting of $n = 3$ for this dataset, but further work may require revisiting this matter.

Section 3.2.4 describes our use of the mean or mode to fill in missing values. Filling all missing values of a given attribute with the same value, however, can significantly affect the underlying distribution of that attribute. Of course, more sophisticated missing value strategies exist and may provide further improvement to the model. For example, preservation of the distribution could be achieved with techniques such as nearest neighbor or linear interpolation [21].

As mentioned above, in this work we treat the *logerror* values as a dependent variable to be predicted. More precisely, though, this work attempts to predict the error in an existing model – the Zestimate currently used by Zillow. So this work could even be considered the second predictor in a two-predictor ensemble with the Zestimate. The Zestimate makes an initial prediction, which can then be adjusted up or down according to the predicted error developed here. Such a characterization of this work may lead to additional insights we do not explore here. For example, different methods of building an ensemble with the Zestimate are possible.

6. SUMMARY AND CONCLUSION

This paper presents efforts in housing price prediction. Several data cleaning steps are described, along with three gradient tree boosting algorithms: XGBoost, LightGBM, and Gradient Boosting Regressor. Experiments show that among the three algorithms, XGBoost is the most effective single algorithm, while an ensemble of XGBoost weighted at 0.7 and Gradient Boosting Regressor weighted at 0.3 gives the best performance overall. These results provide interesting further evidence of the effectiveness of ensembles with *distinct* members, even when those individual members may have flaws, as is the case in particular with Gradient Boosting Regressor.

7. ACKNOWLEDGEMENTS

This work uses the Extreme Science and Engineering Discovery Environment (XSEDE), supported by National Science Foundation grant ACI-1548562. We also thank DePauw University and the Science Research Fellows Program for providing funding for this project.

8. REFERENCES

[1] Dietterich, T.G., et al.: Ensemble methods in machine learning. Multiple classifier systems **1857** (2000) 1–15

[2] Martin, A.: Zillow's home value prediction (Zestimate). www.kaggle.com/c/zillow-prize-1 Accessed: 2017-7-21.

[3] Novak, J.: Zillow: Outliers in training set. https://www.kaggle.com/c/zillow-prize-1/discussion/33710 (2017) Accessed: 2017-7-24.

[4] Ghasemi, A., Zahediasl, S.: Normality tests for statistical analysis: a guide for non-statisticians. International journal of endocrinology and metabolism **10**(2) (2012) 486

[5] Zwillinger, D., Kokoska, S.: CRC standard probability and statistics tables and formulae. CRC Press (1999)

[6] Osborne, J.W.: Improving your data transformations: Applying the box-cox transformation. Practical Assessment, Research & Evaluation **15**(12) (2010) 1–9

[7] Popescul, A., Pennock, D.M., Lawrence, S.: Probabilistic models for unified collaborative and content-based recommendation in sparse-data environments. In: Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence, Morgan Kaufmann Publishers Inc. (2001) 437–444

[8] McGinnins, W.: Beyond one-hot: An exploration of categorical variables. http://www.willmcginnis.com/2015/11/29/beyond-one-hot-an-exploration-of-categorical-variables/ (2015) Accessed: 2017-7-24.

[9] Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Annals of statistics (2001) 1189–1232

[10] Breiman, L., et al.: Arcing classifier (with discussion and a rejoinder by the author). The annals of statistics **26**(3) (1998) 801–849

[11] Li, C.: A gentle introduction to gradient boosting. URL: http://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/slides/gradient_boosting.pdf (2014)

[12] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12** (2011) 2825–2830

[13] Microsoft: LightGBM GitHub repository. https://github.com/Microsoft/LightGBM/wiki/Features (2016) Accessed: 2017-7-25.

[14] Jin, R., Agrawal, G.: Communication and memory efficient parallel decision tree construction. In: Proceedings of the 2003 SIAM International Conference on Data Mining, SIAM (2003) 119–129

[15] Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in mpich. The International Journal of High Performance Computing Applications **19**(1) (2005) 49–66

[16] Shi, H.: Best-first decision tree learning. PhD thesis, The University of Waikato (2007)

[17] Ranka, S., Singh, V.: Clouds: A decision tree classifier for large datasets. In: Proceedings of the 4th Knowledge Discovery and Data Mining Conference. (1998) 2–8

[18] Li, P., Wu, Q., Burges, C.J.: Mcrank: Learning to rank using multiple classification and gradient boosting. In: Advances in neural information processing systems. (2008) 897–904

[19] Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, ACM (2016) 785–794

[20] Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G.D., Roskies, R., Scott, J.R., Wilkins-Diehr, N.: XSEDE: Accelerating scientific discovery. Computing in Science & Engineering **16**(5) (2014) 62–74

[21] Junninen, H., Niska, H., Tuppurainen, K., Ruuskanen, J., Kolehmainen, M.: Methods for imputation of missing values in air quality data sets. Atmospheric Environment **38**(18) (2004) 2895–2907