# Machine Learning Engineer Nanodegree

## Capstone Project - Predict logerror for Zestimate

Lei Pan

November 14th, 2017

## I. Definition

### Project Overview

- Zillow created “Zestimate” which gives customers a lot of information about homes and housing markets at no cost by using publicly available data.
- 7.5 million statistical and machine learning models that analyze hundreds of data points on each property are used by Zillow to create and improve “Zestimate”. They improved median margin of error from 14% to 5%. Zillow announced a Kaggle competition to improve the accuracy of “Zestimate” even further.
- Zillow competition has two rounds. The first round is to build a model to predict Zillow residual error. The final round is to build a home evaluation algorithm from ground up using all external data. My project will focus on the first round of the competition. The goal of capstone project is to build a model to improve Zillow residual error.
- This is a very typical supervised machine learning problem, because supervised learning algorithms learns and analyzes labeled training data and then generates function to predict output. Zillow gave the datasets of log error between Zestimate price and actual price for both 2016 and 2017 which are labeled data as well as Zillow asked for a prediction for log error. Similar machine learning tasks are weather apps predict the temperature for a given time and spamming emails prediction based on prior spamming information.

### Problem Statement

- A machine learning computer program is said to learn from experience E with respect to some class of task T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. In this capstone project:
    - P is Mean absolute error of predicted log error and actual log error.
    - T is The log error prediction task.
    - E is The process of the algorithm examining a large amount of historical data of log error.
- This is the link to all the datasets I am using:
- https://www.kaggle.com/c/zillow-prize-1/data
- Train_2016.csv,properties_2016.csv,Train_2017.csv,properties_2017.csv

## Solution Statement

- Since the goal is to predict the log error between Zestimate and actual price and we have all the training and testing dataset for it, this is a very clear supervised learning problem for me. Based on the research I've done regarding the supervised learning algorithms, lightGBM and XGBoosting are clear winners for this problem statement in terms of accuracy and performance. First, I will clean up dataset including dealing with null values and missing values as well as converting non-numerical data to numerical data and remove outliers. Then I will do feature selections to find most important features. Then I will build benchmark model, lightGBM model, and XGBoosting model and tune parameters to get the predicted results and mean absolute errors. Finally, I will ensemble two models together to get the best results.

## Metrics

- Models are evaluated on Mean Absolute Error between the predicted log error and the actual log error. The log error is logerror=log(Zestimate)−log(SalePrice)
- If I can build a model with a MAE that's less than the MAE of base model, then the model passes evaluation.
- The reason I chose this evaluation metrics is that 1) this is provided by Zillow to evaluate actual competition. 2)Since every Zillow competition participant uses and publishes this evaluation metrics, I can compare my MAE with all the MAEs for Zillow competition participants' models to get a sense how well I am doing among all the professionals around the world.
- MAE is a good measurement for this project because of the nature of the datasets. The targeted value - logerror distributes from -4 to 4. It ranges from

negative value to positive value. Mean absolute error can take care of negative value issue. It gives proper MAE regardless of negative or positive data.

- $$\text{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i| .$$

## II. Analysis

### Data Exploration

In the data exploration phase, the first thing I did is to have a peek into the training data that I will be using. After I read data from train_2016.csv and properties_2016.csv, I printed out the first 3 rows of the datasets. I got a sense of what kind of data that train and properties datasets have and what kind of columns they have. Train data set only has 3 columns - parcelid, logerror, and transactiondate. On the other hand, property data set has 58 columns. I need to understand what those columns mean so that I can better understand data it represents. I read the data from zillow_data_dictionary.xlsx to understand every single column better. For example, I was not exactly sure about what column - hashottuborspa represents before I read data from zillow_data_dictionary. After I read dictionary data, I understand that the feature represents if the home has a hot tub or spa or not.

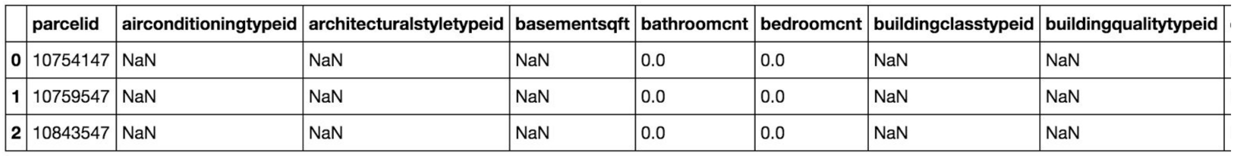
Just couple rows of data can't tell me how much of the data I am dealing with, I need more information. So I printed out the shape of train data set and property data set. The shape of train_2016 dataset is (90275, 3) and shape of  properties_2016 dataset is (2985217, 58). Train dataset has 90275 row and properties has 2985217 rows. Since I want both information to do training and data exploration, I merged two datasets together before I did further analysis. I joined two tables together by parcelid. After I merged it, I printed first 3 rows out to do a spot check. Data looks good after merge.

After understanding and merging the data, I started looking at missing values. We can't get a great result for any data analysis or machine learning problems without a great data input. A great data input means a dataset without false input, abnormal input, skewed input and etc. That's why the first thing I looked into is missing values. I checked null values for every single column and summarized them together as well as sorted them by columns names and then plotted it into a graph. In the graph, I can see most of columns have big amount of missing values and only small percentage of columns don’t have missing values. This gives me an idea about what I need to do for data processing before I start using machine learning models.

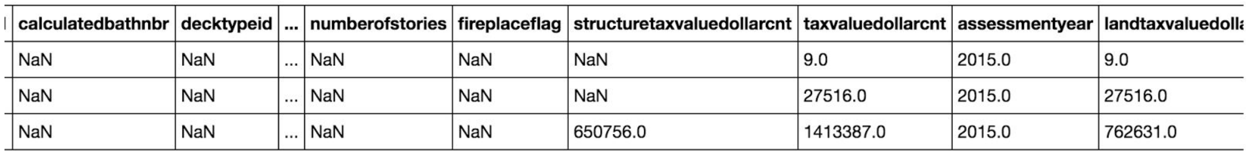
Since for the models I want to use for this problem don’t process categorical data, I need to deal with non numerical data. I grouped data by data type and checked what kind of data types we have. The data types that we have in this dataset are 1 int, 53 float, 1 datatime, 5 objects. That 5 objects will be needed to converted into numerical data later before using machine learning models. I will talk about outliers later in data exploratory visualization part.

## A Sampling of the Data and Statistics

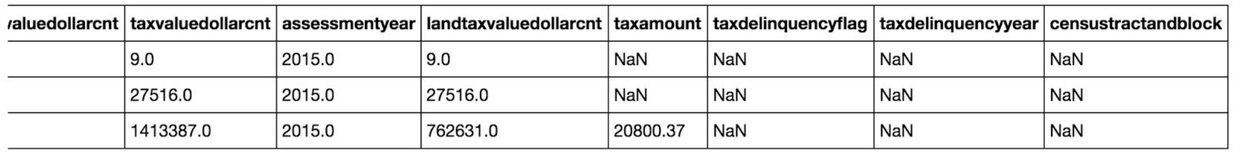
There is a sampling of the data. It’s not easy to see all of them here. Please refer to A Peek inside the Datasets section in capstone.ipynb for more information.



| | parcelid | airconditioningtypeid | architecturalstyletypeid | basementsqft | bathroomcnt | bedroomcnt | buildingclasstypeid | buildingqualitytypeid |
|---|---|---|---|---|---|---|---|---|
| 0 | 10754147 | NaN | NaN | NaN | 0.0 | 0.0 | NaN | NaN |
| 1 | 10759547 | NaN | NaN | NaN | 0.0 | 0.0 | NaN | NaN |
| 2 | 10843547 | NaN | NaN | NaN | 0.0 | 0.0 | NaN | NaN |

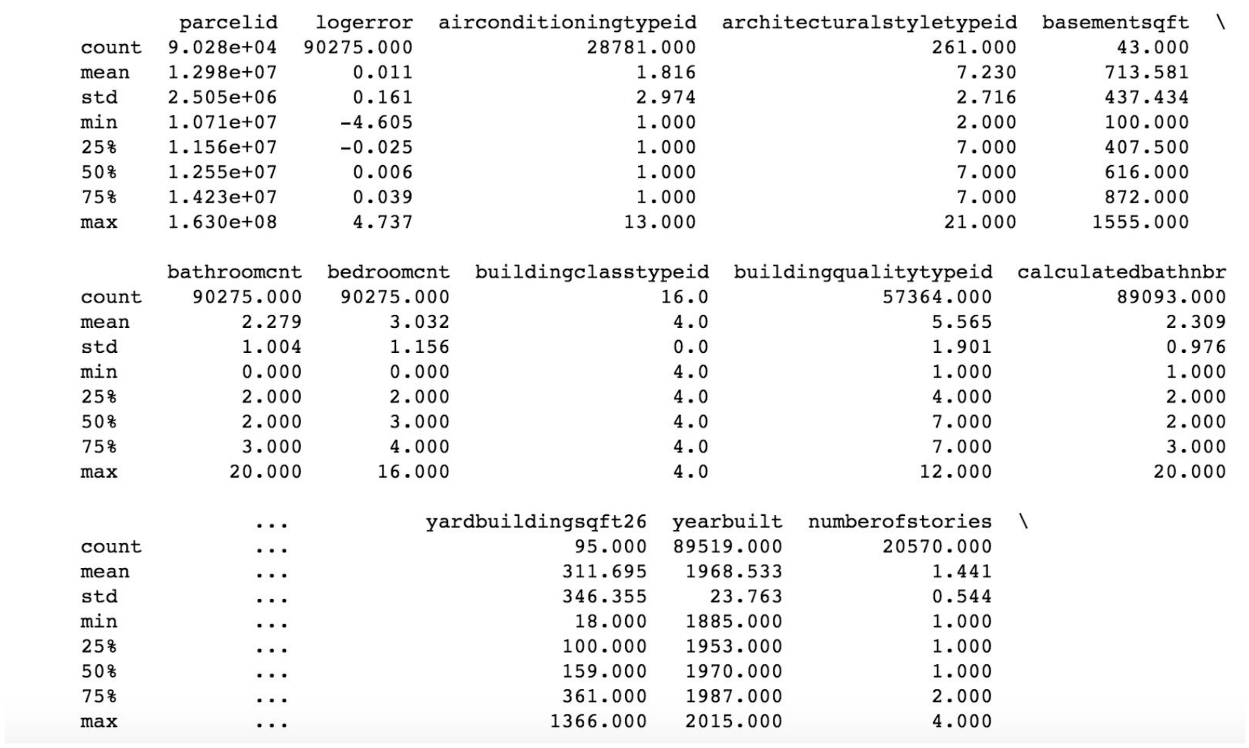


| calculatedbathnbr | decktypeid | ... | numberofstories | fireplaceflag | structuretaxvaluedollarcnt | taxvaluedollarcnt | assessmentyear | landtaxvaluedolla |
|---|---|---|---|---|---|---|---|---|
| NaN | NaN | ... | NaN | NaN | NaN | 9.0 | 2015.0 | 9.0 |
| NaN | NaN | ... | NaN | NaN | NaN | 27516.0 | 2015.0 | 27516.0 |
| NaN | NaN | ... | NaN | NaN | 650756.0 | 1413387.0 | 2015.0 | 762631.0 |



| valuedollarcnt | taxvaluedollarcnt | assessmentyear | landtaxvaluedollarcnt | taxamount | taxdelinquencyflag | taxdelinquencyyear | censustractandblock |
|---|---|---|---|---|---|---|---|
| | 9.0 | 2015.0 | 9.0 | NaN | NaN | NaN | NaN |
| | 27516.0 | 2015.0 | 27516.0 | NaN | NaN | NaN | NaN |
| | 1413387.0 | 2015.0 | 762631.0 | 20800.37 | NaN | NaN | NaN |

There are too many columns. It’s not easy to show all of them here. Please refer to capstone.ipynb descriptive statistics section for more information. The most important one is logerror. You can see from the statistics logerror ranges from -4.605 to 4.737.



```
        parcelid   logerror  airconditioningtypeid  architecturalstyletypeid  basementsqft  \
count  9.028e+04  90275.000              28781.000                   261.000        43.000
mean   1.298e+07      0.011                  1.816                     7.230       713.581
std    2.505e+06      0.161                  2.974                     2.716       437.434
min    1.071e+07     -4.605                  1.000                     2.000       100.000
25%    1.156e+07     -0.025                  1.000                     7.000       407.500
50%    1.255e+07      0.006                  1.000                     7.000       616.000
75%    1.423e+07      0.039                  1.000                     7.000       872.000
max    1.630e+08      4.737                 13.000                    21.000      1555.000

       bathroomcnt  bedroomcnt  buildingclasstypeid  buildingqualitytypeid  calculatedbathnbr
count    90275.000   90275.000                 16.0              57364.000          89093.000
mean         2.279       3.032                  4.0                  5.565              2.309
std          1.004       1.156                  0.0                  1.901              0.976
min          0.000       0.000                  4.0                  1.000              1.000
25%          2.000       2.000                  4.0                  4.000              2.000
50%          2.000       3.000                  4.0                  7.000              2.000
75%          3.000       4.000                  4.0                  7.000              3.000
max         20.000      16.000                  4.0                 12.000             20.000

              ...          yardbuildingsqft26  yearbuilt  numberofstories  \
count         ...                      95.000  89519.000        20570.000
mean          ...                     311.695   1968.533            1.441
std           ...                     346.355     23.763            0.544
min           ...                      18.000   1885.000            1.000
25%           ...                     100.000   1953.000            1.000
50%           ...                     159.000   1970.000            1.000
75%           ...                     361.000   1987.000            2.000
max           ...                    1366.000   2015.000            4.000
```

## Exploratory Visualization

The first exploratory data visualization I did is Transaction Date exploration. I want to understand how data was distributed in this 12 months in this dataset. I counted transactions by month and put them into a new column called transaction_month. I then displayed them using seaborn barplot. X axis is transaction month from Jan to Dec. Y axis is transaction counts. According to the graph, we know that we have good amount of data from Jan to Oct 2016. We have less transaction data from November and December 2016.

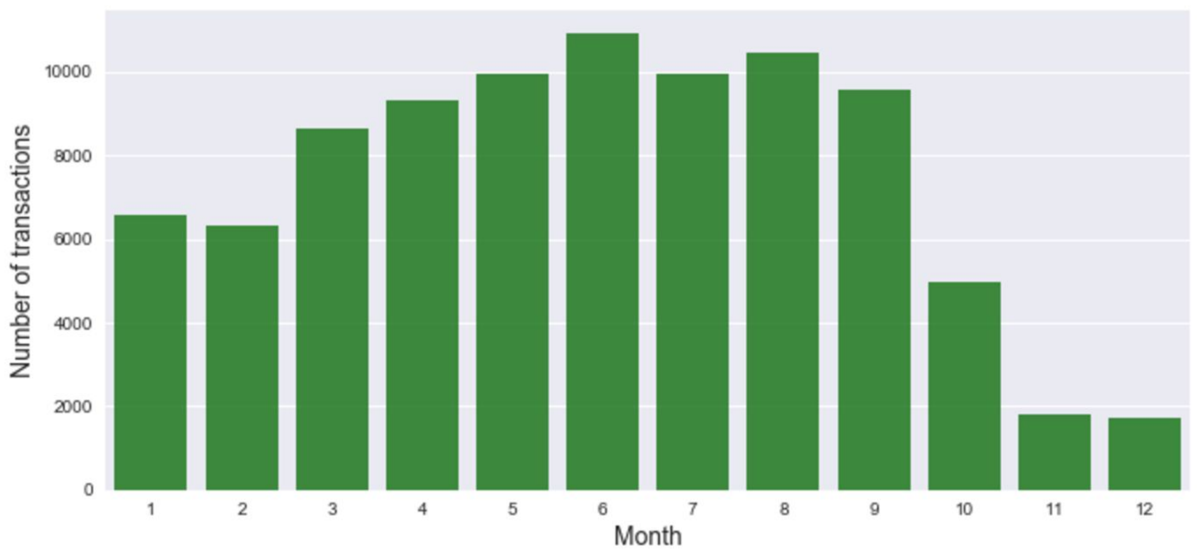


Figure 1

Then the second exploratory visualization that I did is logerror visualization. Logerror is the target values in this problem statement, let's explore it! I sorted all the logerror values for every single row and displayed it by using seaborn regplot. X axis is logerror values and Y axis is indexes of logerror values. According to the graph, Zestimate is doing pretty good. Most of the transactions have very less log errors or 0 log errors. It did have some outliers.

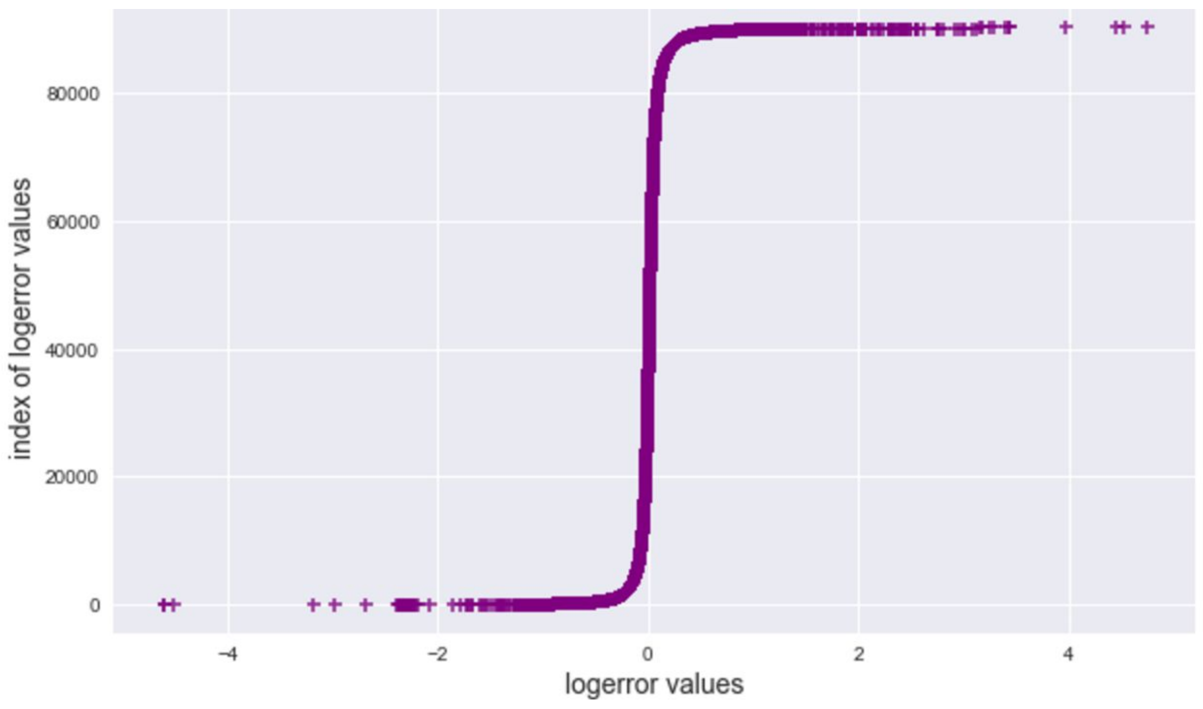


Figure 2

After logerror values exploration, I did want to know what kind of distribution logerror has. So I took out outliers and used seaborn displot to display the distribution for logerror.

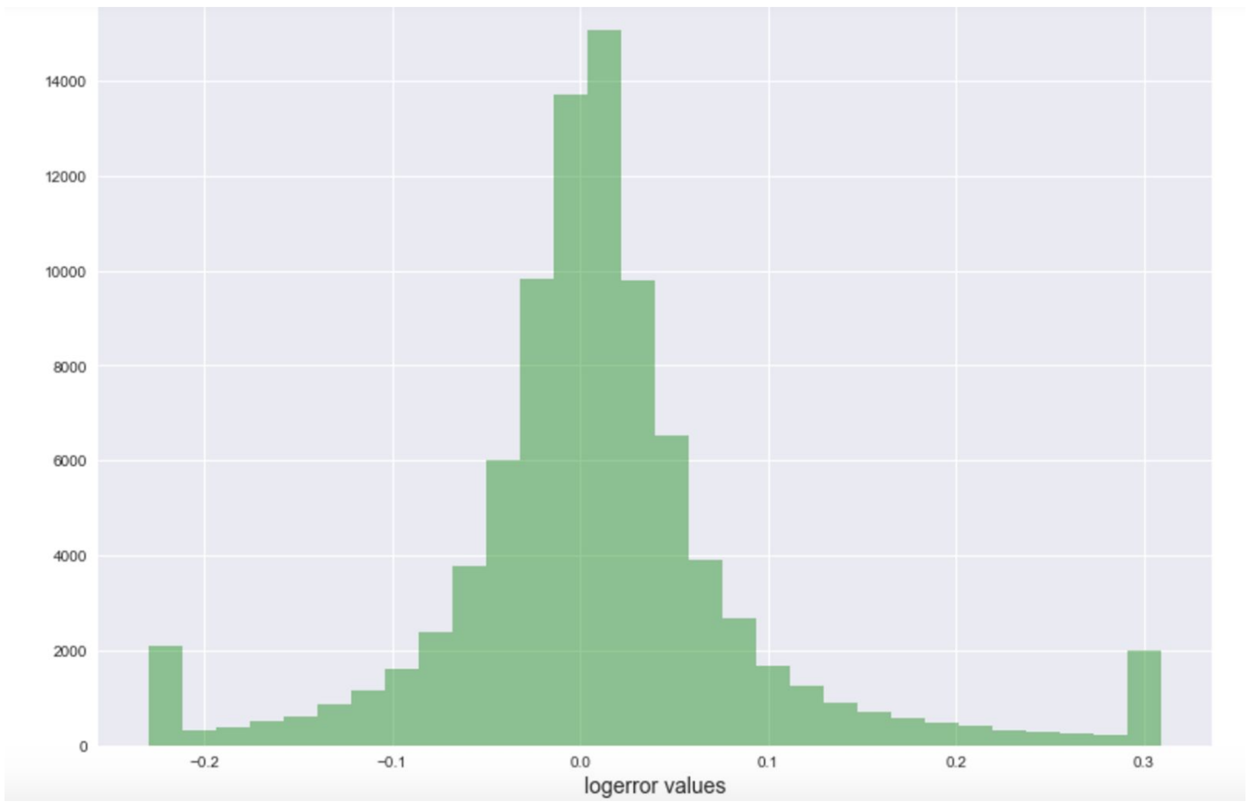


Figure 3

We can see from graph above, it is a very good normal distribution. Since logerror = log(Zestimate) - log(actual price), negative logerror value means Zestimate underestimated, positive value means Zestimate overestimated. We can see from this normal distribution, it estimates pretty accurate most of the time. We need to find out when Zesimate does well and when it doesn't. We need to find out correlations between different features and logerror.

The next thing that I did is to explore correlations between logerror and features. I filled out NaN data first and then calculated coefficient for every column with logerror and then put those coefficient values and columns names to a new data frame. I used matplotlib subplot to display the coefficient of every column with logerror. The Y axis is all the column names and X axis is coefficient of every column with logerror. Just as my intuition says, the graph shows that finished square feet, calculated finished square feet and etc have high and positive coefficient with logerror. Only small amount of features have high coefficient.

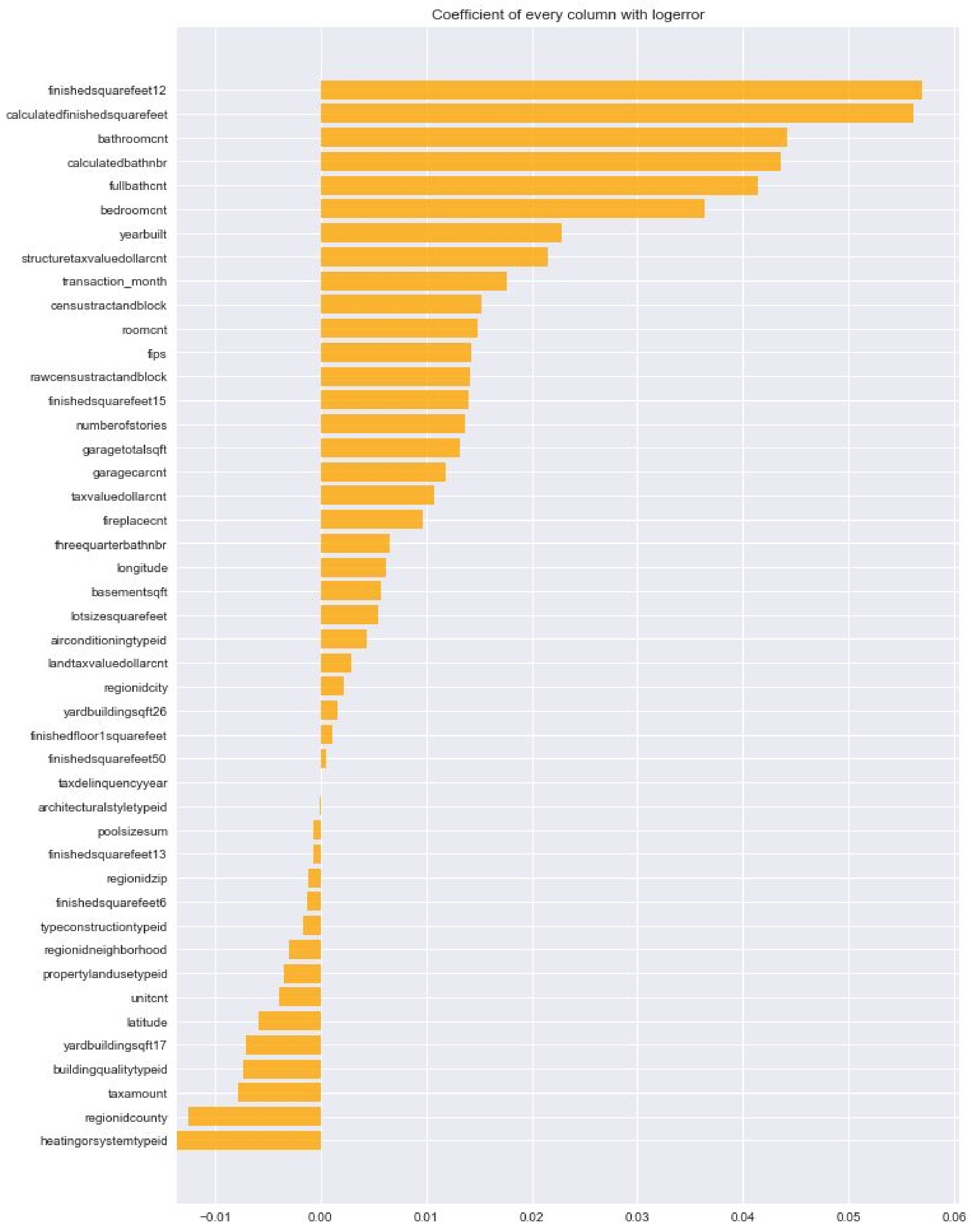

Coefficient of every column with logerror
finishedsquarefeet12
calculatedfinishedsquarefeet
bathroomcnt
calculatedbathnbr
fullbathcnt
bedroomcnt
yearbuilt
structuretaxvaluedollarcnt
transaction_month
censustractandblock
roomcnt
fips
rawcensustractandblock
finishedsquarefeet15
numberofstories
garagetotalsqft
garagecarcnt
taxvaluedollarcnt
fireplacecnt
threequarterbathnbr
longitude
basementsqft
lotsizesquarefeet
airconditioningtypeid
landtaxvaluedollarcnt
regionidcity
yardbuildingsqft26
finishedfloor1squarefeet
finishedsquarefeet50
taxdelinquencyyear
architecturalstyletypeid
poolsizesum
finishedsquarefeet13
regionidzip
finishedsquarefeet6
typeconstructiontypeid
regionidneighborhood
propertylandusetypeid
unitcnt
latitude
yardbuildingsqft17
buildingqualitytypeid
taxamount
regionidcounty
heatingorsystemtypeid
−0.01
0.00
0.01
0.02
0.03
0.04
0.05
0.06

Figure 4

From data visualization above, we found out some important features correlated to logerror. Let's pick up one of them to further examine it.

So I went on and started picking up couple major features to examine their relationship with logerror. First one is calculated finished square feet.You can see from the graph (Ignore the outliers), as the size of Calculated Finished Square Feet increases, logerror decreases. That means Zestimate estimates well for bigger size houses.

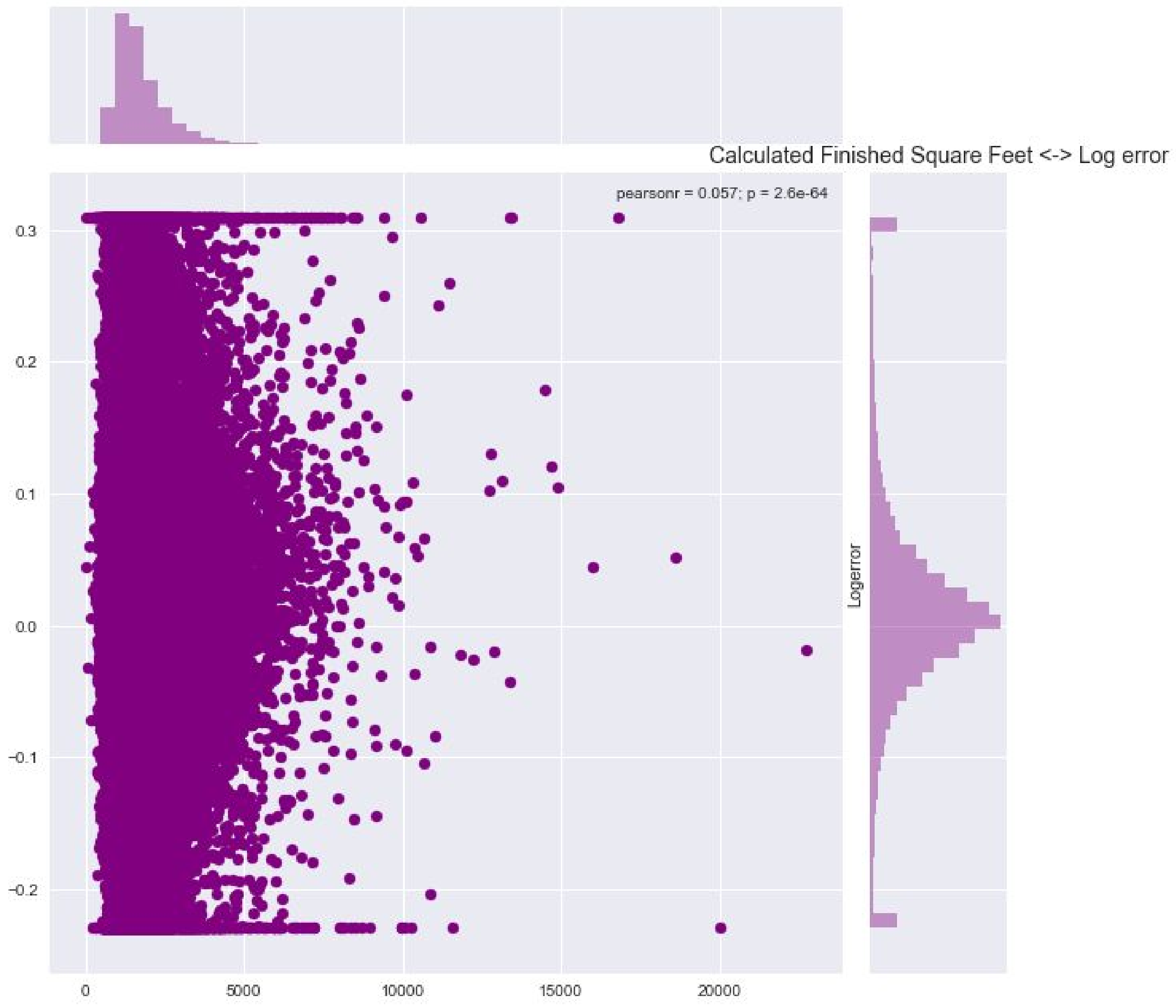


Figure 5

The second one is Finished Square Feet12.

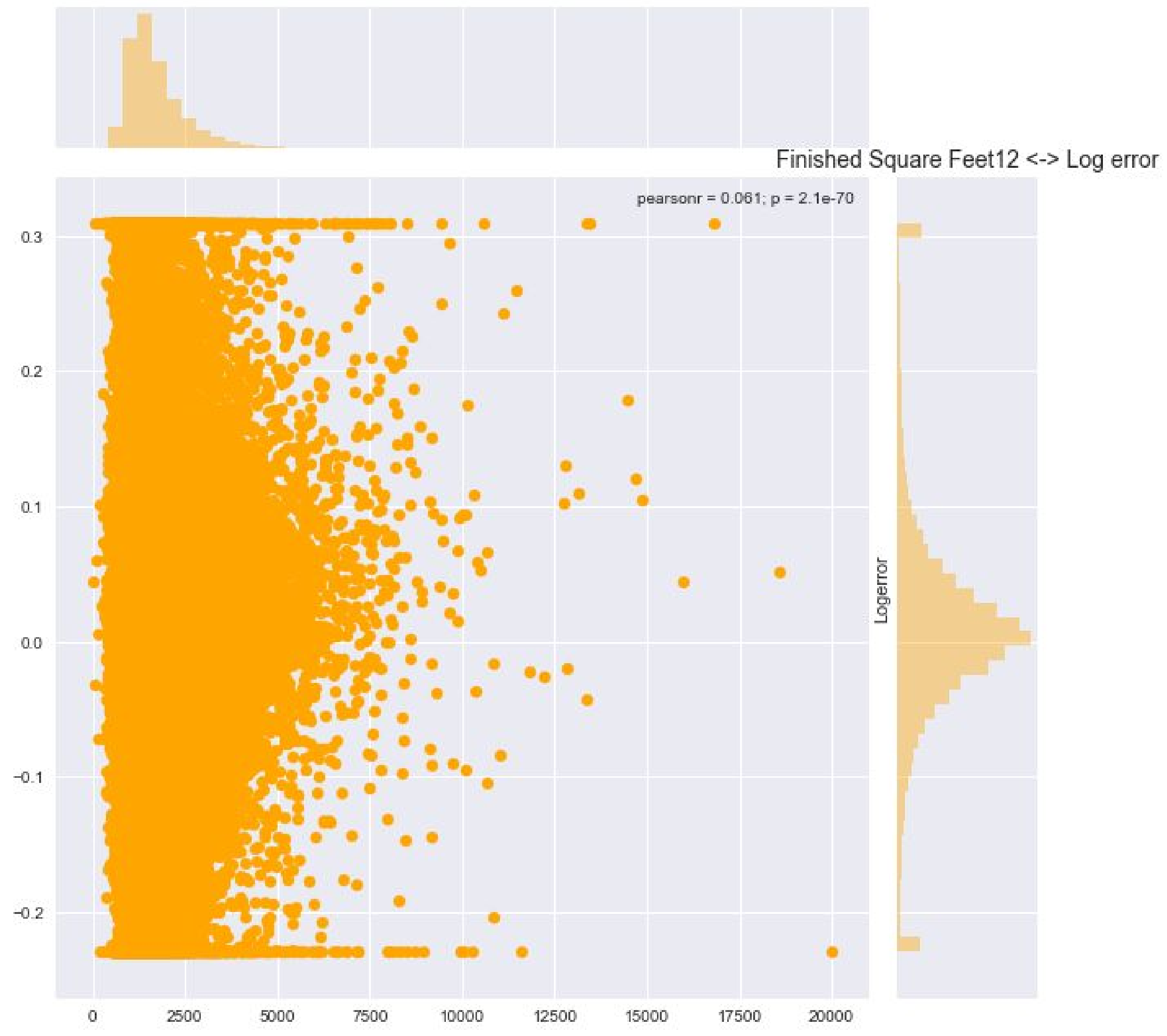


Figure 6

You can see from the graph above (Ignore the outliers), as the size of Finished Square Feet12 increases, logerror decreases. That means Zestimate estimates well for bigger size houses.

The third one is calculatedbathnbr.

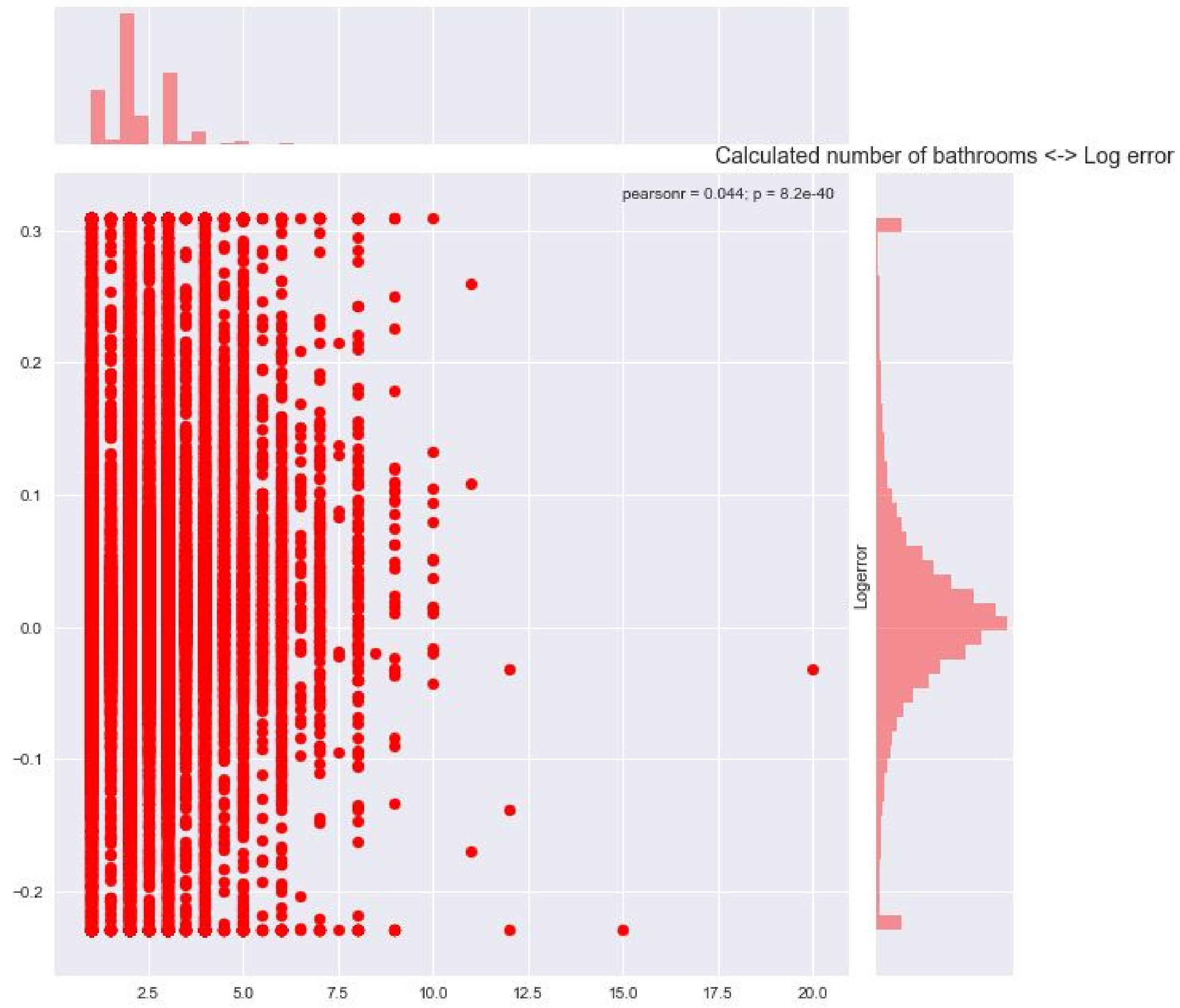


Figure 7

You can see from the graph above (Ignore the outliers), as the size of Calculated number of bathrooms increases, logerror decreases a little bit. This one is not as strong correlated as the previous one.

In summary, finishedsquarefeet12, calculatedfinishedsquarefeet, bathroomcnt, calculatedbathnbr, fullbathcnt, bedroomcnt,censustractandblock,transaction_month, structuretaxvaluedollarcnt,yearbuilt are top 10 correlated features.

## Algorithms and Techniques

Because the problem I am solving in the project is clearly a supervised learning problem. I can choose one of the supervised learning algorithms. Based on the research

I’ve done regarding the supervised learning algorithms, lightGBM and XGBoost are clearly winners for this problem statement in terms of accuracy and performance. First, I cleaned up dataset including dealing with missing values as well as converting non-numerical data to numerical data and remove outliers. Those two algorithms need valid numerical data to train. Besides cleaning up data, I also picked up some important parameters for both lightGBM and XGBoost to train such as number of leaves and numbers of rounds boosting. Then I did feature selections to find most important features. Then I built lightGBM and XGBoost models and tune parameters to get the predicted results and mean absolute errors. Finally, I combined two models together to get the best results.

LightGBM is a fast and high performance gradient boosting algorithm based on decision trees. The difference between lightGBM and other decision tree algorithms is that it splits the tree leaf wise while others split the tree level wise. The leaf wise algorithm can reduce more loss with high accuracy and super fast performance. XGBoost stands for eXtreme Gradient Boosting. Let’s quote creator’s words to explain XGBoost: *“The name xgboost, though, actually refers to the engineering goal to push the limit of computations resources for boosted tree algorithms. Which is the reason why many people use xgboost.” - Tianqi Chen (Creator of XGBoost)*. It’s an implementation of gradient boosting decision tree to improve performance and speed of gradient boosting decision tree. A lot of Kaggle competition winners used XGBoost for their models.

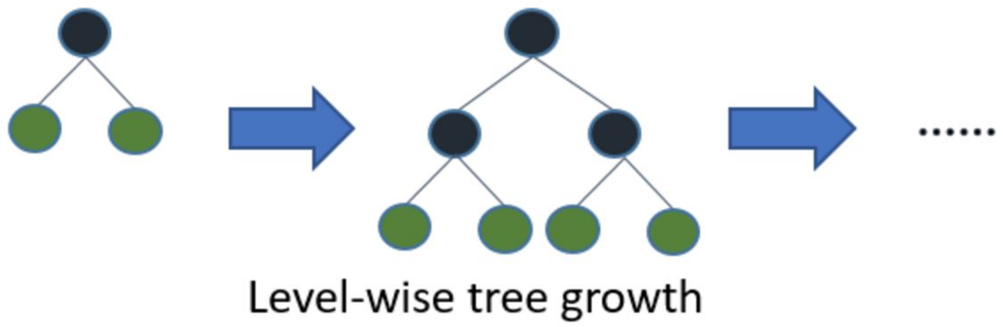


Figure 8 XGBoost - Level-wise tree growth

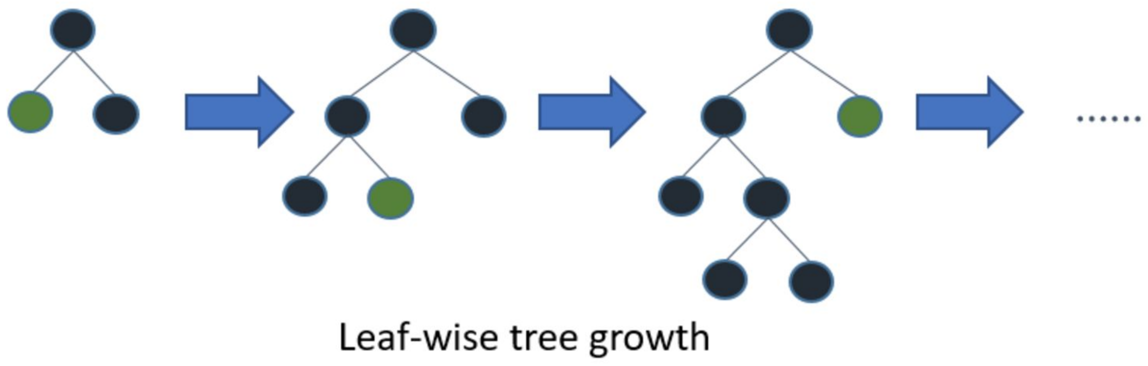
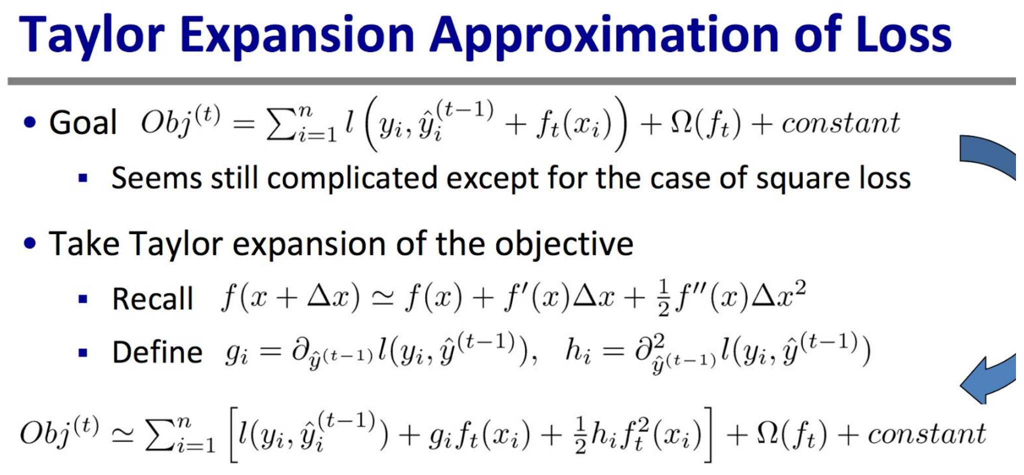


Figure 9 LightGBM leaf-wise tree growth
(this two tree graphs are from https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/)

Let's explain those algorithms a little bit further. Boosting refers to a group of algorithms that can convert weak learns to strong learners. 1) base learner assigns equal weight to each observation. 2) It will assign higher weights to observations having prediction errors, if there is any errors caused by first base learning algorithm. 3) repeat second step until higher accuracy is achieved or limit is reached. 4) combine all the weak learners and generate a strong learner which improves prediction power. There are different types of Boosting such as AdaBoost (**Ada**ptive **Boost**ing), Gradient Tree Boosting, and XGBoost. Let's talk about Gradient Tree Boosting first, then I will get into XGBoost and lightGBM. Gradient boosting has 3 elements: 1) A loss function 2) A weaker learner 3) An additive model to add weaker learners to reduce the loss function. Loss function can be defined according the problem being solved. Decision trees are used as the weak learner in gradient boosting. Regress trees are used so that subsequent model outputs can be added together and fix the residuals in the prediction. Trees are added one at a time. A gradient descent procedure is used to minimize the loss when adding trees. Gradient boosting can overfit very quickly since it's a greedy algorithm. Regularization methods which penalize different parts of the algorithm and improve the performance by decreasing overfitting. This is where XGBoost comes into the picture.

XGBoost is built on base Gradient Boosting Model with the goal of improving speed and accuracy of original model. If you understand Gradient Boosting Trees, you understand most of part of XGBoost. Let's talk about what XGBoost does different from Gradient Boosting Trees. First, XGBoost uses second-order gradient of the loss function (add to the first-order gradient) based on Taylor expansion of the loss function. Taylor expansion of different types of loss functions can be plugged into the same algorithm for greater generalization. Second, XGBoost transforms the loss function into a complicated objective function which has regularization terms. This new transformation - extension of the loss function adds penalties when it adds new decision tree leaves to the model. Penalties are proportional to the size of the leaf weights. This prevents the growth of the model and prevents overfitting. In addition to accuracy improvement, it also improves speed a lot because XGBoost uses cache and memory optimization. XGBoost performs so much better than Gradient Boosting Trees. Taylor expansion:



## Taylor Expansion Approximation of Loss

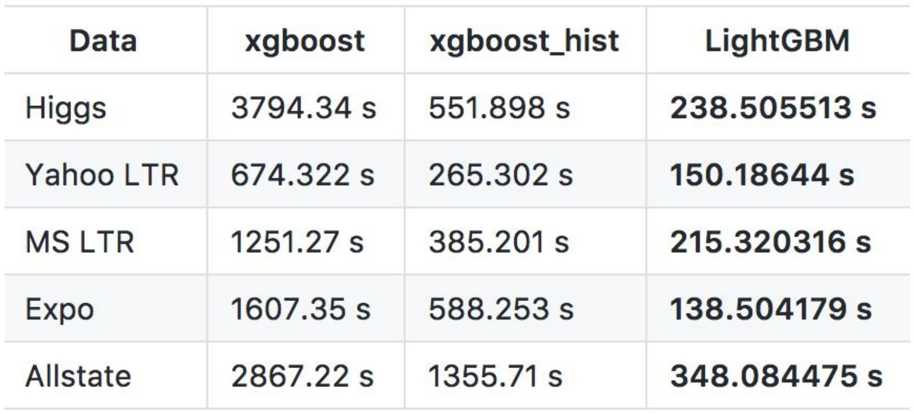
- Goal $Obj^{(t)} = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) + constant$
    - Seems still complicated except for the case of square loss
- Take Taylor expansion of the objective
    - Recall $f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2} f''(x)\Delta x^2$
    - Define $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)}), \quad h_i = \partial^2_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$

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

Equations are from (https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf)

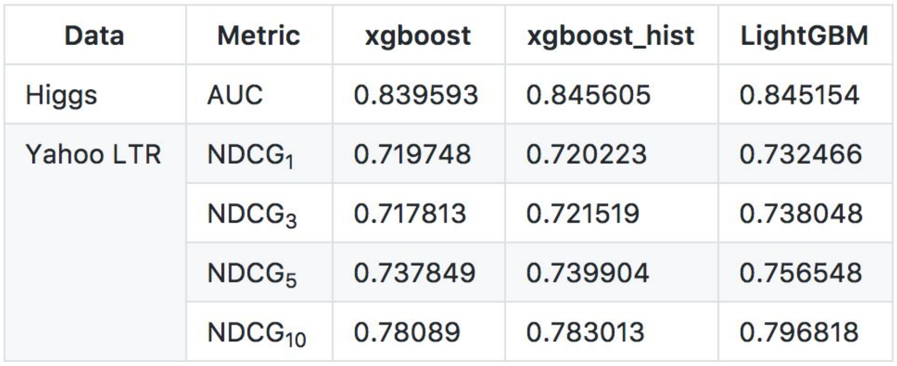
LightGBM is the new challenger for XGBoost. LightGBM is also a Gradient Boosting framework. If you understand Gradient Boosting Trees, you understand most of part of lightGBM. LightGBM is very similar to XGBoost. It optimizes performance. It gets better accuracy than XGBoost. It's faster than XGBoost. As I discussed above, one thing lightGBM does differently is to grow trees leaf wise rathern than the way XGBoost grows the tree - level wise. In this way, it can control overfitting even further and terminate growth at optimal point, which makes it more accurate and faster. Here are some performance comparison between XGBoost and lightGBM from lightGBM github: (https://github.com/Microsoft/LightGBM/blob/master/docs/Experiments.rst#comparison-experiment)

Speed



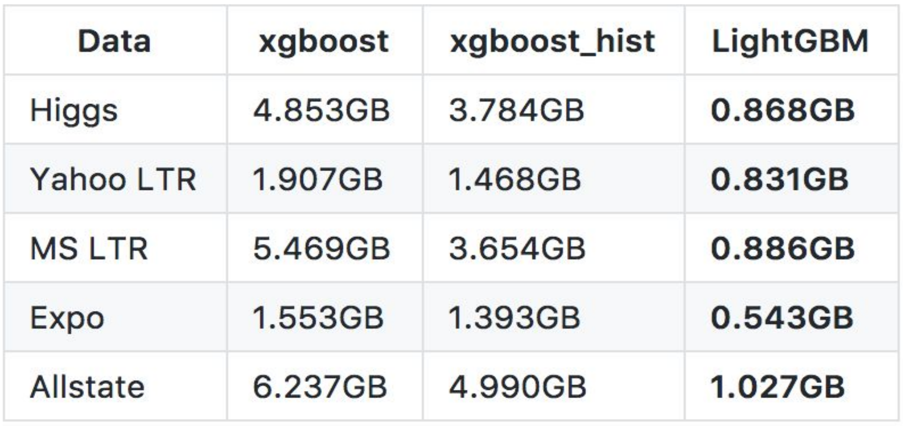
| Data | xgboost | xgboost_hist | LightGBM |
|---|---|---|---|
| Higgs | 3794.34 s | 551.898 s | **238.505513 s** |
| Yahoo LTR | 674.322 s | 265.302 s | **150.18644 s** |
| MS LTR | 1251.27 s | 385.201 s | **215.320316 s** |
| Expo | 1607.35 s | 588.253 s | **138.504179 s** |
| Allstate | 2867.22 s | 1355.71 s | **348.084475 s** |

Accuracy



| Data | Metric | xgboost | xgboost_hist | LightGBM |
|---|---|---|---|---|
| Higgs | AUC | 0.839593 | 0.845605 | 0.845154 |
| Yahoo LTR | $NDCG_1$ | 0.719748 | 0.720223 | 0.732466 |
| | $NDCG_3$ | 0.717813 | 0.721519 | 0.738048 |
| | $NDCG_5$ | 0.737849 | 0.739904 | 0.756548 |
| | $NDCG_{10}$ | 0.78089 | 0.783013 | 0.796818 |

Memory utilization



| Data | xgboost | xgboost_hist | LightGBM |
|---|---|---|---|
| Higgs | 4.853GB | 3.784GB | **0.868GB** |
| Yahoo LTR | 1.907GB | 1.468GB | **0.831GB** |
| MS LTR | 5.469GB | 3.654GB | **0.886GB** |
| Expo | 1.553GB | 1.393GB | **0.543GB** |
| Allstate | 6.237GB | 4.990GB | **1.027GB** |

## Benchmark

During my research, I found one very good baseline lightGBM model with a very less mean absolute error. I can build and tune my first lightGBM model based on this model. This is the model I refer to:

https://www.kaggle.com/guolinke/simple-lightgbm-starter-lb-0-06487 The reason I chose this benchmark model is that 1) This model gives a pretty good starting score. 2) It sets up some pretty good parameters such as learning rate, number of leaves, boosting type , objective as regression and metric as Mean Absolute Error. This model perfectly fits the problem and it runs fast. There is one thing I do want to discuss regarding the model. It’s said the MAE (mean absolute error) is 0.06487 on Kaggle site. However, when I ran it on my Jupyter notebook, it gave MAE as 0.06966. My suspect is that they trained and tested with different datasets. I posted my question on the kennel discussion. Nobody replied my question yet. Regardless, I will use 0.06966 as my baseline MAE. That’s the score I targeted at for my model tuning and model selection.

## III. Methodology

### Data Preprocessing

Since both models need data to be well preprocessed before training, I did preprocess data. The first thing I did for data preprocessing is to merge two datasets - train dataset and property dataset together by parcel Id. Then I did data visualization to find out that most of columns have NaN data which needs to be taken care of. I filled out those NaN data with the mean of every single column. Besides NaN data, this merged dataset also has non numerical data. I found out those 5 objects fields. I used LabelEncoder to convert those objects to numerical data. After I dealt with missing and non numerical data, I then found out the correlation of each column with targeted value - logerror and also found out the feature importance for every single column. Then I finally excluded some unimportant features from merged dataset based on those findings when I trained models.

### Implementation

After preprocessing the data, I implemented benchmark model with cross validation. First, I dropped couple unimportant features from merged data set before I assigned it to training data and targeted values. Then I splitted up the training dataset at 90000 to make part of it as training set and another part of it as validation set. I loaded both final training dataset and targeted values into lightGBM dataset to speed up the performance.

Before training, couple parameters was picked up and set them to the values this model set them to. Those parameters are min_data, objective, learning_rate, min_hessian, sub_feature, boosting_type, num_leaves, and metric. Min_data is the minimum number of data in one leaf. This can be potentially to used to control overfitting. Benchmark model used 500 as min_data. This is a petty optimal choice. I tried lower and higher number of min_data, they all gave worse performance than 500. Regression was chosen for objective function because this is a regression supervised learning problem. Learning rate was set to 0.002. This can be used to control training speed. Lower rate lowers overfitting speed. Min_hessian means min number of hessians in a leaf for a split. It was set to 1. Higher value potentially decrease overfitting. Sub_feature is alias for feature_fraction. When this value is less than 1.0, it will make lightGBM randomly select part of features on each iteration. Sub_feature was set to 0.5. So 50% of features were randomly selected before training each tree. This can be used to deal with training speed and overfitting. For boosting_type, gbdt was chosen. Gbdt means traditional gradient boosting decision tree. Num_leaves means number of leaves in one tree. Num_leaves was set to 60. The last parameter is metric. Since I use mean absolute error as evaluation metric for this model, this parameter was set to mae. After all the parameters were set, training was started with one extra parameter - number of iterations. Number of iteration was set to 700.

After 700 iteration training with test data and cross validation data set, a benchmark model was born. In order to evaluate the benchmark model using mean absolute error, test dataset preparation was performed. Train_2017 and properties_2017 were used as test dataset. Test dataset preparation process was the same as training dataset preparation. Test data sets was merged first. Then missing values were filled out and non numerical data was transformed. When test data was ready, benchmark model was used to predict targeted values (logerror) for test dataset. Mean_absolute_error function from skLearn was used to get MAE score. Predicted targeted values and actual target values (logerror) were put into mean_absolute_error. MAE score for benchmark model is 0.06966.

After benchmark model was tested out, couple improvements were made from this model and couple new model were implemented based on this model. The first model was made after benchmark model used exactly the same parameters and techniques. The only difference is that this new model set different values for all this parameters after try and errors as well as tuning. The first new model was called lgb_model. MAE for lgb_model is 0.06950. The second new model was made used different algorithm - XGBoost and different parameters. It’s called xgb_model. MAE for xgb_model is 0.06962. The third new model was made was combined model. This model combined two models - lgb_model and xgb_model together. MAE for combined model is 0.06939. I will talk about more regarding tuning in the next section - refinement.

## Refinement

As I discussed above, the evolution of model selection is from benchmark model to 1)lgb_model;2)xgb_model;3)combination of lgb_model and xgb_model. The final model is the combination of lgb_model and xgb_model.

A lot of tuning was performed on the benchmark model to derive lgb_model. The first thing I did to tune benchmark model was to test out optimal cross validation split. The original number is 90000. Since the total row numbers of training set is 90275. That 90000 split didn’t give cross validation a lot of data to deal with. So I used split as 80000, MAE was improved from 0.06962 to 0.06960. And I kept going and tried 70000 (MAE 0.06961), 79000 (MAE 0.06960). Finally I found 75000 (MAE 0.06959). Learning rate could be used to control speed and overfitting. Learning rate was changed from 0.002 to 0.001, MAE was improved from 0.06959 to 0.069584. Number of iterations was changed to 600, I didn’t see a lot of improvement, so I kept it as 700. Number of leaves could used to control overfitting as well. It was increased from 60 to 70. MAE was improved from 0.06958 to 0.06957. Then I increased it again from 70 to 200. MAE was improved from 0.06957 to 0.069569.

I wasn’t satisfied with the result. I was looking for some breakthrough that could lead to much bigger improvement. Then I started looking at features that I selected. The first thing I did was to pick all the less important features based on all the feature correlation and feature importance analysis I did earlier. So I dropped all those features - 'airconditioningtypeid’,'architecturalstyletypeid’,'basementsqft’,'buildingclasstypeid’,'deck typeid’,'threequarterbathnbr’,'finishedsquarefeet6’,'finishedsquarefeet13’,'finishedsquare feet50’,'fireplacecnt’,'fireplaceflag’,'fullbathcnt’,'hashottuborspa’,'longitude’,’parcelid', ’poolcnt’, ’poolsizesum’, ’pooltypeid10’, ‘pooltypeid2’, ‘pooltypeid7’, ’propertycountylandusecode’,'propertyzoningdesc’,'regionidcity’,'storytypeid’,'unitcnt’,'yar dbuildingsqft26’,'assessmentyear’,’taxdelinquencyflag’,’taxdelinquencyyear'. The result was bad. MAE was getting so much worse. I had to revert back to the original feature selection. The original feature selection - dropped those unimportant feature : 'parcelid', 'logerror', 'transactiondate', 'propertyzoningdesc', 'propertycountylandusecode'. After some trial and error, I take out feature 'taxdelinquencyyear' and MAE was improved from 0.069569 to 0.06950. MAE for lgb_model is 0.06950.

Then I set up a xgb_model. I used some basic parameters for xgb_model such as 'eta': 0.037, 'eval_metric': 'mae', 'lambda': 0.8, 'alpha': 0.4, 'max_depth': 6, 'subsample': 0.90, 'objective': 'reg:linear', 'base_score': y_mean, 'silent': 1. Eta is learning rate which is the same as how I explained learning rate earlier for lightGBM. It can be used to control overfitting. Eval metric was set to mean absolute error. Lambda is L2 regularization term on weights. Alpha is L1 regularization term on weights. Both of them can be used to make model more conservative when it is increased. The difference between L1 and L2 is that L2 is the sum of squared weights and L1 is only the sum of weights. Max depth is the maximum depth of a tree. It could make model more likely to be more complex and overfitting when it gets increased. Subsample means the ratio of training dataset that XGBoost randomly chooses to grow trees. This can be used to prevent overfitting. Objective function is linear regression because this is a regression supervised learning problem. Base score is the initial prediction score of all dataset. Silent 0 means printing out message and silent 1 means silent mode - don't print out message. Number of boosting round was set to 150.

After basic parameters were set up for XGBoost, I trained model using training dataset. MAE was 0.069794 for basic XGBoost model. Given how many Kaggle winners used this algorithm, it's petty disappointing. So I tuned couple parameters to make it better. The first thing I tuned was max depth. This is one parameter we can use to control overfitting. I changed it from 6 to 8. MAE was getting so much worse from 0.069794 to 0.069945. I knew it's overfitting already. So I reduced it from 6 to 5. MAE was improved from  0.069794 to 0.069713. Then I started tuning number of boosting round. I changed it from 150 to 180 and MAE was improved from 0.069713 to 0.069627. I tried to changed it higher than 180, MAE was getting worse. Then I reduced number of boosting round from 180 to 170. MAE was improved from 0.069627 to 0.069625. Then I reduced it again from 170 to 160. MAE was improved from 0.069625 to 0.069623. 0.069623 was the final MAE for xgb_model that I built.

After training on lightGBM and XGBoost models, I found out that lightGBM is faster and more accurate. That being said. XGBoost is still pretty good and much better than traditional gradient boosting trees. Then I thought why not just combine the two together to make a better meta model. The final model was derived from these two models. Each one of them has different weight. I tuned the final model by weight. Since lgb_model performed better than xgb_model, I knew the final model would have light more weight on lgb_model and less weight on xgb_model. Regardless, I started with a little higher weight on XGBoost and worked all the way down and compared MAEs. First, I set XBGoost weight as 0.7 and MAE was 0.06945. I then reduced it to 0.6 and MAE was 0.06941. Then reduced it again to 0.5 and MAE was 0.069398. Then reduced it again to 0.4 and MAE was 0.069392. And then reduced it again to 0.3 and MAE was 0.069400. So when XGBoost weight was 0.4, combined model got the best MAE. XGBoost weight was set to 0.4 and LightGBM weight was set to 0.6. That's the final model! The final

MAE for the combined model was 0.069392. Compared to benchmark model, MAE was improved from 0.06966 to 0.069392.

I did face couple problems. I planned to use K-fold cross validation for lightGBM model. However, sklearn k-fold cross validations didn’t work well with lightGBM. I had to change it back to normal cross validation. The second problem is grid search - hyperparameter optimization. I also tried to use sklearn grid search on lightGBM and it didn’t work well. lightGBM is new and powerful. One downside of being so new is lack of documentations and community support. I am glad I tested out this algorithm and I should contribute back to lightGBM community to make it better.

## IV. Results

## Model Evaluation and Validation

Let’s summarize what’s the final model. The final model is the combination of lgb_model and xgb_model. Xgb weight is 0.4 and lgb weight is 0.6. The final MAE is 0.06939.

As I explained earlier, min_data, objective, learning_rate, min_hessian, sub_feature, boosting_type, num_leaves, and metric were set for lightGBM model. After tuning, lgb_model used 500 as min_data which is an optimal choice. Regression was chosen for objective function because this is a regression supervised learning problem. Learning rate was set to 0.001. Lower rate was used to lower overfitting speed. Min_hessian was set to 1. Higher value potentially decrease overfitting. Sub_feature was set to 0.5. So 50% of features were randomly selected before training each tree to deal with overfitting and speed up performance. Gbdt was chosen for boosting_type. Gbdt means traditional gradient boosting decision tree. Num_leaves was set to 200 to boost performance. Meric was set to mae since it is the evaluation metric for this project. Number of iteration was set to 700. Key parameters were set to optimal values.

After tuning for xgb_model, those are the parameters set for xgb_model  'eta': 0.037, 'eval_metric': 'mae', 'lambda': 0.8,  'alpha': 0.4, 'max_depth': 5, 'subsample': 0.90, 'objective': 'reg:linear', 'base_score': y_mean, 'silent': 1. Lower Eta - learning rate was set to to control overfitting. Eval metric was set to mean absolute error. Lambda and alpha were set to a lower value to model less conservative. Max depth was set to 5 because higher max depth causes overfitting and lower max depth causes underfitting according to my parameter tuning test. Subsample to set to 0.9 to prevent overfitting. Objective function is linear regression because this is a regression supervised learning problem. Base score was set to the mean of targeted values to give the initial prediction score of all dataset. Number of boosting round was set to 160 because higher boosting rounds and lower boosting rounds made MAE worse according to tuning test.

The combined model of lgb_model and xgb_model was the chosen model. The weight of lgb_model and xgb_model is 0.6, 0.4 respectively. Lgb model has higher weight because it performed better than xgb model. To get a better result, lgb model would have to take more weights. The weights were chosen based on several tuning tests to get the optimal MAE. Optimal weights were chosen in this case. The combined model performed better than any one of them. The final model: 0.4*xgb_model.predict(xtest) + 0.6*lgb_model.predict(xtest)

The final model was tested out against 3 different test dataset. The first one was the merged dataset from 2017 Jan to March, MAE for it was 0.070981. The second one was the merged dataset from 2017 April to June, MAE for it was 0.06834. The third one was the merged dataset from 2017 July to Sep, MAE for it was 0.069227. We can see from those results, the final model performed pretty consistent regardless of test dataset changes. Besides, I've built model with different training set earlier as well. I have used data from 2016 Jan to Oct instead of the whole year of 2016 to build exactly the same final model. And the final MAE for that model for the same testing dataset was 0.06950 which is little bit worse than our MAE 0.06939 which I got by using more data (2016 Jan to Dec). Feeding more data to the model did improve performance from 0.06950 to 0.06939 (MAE). Regardless of training data and testing data changes, the final model performed pretty consistently and the way of deriving final model got demonstrated to be valid.

## Justification

The final model performed better than benchmark model. It improved MAE from 0.06966 to 0.06939. As I discussed above, the final model is 0.4*xgb_model.predict(xtest) + 0.6*lgb_model.predict(xtest) - a combination of lgb model and xgb model. The benchmark model already used very good algorithm lightGBM with some optimal parameters and cross validations. It gave a very good MAE for this problem statement and presented a challenge to tune.

There are couple reasons why I think the final model is valid and solved the problem. First, let's talk about test harness. Test harness is the data I used to train and test for an algorithm to get MAE measurement. During the experimentation, I tested out multiple algorithms using same test harness against the same measurement metric - MAE. I used the same train and test dataset for 4 algorithms - benchmark model, improved lightGBM model, XGBoos Model, combined model of lightGBM and XGBoost. MAEs for them are 0.06966, 0.06950, 0.06962, 0.06939 respectively. Since a group of different learning algorithms performed consistently well on the problem, the problem is pretty learnable. The selected data has pretty good learnable structure for algorithms to learn from. Besides, we can see the performance of combined model exceeded the other models from the experimentation. Second, test data and training data sets were well prepared including shape spot check, data visualization for distribution, filling out missing values, transforming non numerical data, and feature selection based on feature importance and tuning. Third, cross validation and parameter tuning were used for the model. Cross validation and parameters tuning can help prevent underfitting and overfitting problems. Please refer to the tuning and cross validation process I talked about earlier to get more information regarding how I derived the optimal parameters and cross validation split. The final model was also tested out against 3 different test dataset and got pretty consistent MAE. Last but not least, I also tested out upper limit - the best MAE the model can ever get. I used the same training dataset as testing dataset, I got MAE around 0.054. This means when everything is perfect and model predicts exactly right for every single targeted value, this is the best MAE you can get - 0.054. This is caused by the intrinsic error you can’t avoid. My final model MAE is 0.06939. It is in a pretty reasonable range considered that upper limit. This problem of predicting logerror is a regression supervised learning problem. The final model is the combination of two regression supervised learning algorithms. The solution perfectly fits the problem statement.

## V. Conclusion

## Free-Form Visualization

There were a lot of visualization performed during experimentation. I have already discussed and displayed some of them in Exploratory Visualization section. The first exploratory data visualization I did is Transaction Date exploration.The second exploratory visualization that I did is logerror visualization. After logerror values exploration, I did want to know what kind of distribution logerror has. I took out outliers and used seaborn displot to display the distribution for logerror. The next one that I did is to explore correlations between logerror and features. I filled out NaN data first and then calculated coefficient for every column with logerror and then put those coefficient values and columns names to a new data frame.Then I went on and started picking up couple major features to examine their relationship with logerror. First one is calculated finished square feet.The second one is Finished Square Feet12.The third one is calculatedbathnbr.

Let’s pick up one visualization I haven’t discuss in Exploratory Visualization section. In order to understand how many missing values we have for every single column, I used matplotlib subplots display total number of missing values for every column. I built a new dataframe with column names and missing values counts. Then I sorted it by missing values counts in descend order. It’s shown that the columns which have most missing values displayed at the top and the columns with less missing values or none missing values displayed at the bottom. X axis is the total number of missing values for every column. Y axis is the column name. From the graph, it’s easily to tell more than half the columns have big number of missing values and around one third of columns don’t have missing values at all. I know what to do with raw data before feed them into models according to this missing values graph. The graph is a good indication and mechanism to give us a good intuition about input data issues.

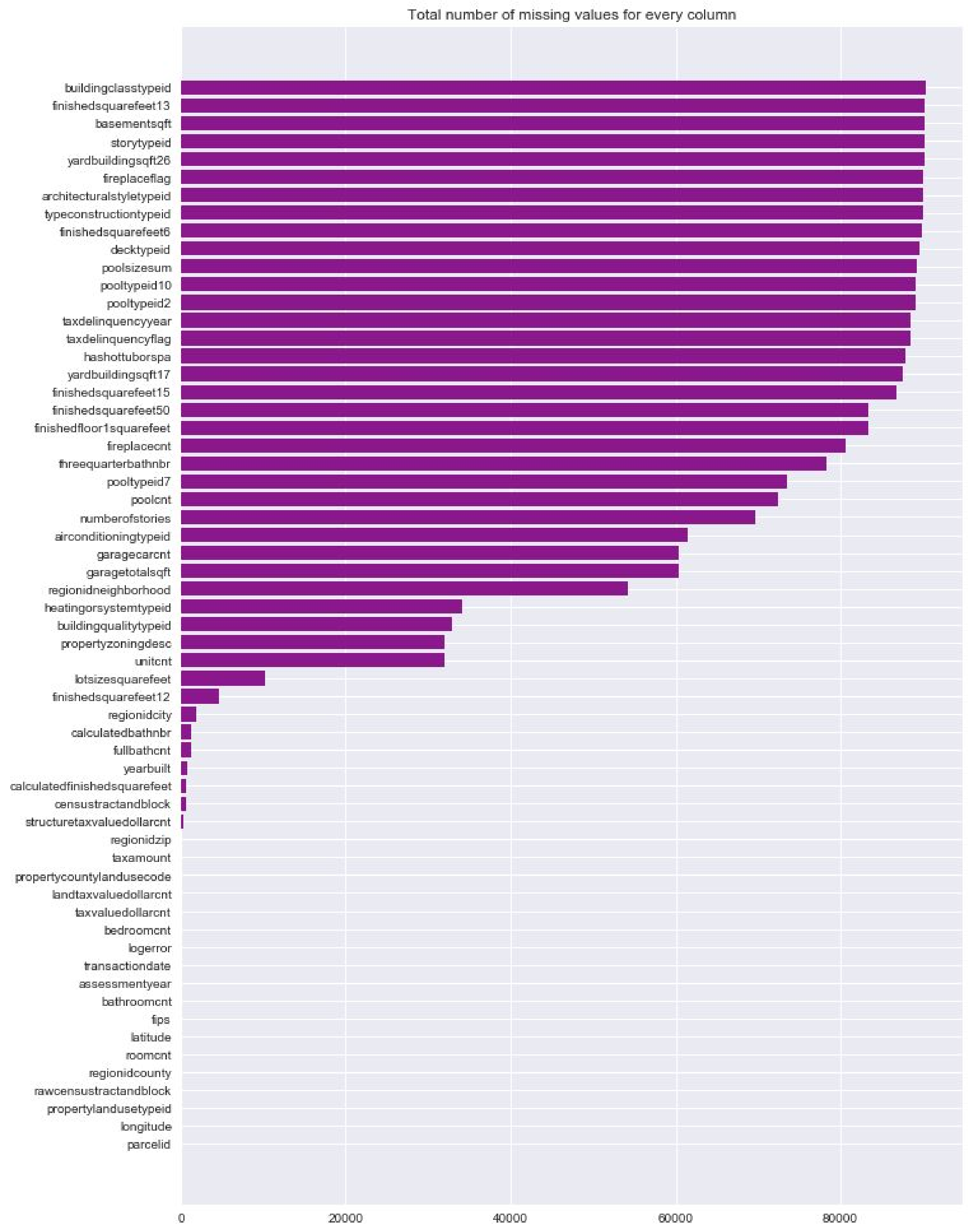


Figure 10

## Reflection

Let’s summarize the end to end problem solution. First, exploratory data visualization was performed on couple aspects of the problem such as input dataset, missing values, non numerical data, distribution of targeted values, and coefficient of features with targeted values. After understanding dataset, data preprocessing was taken out to make sure training dataset was well prepared and valid. Then benchmark model was implemented with some optimal parameters and cross validation as well as tested it out with testing data against performance metric MAE. After benchmark model implementation, cross validation was improved and parameters were tuned to generate the second model - lightGBM model. lightGBM model was trained and tested out against MAE. It got a better MAE. XGBoost model was implemented and tuned after that. Eventually, a meta model - the combination of lightGBM and XGBoost models was born. The final combined model was tested out against 3 different testing datasets to check its consistency. The final model got the best MAE among all the models and it performed pretty consistent across all different test datasets. I have already discussed why I think the final model is valid in Justification section.

Let’s talk about some interesting or difficult aspects regarding the project. One aspect of this project I found is really interesting is technical challenge. Because I used a very good benchmark model - lightGBM as starter and the benchmark model used cross validation and some optimal parameters. LightGBM is considered the new winning algorithm for Kaggle competitions. It’s not that easy to challenge the benchmark model. During the experimentation, I failed so many time to tune but eventually I was able to find the crucial parameters to tune and modified cross validation as well to get better performance. There is another reason why the technical part is so interesting. I’ve got chance to try out both lightGBM and XGBoost models. XGBoost is the winning algorithm for many Kaggle competitions because it has so much better performance in terms of speed and accuracy than traditional gradient boosting trees. LightGBM just came into the game. It’s pretty new. It’s the new challenger. People claimed that lightGBM is much better than XGBoost. I’ve tested it out and it turned out that lightGBM was faster and more accurate. The most exciting part is the part where I combined those winning algorithms together to produce a better model than any one of them alone. This is just rewarding to see the experiment went the direction you wanted it to go to.

Difficult aspect is also the technique part. I didn't know any of those algorithms before I started the project. I picked it up while I was doing the project. I spent some time to set them up and debug them when some weird errors showing up. Because LightGBM is so new and it's lack of community support and documentations, It's hard to find out and learn solutions. For example, for sklearn k-fold cross validation, I tested it out with lightGBM, it just didn't compile; so I changed it back to normal cross validation - test data split. XGBoost also has similar issues. When you try something really new, those issues are going to happen. But I learned a lot during experimentation and I fixed some difficult and weird errors just by using intuitions. I should contribute back to their github community with my solutions.

All in all,  I believe the solution could be used in general settings because all the reasons I demonstrated earliers such as it performed well for test harness (training data has good learnable structure ), datasets were well preprocessed, cross validation and parameters tuning were carried out, it got really good MAE.

## Improvement

There are definitely further improvements that could be done on the algorithms I chose for this project. For example, k-fold cross validations could be done for both LightGBM and XGBoost if I figure out how to implement it with LightGBM and XGBoost. It's the same for grid search - hyperparameter tuning. If grid search could be done for LightGBM and XGBoost, that might improve algorithm performance. Since I didn't figure out how to use sklearn grid search with lightGBM and xgboost models, I manually tuned all the parameters. A grid search might do a better job than manually tuning parameters. Another thing that I can think of is to use deep learning neural nets to improve algorithm. The feature selection process is pretty manual in my solution. If feature selections can be done by deep learning neural nets, it would probably generate much better feature selections and much better results. Deep neural nets should be able to pick up much better relevant feature selections.

## References

Some ideas are inspired from those kernel discussions on Kaggle. Thanks for all their work!!

- https://www.kaggle.com/c/zillow-prize-1/kernels
- https://www.kaggle.com/philippsp/exploratory-analysis-zillow
- https://www.kaggle.com/viveksrinivasan/zillow-eda-on-missing-values-multicollinearity
- https://www.kaggle.com/aharless/xgboost-lightgbm-and-ols
- https://www.kaggle.com/guolinke/simple-lightgbm-starter-lb-0-06487

- https://machinelearningmastery.com/metrics-evaluate-machine-learning-algorithms-python/
- https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/
- https://datascience.stackexchange.com/questions/18903/lightgbm-vs-xgboost
- https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/
- https://lightgbm.readthedocs.io/en/latest/
- https://xgboost.readthedocs.io/en/latest/