Capstone Project for Udacity Machine Learning Nanodegree

Pär Steffansson

August 31, 2017

Contents

1 Project Definition

An overview of the project definition is described below. For more details see the competition Zillow Prize [1].

1.1 Project Overview

I selected the competition Zillow Prize [1] found at Kaggle [2] as the Capstone project in my *Machine Learning Engineer Nanodegree* provided by Udacity [3].

Zillow [4] is the leading real estate and rental marketplace dedicated to empowering consumers with data. They launched the Kaggle competition Zillow Prize [1] to improve their ability to predict house prices.

I selected this competition to learn from the rich source of knowledge the community around these competitions provide. Comparing and learning from public Kernels provided gives a good benchmark of cutting edge models for these kind of problems.

1.2 Problem Statement

They have been developing their own model to predict prices for years. The problem at hand is to see if the Kaggle community can improve on it creating an even better model.

Zillow provide an error from using their own model trying to predict estate prices. The problem is to predict that error. This is a regression task and a weighted solution of three models will be used. First model is a simple model where the mean of the training data is the prediction for all estates. Second and third model are two different tree based Gradient boosting models called LightGBM [7] and XGBoost [8].

1.3 Metrics

Zillow is asking to predict the *logerror* between their Zestimate model and the actual sale price, given all the features of a home. The *logerror* is defined as

$$logerror = log(Zestimate) - log(SalePrice)$$

Mean Absolute Error (MAE) is then used as the metric on the logerror individually for Mean, LightGBM [7], XGBoost [8], and Weighted model. It's simple and fast to compute and works well with selected models. The formula is

$$MAE(y_{true}, y_{pred}) = \frac{1}{n}\sum_{i=1}^{n} |y_{pred}(i) - y_{true}(i)|$$

2 Analysis

2.1 Data Exploration

This data exploration is inspired by the Simple Exploration Notebook - Zillow Prize [5] written by Kaggle Grandmaster SRK [6].

We will be working with following data files

- *properties_2016.csv* contains all the properties with their home features for 2016. Note: Some 2017 new properties don't have any data yet except for their parcelid's. Those data points should be populated when properties_2017.csv is available.
- *train_2016.csv* contains the training set with transactions from 1/1/2016 to 12/31/2016.
- *sample_submission.csv* is a sample submission file in the correct format

2.1.1 Training Set File

The training set in file *train_2016.csv* contains 90275 rows and three columns

- *parcelid* is the id of the property.
- *logerror* is the log of the error comparing the log of the actual price and the log of the predicted price.
- *transactiondate* is the date when the property was sold

Each row correspond to a property transaction and there will be more than one row in this file with the same parcelid if the property has been sold more than ones during 2016. In fact counting there are

- 90026 properties that have been sold ones
- 123 properties that have been sold twice
- and one property that was sold three times.

Looking at Figure 1 the logerror has a nice normal distribution centered around zero.

According to Zillow [1] the training data has all the transactions before October 15, 2016, plus some of the transactions after October 15, 2016. Looking at Figure 2 from January to September there are about 6000-11000 transactions per month while dropping to under 2000 in November to December.

2.1.2 Properties

Property data file *properties_2016.csv* is a lot larger with 2985217 rows and 58 columns describing home features. It seems like some of the columns contain less information in the form of NaN values. Looking at Figure 3 more than half of the data is not there and the data loss is unevenly distributed on features. About 30 percent of the features will most likely not contribute to a better model when they do not contain any information.

There are many features, so let take a look at the correlation between the logerror and these features to see which ones seems to be more likely to improve the prediction than others. Looking at Figure 4 we can see

- correlation is low in general which indicate that improving prediction will be hard
- a few features are missing correlation most likely because there are only one value

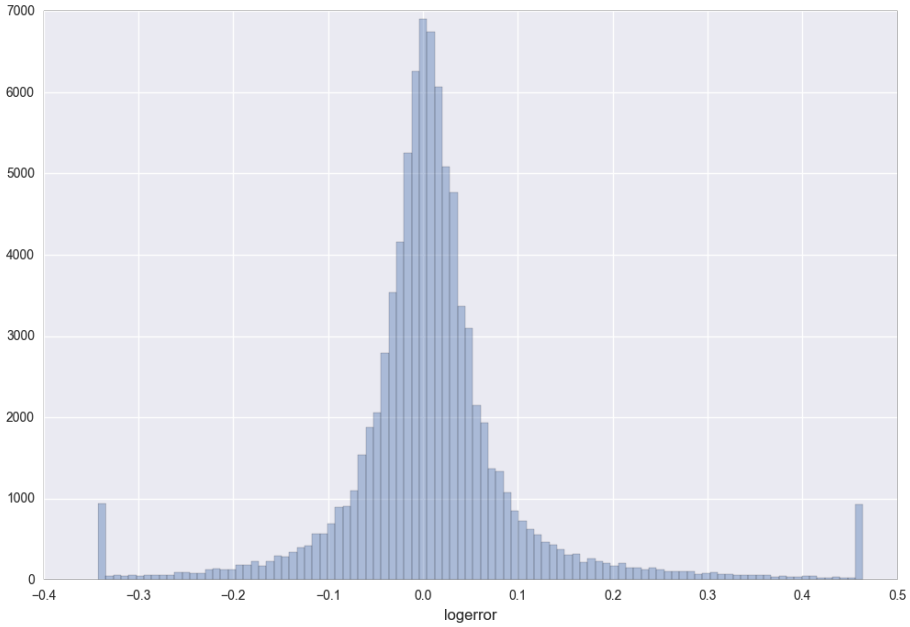

Figure 1: Logerror distribution

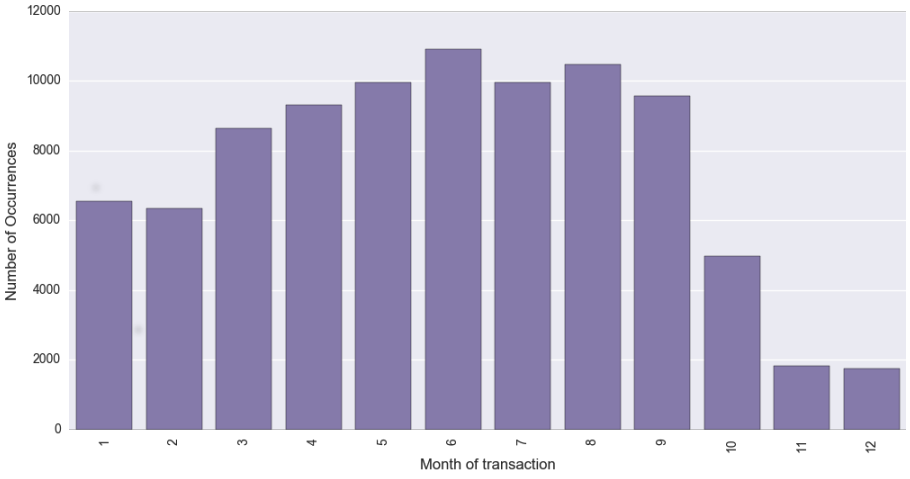

Figure 2: Number of transactions over time

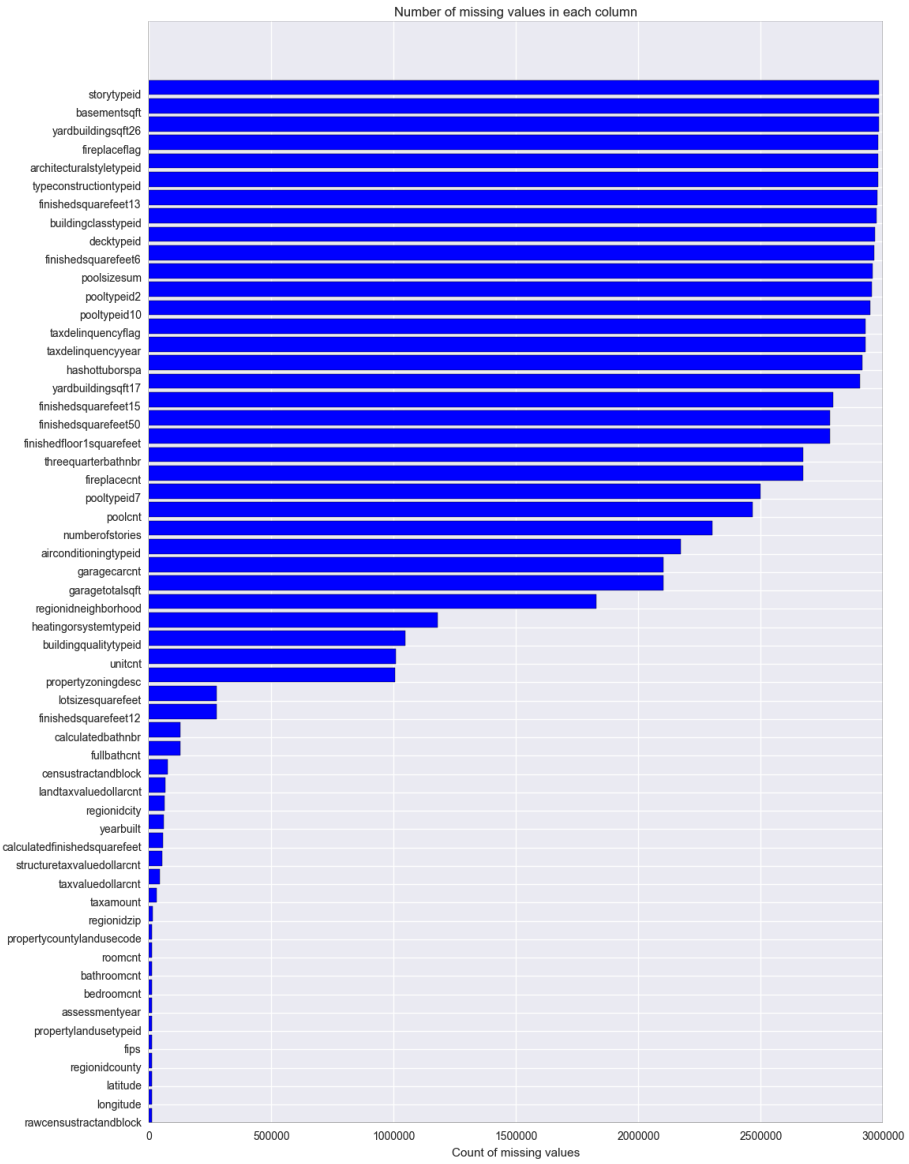

Figure 3: Amount of information in features

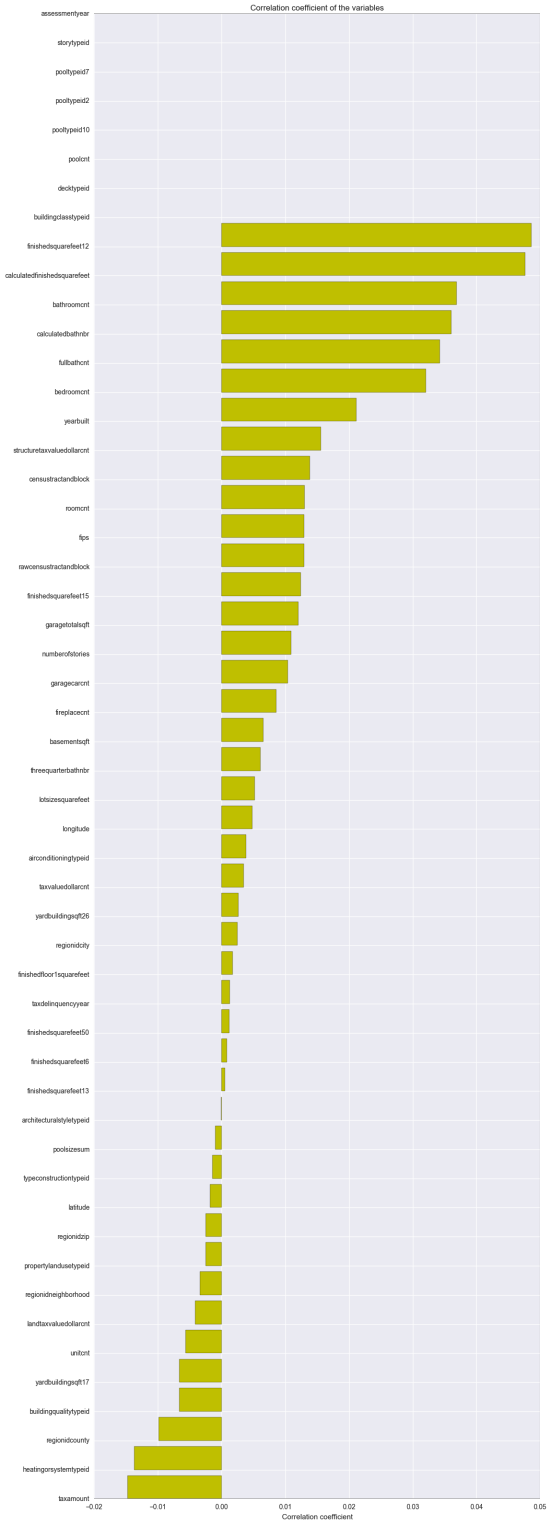

Figure 4: Correlation between features and the log error

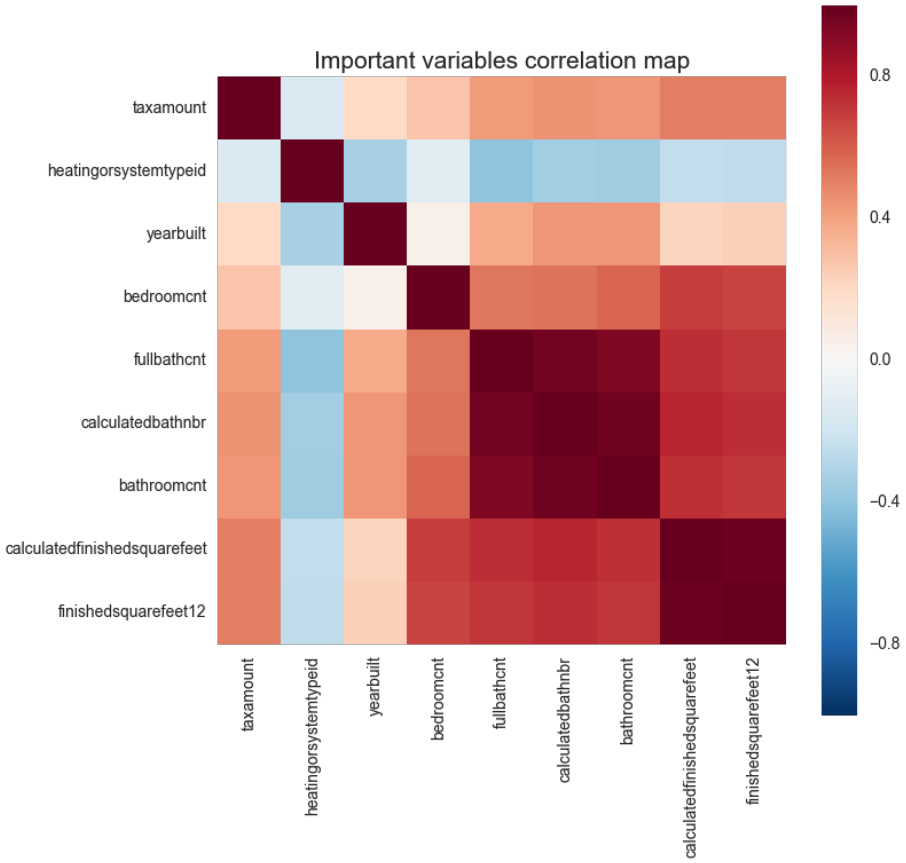

Figure 5: Correlation between features and the log error

If we look at features with a correlation less than -0.01 and more than 0.02 we may filter out nine features. High correlation between these features might indicate that they contain the same type of information which most likely makes prediction harder than if little correlation exists. Looking at Figure 5 show quite high correlation between features, especially between features relating to room counts and house size.

2.2 Algorithms and Techniques

2.2.1 Overview

I have selected the approach of weighing the result of following three different algorithm.

- *Mean* which is the simple prediction of predicting all by the mean of the training data. The logerror seems to follow the normal distribution and this is the mean that we are after and is estimated by the mean of the training data.
- *LightGBM* [7] is a gradient boosting framework that uses tree based learning algorithms. It is a high speed, high accuracy, and large scale data library.
- *XGBoost* [8] implements the gradient boosting tree based model as well. The library is optimized to be a scalable, portable and accurate library.

LightGBM [7] and XGBoost [8] are tree based Gradient boosting models. The result from these three models will then be weighted according to the best weights found using the training set. Often the result of weighing the result from different models creates a better result then any of the models by them self

2.2.2 Gradient Tree Boosting Models

Gradient tree boosting can be used for regression and classification which produce prediction by ensemble of weak decision trees.

A decision tree predicts a label by starting at the root of the tree, selecting branches at every node using the feature base statement defined at each node, ending up in a leaf which contains the prediction. A weak decision tree is a tree that have few leaves (generally 4-8 leafs).

In Gradient boosting weak decision trees are added one at a time. Once a tree is added, it will not change. A new tree is added such that the loss is reduced. This is done by parameterize the tree and changing the parameters using the gradient decent.

Gradient boosting tree has excellent accuracy which is the main reason for using it in this problem. Correlation between the logerror and features are low which indicate that a sensitive algorithm is needed to be able to predict the logerror. However, Gradient boosting trees are prone to overfit so extra caution must be taken tuning parameters.

In general, Gradient tree boosting models has proven to produce good predictions.

2.3 Benchmark

The closer to zero the *Mean Absolute Error* metric is, the better the model performance. The model result will also be compared with the *Zero model* which is the simple model of predicting the logerror to always be zero independent of feature values. The error metric of the Zero model is used as an upper level when comparing. The Zero model will product a MAE logerror value of 0.053989.

3 Methodology

3.1 Data Preprocessing

The data is prepared in following steps before fed to LightGBM [7] and XGBoost [8].

1. Remove outliers to prevent the model to reduce generalization due to data points not likely to reflect possible future data. Outliers was defined as before the first and after the 99 percentile.
2. Convert types so that the data can be used in LightGBM [7] and XGBoost [8]
3. Enriching the train data in file *train_2016.csv* with property data in file *properties_2016.csv* by joining on *parcelid*
4. Adding the month of the transaction as a feature
5. Removed column *transactiondate* since these will most likely not improve the prediction
6. Split data into train and test

3.2 Implementation

3.2.1 Libraries

Numpy is used to handle all statistical ground work, like removing outliers, converting data types, and mean computations. Scikit-learn was used to compute MAE and split data into a train and test set. Panda was used to read and write data to disk and merge data sets. LightGBM [7] and XGBoost [8] are two libraries that need to be downloaded and installed separately.

3.2.2 Overview

The implementation is divided up into two parts, training and the test part. The data is also divided into two parts. The test size of the data is 33% while the train size is 67%. A random state seed was used everywhere to enable identical runs.

The Mean, LightGBM [7], and XGBoost [8] is first trained individually. Parameters for LightGBM [7] and XGBoost [8] are tuned independent of each other. Once all three models are trained and tuned, the weights are trained by randomly select the weights over and over again until a minimum MAE is obtained on the weighted sum. To prevent overfitting the weights, a minimum weight of 0.1 is set for LightGBM [7] and XGBoost [8].

3.2.3 Steps

The overall implementation steps are

1. Preprocessing and train for LightGBM [7] using train data
2. Preprocessing and train for XGBoost [8] using train data
3. Find the best weights for the tree models by randomly trying different values on the train result from above. I selected a minimum weight of 0.1 for the LightGBM [7] and XGBoost [8] models.
4. Predict *logerror* for test data using LightGBM [7]
5. Predict *logerror* for test data using XGBoost [8]
6. Predict final *logerror* for test data weighing together LightGBM [7], XGBoost [8], and the constant model using the weights from above.
7. Compute the performance using the Mean Absolute Error metric
8. Adjust configuration parameters for LightGBM [7] and XGBoost [8] and start over at the top

3.2.4 Complications

Preparing the data so that LightGBM [7] and XGBoost [8] would not crash took a lot of time. Data type transformations were needed.

The sum of the weights for the three models need to be 1.0 while each weight is positive. At first I did not check for positive weights and this generated a less accurate Weighted model.

Before adding the lower limit on the weights of 0.1 for LightGBM [7] and XGBoost [8] the XGBoost [8] got most of the weights which generated a worse test. The lower limit contributed to allowing the Weighted model to generalize better.

3.3 Refinement

Major parameters for both models where tuned through the steps of changing one parameter at a time to different values while rerunning the prediction and see if the train and test data result improved. In case of

1. neither train result nor test result improved indicates that the change made made the model worse and the change was reverted
2. train result improved but the test result become worse indicates that the change made made the model overfit and the change was reverted
3. train result and the test result improved indicates that the change made the model generalize better and the change was kept

Following improvements were made per model.

3.3.1 LightGBM

- Changed learning rate from default value of 0.1 to 0.0021 which mitigated overfitting
- Changed number of leaves in one tree from default value of 31 to 512 which improved result

3.3.2 XGBoost

- Changed the step size shrinkage (eta) from default value of 0.3 to 0.03295 which mitigated overfitting
- Changed the maximum depth of a tree from default value of 6 to 8 which improved the model without overfitting it by increasing complexity
- Changed the subsample ratio from default value of 1 to 0.8 which mitigated overfitting

4 Results

4.1 Mean Absolute Errors

Mean Absolute Error and Weights for the **training** data was

Model	**Error**	**Weights**
Mean	0.053027	0.002130
LightGBM	0.049524	0.104294
XGBoost	0.048911	0.893574
Weighted	0.048946	

Mean Absolute Error and Weights for the **test** data was

Model	**Error**
Mean	0.053806
LightGBM	0.053157
XGBoost	0.053021
Weighted	0.052987

4.2 Model Evaluation, Validation, and Justification

The robustness is checked for the Weighted model by running it on the separate test data.

Parameters been tuned individually and care been taken to not overfit looking at the Mean Absolute Error for training as well test data.

Changing the *random state seed* will pick a different train set, pick a different test set, and train models using different entropy. As you can see below, different seeds the test data generates a predictive and good result showing the robustness of the model. Following are the seeds and errors from the different models on the test data set.

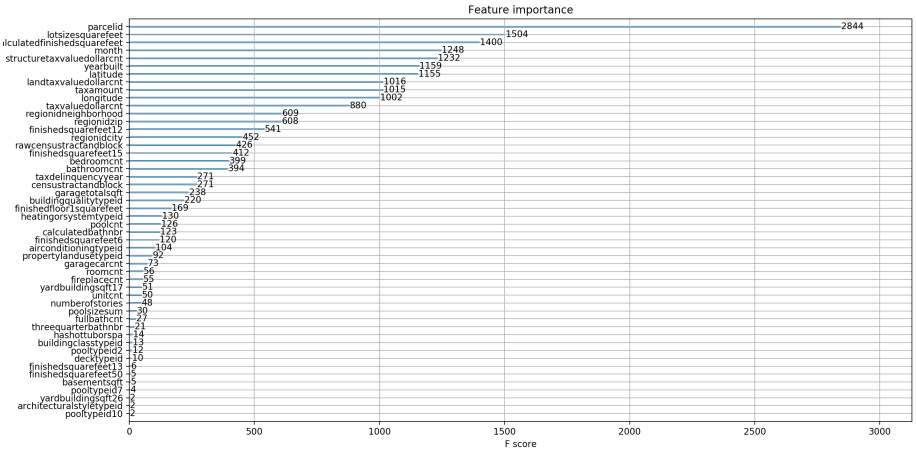

Figure 6: Feature importance according to XGBoost

Seed	Zero	Mean	LightGBM	XGBoost	Weighted	Diff
42	0.053989	0.053806	0.052987	0.053157	0.053021	0.000968
43	0.053258	0.053010	0.052441	0.052355	0.052313	0.000945
44	0.052686	0.052544	0.051925	0.051949	0.051889	0.000797
45	0.053565	0.053414	0.052846	0.052689	0.052660	0.000905

The *Diff* column is the difference between Zero model and the Weighted model and it seems to stay steady above **0.000797**. This indicates that the Weighted model can find and use some information in the data to make a better prediction than just a simple zero approach. That said, it is still a long way down to zero which means that there a lot of room for improvement.

5 Conclusion

The correlation between the logerror and different features are low (Figure 5) and even if the Weighted model is utilizing some of the little information that exists to improve the result towards zero, it is not with much. It improves the MAE result with about 1.7% compared with the Zero model.

5.1 Feature Importance

Gradient Tree Boosting Models give the opportunity to estimate how important each feature is to the result at hand. Looking at figure 6 we can see that the *parselid* sticks out which is a surprise. It seams like the *parselid* is created in a context appending additional information which seems to be important. Size of lot, finished size, and month of transaction is also important. Pool and architecture types seems not to be important.

5.2 Reflection

Zillow asked to predict the logerror which is an error produced when they try to predict prices for real estates. Predicting the logerror turned out to be hard which indicate that Zestimate algorithm used by Zillow is doing a good job. Their ability to predict house prices seams to be quite good. Even so, an improvement was made.

Another interesting observation is that the weighing the predictions the Mean, LightGBM [7], and XGBoost [8] generates a better overall MAE then for any individual model.

5.3 Improvement

It could be possible to tune the Weighted model some more but I think that it would not give much better result. To make major improvements, new information is needed.

Adding more models could improve the result, but most likely not with much. Continue tuning parameters might improve the result some more, but most likely not with much.

Adding more information is probably a better approach Using the transaction date to extract the day of the week would add information that is currently hidden. Longitude and latitude could also be used to compute distances to attractive locations like the sea or bigger cities.

References

[1] Zillow Prize
`https://www.kaggle.com/c/zillow-prize-1#description`

[2] Kaggle
`https://www.kaggle.com`

[3] Udacity
`https://www.udacity.com`

[4] Zillow
`https://www.zillow.com`

[5] Simple Exploration Notebook - Zillow Prize
`https://www.kaggle.com/sudalairajkumar/simple-exploration-notebook-zillow-prize`

[6] Kaggle Grandmaster SRK
`https://www.kaggle.com/sudalairajkumar`

[7] LightGBM
`https://github.com/Microsoft/LightGBM`

[8] XGBoost
`https://xgboost.readthedocs.io/en/latest/`