Capstone Project

Machine Learning Engineer Nanodegree

Saurabh Sharma

Dec 4, 2017

Definition

Project Overview

A home is often the largest and most expensive purchase a person makes in his or her lifetime. To market a house competitively or buy it at the right price, sellers / buyers would need to research the final sale prices of similar properties in their community. They need as much sales information as possible of the deals made in their area. An algorithm that can provide estimated value of the target property even in a reasonable range can be greatly helpful.

Since the Zestimate, Zillow's proprietary home value prediction algorithm already exists, there is a constant need to keep improving it. Hence Zillow has shared the residual errors from predictions of a version of its production deployed algorithm asking the participants to create a model that can accurately predict these errors.

The dataset has been provided by Zillow as a part of the competition here.

Problem Statement

To develop an algorithm that makes predictions about the future sale prices of homes by building a model to improve the Zestimate residual error.

The tasks involved are the following:

1. Download and preprocess the data

2. Train a regression model that can minimize the mean absolute error between the predicted and the actual log error.

3. Each Input sample represents a real estate property. A real estate property can be a residential house, commercial setting or even an empty land. Following are some examples of the features of an input sample:

- Total area (finished or unfinished) of the property
- Total assessed value of the property
- Neighborhood, region, city, zip, state of the property
- Property Zoning code assigned by a municipal authority
- Kind of material used to build the house if applicable
- Total number of bathrooms, bedrooms, pools, patios, spas, garages, floors, fireplaces etc. on the property if applicable

- And so on.

4. The output or target variable is an error value called logerror that represents the difference between the log (Zestimate) and log (Actual Sales Price). Zestimate is the Zillow's prediction of the monetary value of that property while actual sales price is the price at which that property actually got sold.

Metrics

The objective is to minimize the **mean absolute error** between the predicted and actual log error (target variable). The definition of log error has been provided by the Zillow in the details of the competition itself. It's defined as :

$$\text{Logerror} = \log(\text{Zestimate}) - \log(\text{SalePrice})$$

Where,
Zestimate is the predicted value of a home as generated by Zillow's proprietary algorithm and Sale price is the actual sale price of the house.

This was the metric defined by Zillow for this round of the competition and hence I too decided to choose the same for this project. There is a rationale behind why this metric was chosen as learnt from Zillow's competition FAQ and my own research. By improving the Zestimate residual error and thus by being able to predict Zillow's home value prediction algorithm as closely as possible, the predictions of the algorithm itself can be improved. Also this can help find signals from the data which the original algorithm might have missed and thus help improve it.

Please note that however since the contributions from this project wouldn't be submitted to the actual Zillow's competition, the improvement in the actual algorithm as mentioned above couldn't be demonstrated as a part of this project.

The plan is also to use R^2 coefficient of determination to determine the goodness of fit of the model. The coefficient of determination for a model is a useful statistic in regression analysis, as it often describes how "good" that model is at making predictions. The values for R^2 range from 0 to 1. A R^2 of 1 indicates that the regression line perfectly fits the data while 0 means it's no better than a random guess.

Since the Zestimate is already expected to be very close to Sale Price, given the fact that it has been developed by world class scientists, even an improvement over the naive model at the 4th decimal place should be considered reasonable.

Analysis

Data Exploration

The competition organizers have provided a full list of real estate properties in three counties (Los Angeles, Orange and Ventura, California) data in 2017. The dataset has details of around 172,000 real estate properties described using 58 features. The file below provides the details of all the features included.

zillow_data_dictionary.xlsx

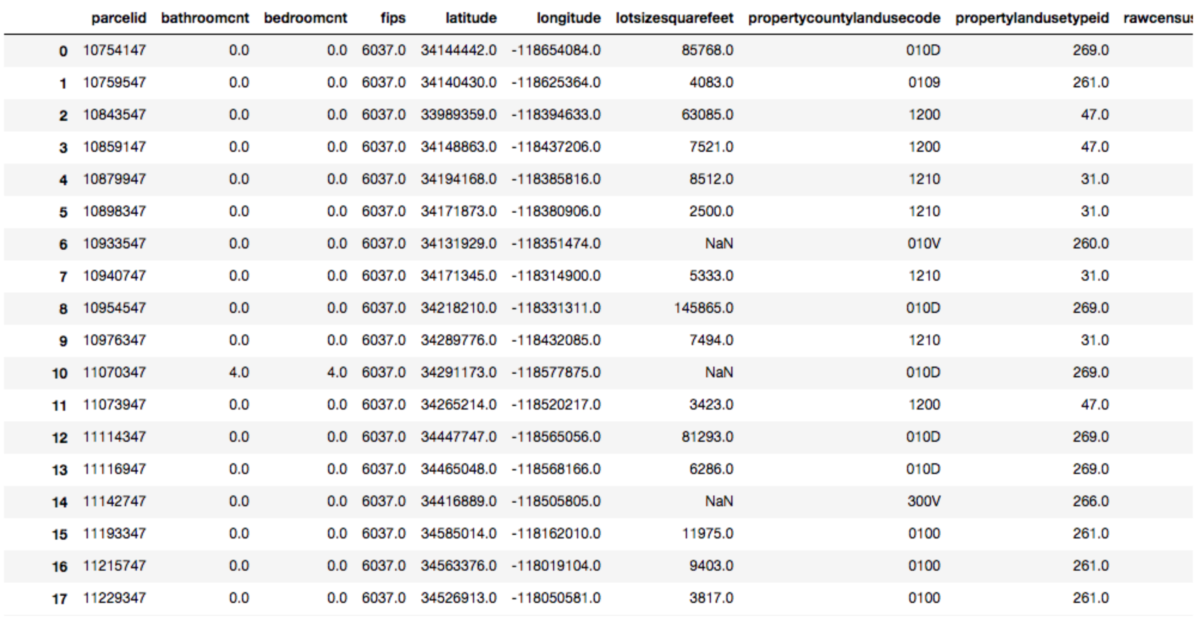
Here are the details of data type as well as the descriptive stats of the features.

The file with the names properties_2016.csv and properties_2017.csv had the data on real estate properties. There are two such files - one for the year 2016, and the other for the year 2017. Both have the information on exactly the same properties except 2017 had some new information added to it. Similarly the file with the names train_2016_v2.zip and train_2017.csv.zip had the data on the log error for the years 2016 and 2017 along with the corresponding transaction dates - the dates on which the properties were sold. A few of the properties were sold more than once in this period of two years. The total number of properties in the properties.. files were around 3 million. But only a little more than 170,000 of these were sold. The participants were expected to train their models on these 170k sold properties and submit their predictions for the rest of the them. These submitted predictions were then compared against the actual log error which Zillow has held back.

Since I was could not be actively involved in the competition, I just took the sold properties data while discarded the data of the rest of the unsold properties. That is, I took the data on the 172k sold properties and decided to train as well as test my model on that only.

Exploratory Visualization

Below is a sample snapshot of how data looks like :



	parcelid	bathroomcnt	bedroomcnt	fips	latitude	longitude	lotsizesquarefeet	propertycountylandusecode	propertylandusetypeid	rawcensu
0	10754147	0.0	0.0	6037.0	34144442.0	-118654084.0	85768.0	010D	269.0	
1	10759547	0.0	0.0	6037.0	34140430.0	-118625364.0	4083.0	0109	261.0	
2	10843547	0.0	0.0	6037.0	33989359.0	-118394633.0	63085.0	1200	47.0	
3	10859147	0.0	0.0	6037.0	34148863.0	-118437206.0	7521.0	1200	47.0	
4	10879947	0.0	0.0	6037.0	34194168.0	-118385816.0	8512.0	1210	31.0	
5	10898347	0.0	0.0	6037.0	34171873.0	-118380906.0	2500.0	1210	31.0	
6	10933547	0.0	0.0	6037.0	34131929.0	-118351474.0	NaN	010V	260.0	
7	10940747	0.0	0.0	6037.0	34171345.0	-118314900.0	5333.0	1210	31.0	
8	10954547	0.0	0.0	6037.0	34218210.0	-118331311.0	145865.0	010D	269.0	
9	10976347	0.0	0.0	6037.0	34289776.0	-118432085.0	7494.0	1210	31.0	
10	11070347	4.0	4.0	6037.0	34291173.0	-118577875.0	NaN	010D	269.0	
11	11073947	0.0	0.0	6037.0	34265214.0	-118520217.0	3423.0	1200	47.0	
12	11114347	0.0	0.0	6037.0	34447747.0	-118565056.0	81293.0	010D	269.0	
13	11116947	0.0	0.0	6037.0	34465048.0	-118568166.0	6286.0	010D	269.0	
14	11142747	0.0	0.0	6037.0	34416889.0	-118505805.0	NaN	300V	266.0	
15	11193347	0.0	0.0	6037.0	34585014.0	-118162010.0	11975.0	0100	261.0	
16	11215747	0.0	0.0	6037.0	34563376.0	-118019104.0	9403.0	0100	261.0	
17	11229347	0.0	0.0	6037.0	34526913.0	-118050581.0	3817.0	0100	261.0	

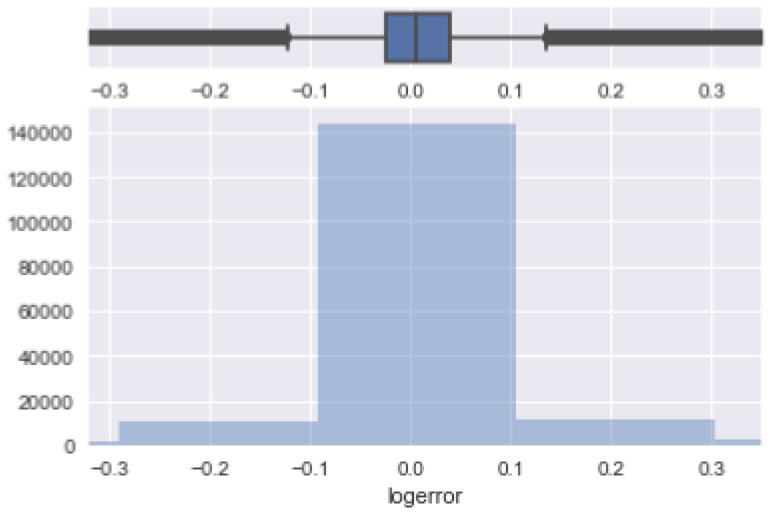
***Since the whole data frame is not visible here, a sample file is being attached below.*

sample.csv

The visualization shows the correlation of the numerical features in the raw data set.

It can be seen that a lot of features are strongly correlated. For e.g. calculatedfinishedsquarefeet is strongly positively correlated to finishedsquarefeet6. This is expected as the former feature is "Calculated total finished living area of the home" while the latter is "Base unfinished and finished area". Similarly, taxvaluedollarcnt which is "the total tax assessed value of the parcel" is strongly positively correlated to landtaxvaluedollarcount which is "the tax assessed net value of the land area of the parcel".

Below visualization shows the distribution of the dependent variable (target variable) :



Algorithms and Techniques

As the outputs are continuous values, so this is a regression problem.

Since the data set has 58 features with a domain of real estate market, it is pretty likely that the data set is going to have a lot of randomness in it.

The reason is real estate market is very heteroscedastic and the fact that a model to be created would be about residuals of an already existing algorithm developed by a team of experts at Zillow, it's going to be very likely that the signals they have missed to capture from the input data set wouldn't have an obvious pattern.

Hence it made sense to go with an Ensemble method which gives more leeway to fit a nonlinear model. By utilising the power of multiple base learners a stronger model than the one generated using a linear regression model can be generated.

More specifically, Xtreme Gradient Boosting or XGBoost method was used to find the right model fit. As per this page:

XGBoost is a software library that one can download and install on their machine, then access from a variety of interfaces. The XGBoost library implements the gradient boosting decision tree algorithm. It is an implementation of gradient boosting machines.This algorithm goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines.Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.This approach supports both regression and classification predictive modeling problems. XGBoost library is laser focused on computational speed and model performance. The library provides a system for use in a range of computing environments, not least:

> *Parallelization of tree construction using all of your CPU cores during training.*
>
> *Distributed Computing for training very large models using a cluster of machines.*
>
> *Out-of-Core Computing for very large datasets that don't fit into memory.*
>
> *Cache Optimization of data structures and algorithm to make best use of hardware.*

Some key algorithm implementation features include:

> *Sparse Aware implementation with automatic handling of missing data values.*
>
> *Block Structure to support the parallelization of tree construction.*
>
> *Continued Training so that you can further boost an already fitted model on new data.*

For me, there was more than one reason to use this algorithm. One, it gave the freedom to use multiple differentiable functions as the objective function, as long as gradient descent is successfully implemented along with it. Second, the underlying concept of XGBoost is that every base learner works on to improve the previous learner's prediction by improving latter's residual. Third and the most important reason to use this method was its computational efficiency. Its parallel architecture made it way faster than the simple Gradient Boosting algorithm which meant a lot higher number of base learners (and thus increase the possibility of getting an improved model) than those created by simple Gradient boost could be tried. To put things in perspective, the time taken to train 1000 estimators on Gradient boosting regressor was almost same as the that taken by 8000 estimators on XGBoost.

Benchmark

The benchmark is a naive model that always predicts the mean of the residual error learned from the training data. To be more specific, the mean of the log error in the training data set (around 75% of the entire data) was found to be around 0.0142. The mean absolute error calculated based on this value for the test set (the rest of the 25% of the entire data) was calculated to be : 0.07114106

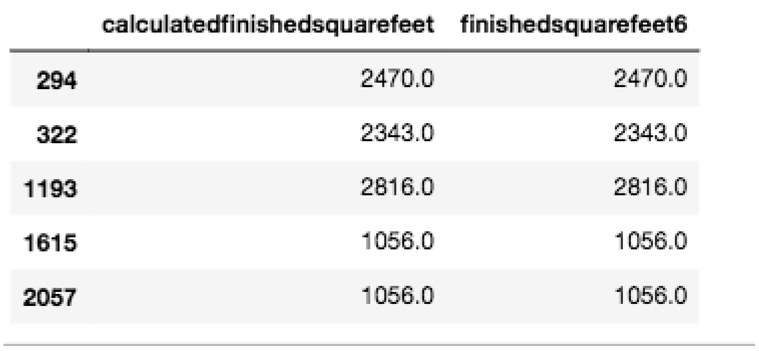
I haven’t tried to create/find a benchmark for time taken for processing and modelling.

Methodology

Data Preprocessing

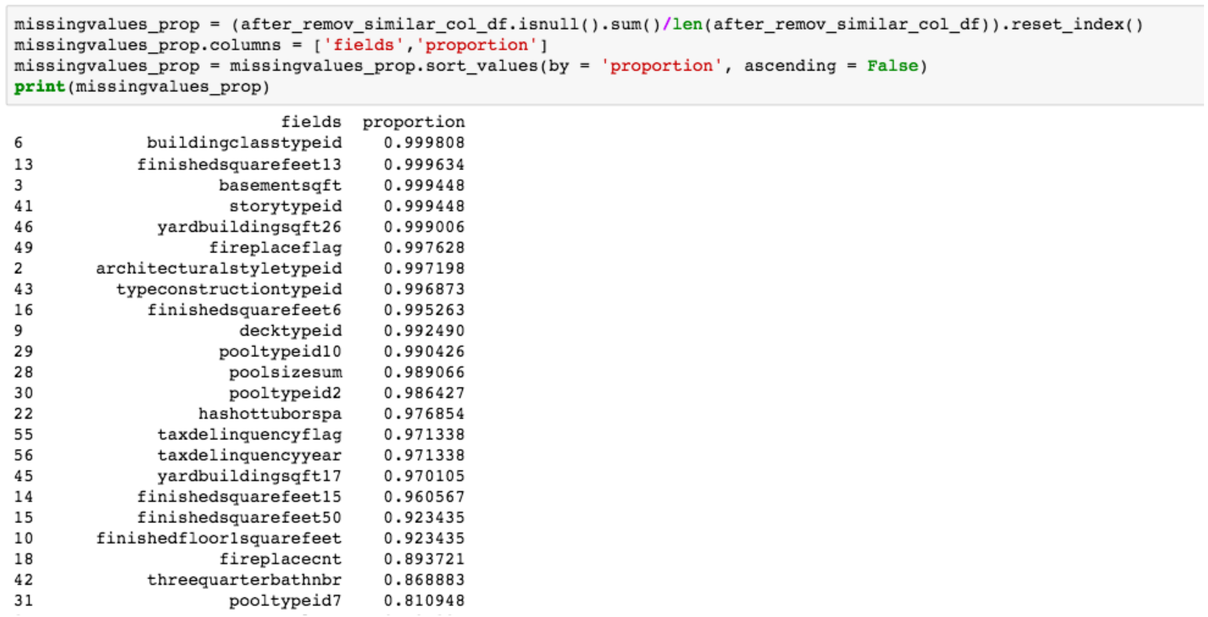
This was perhaps the single most important step in the entire project, consuming the lion’s share of the time. It started with getting a feel of the data by exploring the properties (as in real estate) and transactions data provided by Zillow. The interrelationship of the features were sussed manually by looking at their values and the subtle information they held.

It was learnt from the Kaggle forums that Zillow has provided both the raw features as well as the their own reengineered features. This resulted in having some redundant features. Hence most of these redundant features were removed. To illustrate this, take the example of calculatedfinishedsquarefeet and finishedsquarefeet6. After I read their definitions from here , I had a hunch the two might be similar and I confirmed it by checking their values as shown below:



	calculatedfinishedsquarefeet	finishedsquarefeet6
294	2470.0	2470.0
322	2343.0	2343.0
1193	2816.0	2816.0
1615	1056.0	1056.0
2057	1056.0	1056.0

The other main issue was the excessive proportion of null values for over a dozen features. I came to know about them by running the following code:



```
missingvalues_prop = (after_remov_similar_col_df.isnull().sum()/len(after_remov_similar_col_df)).reset_index()
missingvalues_prop.columns = ['fields','proportion']
missingvalues_prop = missingvalues_prop.sort_values(by = 'proportion', ascending = False)
print(missingvalues_prop)
```

```
                       fields  proportion
6           buildingclasstypeid    0.999808
13          finishedsquarefeet13    0.999634
3                  basementsqft    0.999448
41                  storytypeid    0.999448
46           yardbuildingsqft26    0.999006
49                fireplaceflag    0.997628
2       architecturalstyletypeid    0.997198
43        typeconstructiontypeid    0.996873
16          finishedsquarefeet6    0.995263
9                   decktypeid    0.992490
29                pooltypeid10    0.990426
28                  poolsizesum    0.989066
30                 pooltypeid2    0.986427
22               hashottuborspa    0.976854
55           taxdelinquencyflag    0.971338
56           taxdelinquencyyear    0.971338
45           yardbuildingsqft17    0.970105
14          finishedsquarefeet15    0.960567
15          finishedsquarefeet50    0.923435
10       finishedfloor1squarefeet    0.923435
18                fireplacecnt    0.893721
42          threequarterbathnbr    0.868883
31                 pooltypeid7    0.810948
```

As can be seen, around 18 features have more than 97% of their values as null. Most of these features had to be unfortunately removed too.

First major step was to combine the real estate properties data with their corresponding transactions data. This was achieved through combining the two files using their parcelid (the unique id associated with a parcel or lot or property).

In the beginning itself, I was able to see a potential for feature transformation by breaking the TransactionDate feature into three separate columns, one each for its Year, Month and Day. I did this since I thought I might be able to derive any seasonality signals if present in the sales of properties using this approach.

As far as the outliers go, I tried to remove as many obvious outliers as possible (like a home with total finished area = 2 sq ft) but for the most part I kept the rest of the outliers since I learnt from the kaggle forums this is where the key to “why Zillow’s algorithm was failing” might lie.

The next logical step was to impute the missing values. This was done using a method based on K-NN approach by supplying the method - latitude and longitude along with the column to be imputed. I found this approach on one of the public kernels of Kaggle. The method trained a model on the non-null values of that column as target while corresponding latitude and longitude feature values as the input variable.

There were two variants of the above function - one to impute columns which could only take discrete values, another for the ones which could take continuous values too.

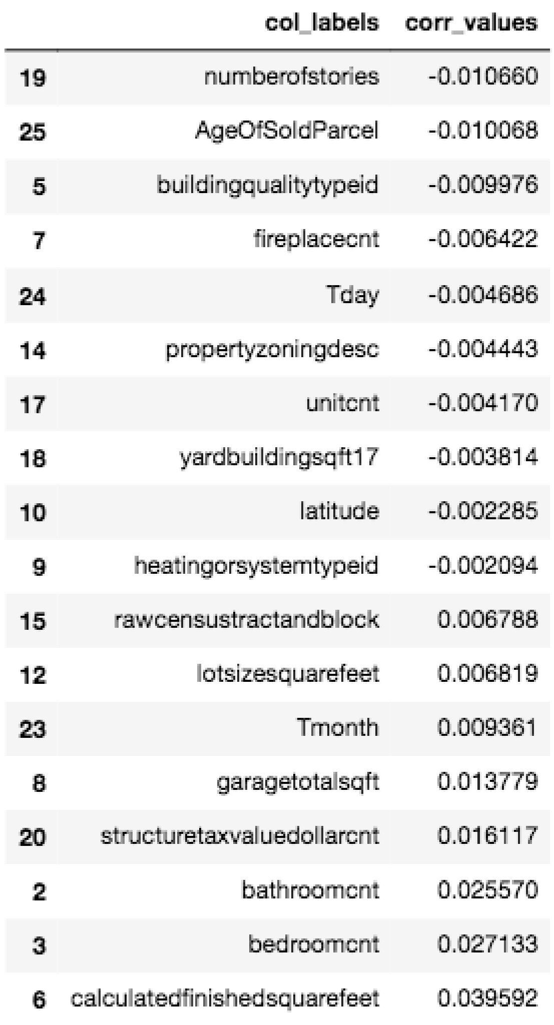
Next issue was not all features were numerical, around 8 of them are of type object. The rest were of type float or int. Hence that would mean encoding the non-numerical features. I used label encoding for this.

I then came across a unique problem where there happen to be a few features such as heatingorsystemtypeid which has a discrete set of values but they are nothing but categories. Thus the values of such a feature can be said to have non-ordinal numerical values. The only approach I could come up was to re-assign the feature ‘s numerical categories with values in order of their population. For e.g. if there is a Fruits feature with apple(id = 1) as 50% of the samples, peach(id=2) as 30% of the samples and orange(id=3) as 20% of the samples. Then I reassigned them as apple = 6, peach = 5, orange = 4.

The next step was to standardize / normalize the features to scale and for this I used RobustScaler, reason being it maintained the original distribution of values and was robust to outliers, which I have decided to keep for the most of the part.

Then using pearson’s correlation coefficients I filtered the top 18 features most correlated (positive or negative). The reason I zeroed in on this number was as I had learnt from the course that a thumb rule for how much data might be required for N features could be 2^N. Since I had a training set of around 130K samples, 18 seemed to be close enough.

Following were the features I ended up with:



	col_labels	corr_values
19	numberofstories	-0.010660
25	AgeOfSoldParcel	-0.010068
5	buildingqualitytypeid	-0.009976
7	fireplacecnt	-0.006422
24	Tday	-0.004686
14	propertyzoningdesc	-0.004443
17	unitcnt	-0.004170
18	yardbuildingsqft17	-0.003814
10	latitude	-0.002285
9	heatingorsystemtypeid	-0.002094
15	rawcensustractandblock	0.006788
12	lotsizesquarefeet	0.006819
23	Tmonth	0.009361
8	garagetotalsqft	0.013779
20	structuretaxvaluedollarcnt	0.016117
2	bathroomcnt	0.025570
3	bedroomcnt	0.027133
6	calculatedfinishedsquarefeet	0.039592

Note that AgeOfSoldParcel is an engineered feature using YearBuilt and Year column derived from the TransactionDate, which also was used to generate Tmonth and Tday features.

Implementation

Although the plan was to use the Ensemble methods, the simpler ones weren't completely discarded. The first models to be tried were Linear regression and Bayesian Ridge regression from sklearn library, followed by Decision trees. There weren't any major hyperparameters to be tuned in these simpler models. Another major advantage these models specially the linear models had over ensemble methods was that they were very fast both for training as well as testing. They were also relatively easy to be plotted for their learning curve performance. Major advantage of Decision tree regressor over linear model was that this can fit functions with a nonlinear form. However this advantage didn't get reflected in their performance. DecisionTreeRegressor has a generalization (testing set) error rate of 0.0711796959502 which is slightly higher than that of Bayesian Ridge (0.07093958932) and Linear Regression (0.070881907705640845).

Looks like the simpler the model the better the generalization has been till now.This may be attributed to the presence of correlated features in the input data.

The learning curves of Linear regression and Bayesian Ridge were almost similar. Same was the case with their predictions too. Decision tree regressor seemed to perform better on the validation error graph. However it generalized pretty badly as was obvious from its predictions vs actuals graph.

Decision tree was visualized using pydot and graphviz. It seemed to be splitting on the most important attributes and correspondingly bring down the mean value of samples at each node except for a few of them where it failed quite badly. This may have contributed to its slightly worse performance.

Finally the ensemble methods were tried. First XGBoost was tried followed by Gradient Boosting regressor, although I could have done it other way round too.

For XGBoost, the following parameters were tuned to optimize the model:

'eta': step size shrinkage used in update to prevents overfitting. Also known as learning rate.

'max_depth': maximum depth of a tree.

'subsample': subsample ratio of the training instance. For e.g., setting it to 0.5 means that XGBoost randomly collected half of the data instances to grow trees.

'colsample_bytree': subsample ratio of columns when constructing each tree.

'objective': Specifies the learning task and the corresponding learning objective.

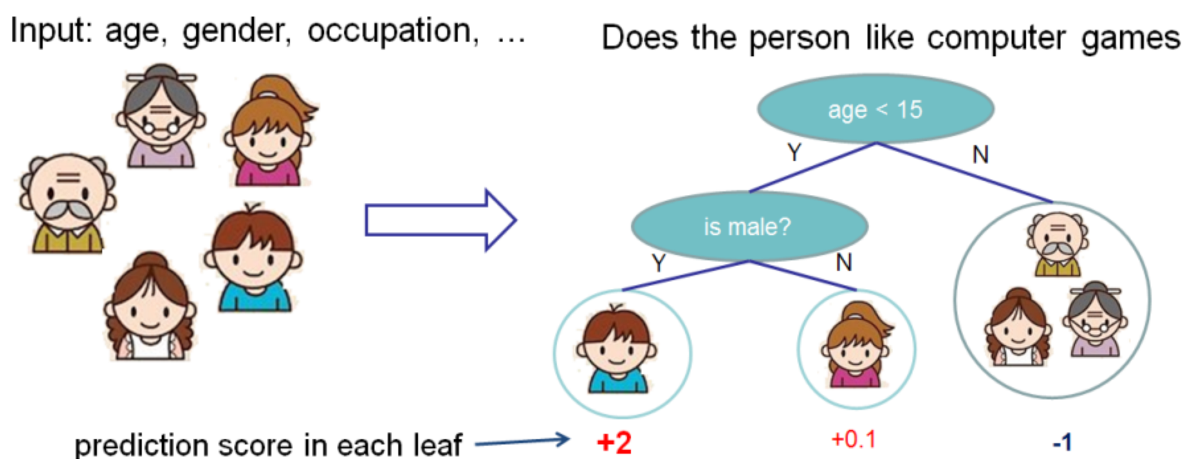
'eval_metric':evaluation metrics for validation data.

'alpha' : L1 regularization term on weights.

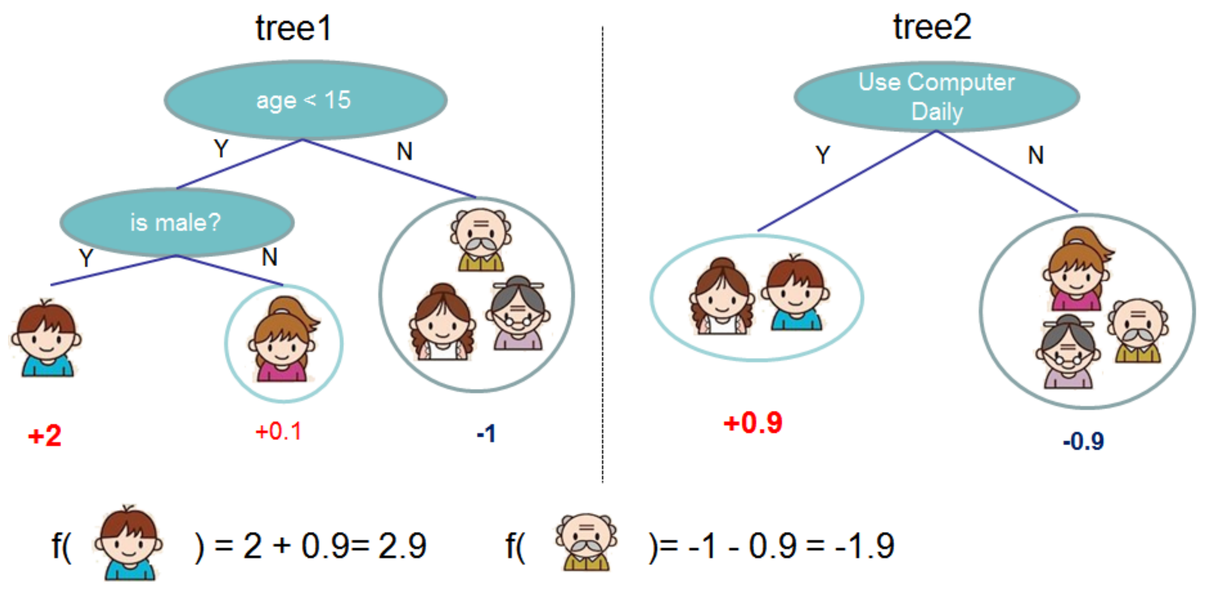
'lambda' : L2 regularization term on weights.

To further explain XGBoost, as given here:

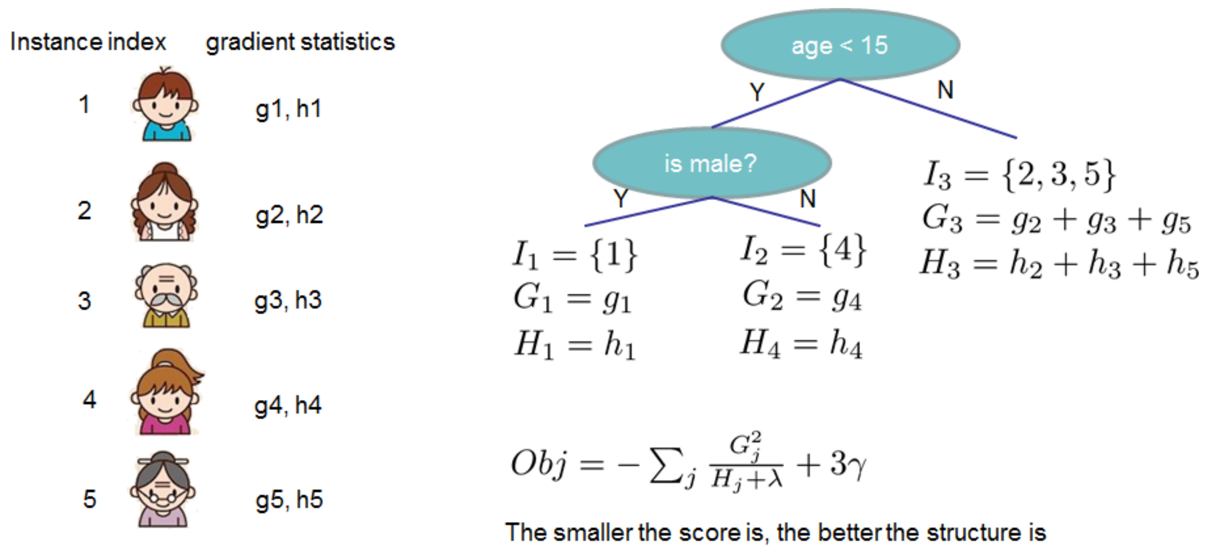
XGBoost is used for supervised learning problems, where we use the training data (with multiple features) to predict a target variable.
To elaborate more, a model in supervised learning usually refers to the mathematical structure of how to make the prediction y given x.The prediction value can have different interpretations, depending on the task, i.e., regression or classification. The parameters are the undetermined part that we need to learn from data.Based on different understandings of yi we can have different problems, such as regression, classification, ordering, etc. We need to find a way to find the best parameters given the training data. In order to do so, we need to define a so-called objective function, to measure the performance of the model given a certain set of parameters.
Now coming back to the model of xgboost: tree ensembles. The tree ensemble model is a set of classification and regression trees (CART). Here's a simple example of a CART that classifies whether someone will like computer games.


Input: age, gender, occupation, ...
Does the person like computer games
age < 15
Y
N
is male?
Y
N
prediction score in each leaf
+2
+0.1
-1

We classify the members of a family into different leaves, and assign them the score on the corresponding leaf.
Usually, a single tree is not strong enough to be used in practice. What is actually used is the so-called tree ensemble model, which sums the prediction of multiple trees together.



Here is an example of a tree ensemble of two trees. The prediction scores of each individual tree are summed up to get the final score.
After introducing the model, let us begin with the real training part. How should we learn the trees? The answer is, as is always for all supervised learning models: define an objective function, and optimize it!
Since it is not easy to train all the trees at once, we use an additive strategy: fix what we have learned, and add one new tree at a time. After we remove all the constants, we can arrive at a specific objective represented by a mathematical function.This becomes our optimization goal for the new tree.



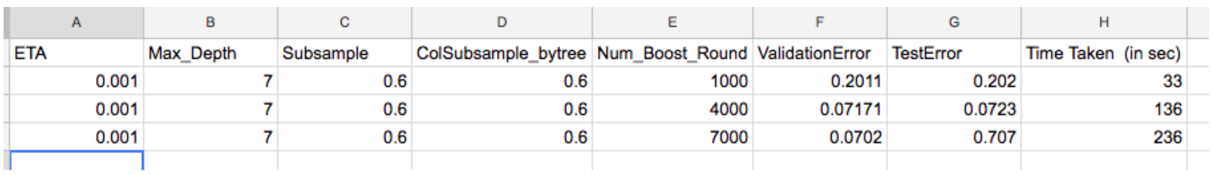
Now that we have a way to measure how good a tree is, ideally we would enumerate all possible trees and pick the best one. In practice this is intractable, so we will try to optimize one level of the tree at a time.
XGBoost is exactly the tool motivated by this principle.

Refinement

The initial performance of XGBoost was quite bad, which I later learned was because of the fact that the XGBoost's algorithm is a greedy algorithm, hence can overfit very fast. After learning this, I started to tune the hyper parameters :

- Reduced the max_depth parameter
- Increased the subsample and col_subsample size
- Increased the alpha and lambda

The following picture shows some instances of hyperparameters tuned. Note that the below tuning was performed on a smaller training set (100k compared to 130k).



A	B	C	D	E	F	G	H
ETA	Max_Depth	Subsample	ColSubsample_bytree	Num_Boost_Round	ValidationError	TestError	Time Taken (in sec)
0.001	7	0.6	0.6	1000	0.2011	0.202	33
0.001	7	0.6	0.6	4000	0.07171	0.0723	136
0.001	7	0.6	0.6	7000	0.0702	0.707	236

This helped improve the performance significantly and I was able to see a generalization error rate of 0.0708735789611. The visualization plot of predictions vs actuals too looked way better than that of Decision Tree Regressor.

The gradient tree regressor on the other hand had already performed way better than others with a generalization error rate of 0.070216390341452412. This was a huge surprise as I had not expected it to beat XGBoost.

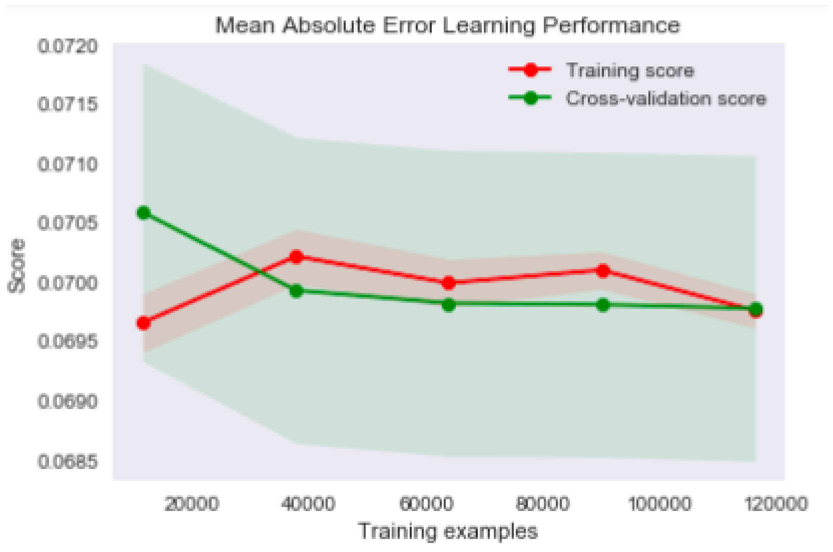
Results

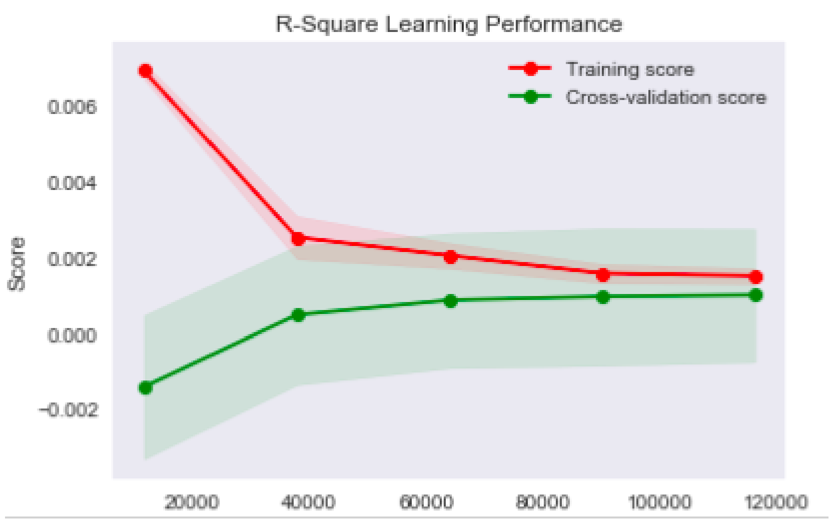
The entire data set was divided into training and testing sets based on approx. 75 - 25 basis. The cross validation was either done using learning curves or using the inbuilt functionality of the models.

A sample demonstration of calling

plot_learning_curve(lr,'Mean Absolute Error Learning Performance', X_train_1st, y_train_1st, cv=10, scoring=performance_metric)

for Linear regression as follows:





The cv parameter in the above function was used to define how many groups the training set to be divided into and the training would be performed on each of these groups minus 1 which in turn would be used for cross validation, thus resulting in ‘cv’ number of trainings. In the end the average error rate would be arrived at and treated as the one to be plotted for either training or validation curve.

There was no learning curve plotted for XGBoost as it had an in built parameter called evals which would take as input training and validation set, and then print the training and cross validation errors during runtime of the model as shown below:

```
[0]     train-mae:0.494982    valid-mae:0.494902
[800]   train-mae:0.237735    valid-mae:0.236994
[1600]  train-mae:0.130695    valid-mae:0.129535
[2400]  train-mae:0.089797    valid-mae:0.088246
[3200]  train-mae:0.076155    valid-mae:0.074428
[4000]  train-mae:0.07202     valid-mae:0.070274
[4800]  train-mae:0.070708    valid-mae:0.068964
```

```
[5600]  train-mae:0.070236      valid-mae:0.068512
[6400]  train-mae:0.070038      valid-mae:0.068344
[7200]  train-mae:0.069945      valid-mae:0.068279
[7999]  train-mae:0.069892      valid-mae:0.068255
```

The validation set in the above scenario was built one time out of training set using approx the 75-25 rule.

For gradient boosting regressor, the validation was performed on the same validation set as that used for XGBoost.

Justification

To make sure the results were comparable all the models were tested on the same test set.

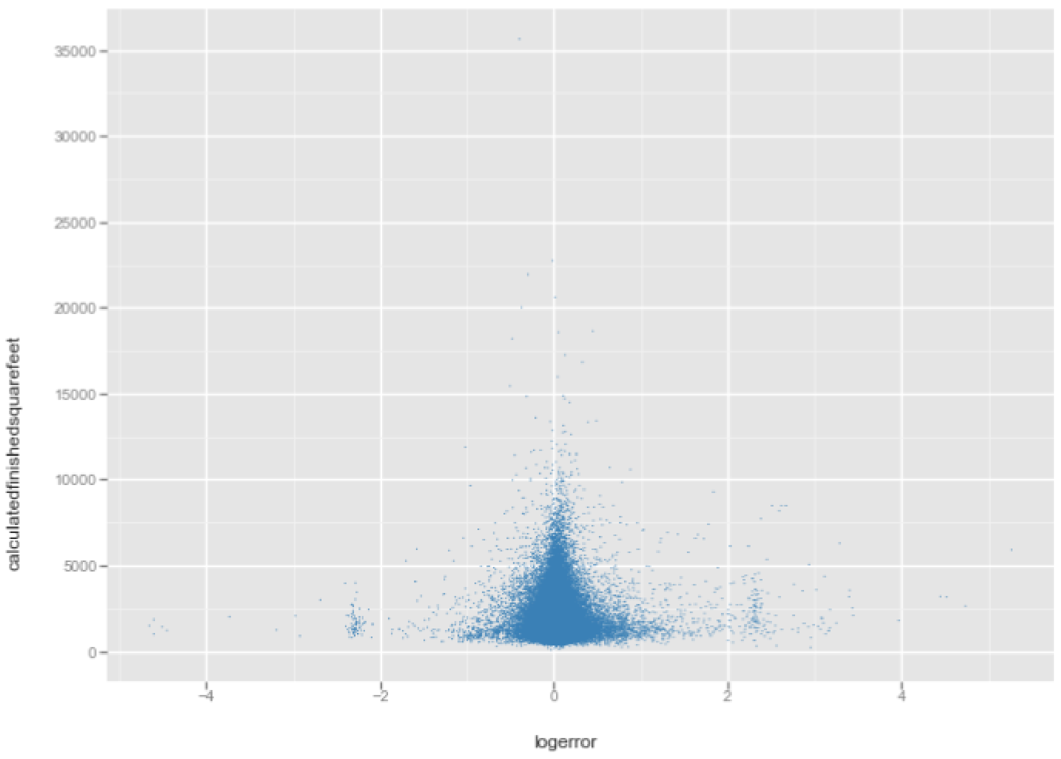
The mean absolute error calculated for the naive model on the test set was calculated to be : 0.07114106, with an R-square value of -6.865877875.

The mean absolute error calculated for XGBoost on the test set was calculated to be : 0.070873578, with an R-square value of 0.008446.

The mean absolute error calculated for Gradient Boosting Regressor on the test set was calculated to be : 0.070217176, with an R-square value of -0.00055398.

As can be seen a high R-square values do not seem to correspond to a low generalization error as it should be. So it becomes difficult to pick which one of the two - XGBoost or Gradient Boosting Regressor is better in this scenario.

Before I jump to conclusion, I understand the signals I am trying to find may well could be hidden deep in the noise of data provided, enough to beat a team of data scientists at Zillow. The log error is nicely *distributed normally* as can be seen from a bivariate analysis between calculatedfinishedsquarefeet and logerror:



Hence, the naive model used as baseline here is “not-so naive” after all as it takes the mean of the training set’s log error.

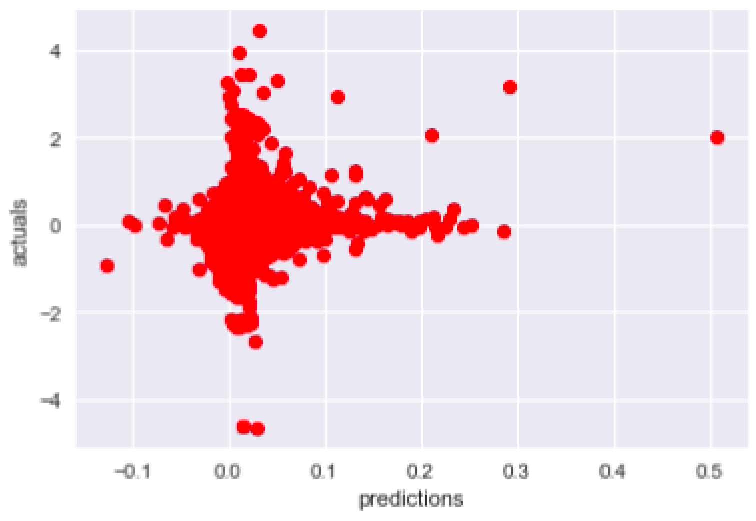
I guess a simple explanation of the results above seems to elude me at the moment.

But one thing can be said with certainty that the problem I was trying to resolve, that was to improve the residual error, is not yet resolved.

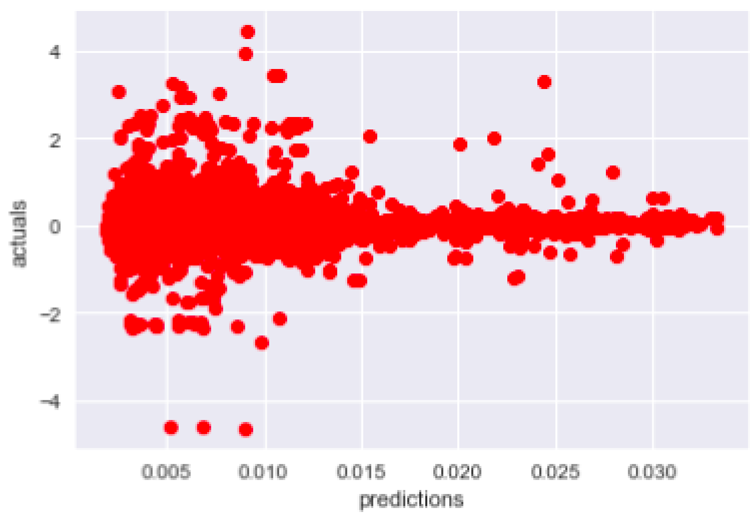
Conclusion

The visualization shows the processed data set with comparatively less correlations between its features as compared to the when I began this project.

Following are the predictions vs actuals of XGBoost:



While following is that of Gradient boosting regressor:



It is obvious that the models developed have not been able to predict the log error, although they have been able to beat the naive model selected here as shown below:

Metric	Model		
	Naive	XGBoost	Gradient Boost Regressor
Mean Absolute Error	0.07114106	0.070873578	0.070217176
R- Square	0	0.008446	-0.00055398

Reflection

The process used for this project :

1. A competition on Kaggle was found that can be relevant to my skill set
2. Dataset was downloaded from Kaggle
3. Benchmark was created using a naive model
4. Preprocessing was performed
5. Different algorithms were tried to come up with the best model
6. XGBoost and Gradient Boosting Regressor both have shown promise, although both are far from the desired target

I found the step 4 to be the most challenging, as this was my first project of this kind and on top of that I also had to familiarize myself with statistics and sklearn implementation.

Improvement

There is a lot of improvement that needs to be done. I am going through a number of options.

One is to get rid of outliers of the target variable to some degree and see if that helps improve the performance. Reason being a lot of public kernels submitted on Kaggle had outliers removed before running their models.

Secondly, try a different approach to represent the numerical categorical variables.

Thirdly, as I can still see a lot of correlation present in the features in the processed dataset I guess I should try to remove that correlation as well without removing the features.

Of course, I can also use another non-linear model such as Light Gradient Boosting Machine along with XGBoost / Gradient Boost and use the combined average.

Change the cross validation data set selection criterion to be based on some feature such as a month might also help.

These are of course just ideas as of now but I definitely will start digging the public kernels of Kaggle as well as the Discussion thread even more as there could be a gold mine of information hiding there somewhere.