Zillow Home Value Prediction

Robert Burrus

Machine Learning Engineer Capstone

Definition

Project Overview

This project is a solution to Zillow’s Home Value Prediction competition on Kaggle. The goal is to predict the *error* in Zillow’s real-estate value predictions. Zillow makes value estimations, called “Zestimate”, for more than 100 million real estate properties in the US. The median margin of error between Zestimate and the actual sales price has dropped from 14% to 5%, making Zillow one of the largest and most trusted sources for real estate information in the US.

To feed their Zestimate prediction algorithm, Zillow gathers data from a range of sources including county records, tax data, home sale listings, home rental listings, and mortgage information. Issues of data completeness, data latency, and data interpretation are critical problems in making Zestimate accurate. However, Zillow does not disclose the implementation details of their algorithm.

Problem Statement

It is important to emphasize that this is not a competition to predict home values; rather, it is a competition to predict the error in Zillow’s predictions. The task is to predict the log-error between the Zestimate and the actual sales price, given the same real-estate features as the Zestimate algorithm. Thus, this is a structured data regression problem, and the relevant algorithms will be applied.

Metrics

The Kaggle competition submissions are evaluated on the mean absolute error between the predicted log error and the log error. The log error is defined as:

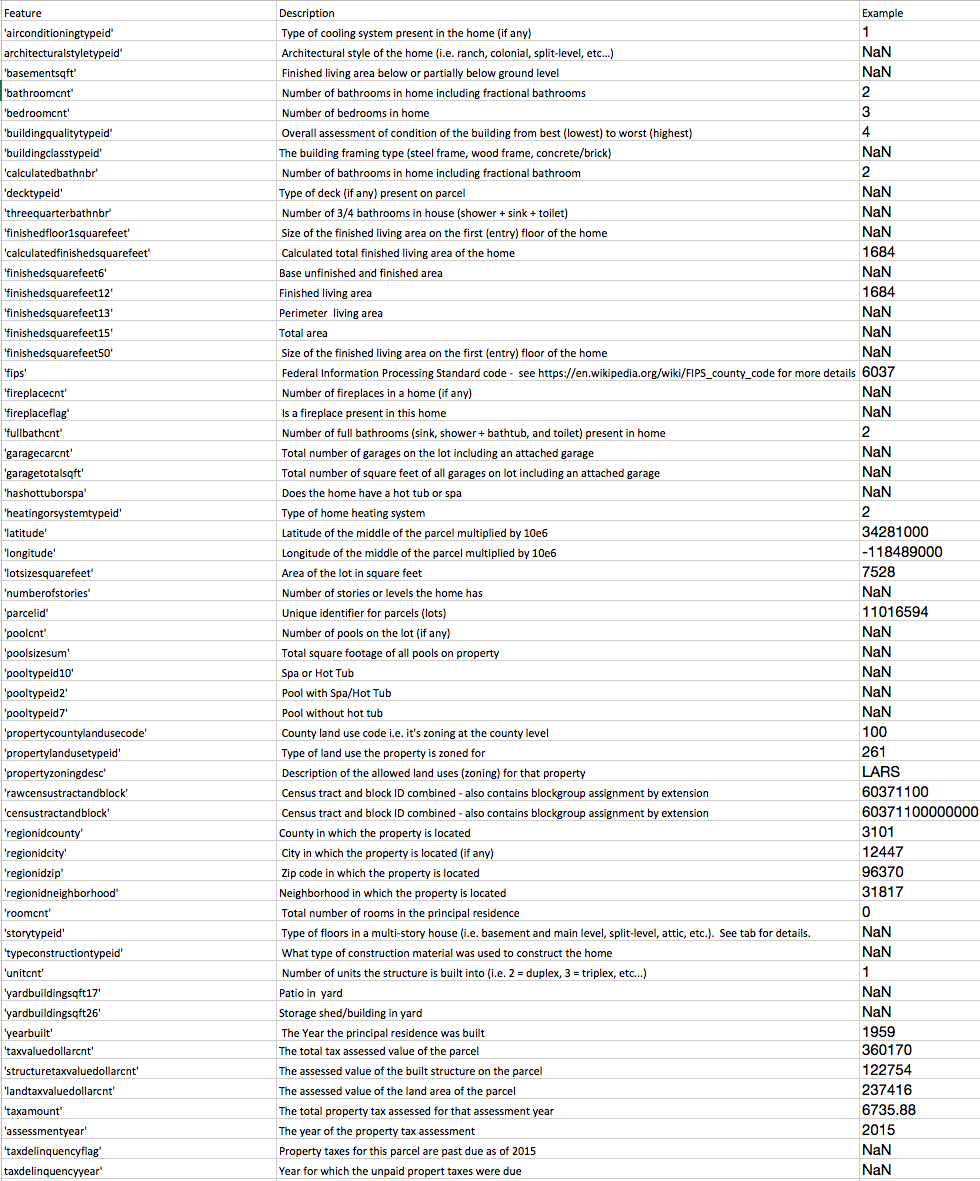
logerror = log(Zestimate) – log(SalesPrice)

Analysis

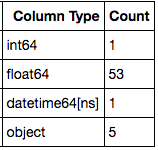
Data Exploration

Zillow provided data for 2,985,217 for real estate properties in counties surrounding Los Angeles, California. Each property has data for 2016 and 2017, bringing the total data set to over 5.9 million entries. Zillow also provides sales prices, transaction dates, and Zestimate-SalesPrice logerrors for the 167,888 properties that were sold between Jan 1, 2016 and Sep 30, 2017. I have combined these sales data with the associated property data to create the training set (68%), validation set (12%), and testing set (20%).

A sampling of the data with feature definitions is found below. Some features, especially “id” features, are not obvious from name alone. Zillow provided a feature dictionary to define the various id codes, and differentiate fields with similar names (e.g. finishedsquarefeet12 vs finishedsquarefeet13):



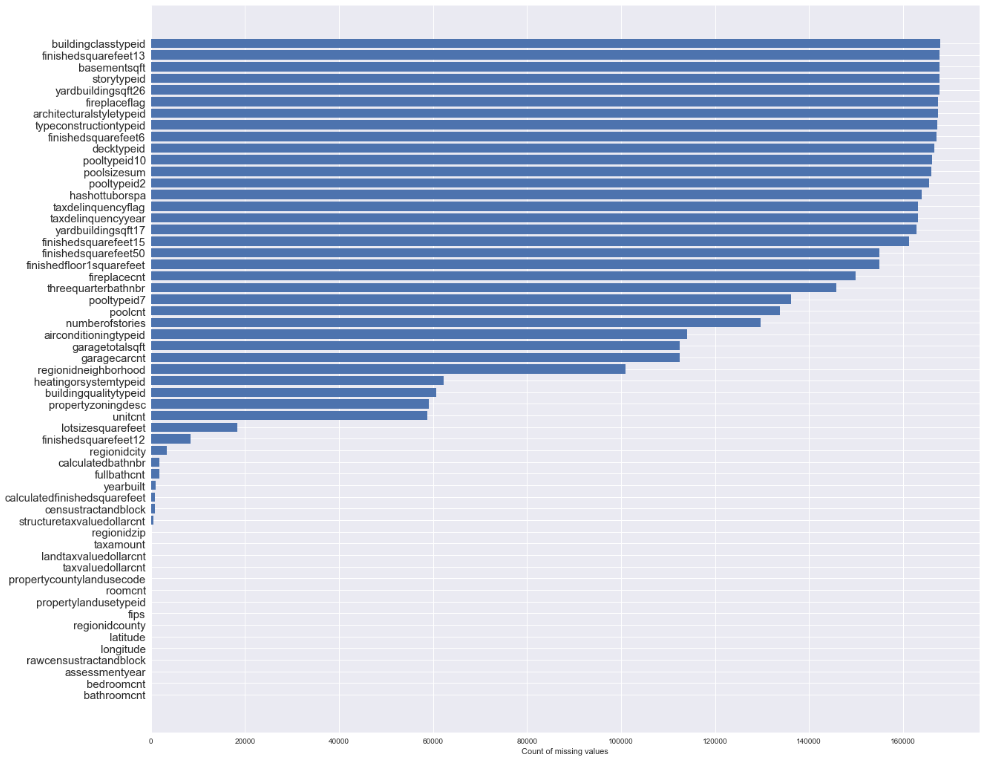
In total, there are 60 features, and the following data type distribution:



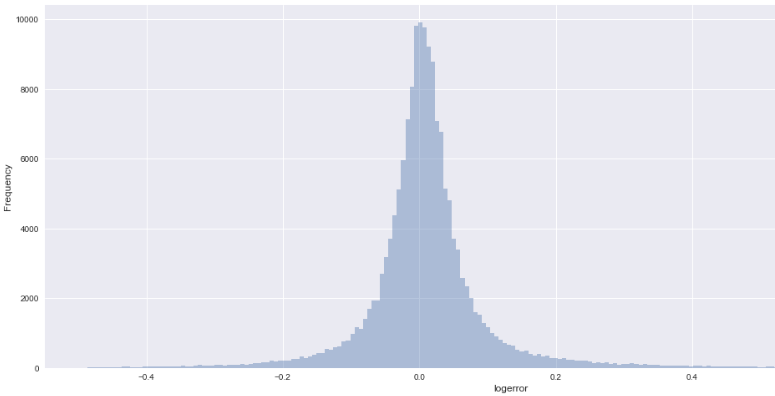
There are 19 categorical fields, 9 of which are ID features. ID data are codes that represent a description. Zillow provides the ID -> description mapping. For example:

|  |  |  |
| --- | --- | --- |
| Feature | ID | Description |
| airconditioningtypeid | 1 | Central |
| airconditioningtypeid | 4 | Geo Thermal |
| propertylandusetypeid | 261 | Single Family Residential |
| propertylandusetypeid | 264 | Townhouse |
| architecturalstyletypeid | 5 | Colonial |
| architecturalstyletypeid | 21 | Ranch/Rambler |

Dealing with missing data is one of the most important aspects of this challenge. There are 18 features that have missing data for >95% of the real estate properties. More on this later. The figure below shows the counts of missing values for each feature:



Logerrors follow a fairly normal distribution:



Algorithms

According to Anthony Goldbloom, the founder and CEO of Kaggle, there are only 2 machine learning approaches that win Kaggle competitions: ensembles of decision trees and neural networks. For structured, or tabular, data competitions, like the Zillow competition, ensembles of decision trees with handcrafted feature engineering is very likely to be the winner. Random Forest used to be the structured data algorithm of choice, but XGBoost has recently emerged and is winning practically every competition in the structured data category.

XGBoost (eXtreme Gradient Boosting) is an implementation of gradient boosting that is designed for speed and performance. In general, boosting produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. Gradient boosting builds the model in a stage-wise fashion like other boosting methods, and it generalizes them by allowing gradient descent optimization of an arbitrary differentiable loss function.

As detailed in the Methodology section, dealing with categorical fields lead me to another gradient boosting library called CatBoost. CatBoost is an algorithm developed by Yandex researchers and has native categorical feature support - allowing you to use non-numeric factors, instead of having to pre-process data or turning it into numbers. This algorithm proved particularly powerful for this dataset.

Benchmark

The repeated success of XGBoost has resulted in wide adoption of the algorithm in machine learning competitions. For this reason, I decided to use XGBoost with no data-preprocessing or algorithmic parameter tuning as the benchmark for which to compare my final solution.

MAE Benchmark: 0.07068891672999054

Methodology

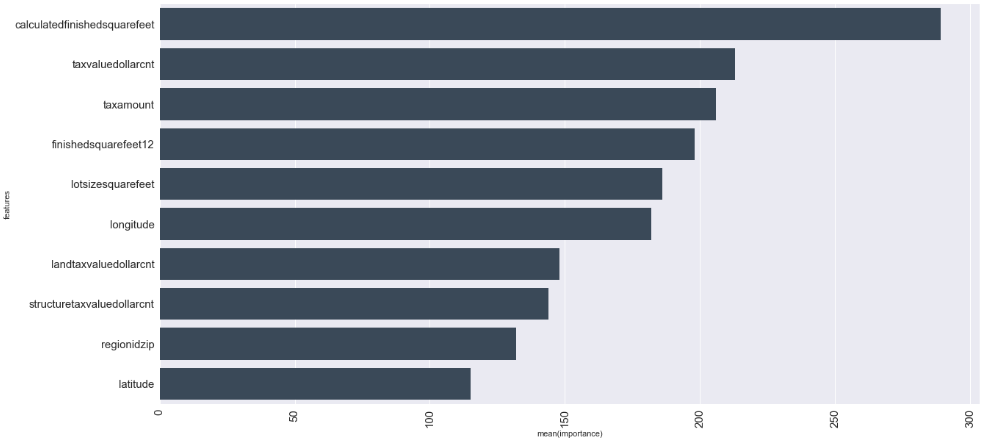
Data Preprocessing

One important aspect of this problem is that the goal is to predict error, thus it is possible that filling in missing data could be covering up an important way in which the model could learn when the Zestimate prediction error will be large. In other words, missing data is a good clue that the Zestimate will not be accurate. As the famous real-estate adage goes, “location, location, location.” If I were to fill in missing latitude / longitude or neighborhoodid, for example, then the model would not have knowledge of important missing data that plays a large role in determining price. Consequently, I have chosen not to fill in missing data with mean, median, or some other technique.

A significant jump in performance usually comes not by parameter tuning or using slightly better models, but by feature engineering. And, as is often the case with structured data problems, feature engineering is the most challenging and time-consuming aspect of the project. Most of my time has been spent on creating and analyzing new features. Feature engineering is the process of transforming raw data into features that better represent the structure to the model. This is a cyclical process that consists of data exploration, creating a new feature, and evaluating the feature importance by training a new XGBoost model. If a newly created feature improved the validation score and ranked in the top 20 of feature importance, then I kept it. I used the validation set MAE score to compare performance.

I created and tested many different features, including various ratios created from tax and square footage features, datetime features, and “binned” float categories. I will start by highlighting a few failures, and then move to the successes.

Running a feature importance analysis on the benchmark XGBoost model gave me a good idea of where to start with engineering new features. Feature importance is a metric that simply sums up the number of times each feature was split on. The top 10 features:



Using these features, I first created “binned” categorical features by grouping float64 features into bins – one bin for each quartile. For example, for calculatedfinishedsquarefeet:

|  |  |  |
| --- | --- | --- |
| Percentile | Value | Number of Examples |
| <25% | <1180 | 33,234 |
| >25 <50% | >1180 <1540 | 33,491 |
| <50 <75% | >1540 <2100 | 33,319 |
| <75 <100% | >2100 | 33,611 |
| Unknown | NaN | 697 |

Each percentile (including Unknown) was a category for the new categorical feature. But these features didn’t produce any meaningful improvement, and did not show up in the top 20 feature importance analysis. Next, I tried a categorical feature for high and low percentile ranges. Again, for example, calculatedfinishedsquarefeet:

|  |  |
| --- | --- |
| Percentile | Number of Examples |
| <0.5% | 636 |
| <1% | 672 |
| <5% | 5356 |
| <10% | 6712 |
| >90% | 6685 |
| >95% | 5371 |
| >99% | 670 |
| >99.5% | 641 |

This feature also didn’t produce meaningful results. I tried this technique for most of the features in the top 10 with no performance improvement. Other failed pre-processing techniques include removing columns with >97% of missing data, and filling in NaN values with 0, -1, and -999.

There were several modifications on existing features that resulted in slight improvements:

1. **ParcelId** -Interestingly, ParcelId, the unique identifier of the real estate property, seems to carry some signal. It isn’t clear how Zillow creates the unique id’s, though I can speculate that there is some combination of location and date/time that is resulting some predictive power.
2. **Transactiondate -** To properly use this feature, it was necessary to split the original datetime format into separate year, month, and day features. It appears that logerrors predictably vary in certain times of the year, and certain times of the month. The resulting features included:
   1. transaction\_year
   2. transaction\_month
   3. transaction\_day
3. **YearBuilt** - re-formatted “years since build”, by subtracting the year built from 2018. This reformatting consistently improved the feature importance.
4. Remove propertcountylandusecode and propertyzoningdesc, both of which were string-based categorical features that contained over 85 and 2,200 different categories, respectively.

My final dataset also included several new features created from existing features. It is important to note that no outside data sources were allowed for this Kaggle competition so augmenting data was not an option. New features included:

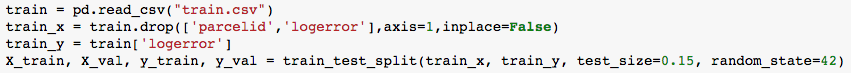
1. **Haversine distances** – determines the great-circle distance between two points on a sphere given their latitudes and longitudes. For each real estate property, I computed 3 haversine distances to downtown LA, Beverly hills, and San Clemente. These points were sufficiently spaced such that, in theory, the model could use them to accurately triangulate a location.
   1. **haversine\_la**
   2. **haversine\_beverlyhills**
   3. **haversine\_sanclemente**
2. **FinishedLivingToLotSizeRatio –** calculated total living area of the home (calculatedfinishedsquarefeet) divided by area of the lot (lotsizesquarefeet)
3. **StructureTaxToLandTaxRatio** – the assessed value of the built structure on the parcel (structuretaxvaluedollarcnt) divided by the assessed value of the land area of the parcel (landtaxvaluedollarcnt)
4. **TaxRatio** – total property tax assessed for that assessment year (taxamount) divided by total tax assessed value of the parcel (taxvaluedollarcnt)

When working with XGBoost, categorical features were required to be one-hot-encoded. I did this with the get\_dummies() method from the pandas library.

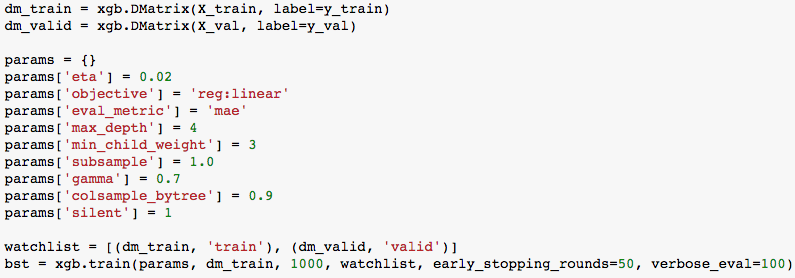
Implementation

Using the XGBoost library is a rather straight forward process. The following pipeline was used for all feature engineering testing.

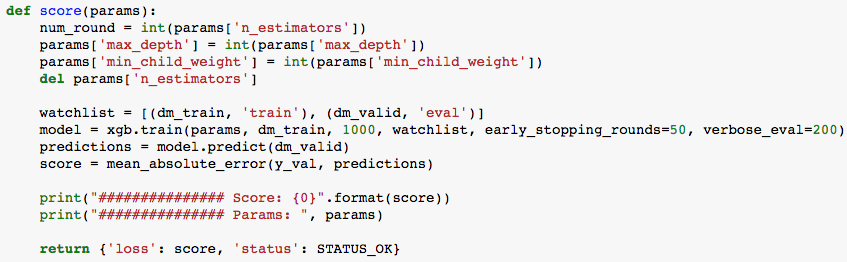
After data preprocessing (creating new feature), I load the training data, divide it into features and labels, then use train\_test\_split to split into training and validation sets.

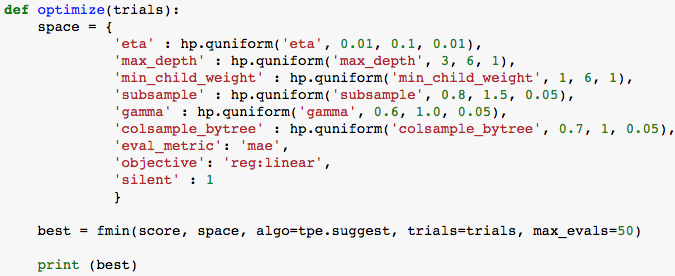


Next, using the XGBoost library, load the data into DMatrix objects – this is a requirement of XGBoost. Model parameters are set using a python dictionary. XGBoost uses the validation set during training to implement early stopping to find the optimal number of boosting rounds.



After all feature optimizations, I used the hyperopt library to fine-tune XGBoost model parameters. Hyperopt is a python library for optimizing over awkward ssearch spaces with real-valued, discrete, and conditional dimensions. The fmin function takes a dictionary defining the search space for each parameter and a function that trains the model with the given parameter configuration.





Refinement

After dozens of feature engineering cycles, my best performing XGBoost model scored MAE = 0.06915645 on the test set – an improvement of 2.1% over the benchmark MAE. This model was trained on features detailed in the Data Preprocessing section, and final model parameters were tuned with hyperopt.

During this trial and error process with XGBoost, I discovered another gradient boosting library, CatBoost, that natively handles categorical features. XGBoost required categorical features be one-hot-encoded, but CatBoost handles categorical features internally. All that is required is passing a list of indices that correspond to the categorical features, and to fill in NaNs with some default value.

My best performing CatBoost model scored MAE = 0.0677197127 on the test set – an improvement of 4.2% over the benchmark MAE.

Results

Model Evaluation

Justification

Conclusion

Visualization

Reflection

The story of this project was less about finding the right algorithm with the right parameters and more about creating good features.

Improvement