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 "Zestimates", 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. Zillow does not disclose the implementation details of their algorithm, but they do offer many insights into Zestimate evaluation and accuracy, and occasionally discuss some of Zestimate's sub-models. Zillow has been researching this modeling challenge for years, and they discuss many insights on their blog. For example, this post discusses the data quality challenges with public records. US housing prices have been steady over much of the last 40 years, but there have been occasional declines. A lot of research is dedicated to modeling these changes in the housing market over time, as seen here, here, and here.

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 gradient boosting libraries, such as XGBoost, often yield good results. As is the case with most structured data problems, the challenge lies in feature engineering and shaping the data so the model can find the most signal. Most gradient boosting libraries will perform similarly on most datasets, so the key is to create new features that yield large performance gains.

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:

$$\text{logerror} = \log(\text{Zestimate}) - \log(\text{SalesPrice})$$

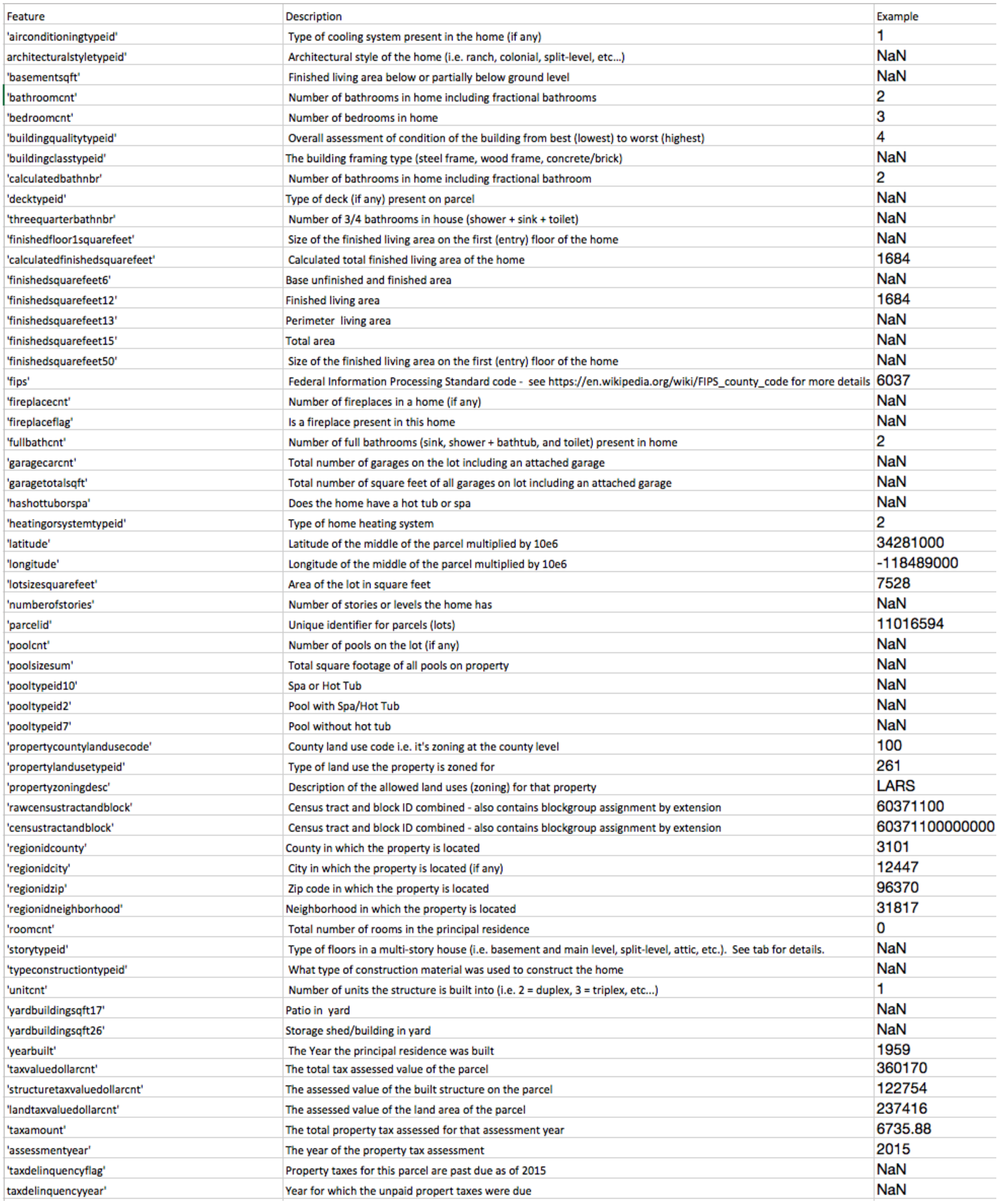
MAE has the benefit of being easily interpreted, which is necessary for Zillow to understand results. MAE also penalizes large and small deviations from the mean error with equal weight, unlike other metrics (RMSE).

Analysis

Data Exploration

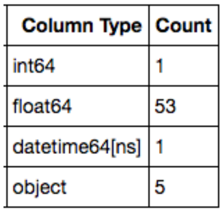
Zillow provided data for 2,985,217 for real estate properties in counties surrounding Los Angeles, California. Each property has data for 2016 and 2017, bringing the total data set to over 5.9 million entries. Zillow also provides sales prices, transaction dates, and Zestimate-SalesPrice logerrors for 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 to differentiate fields with similar names (e.g. finishedsquarefeet12 vs finishedsquarefeet13):



Feature	Description	Example
'airconditioningtypeid'	Type of cooling system present in the home (if any)	1
architecturalstyletypeid'	Architectural style of the home (i.e. ranch, colonial, split-level, etc…)	NaN
'basementsqft'	Finished living area below or partially below ground level	NaN
'bathroomcnt'	Number of bathrooms in home including fractional bathrooms	2
'bedroomcnt'	Number of bedrooms in home	3
'buildingqualitytypeid'	Overall assessment of condition of the building from best (lowest) to worst (highest)	4
'buildingclasstypeid'	The building framing type (steel frame, wood frame, concrete/brick)	NaN
'calculatedbathnbr'	Number of bathrooms in home including fractional bathroom	2
'decktypeid'	Type of deck (if any) present on parcel	NaN
'threequarterbathnbr'	Number of 3/4 bathrooms in house (shower + sink + toilet)	NaN
'finishedfloor1squarefeet'	Size of the finished living area on the first (entry) floor of the home	NaN
'calculatedfinishedsquarefeet'	Calculated total finished living area of the home	1684
'finishedsquarefeet6'	Base unfinished and finished area	NaN
'finishedsquarefeet12'	Finished living area	1684
'finishedsquarefeet13'	Perimeter living area	NaN
'finishedsquarefeet15'	Total area	NaN
'finishedsquarefeet50'	Size of the finished living area on the first (entry) floor of the home	NaN
'fips'	Federal Information Processing Standard code - see https://en.wikipedia.org/wiki/FIPS_county_code for more details	6037
'fireplacecnt'	Number of fireplaces in a home (if any)	NaN
'fireplaceflag'	Is a fireplace present in this home	NaN
'fullbathcnt'	Number of full bathrooms (sink, shower + bathtub, and toilet) present in home	2
'garagecarcnt'	Total number of garages on the lot including an attached garage	NaN
'garagetotalsqft'	Total number of square feet of all garages on lot including an attached garage	NaN
'hashottuborspa'	Does the home have a hot tub or spa	NaN
'heatingorsystemtypeid'	Type of home heating system	2
'latitude'	Latitude of the middle of the parcel multiplied by 10e6	34281000
'longitude'	Longitude of the middle of the parcel multiplied by 10e6	-118489000
'lotsizesquarefeet'	Area of the lot in square feet	7528
'numberofstories'	Number of stories or levels the home has	NaN
'parcelid'	Unique identifier for parcels (lots)	11016594
'poolcnt'	Number of pools on the lot (if any)	NaN
'poolsizesum'	Total square footage of all pools on property	NaN
'pooltypeid10'	Spa or Hot Tub	NaN
'pooltypeid2'	Pool with Spa/Hot Tub	NaN
'pooltypeid7'	Pool without hot tub	NaN
'propertycountylandusecode'	County land use code i.e. it's zoning at the county level	100
'propertylandusetypeid'	Type of land use the property is zoned for	261
'propertyzoningdesc'	Description of the allowed land uses (zoning) for that property	LARS
'rawcensustractandblock'	Census tract and block ID combined - also contains blockgroup assignment by extension	60371100
'censustractandblock'	Census tract and block ID combined - also contains blockgroup assignment by extension	60371100000000
'regionidcounty'	County in which the property is located	3101
'regionidcity'	City in which the property is located (if any)	12447
'regionidzip'	Zip code in which the property is located	96370
'regionidneighborhood'	Neighborhood in which the property is located	31817
'roomcnt'	Total number of rooms in the principal residence	0
'storytypeid'	Type of floors in a multi-story house (i.e. basement and main level, split-level, attic, etc.). See tab for details.	NaN
'typeconstructiontypeid'	What type of construction material was used to construct the home	NaN
'unitcnt'	Number of units the structure is built into (i.e. 2 = duplex, 3 = triplex, etc...)	1
'yardbuildingsqft17'	Patio in yard	NaN
'yardbuildingsqft26'	Storage shed/building in yard	NaN
'yearbuilt'	The Year the principal residence was built	1959
'taxvaluedollarcnt'	The total tax assessed value of the parcel	360170
'structuretaxvaluedollarcnt'	The assessed value of the built structure on the parcel	122754
'landtaxvaluedollarcnt'	The assessed value of the land area of the parcel	237416
'taxamount'	The total property tax assessed for that assessment year	6735.88
'assessmentyear'	The year of the property tax assessment	2015
'taxdelinquencyflag'	Property taxes for this parcel are past due as of 2015	NaN
taxdelinquencyyear'	Year for which the unpaid propert taxes were due	NaN

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

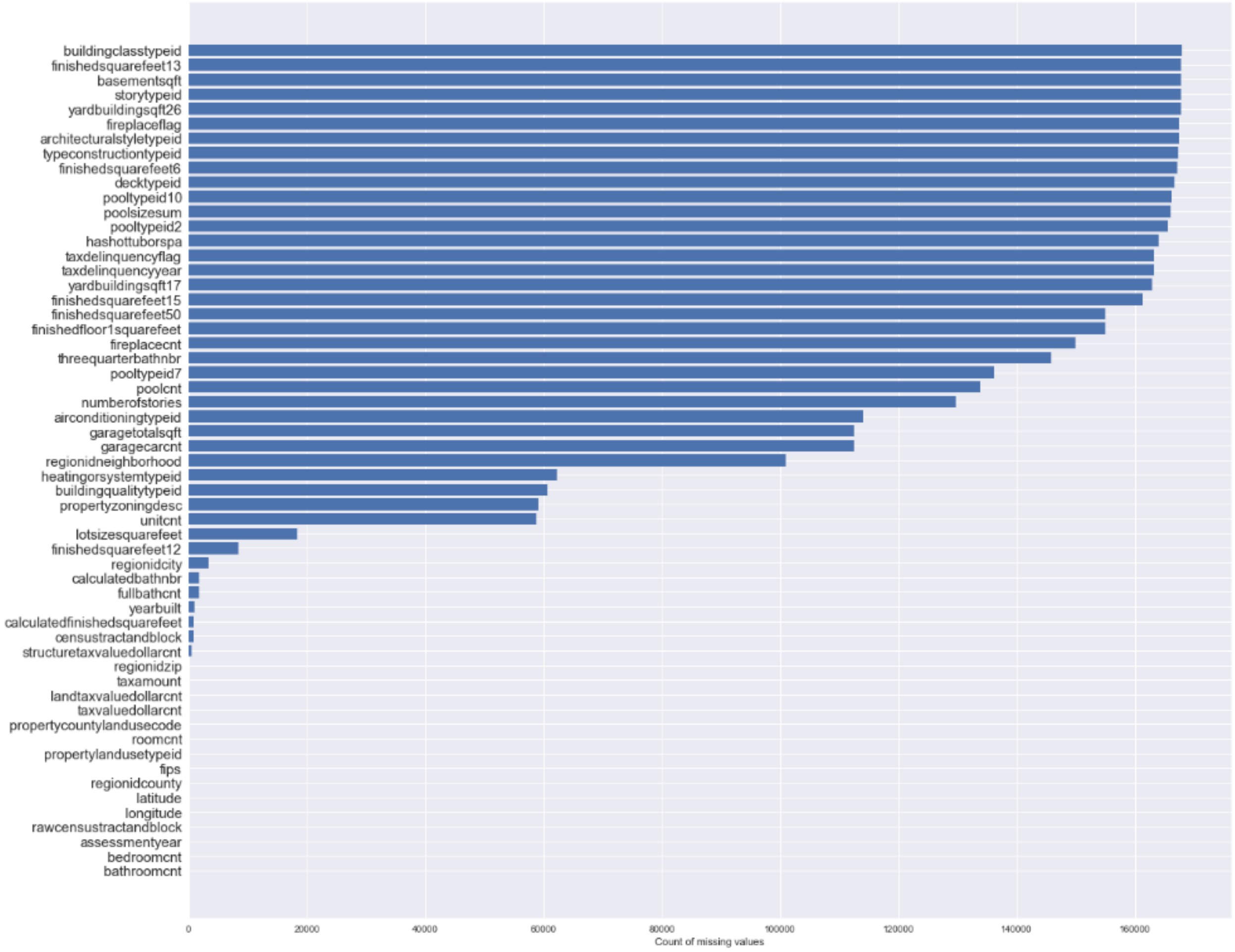


Column Type	Count
int64	1
float64	53
datetime64[ns]	1
object	5

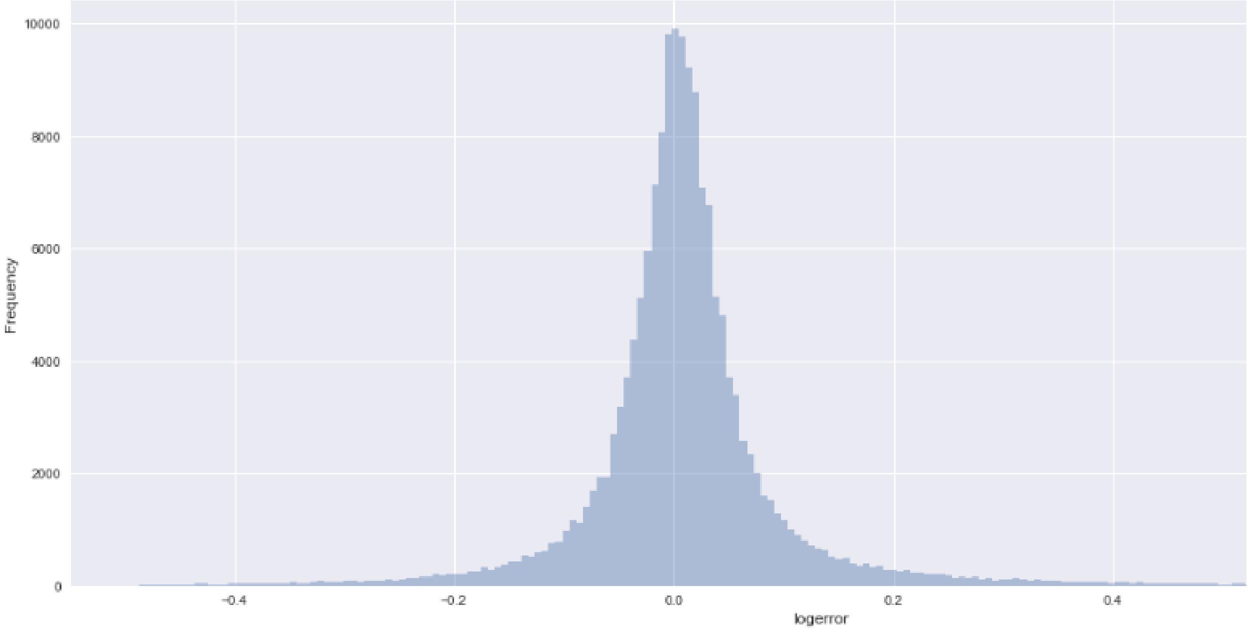
There are 19 categorical fields, 9 of which are ID features. ID data are codes that represent a description. Zillow provides the ID -> description mapping. For example:

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. Note: outliers (>99 percentile, <1 percentile) are left off the graph:



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 will be 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 turn 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

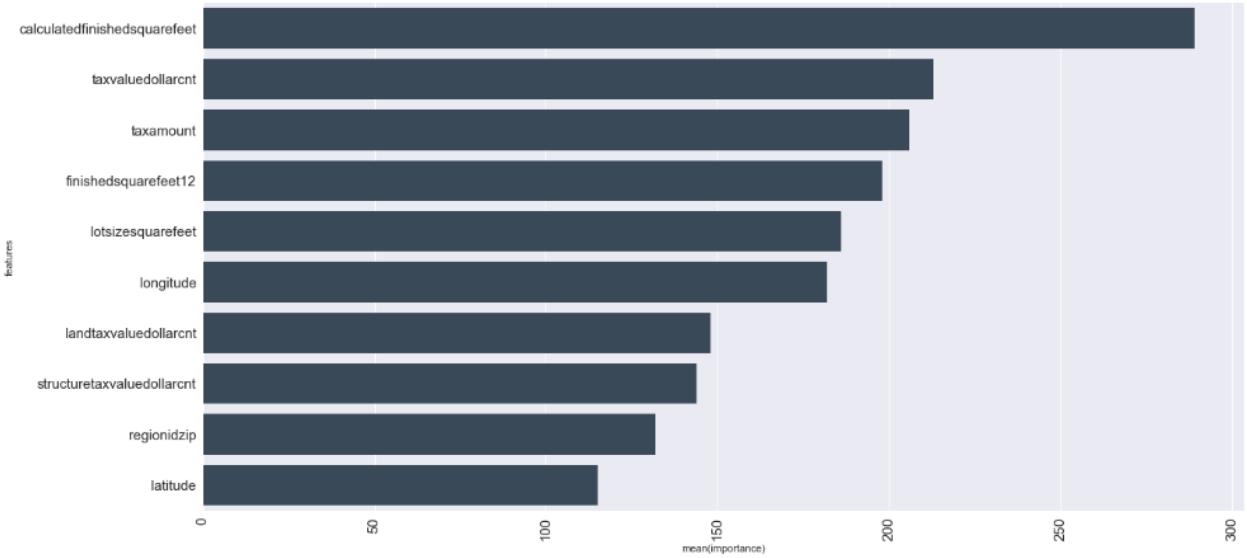
Data Preprocessing

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:
    a. transaction_year
    b. transaction_month
    c. 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:

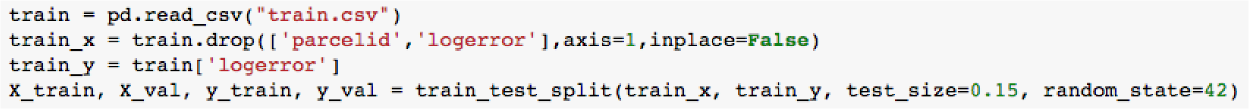
1. **Haversine distances** – determines the great-circle distance between two points on a sphere given their latitudes and longitudes. For each real estate property, I computed 3 haversine distances to downtown LA, Beverly hills, and San Clemente. These points were sufficiently spaced such that, in theory, the model could use them to accurately triangulate a location.
    a. haversine_la
    b. haversine_beverlyhills
    c. 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

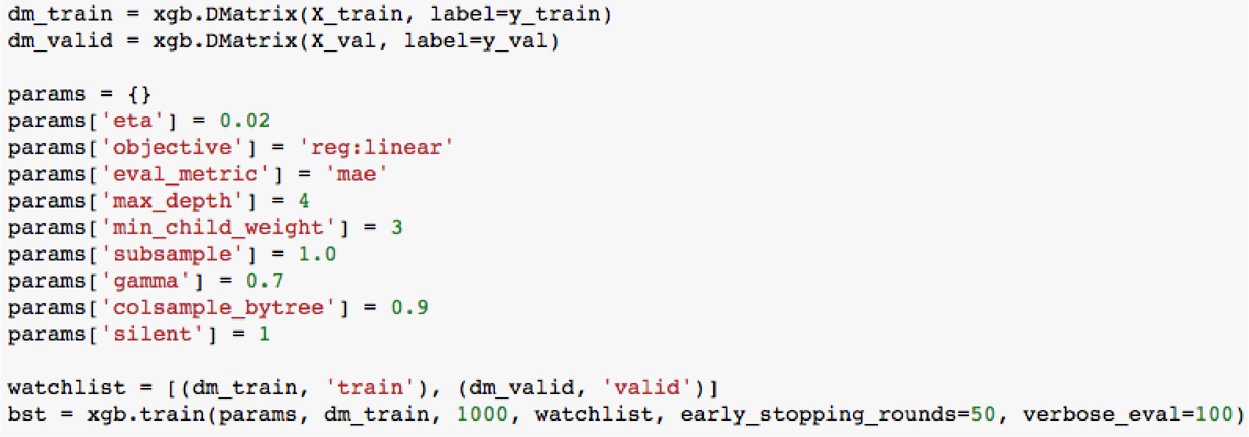
Using the XGBoost library is a rather straight forward process. I didn't face any particularly difficult coding challenges because the XGBoost, CatBoost, and hyperopt libraries have intuitive APIs and useful options, such as automatically calculating feature importance. The data didn't require any unusual visualization techniques, and could be explored and manipulated with the usual python libraries. 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.



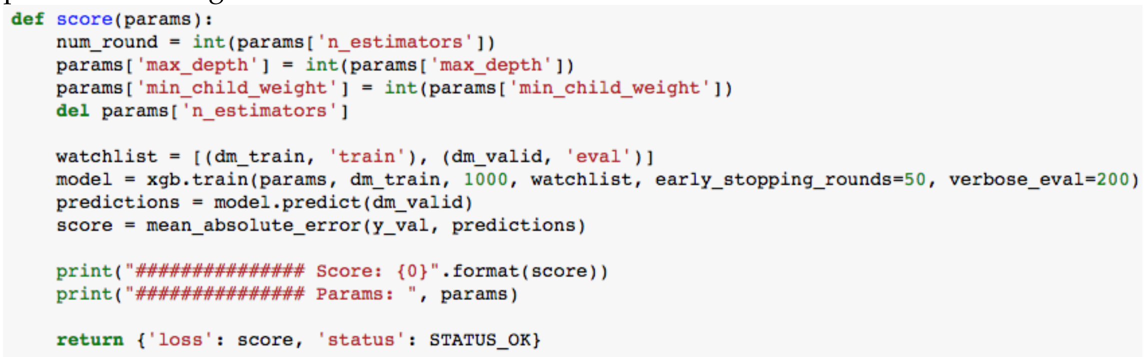
```
train = pd.read_csv("train.csv")
train_x = train.drop(['parcelid','logerror'],axis=1,inplace=False)
train_y = train['logerror']
X_train, X_val, y_train, y_val = train_test_split(train_x, train_y, test_size=0.15, random_state=42)
```

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.

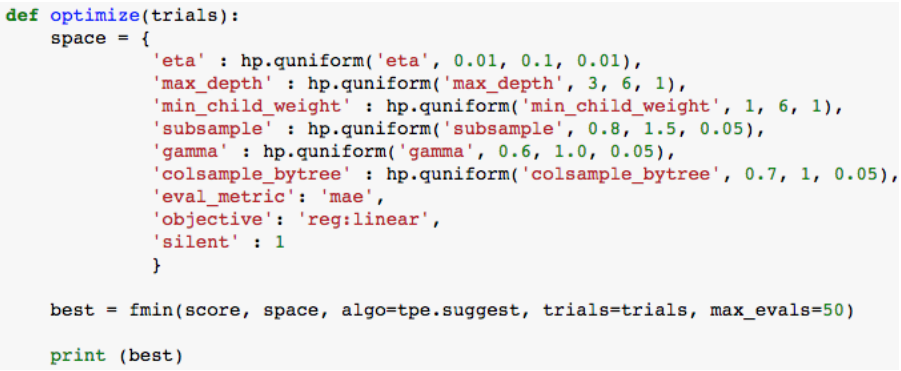


```
dm_train = xgb.DMatrix(X_train, label=y_train)
dm_valid = xgb.DMatrix(X_val, label=y_val)

params = {}
params['eta'] = 0.02
params['objective'] = 'reg:linear'
params['eval_metric'] = 'mae'
params['max_depth'] = 4
params['min_child_weight'] = 3
params['subsample'] = 1.0
params['gamma'] = 0.7
params['colsample_bytree'] = 0.9
params['silent'] = 1

watchlist = [(dm_train, 'train'), (dm_valid, 'valid')]
bst = xgb.train(params, dm_train, 1000, watchlist, early_stopping_rounds=50, verbose_eval=100)
```

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



```
def score(params):
    num_round = int(params['n_estimators'])
    params['max_depth'] = int(params['max_depth'])
    params['min_child_weight'] = int(params['min_child_weight'])
    del params['n_estimators']

    watchlist = [(dm_train, 'train'), (dm_valid, 'eval')]
    model = xgb.train(params, dm_train, 1000, watchlist, early_stopping_rounds=50, verbose_eval=200)
    predictions = model.predict(dm_valid)
    score = mean_absolute_error(y_val, predictions)

    print("################ Score: {0}".format(score))
    print("################ Params: ", params)

    return {'loss': score, 'status': STATUS_OK}
```

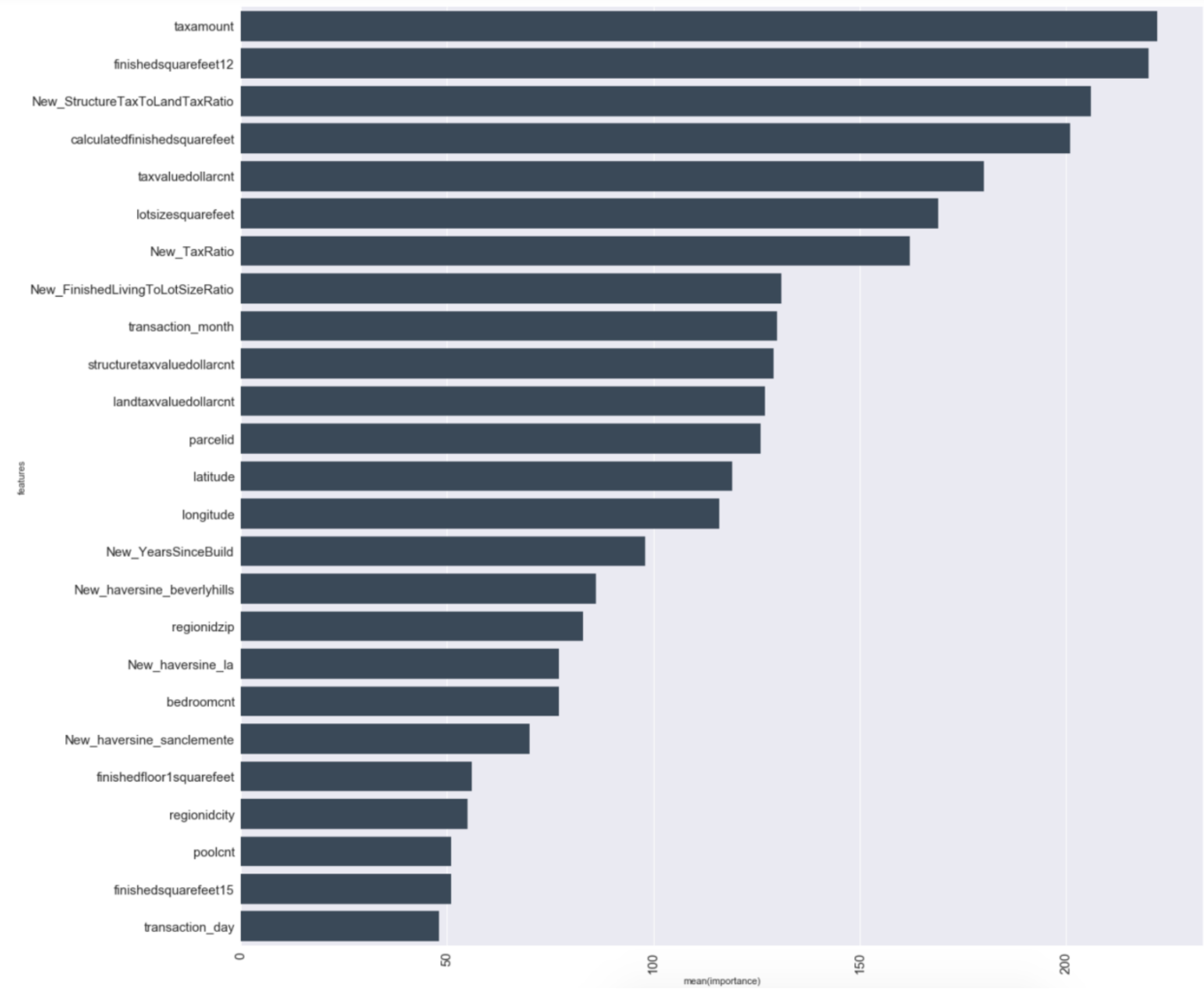


```
def optimize(trials):
    space = {
             'eta' : hp.quniform('eta', 0.01, 0.1, 0.01),
             'max_depth' : hp.quniform('max_depth', 3, 6, 1),
             'min_child_weight' : hp.quniform('min_child_weight', 1, 6, 1),
             'subsample' : hp.quniform('subsample', 0.8, 1.5, 0.05),
             'gamma' : hp.quniform('gamma', 0.6, 1.0, 0.05),
             'colsample_bytree' : hp.quniform('colsample_bytree', 0.7, 1, 0.05),
             'eval_metric': 'mae',
             'objective': 'reg:linear',
             'silent' : 1
             }

    best = fmin(score, space, algo=tpe.suggest, trials=trials, max_evals=50)

    print (best)
```

Refinement

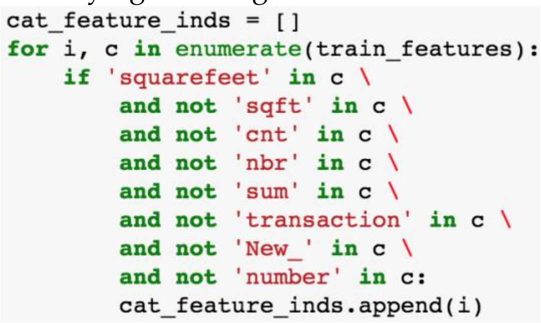
After dozens of feature engineering cycles, my best performing XGBoost model scored MAE = 0.06915645 on the test set – an improvement of 2.1% over the benchmark MAE. This model was trained on features detailed in the Data Preprocessing section, and final model parameters were tuned with hyperopt.

Top 25 features for the XGBoost model:



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. This model takes the same pre-processed data and new features as the XGBoost model, with the only difference being that one-hot-encoding isn't necessary, and NaNs are replaced with a default value: -999.

Identifying the categorical feature indices:



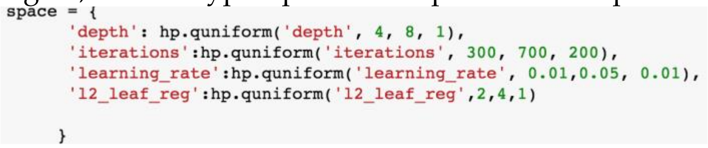
```
cat_feature_inds = []
for i, c in enumerate(train_features):
    if 'squarefeet' in c \
        and not 'sqft' in c \
        and not 'cnt' in c \
        and not 'nbr' in c \
        and not 'sum' in c \
        and not 'transaction' in c \
        and not 'New_' in c \
        and not 'number' in c:
        cat_feature_inds.append(i)
```

Results

Model Evaluation, Justification, and Visualization

My best performing CatBoost model scored MAE = 0.0677197127 on the test set – an improvement of 4.2% over the benchmark MAE. The training MAE score was 0.06544, so the model is slightly overfitting the training data, but is still generalizing well to unseen data.

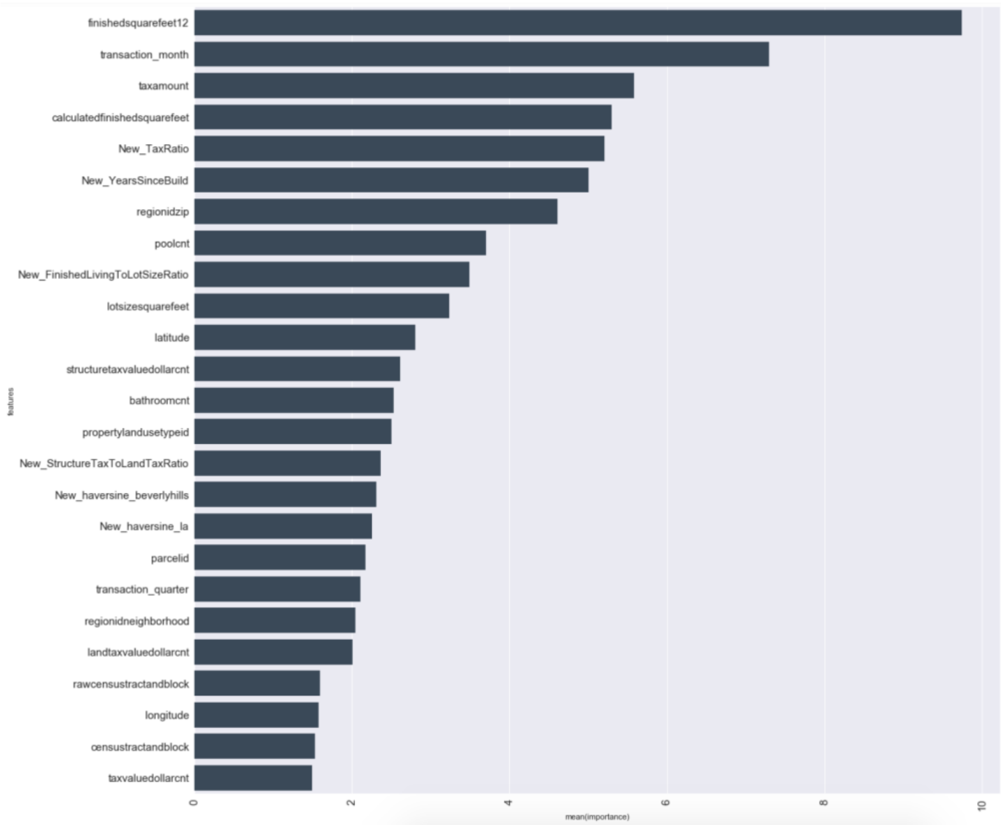
Again, I used Hyperopt to final optimal model parameters with the following search space:



```
space = {
        'depth': hp.quniform('depth', 4, 8, 1),
        'iterations':hp.quniform('iterations', 300, 700, 200),
        'learning_rate':hp.quniform('learning_rate', 0.01,0.05, 0.01),
        'l2_leaf_reg':hp.quniform('l2_leaf_reg',2,4,1)

    }
```

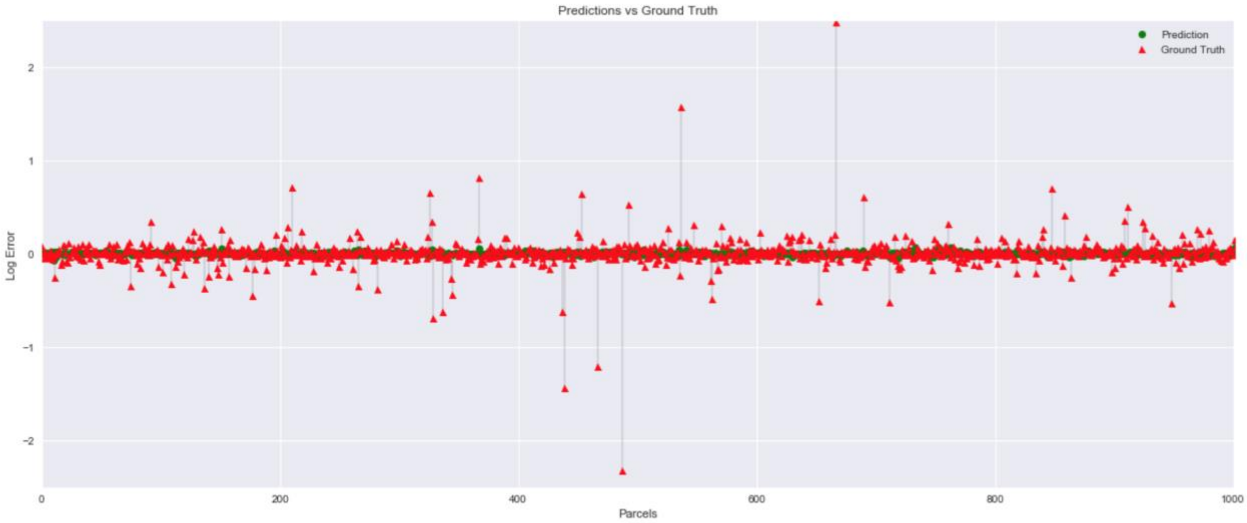
The final model parameters were:
Depth: 4
Iterations: 600
Learning Rate: 0.03
L2_leaf_reg: 3

Eight of the engineered features ranked in the top 25 in feature importance analysis. Feature importance for the final CatBoost model:



Interestingly, the model does not accurately predict large log errors. It is very biased towards predicting small error. This makes the model robust to outliers and small changes in the data, but not able to predict the outlier data. Indeed, it could be argued that this failure means the model is not an adequate solution to the problem, because the huge prediction errors in outliers are driving up the MAE score.

Figure below: The first 1,000 parcel predicted log error vs ground truth log error. Note that large ground truth log errors are never predicted accurately. The model is very biased towards small log errors.



Conclusion

Reflection and Improvement

Starting with the competition-winning XGBoost library, I went through many cycles of feature engineering, data manipulation, and XGBoost validation testing to determine an optimal feature set for this project. This process was really the story of this project - it was less about finding the right algorithm with the right parameters and more about creating good features. I enjoyed this aspect of the project because it required a lot of experimentation and creativity. Ultimately, the CatBoost model provided the best performance on the test set that resulted in over 4% improvement on the benchmark model. This is a positive result, but there is no real way to determine if this problem is "solved". Ultimately, Zillow is trying to determine how their Zestimate algorithm can be improved. Therefore, feature importance analysis on a model that is predicting error in the Zestimate will likely be valuable information to them. It is up to Zillow evaluate the effects of this analysis.

I have no doubt that further feature creation could result in a significant jump in MAE. One limiting factor for feature creation is the restriction on outside data sources. This makes serious data augmentation rather difficult. As shown previously, tax features and location features carry the most predictive power. I suspect that some more powerful features can be derived from them. In particular, it would be important to figure out what cases the ground truth log errors are very large, and work in some way of handling this, as the failure to predict large log errors is driving up the MAE score.