Machine Learning Engineer Nanodegree

Capstone Project

Lin Muqing
Jan 13th, 2018

I. Definition

Project Overview

I choose the Kaggle "Zillow Prize: Zillow's Home Value Prediction" round 1[1] competition as my capstone project. Zillow is a US real estate firm that has an in-house parcel value prediction model called Zestimate. In the first round of the competition, participants need to be able to build a model to use given features to predict the log error between the actual transaction price and the Zestimate valuation.

Related dataset is provided by Kaggle[2], including data of properties of over 3 million houses, that is a collection of features for each parcel_id; and Zestimate prediction error data of actual sales, each sale record includes error, sale-date and the parcel_id involved. There are two versions of these data, properties evaluated at beginning of 2016 and actual sales of 2016, and that for 2017. Be noted that the Zestimate predicted prices are generated by the same model, but with different properties data for 2016 sales and 2017 sales. Training data includes all sales of 2016 months 1 – 9, part of sales of 2016 months 10 – 12; all sales of 2017 months 1 – 8 and part of sales of 2017 month 9. Training data release is divided into 2 rounds, the first contains only 2016 data, which is available since the beginning of the competition; the second is 2017 data, which only takes place 2 weeks before deadline. As part of Kaggle rules, there are 2 testing datasets, the public Leaderboard (Public LB), which is used by participants to view their models' out-of-sample performance before final submission; and the private Leaderboard (Private LB), which is the evaluation dataset to get participants' final score and ranking. Public LB's data is not available, but participants can view the performance of models on this dataset multiple times. Private LB's data is not available neither and participants can only choose 2 models before deadline for submission on this dataset to view the score and use the better one of the two to get final ranking. For Zillow Prize-1, Public LB contains part of sales of 2016 months 10 – 12 and Private LB contains part of sales of 2017 months 10 and 12 and all sales of 2017 month 11. Be noted that competition submission deadline is 2017 Oct. 17th, so Private LB is a truly out-of-sample evaluation, with no potential leakage.

One thing special about this competition is that, besides a standard Kaggle medal, ranking into the first 100th is a more important goal for all participants, because only the first 100th in Zillow stage 1 competition can enter stage 2, where the 1 million dollar prize money lies.

For this capstone report, I am not going to chronologically record all the stuff I have done for the

competition, it was quite a mess and would be confusing. Instead, I will only record several most valuable trials. Regarding dataset, I will directly use the combined version of 2016 and 2017, as it seems trivial to document the fact that model performance gets better when I get more training data as 2017 data gets released.

Problem Statement

This is a classic supervised learning problem, with training data of properties of each transaction, including properties of parcel being transacted and time of transaction, as input x and log difference between Zestimate and actual transaction price, i.e. log error = log(Zestimate) – log(SalePrice), as output y. And it is a regression problem, as target y, the log error, is continuously distributed. Be noted that while each parcel has a fixed features specification, it could be traded many times and each time the transaction value could be different, also, since we do not know which houses are, or will be, actually traded in the testing dataset, participants need to make prediction for all of over 3 million houses for both 2016 and 2017 for each month of 10, 11, and 12, and Kaggle will pick up those with actual sales on given year-month combination to evaluate performance.

Metrics

There is an assigned model performance evaluation metric in this competition: average Mean Absolute Error between the model predicted log error and true log error, i.e. $\frac{1}{n}\sum_1^n |\text{predicted log error} - \text{true log error}|$. It is reasonable to use MAE instead of MSE here, as MSE implicitly gives more weight on samples that have larger absolute error to predict the logerror. Here the logerror distribution is highly heavy tailed, and very likely we will do badly on those extreme values than others. So if using MSE, we will try to improve prediction on those large logerror samples while sacrificing accuracy on others, i.e. for parcel value prediction, it will sacrifice overall accuracy when trying to do better on those we do very bad before, which I believe would not be a preferred solution for business. Using MAE means one unit error reduction in those with large errors is equally valuable to us as in those with small errors.

II. Analysis

Data Exploration

We have total of 57 raw features in the provided property data, data exploration is conducted on these features only. Since we are predicting the errors of a fine-tuned model, we don't really expect any remaining significant linear patterns between features and prediction target. And as mentioned later, since boosted-tree model is used here, we don't worry about collinearity. So instead of first looking at a corr matrix among features and target, we directly look at each one of the features in the following three angles:

1. Feature vs. log error patterns from local regression. A good feature should see different values of mean-log error at different area.
2. Feature vs. abs log error patterns from local regression. Abs log error is not directly related to original problem, but due to heavy tailed log error distribution, we could significantly improve results if we could predict well on the large error area. A potential improvement is to first predict where Zestimate makes a large error and we can build a different model for that.
3. Density: significance of contribution of feature to prediction should consider the sample density in the 'patterned' area.

Features are subjectively classified into 4 categories, ranked from 1 to 4:

1. Very good features: with low missing rate and good patterns against log error at high density area.
2. Good features: they have class-1 potential but with higher missing rate or not significant pattern.
3. High cardinal: categorical features with high cardinality.
4. Bad features: either very high missing rate (over 90%) or hardly any pattern.

All the details could be found in data_explore.html, and summarization of key information could be found in data/features_info.csv, be noted that a better readable naming is created for each feature and this will be used in the rest part of the project.

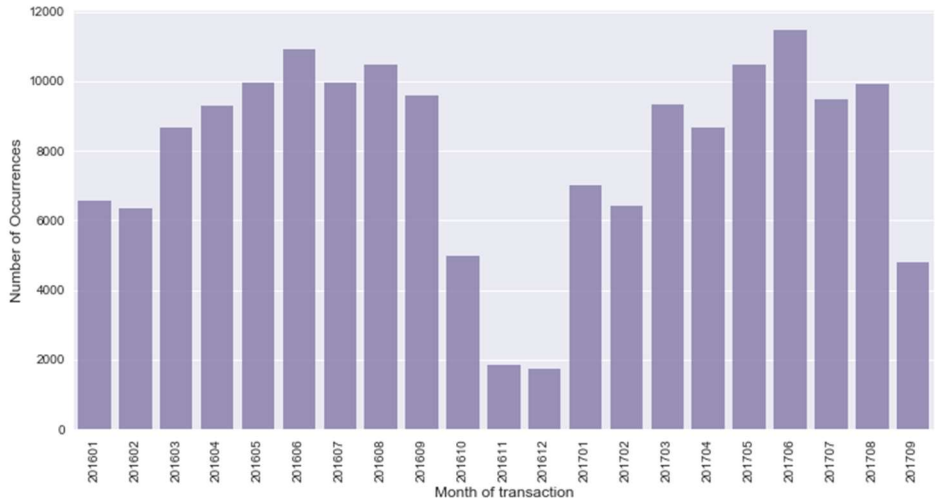
It is worth noting that this hand-labeling of features turns out to be no more efficient than boosted-tree's feature importance analysis. It helps no more than providing a more concrete idea of what we have in hand. In fact these plots could be misleading in the following 2 aspects:

1. For highly concentrated numerical features, 'pattern' across the whole value domain could overshadow the local structures at high-density area that we really care about (e.g. area_living_type_12).
2. Scale of pattern-plot could be dominated by the abnormal-behaving low-density area and difference at high-density area gets visually shrunk (e.g. type_air_conditioning).

Exploratory Visualization

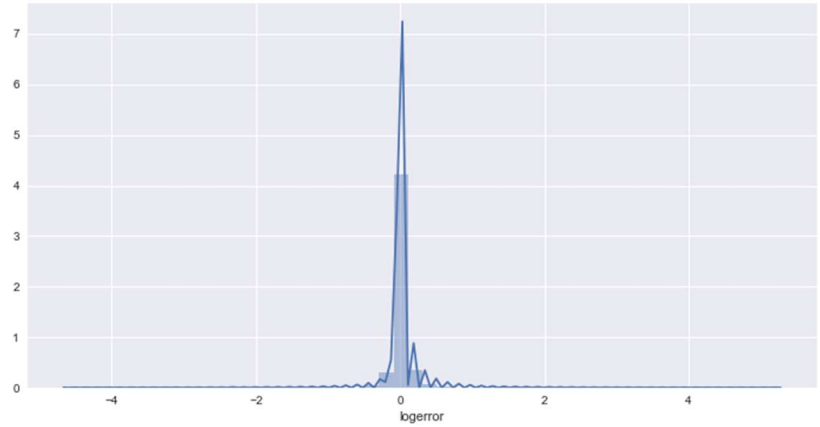
First we get an idea of overall training data.

Let's take a look at number of samples of each month:



As described in data set section, only part of 2016, 10, 11, 12 data is provided in training. Seasonality effect is strong, and sample size distribution needs to be considered for CV design.

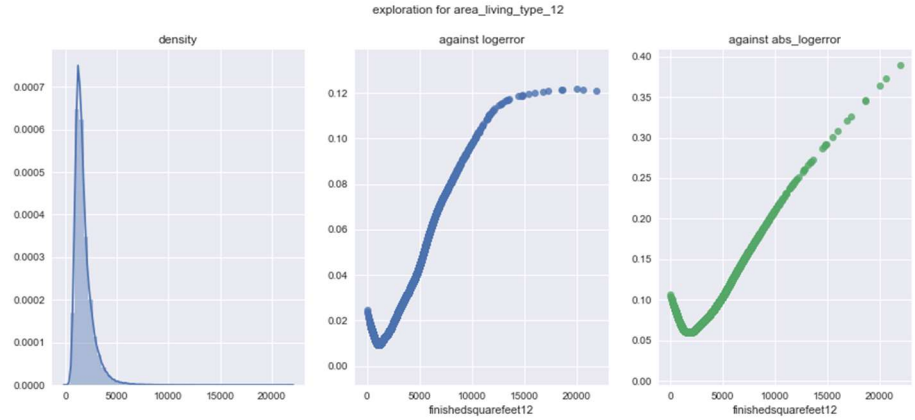
And the distribution of log errors:



Zestimate really does a good job, the log error is already noise-like, well symmetric, close to zero; but heavy tailed, meaning outlier might need to be handled.

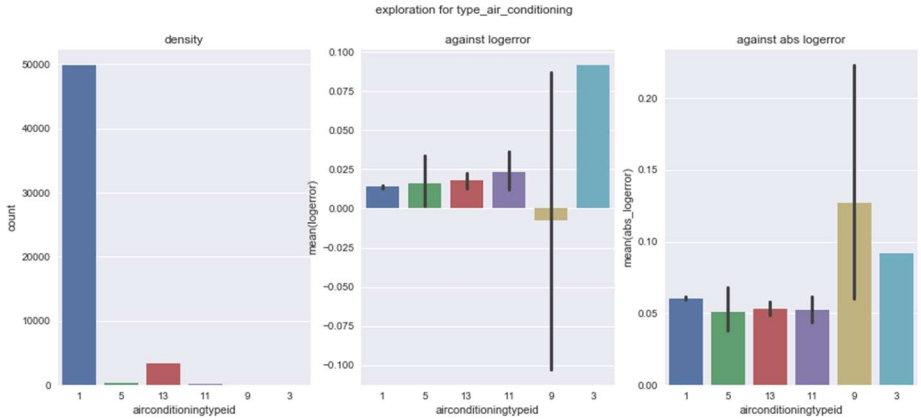
Also, we present visualization of one example feature of each of class 1, 2, and 4. There is no visualization of high-cardinal features.

1. Class-1: area_living_type_12



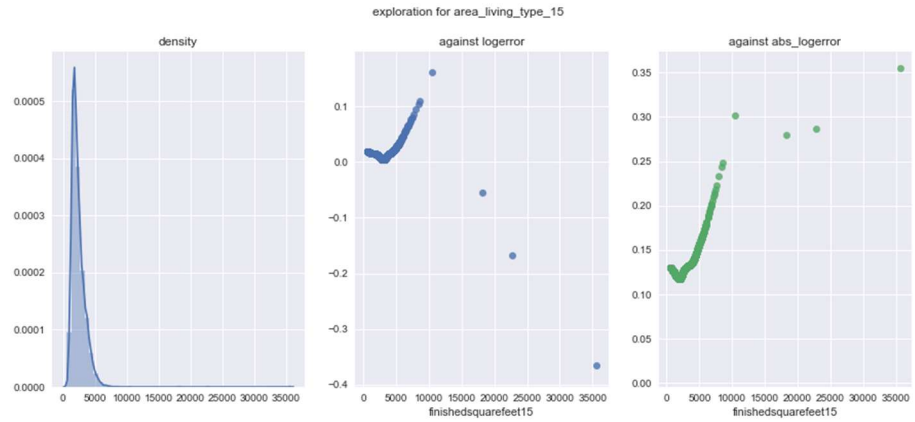
With low missing rate, area_living_type_12 looks like a good feature.

2. Class-2: type_air_conditioning



Missing rate is around 70%, high density area is only type-1 and type-13. There is slight difference between the mean-log-error of the two groups. Type 13 has sample size of around 3k, so tree can make several valid splits in there and potentially capture the pattern.

3. Class-4: area_living_type_15



This is a special feature, besides the over 90% missing rate, it seems the pattern is weak in the high density area. However, later in feature importance analysis, it turns out to be quite useful.

Algorithms and Techniques

I choose to use GBDT (Gradient Boosted Decision Tree) model family to solve the problem. Tree is non-linear and sufficiently expressive, and boosting mitigates over-fitting. Random Forest is less ideal in the sense that trees are not related to each other, new trees does make use of information of previous ones. AdaBoost does not fit well as its main contribution is to make smart combinations of existing weak predictors, but we are building the model from scratch, another algorithm is needed to first find those predictors. After all, the most direct reason for choosing GBDT is its reputation in the Kaggle community, GBDT model family has been proved well effective in multiple competitions.

Idea of GBDT is that, each new predictor fits to gradient of the loss function to previous predictors' prediction results. Think of the process of training of GBDT as Gradient Descending to find parameters of other models like neural network, just the parameters GBDT looks for are the predicted values for each tree leaf. There are three GBDT models mentioned in the report: XGBoost[3], LightGBM[4] and CatBoost[5].

XGBoost makes use of both first and second derivative of loss function to minimize it, while traditional GBDT uses only first derivative. And there are many other useful features, like including regularization in both size of predictions and tree complexity, see reference for all details. It is worth explaining the difference between leaf-wise tree and depth-wise tree. XGBoost (and LightGBM below) is leaf-wise tree, and CatBoost is depth-wise. During the process of finding the best structure of each tree, depth-wise algo will look for the best splits for each node at depth n before going to depth n+1, while leaf-wise algo will search through all the leaf nodes at each iteration, find the best split for each, then pick the one with most improvement for this iteration. So leaf-wise algo normally will find a much deeper tree then depth-wise algo,

LightGBM is a computationally improved version of XGBoost in terms of strategic subsampling and taking advantage of sparse features. So it is much faster, but less accurate in training. However it turns out to be no worse in prediction with testing data, which could be possibly explained by less over-fitting from less accuracy. It is worth mentioning that LightGBM provides automatic handling of categorical features by sorting the classes by gradient information[6] and try splits only at class boundaries.

CatBoost uses a different strategy in mainly 3 aspects: (1), depth-wise tree; (2), categorical features are also auto-handled, but it uses a customized logic to transform them into numerical features; (3), special handling of biases in GBDT[7]. Besides, according to source code, it uses a different way to achieve L2 regularization, and it seems only first order derivative is used.

For my model, LightGBM is chosen against XGBoost for mainly two reasons:

1. XGBoost's default API does not have MAE as loss function. Although API for customized loss function is provided, without sureness of correct implementation, I would go with LightGBM, where MAE can be directly configured.
2. XGBoost has to handle categorical variables with one-hot encoding. It is less ideal for four reasons that I can think of:
    a) There is no way to directly handle high-cardinal features.

b) With sub-sampling on features, such setup would give higher weights on categorical features.
c) For categorical features with more than 4 classes, with one-hot encoding, each split can only look at one class, which loses the big picture of the whole structure.
d) It expands number of effective columns, which expands data-size (important here as we have over 3 million rows to predict).

Several important hyper parameters that I tuned:

- Learning_rate: learning rate for GBDT is of the same meaning as Gradient Descending, i.e. how fast the 'parameters' update in each iteration. This parameter exists for both LightGBM and CatBoost.
- Number_leaf: this parameter only exists for LightGBM. For leaf-wise algo, number_leaf is a more characteristic feature to represent complexity than depth.
- Depth: CatBoost is depth-wise tree, so, as mentioned above, only depth parameter is provided, and we use this to control tree complexity.
- Min_data_in_leaf: this is a regularization parameter and only exists in LightGBM. With this parameter, a node will not be considered for splitting when number of samples it contains is too small, so that over-fitting is mitigated. Be noted, LightGBM also provides L1 and L2 regularization parameters, which I choose to be left as 0. From some testing trails in parameter search, these regularization parameters compensate for each other, e.g. holding everything else the same, small min_data_in_leaf and large L1 param can give similar results as a large min_data_in_leaf and small L1 param; so tuning one regularization param is sufficient.
- L2_leaf_reg: this is CatBoost's regularization param. It handles overfitting is an uncommon way. According to source code, value of each leaf node in a RegressionTree is given as sum(sample_value) / (n_sample + l2_leaf_reg). Its effect is to have much stronger regularization for smaller nodes than big nodes.

Benchmark

I choose 2 benchmarks for this problem.

First is median prediction. Since the target is already noise-like, a naïve median prediction provides a baseline.

Second is a raw LightGBM model, 'raw' meaning we take no feature engineering and uses only provided properties as features, even it is well expected that seasonality effect exists, Let's check the contribution from handling seasonality in model iterations. Even it is said that 'raw' features are used, preprocessing of raw features, as described in Data Processing section, is also applied in this benchmark model. Besides, hyper parameters have been tuned with random-search, details will be explained in implementation section.

Performance of the benchmarks and all the other model trials will be demonstrated in the Results section.

III. Methodology

Data Preprocessing

General preprocessing of properties (implementation of 1-6 to be found in data_prep.py, and 7 to be found in model prep function in cat_boost_models.py and lgb_models.py respectively):

1. Transform num-typed and bool-typed categorical features to string-type. For example, FIPS is read as 4-digit int, and pool_count only has values of 0-1. This transformation may not be necessary, but it ensures avoiding any potential floating-point precision issues and achieves cleanness of data-types.
2. Extract information from census_tract_and_block data. Raw_census_tract_and_block is float-like and census_tract_and_block is int-like. But if we carefully look at the values of those numbers, it could be told that each contains 3 pieces of information:
   a) First 4 digits are the same of FIPS, which is duplicated information.
   b) Next 6 digits (in raw_census_tract_and_block, it is in xxxx.xx format, which makes the info interpreted as a float) are census_tract code.
   c) Last 4 digits are block code.

   So these two columns are extracted as raw_census, raw_block and census, block columns.
3. Cleaning small groups of categorical features. For categorical features, there are some groups whose values only appear in testing dataset but not in training dataset. How the model implementation handles these values in predicting has not been explained in model doc, and I didn't took the trouble to look into source code, so I simply set their values to nan when predicting for testing data set.
4. Cleaning area_garage and num_garage. There is some obvious inconsistency between these two variables, where there exist houses with zero number of garages but having non-zero garage area. This kind of rows makes up of around 6 percent of all properties data. Missing ratio of area_garage is around 70%, so I think it provides sufficient information to fill in zero-garage area according to number of garages. So for such a row, given its number of garage, use average of garage area of all other samples with the same number of garage to fill in for its garage area.
5. Propagating of area_pool from num_pool. First, number of pool only takes values of 1 and NA, so it is reasonable to take NA as 0 pool, then num_pool has no missing. On the other hand, original area_pool has over 99% of missing rate, unlike garage case, it is hard to fill in area_pool data from other samples. So I simply fill in NA area_pool as 0 for rows with 0 num_pool, so that 0 pool_area and NA pool_area can be differentiated.
6. Fill in NA with 0. For num_fireplace and num_34_bathroom, data only contains NA but no 0, so I think it is reasonable to consider NA as 0 for them.
7. Following pairs of features contain almost the same information.
   a) num_bathroom_assessor – num_bathroom
   b) code_county – fips
   c) area_living_type_12 – area_living_finished_calc
   d) area_firstfloor_assessor – area_firstfloor_zillow

So I only kept one (later one in the listing above) for each pair.

Details for each one of the features could be found in data_explore.html.

Preprocessing for LightGBM:

LightGBM auto-handles categorical variables, just need:

1. Categorical features have to have their values being mapped to an int.
2. Model should know which features are used as categorical, I choose to mark data-type of these feature columns as 'category' in pandas.DataFrame.

One thing to take special care is that, we need to make sure the created mapping from string labels to int is consistent in all datasets, i.e. testing & training, 2016 data and 2017 data.

Preprocessing for CatBoost:

CatBoost also auto-handles categorical variables. The handling is easier than LightGBM, just need to provide the indexes of categorical columns. So we need to keep information of that. Another thing is handling of NA for numerical variables, I have not tried the effect of leaving NAs there, just to mimic the Python examples on CatBoost Github, I set NAs to -999, which is fine here as all the variables are non-negative.

Be noted that extreme values are not handles in this section because all the training data are from actual transactions. Let's first see how the original data performs, then see how extreme values affects model behavior in model iterations.

Implementation

Both LightGBM and CatBoost have provided handy API to perform CV, training and prediction. The versions I use are 2.0.5 for LightGBM and 0.2.5 for CatBoost. These are not the lasted published versions, I use the same versions as those I used when I participated the competition.

Parameter tuning (implementation to be found in cat_boost_model.py and lgb_models.py):

Hyper-parameters of each model are tuned through random search.

- Given a parameter set, built-in CV function for LightGBM and CatBoost would provide the average evaluation metric, which is MAE as described in Definition section, across number of folds, which I choose to be 5 for both.
- Total number of iterations is set to be high, 12000 for LightGBM and 3000 for CatBoost, and an early stopping condition is set, 100 for LightGBM and 50 for CatBoost. Early stopping works as following, folds average MAE is calculated at each iteration, so at iteration n, we have the metric value as V(n), then for each of V(n + 1) to V(n + m), if none of them gets a better than V(n), the CV process will be recorded as terminated at n, and m here is the early-stopping-rounds parameter.
- The returned value of CV is a list of evaluation metrics, one value for each iteration, so besides the final performance of a parameter set input, we also get the corresponding number-rounds parameter by checking the length of the output.
- I have set the stratification flag to False, if it is set to True, the folds data would be balanced

according to a certain group. I don't think it is necessary here because the data set is sufficiently large and here CV is just for parameter search, a more careful design would have been performed in the real CV process.

- I do 100 searches for LightGBM and 50 for CatBoost, and pick one(s) at the top to use.

Random-searched parameters for LightGBM:

- Learning_rate: log linear 0.1 ** Uniform(1, 3). It has max of 0.1 and min of 0.001, but gives more trails at small end in the linear space between min and max.
- Min_data_in_leaf: Uniform(100, 600).
- Num_leaf: Uniform(30, 80).

Random-searched parameters for CatBoost, CatBoost has been advertised as default parameters would perform well in many problems, so I am not going too much away from default.

- Learning_rate: default 0.03, search Uniform(0.015, 0.045).
- Depth: default 6, I am searching Uniform(5, 7).
- L2_leaf_reg: default 3, I am searching Uniform(1, 9).

There is no strict reasoning of the choice of the searching range of the parameters above. They are just set around some benchmark, which is default values for CatBoost, and values used in some public kernels in the Kaggle-Zillow1 forum for LightGBM. For final chosen parameters please see params.py and reasoning in model_iterations.html.

Feature Importance (implementation to be found in features.py):

Feature importance analysis is only performed under LightGBM. Since tree is robust to collinearity and useless features, features' importance information is not used for feature selection, but only used to get a general idea of how each feature contributes to help understand the model, and used for feature engineering mentioned below.

A trained LightGBM model has 'feature_importance' method, which collects feature importance information in 2 aspects, split and gain. Among all iterations and all splits of each iteration, split evaluates how many times each feature has been used for splitting, gain evaluates the total score improvement (decreasing of total loss) from using this feature for splitting. For each aspect, each feature has a score and a rank. The two ranks are not necessarily the same, but usually not very far away from each other. So an average ranking is used to evaluate one feature.

Feature Engineering (implementation to be found in data_prep.py):

There are 2 types of feature engineering I have used:

- Tax related values: these include tax_value features, which are the assessed value of the parcel on the assessment year (usually the previous year), which provides valuable information on how much this parcel might be of now, and actual tax amont. There are totally 3 types of tax values: tax_value_total, tax_value_land and tax_value_structure, where tax_value _total = tax_value _land + tax_value _structure. From tax related values, more information about a parcel can be extracted:
    1. dollar_taxvalue_structure_land_diff: tax_value_structure – tax_value _land

2. dollar_taxvalue_structure_land_absdiff: abs(tax_value _structure – tax_value _land)
3. dollar_taxvalue_structure_land_diff_norm: (tax_value _structure – tax_value _land) / tax_value _total
4. dollar_taxvalue_structure_land_absdiff_norm: abs(tax_value _structure – tax_value _land) / tax_value _total
5. dollar_taxvalue_structure_total_ratio: tax_value_total / dollar_tax
6. dollar_taxvalue_total_dollar_tax_ratio: tax_value_structure / tax_value_total

For my solution, these values are not directly used for prediction, but for functionalities described below.

- Numerical_groupby_categorical features: for any pair of numerical feature and categorical feature, we can group samples by category classes, and within each group, we can calculate:
  1. group_mean: average of numerical feature value within the group.
  2. Where each sample is located relative to group_mean: (num – group_mean) / group_mean
  3. How far each sample is away from group_mean: abs(num – group_mean) / group_mean.
  4. group_count: number of samples within the group.

  Be noted this group_by feature engineering should be performed on the 3 million property dataset. One reason is more data leads to be more robust results, and more importantly, we need to keep training and testing consistent.

With above 2 techniques, I can transform high cardinality categorical features (class 3 features) to numerical by (implementation to be found in features.py):

1. Choose a numerical feature set: I only used the 4 original tax related features and 3, 4, 5, 6 of tax derived features.
2. For a class 3 feature, create num_groupby_cat features 1 and 4 from numerical features above and the chosen class 3 feature and add them into the benchmark LightGBM model.
3. Train the model and collect feature importance information.
4. Among the chosen class 3 feature related features, mark the one with highest ranking to be used to replace the original categorical one in the final model.
5. Do 2 to 4 separately for each class 3 feature, i.e. each time only derived features of one class 3 feature is used in model.

Two-step LightGBM (implementation to be found in models.py):

It would be shown that including sale month in features will significantly boost performance. But there are two ways I can think of to include sale month information. The straight forward way is to directly use sale month as a predicting feature. Another way is, after training with a LightGBM model with no sale month, we can collect in-sample true target and predicted target for a specific month group, take the difference, add another layer of LightGBM on top of that to capture this extra information. This process can be understood as a type of model stacking, just instead of stacking a different model family with the same dataset, we stack the same model family with different dataset. So the steps are:

1. Train LightGBM (step-1) without sale_month, using original y as target.
2. Collect (in_sample_y_orig – in_sample_y_pred) as y_step2.

3. Train LightGBM (step-2) with sale_month as feature, using y_step2 as target.
4. For prediction in testing data:
   y_pred = model_step1(step1_features) + model_step2(step2_features.)

Be noted:

- For this month-specific training task, due to reduction in sample size and more 'noisy' of training target, step2's parameters needs to be separately tuned. Searched hyper parameters are the same, but the way I present the search results is slightly modified. Because variance of CV evaluation metric gets larger due to smaller sample size, I perform evaluation on each month with selected sample size of 9000, which is the size of target month (10, 11, 12) in training data. So the presented mean and stddev of performance metric is the average of all months. Most importantly, number_rounds is presented as mean and stddev across all months. For blending models, different number_rounds values are used for the each tuned parameter set.
- For the same reason above, a smaller set of features for this step is chosen (hand-picked features are in params.step2_keep_only_features). The idea is, besides training for sale_month, which would be mainly a median adjustment, some custom local structures for each month could be found.

Refinement

Below is the list of models I have tried (implementation to be found in models.py).

- Median (benchmark): as described in benchmark section.

At this point, lgb_models.param_search_raw() is run, for consistent comparison, params.lgb_raw is used for all following models until next search point is used.

- RawLGB (benchmark): as described in benchmark section.
- RawLGBSubCol: same as RawLGB, but use only hand-picked class 1 and class 2 features as described in Data Exploration section.
- RawLGBIncMon: same as RawLGB, but include sale_month as prediction feature. For all models in this report, due to imbalanced sample size, months 10, 11, 12 are grouped into one month for modeling.
- RawLGBIncMonOutlierRm: same as RawLGBIncMon, but with outlier removed. Here by 'outliers' I mean extreme values in y. Extreme values are defined as those below 0.5 percentile and above 99.5 percentile y values among the whole training data. These two thresholds are then hard-coded (see data_prep.rm_outlier) for consistency in CV.
- LGBOneStep: finalized one-step LGB model, same as RawLGBIncMonOutlierRm, but use engineered features for class 3 features, as described in Implementation section.
  Selected features are:

raw_var	raw_var_rank	picked_var	picked_var_rank
block	32	dollar_taxvalue_total_dollar_tax_ratio	11
census	50.5	dollar_taxvalue_structure_land_absdiff_norm	7
code_county_landuse	26	dollar_taxvalue_total_dollar_tax_ratio	18
str_zoning_desc	22.5	dollar_taxvalue_total_dollar_tax_ratio	13.5
raw_block	31	dollar_taxvalue_total_dollar_tax_ratio	12

raw_census	51.5	dollar_taxvalue_structure_land_absdiff_norm	9
code_city	20	dollar_taxvalue_structure	13.5
code_neighborhood	21	dollar_taxvalue_structure_land_absdiff_norm	13.5
code_zip	12.5	dollar_taxvalue_structure_land_absdiff_norm	10.5

Be noted, implementation of these features are directly included in data_prep step in code, for faster following implementation iteration.

- LGBOneStepDefaultParam: same as LGBOneStep, but uses LightGBM default values for searched hyper parameters. This is just a demonstration of effectiveness of parameter random search.

At this point, since the OneStep model is finalized, with replaced features and extreme value handling, corresponding parameter search lgb_models.param_search_one_step is run, and params.lgb_step1_i are used for LGBOneStepBlending and step 1 of LGBTwoStep models.

- LGBOneStepBlending: equal-weighted blending of 5 LGBOneStep models, each use a different set of searched parameters.

At this point, lgb_models.param_search_step2() is run and params.lgb_step2_i are used for step 2 of LGBTwoStep models.

- LGBTwoStep: As described in Implementation section, a hard-coded month-layer is added.
- LGBTwoStepBlending: as LGBOneStepBlending, we can use more than one set of parameters to make the model more robust. Since there are two steps now, here is a description of how blending is performed to avoid ambiguity:
    1. Train step1 model with each step1 parameter set.
    2. Blend trained step1 models and make prediction for in-sample targets.
    3. Get in-sample true_target – pred_target diff.
    4. Use this diff as step2_target to train step2 model with each step2 parameter set.
    5. For prediction on testing data, step1 predict is the blending of all step1 models, step2 predict is the blending of all step2 models, and final predict is the sum of the two.

At this point, we need parameter search for CatBoost models, cat_boost_models.param_search() is run, as explained below, only one param set is used for both CatBoost and CatBoostBlending, that is params.catboost_tuned.

- CatBoost: CatBoost prediction with single parameter set. It includes sale_month as feature, has outlier removal but uses class 3 features as categorical.
- CatBoostBlending: CatBoost's training relies on a seed input, and it plays a key role in the training algo (how the samples are shuffled for bias-avoiding algo). So unlike LightGBM, the blending for CatBoost is from using different training seed with same tuned hyper parameter.
- CatBoostDefaultParam: again, to demonstrate effectiveness of param search for CatBoost.
- BlendingLGB: Equal weighted blending of LGBOneStepBlending and LGBTwoStepBlending.
- BlendingAll: Blending of LGBOneStepBlending, LGBTwoStepBlending and CatBoostBlending with weight 1:1:2.

Discussion of evaluation of each model version needs to be combined with description of validation framework and validation results. So I leave this to justification section.

IV. Results

Model Evaluation and Validation

Model evaluation consists of 2 parts: local CV and LB score (and ranking). Private LB ranking would be the final evaluation, but we can check if local CV and public LB ranking provides reasonable information to help choose the best model for Private LB. For competition, only two 'best' models can be chosen for final submission. A good CV framework should be used together with Public LB score to determine the 'best' two. Be noted that making decision solely on Public LB score is risky, it is a fixed dataset and can be easily over-fitted.

On local CV side, although available API has been provided in sklearn, LightGBM and CatBoost, I write one myself, for two reasons.

1. The 'model' I am validating here is in generic term, it is a series of operations to get test_y by training with train_x and train_y and then applying to test_x. Like applying extreme value handling and 2-step LGB, compared to fitting them to existing API, it is much easier to customize CV.
2. As shown later, month effect is critical to get a good performance in this competition. CV should take that into consideration. Meanwhile, with the characteristic sample size of target months, it has to be customized to account for that. To be specific, public LB only includes 2016 10, 11 and 12 whose training data is noticeably less than the others (see Data Exploration part); private LB only includes 2017 10, 11, and 12 which has no corresponding training data at all.

3 types of local CV are constructed, reason for choosing month 4, 5, 6 as target are explained in model_iterations.html:

1. n_folds CV on all data, stratified by month.
2. Targeting for public LB, part of data of 2016 4, 5, 6 is held out as validation and the rest is included in training, and all data of 2017 4, 5, 6 is not used at all.
3. Targeting for private LB, all data of 2017 4, 5, 6 is held out as validation, and only part of 2016 4, 5, 6 is used for training, rest is not used at all.

So in general for each model version, we make up to 7 outputs (I have pulled the LB score and ranking information from internet and saved at data/public_lb.csv and data/private_lb.csv, so the ranking can be easily found through searchsorted without manually locating each score.):

- CV_stratified_avg
- CV_public_LB
- CV_private_LB
- public_LB_score
- public_LB_rank
- private_LB_score
- private_LB_rank

With some exceptions:

1. 2-step LGB is specific for LB testing data structure, the second layer can only be built on a

specific month's data, so CV_stratified_avg is not available for it.

2. Blending models' takes a long time to run, its effectiveness is straight forward reasonable so it is local CV results are demonstrated for LGBOneStep only; for others, I only show LB results.

To generate the evaluation results, for each recorded model, first run the submit function to generate submission file and make submission to Kaggle, manually copy publicLB score and privateLB score to code, then run analysis to get summarized evaluation of the model. This process (except the manual part) could be found in model_iteration.html.

Be noted that public LB ranking is somehow misleading, because the tax info in 2017 data is quite indicative for 2016 prices, using that to predict 2016 log errors suffers from forward-looking, which can make 2016 prediction really good, but useless in prediction for real test data. By only submit the predicted results, Kaggle cannot forbid people from using this information leak, it can be well assumed some high ranking public LB submissions took advantage of this leak.

The CV results of all the model versions in the Refinement section:

Model_name	cv_avg	cv_public_LB	cv_private_LB	score_public_LB	rank_public_LB	score_private_LB	rank_private_LB
median	0.068860	0.068620	0.067027	0.065361	3257	0.076327	2839
lgb_raw	0.067896	0.068046	0.066048	0.064372	763	0.075287	501
lgb_raw_sub_col	0.068001	0.068118	0.066113	0.064480	1406	0.075371	680
lgb_raw_inc_mon	0.067825	0.067872	0.065918	0.064157	232	0.075008	81
lgb_raw_inc_mon_outlier_rm	0.067824	0.067878	0.065927	0.064134	181	0.074958	58
lgb_1step	0.067783	0.067829	0.065848	0.064163	242	0.074958	58
lgb_1step_default_param	0.068111	0.068064	0.066128	0.064316	563	0.075233	435
lgb_1step_blending	0.067746	0.067808	0.065812	0.064153	225	0.074928	50
lgb_2step	0	0.067974	0.065906	0.064160	236	0.074925	49
lgb_2step_blending	0	0.067955	0.065902	0.064138	192	0.074914	44
catboost	0.067818	0.067913	0.065921	0.064135	183	0.074899	40
catboost_default_param	0.067917	0.068093	0.066039	0.064223	433	0.075040	113
cat_boost_blending	0	0	0	0.064062	50	0.074835	26
blending_lgb	0	0	0	0.064130	175	0.074900	40
blending_all	0	0	0	0.064016	27	0.074785	16

Justification

Comparing the performance of the model versions, there are some interesting check points:

1. Median model benchmark, we don't really expect much from it, but it is surprised to see that, with a total of 3779 teams participating, quite a number of them are even doing worse than the median prediction.
2. Raw LightGBM benchmark, sufficient improvement has been made from median model, let's see how much more we could do. A simple in-sample seasonality check (see model_iteration.html) shows a necessity of inclusion of sale_month as predictor.
3. Raw LightGBM with selected features, before checking seasonality, let's first see how much

contribution our manual feature selection can make. It turns out to be worse than raw model, which is yet not hard to explain. One way to see this is from the feature importance analysis of the Raw model (see model_iteration.html), some class-4 features are well contributing, and for class-3 features, the ordering may be hard to intuitively understand, but it does contribute. Another way to think of it is, besides the two deficiencies, as described in Data Exploration section, of the visualization method we used for manual selection, another important missing is failure to understand the feature value from the big picture, i.e. the relationship between features. Pattern may not exist if we directly look at the relationship between one feature and the target, but if we take a subset of the samples based on grouping of another feature, things could be different. This is hard to visualize in Exploration stage.

4. Include seasonality, key step, significant improvement, which directly brings me into the 100^{th} club.
5. Take a look at the effect of the outlier. Although it is reasonable to consider all the target values are valid, there is another angle making the outlier removal on target side to be logically necessary. We are building a model to collect patterns to predict for the future, some patterns are easy to catch and others are not. If the extreme values are hard to catch and may pollute the model we want to build for non-outliers, then remove them from model building could be beneficial. For testing-data, we cannot do anything with the outliers anyway, at least do better for non-outliers. From CV results, there is almost no difference. But public LB score confirms the potential benefit of this idea (at least no harm), and it is further confirmed in private LB final score.
6. Then feature engineering for class-3 features. On the contrary to outlier-removal, CV results shows noticeable improvement, yet no difference for LBs. Well, hard to say if this idea really contributes, but still, at least no harm, let's use it in the final model.
7. Now, go one step back to see the effect of random-parameter search. Just have to say, parameter search is really critical for LightGBM.
8. Blending of LightGBM with different parameter set, consistent improvement has been observed in both CV and LBs, although not much. Well, effectiveness of blending is easy to intuitively understood, and observations confirm this expectation. Let's keep this blending routine for final submission, and it is not really necessary to confirm each one of them in CV.
9. 2-step LightGBM, honestly, this model version is not really necessary, but during actual competition, with a CV bug, I missed the value of the direct 1-step model and had to use this one for final submission, but it proves to be effective. Yet, it performs much worse than the 1-step one in CV results, but surprisingly slightly outperforms on the LBs. So we can tell that it does contain information, but not quite stable across different dataset, original complete tool set is better than hand-crafted.
10. Blending of 2-step LightGBM, same observations as above, compared to blending of 1-step LightGBM.
11. CatBoost, same as 2-step LightGBM, worse CV, but better LB, not so stable.
12. Interestingly, CatBoost has provided a much better default parameter set than LightGBM, but not so hard-to-defeat as advertised. Contribution from parameter search is consistent.
13. CatBoost blending, as expected, does better than CatBoost.
14. Blending of LightGBM, as expected, does better than anyone used alone. But improvement is

quite little, this can be explained by the fact that the two models use essentially the same core algo, which is why I use 1:1:2 for final blending instead of 1:1:1.

15. Blending of all, now we see the power of blending of different types of algo, and we get a gold medal!

Getting back to the simulated competition process, with only local CV and public LB, the decision for final submission is not hard to make. We would have a good bet on blending_all, yet with concern of instability of CatBoost and 2-step LightGBM, we can include lgb_1step_blending as the other choice for safety.

V. Conclusion

Free-Form Visualization

There is no more visualization, the table in Results sections tells everything.

Reflection

The key components for the final solution of the problem are:

1. Data cleaning, including
    a) Making sure of consistent mapping for categorical features for LightGBM.
    b) Information extraction for census and block data.
    c) Cleaning of garage and pool data.
2. Hyper-parameter tuning.
3. Feature engineering and inclusion of sale_month as feature.
4. Removal of extreme valued samples for training.
5. Blending of models.
6. Customized CV framework.

The key takebacks are:

1. I think data cleaning is crucial here. Although it has been shown that simple parameter tuning would make a huge difference for LightGBM, I think it is reasonable to expect it to be a routine, no one really directly uses default parameters. For CatBoost, parameter tuning is not even that necessary. Contribution from feature engineering and extreme value handling is shown to be little from Results section. Yet the LGBOneStep model would have ranked me up to 58th, from the forum, other competitors have tried a lot of stuff, yet most of them cannot even beat this quite 'raw' model, I would credit the winning to data cleaning part.
2. The effect of different types of model blending is interesting. Blending of 2 LightGBM models is much weaker than blending of LightGBM and CatBoost. That says, the more different models' underlying logics are, the more gains we get from blending. So, in the future, when one model has been researched enough, compared to blending with more differently tuned same model, it would be much more efficient to try a completely different model.
3. The importance of correctly-set CV. Here in this report it is not quite obvious how much

contribution the CV framework makes, it just looks everything is consistent. This correct CV framework is developed from my experience in actual competition. During actual competition, I only did average CV, there was no customized CV against public LB and private LB. When I found out that seasonality is important, I manually did the by-month bias analysis like in model_iteration.html, but with a bug, and reached the wrong conclusion that directly include month in 1-step LightGBM does not help capture this effect. That's how I came up with the idea of 2-step LightGBM. Actually, near the end of the competition, I had thought to adjust model to specifically make better prediction for 2017 only, but there was no public testing benchmark for it, and without enough time to develop a customized CV, I had to give up the thought.

The final model version in this report gained a rank of 16, which is a gold medal. In my actual competition, I had some bugs in data processing, no well-designed CV framework and not enough time to run enough blending (I only got to know CatBoost less than 24 hours before deadline, final CatBoost version only had blending of 3). Luckily, with the 2-step idea and CatBoost, I got ranked 23 in the actual competition. It looks not far away between 23rd and 16th, but that's the difference between a silver medal and a gold medal. So a carefully planned and well-organized research process if very important.

Improvement

Winning team has score 0.0740861, my best model has score 0.0747848, while the 100th score is 0.0750251. So there is quite a lot space to improve.

Here are several things that I think could be beneficial:

1. Little feature engineering has been done. Due to the noisiness of data, we should be very careful to determine how each engineered feature contributes. I actually did a broad but not accurate search of group_by features in actual competition, with no success. I did it iteratively by adding a group of new features for each categorical variable and keep the top ones. Simply using feature importance ranking to determine a new feature's value is not a robust solution, value of a group of features should only be determined by CV and LB score. But a complete search of that would require 2^n CV runs, where n is number of new features in total. So probably intuition would play a more important role here.
2. Extreme value handling on the feature side. For now, only outlier handling on the target side has been performed, with logic explained above. We could try the same on the feature side, for example, extremely large or extremely small houses may be hard to correctly predict because of less counterparts. However that requires much more efforts, as there are much more columns to look at on feature side than on target side.
3. Unsupervised learning on feature side. From feature info summarization, there are quite some missing values. However, some columns are related to each other, and potentially information from one column can be used to fill missing values of another, like I did for garage area and garage counts. And the special structure of this problem makes this direction extremely valuable. Although we only have around 160k training data in the supervised learning part, we have around 3 million data on the feature side. There is a lot to dig.

4. As mentioned in Data Exploration section, one different way to structure the problem is to first predict whether a sample would be of large error or not, and build a different model for each case, than perform a weighted sum of the two as final prediction. I tried this in actual competition, with no success. This idea might work with more fine-tunning.
5. Blending with more model types. One candidate that immediately pops out of my head is neural network.

References:

[1] https://www.kaggle.com/c/zillow-prize-1

[2] https://www.kaggle.com/c/zillow-prize-1/data

[3] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In 22nd SIGKDD Conference on Knowledge Discovery and Data Mining, 2016

[4] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In Advances in Neural Information Processing Systems (NIPS), pp. 3149-3157. 2017.

[5] https://tech.yandex.com/catboost/doc/dg/concepts/about-docpage/

[6] https://github.com/Microsoft/LightGBM/issues/699

[7] Anna Veronika Dorogush, Andrey Gulin, Gleb Gusev, Nikita Kazeev, Liudmila Ostroumova Prokhorenkova, Aleksandr Vorobev. Fighting biases with dynamic boosting. arXiv:1706.09516