Zillow Prize: Zillow's Home Value Prediction (Zestimate)

1. Introduction and problem description

Zillow Prize, a competition with a one million dollar grand prize, is challenging the data science community to help push the accuracy of the Zestimate even further. Winning algorithms stand to impact the home values of 110M homes across the U.S.

In this million-dollar competition, participants will develop an algorithm that makes predictions about the future sale prices of homes. In the qualifying round, you'll be building a model to improve the Zestimate residual error.

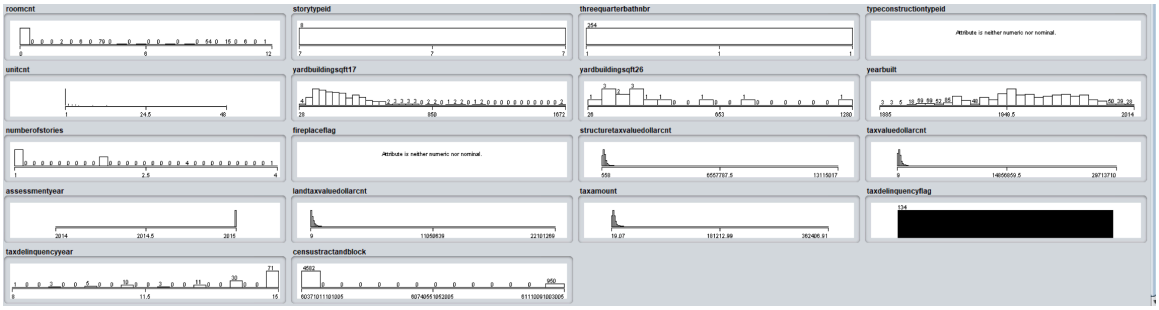
Submissions are evaluated on Mean Absolute Error between the predicted log error and the actual log error. The log error is defined as logerror = log (Zestimate) – log (SalePrice) and it is recorded in the transactions training data.

2. Related work and Dataset description

We participated the Zestimate competition, knowing the rules and background in detail and download original dataset.

Dataset URL: https://www.kaggle.com/c/zillow-prize-1/data

Dataset "properties_2016.csv" distribution using Weka:



Techniques of using: Anaconda, Spyder, Weka, Sublime Text

Experimental methodology: Regression

Coding language / technique to be used: Python / Regression

The dataset contains 4 csv documents:

The properties_2016.csv contains all the features corresponding to a house id. This document has 2985217 rows and 58 columns, such as house id, house square feet, whether there is a certain type of equipment, the location, etc.

The sample_submission.csv gives us a sample form to submit.

The train_2016_v2.csv shows the training data detail of each house id, which will be used to train the algorithm to form a model.

The Zillow_data_dictionary.xlsx illustrates the meaning of each attribute in properties_2016.csv to help us understand the dataset.

3. Pre-processing techniques

In order to make the prediction more accurate and efficient to run, we made the following analysis on the input datasets.

First, we explore the properties_2016 file to find out some features of it.

```
prop_df = pd.read_csv("../input/properties_2016.csv")
```

```
prop_df.shape
```

We get 2985217 rows and 58 columns of data.

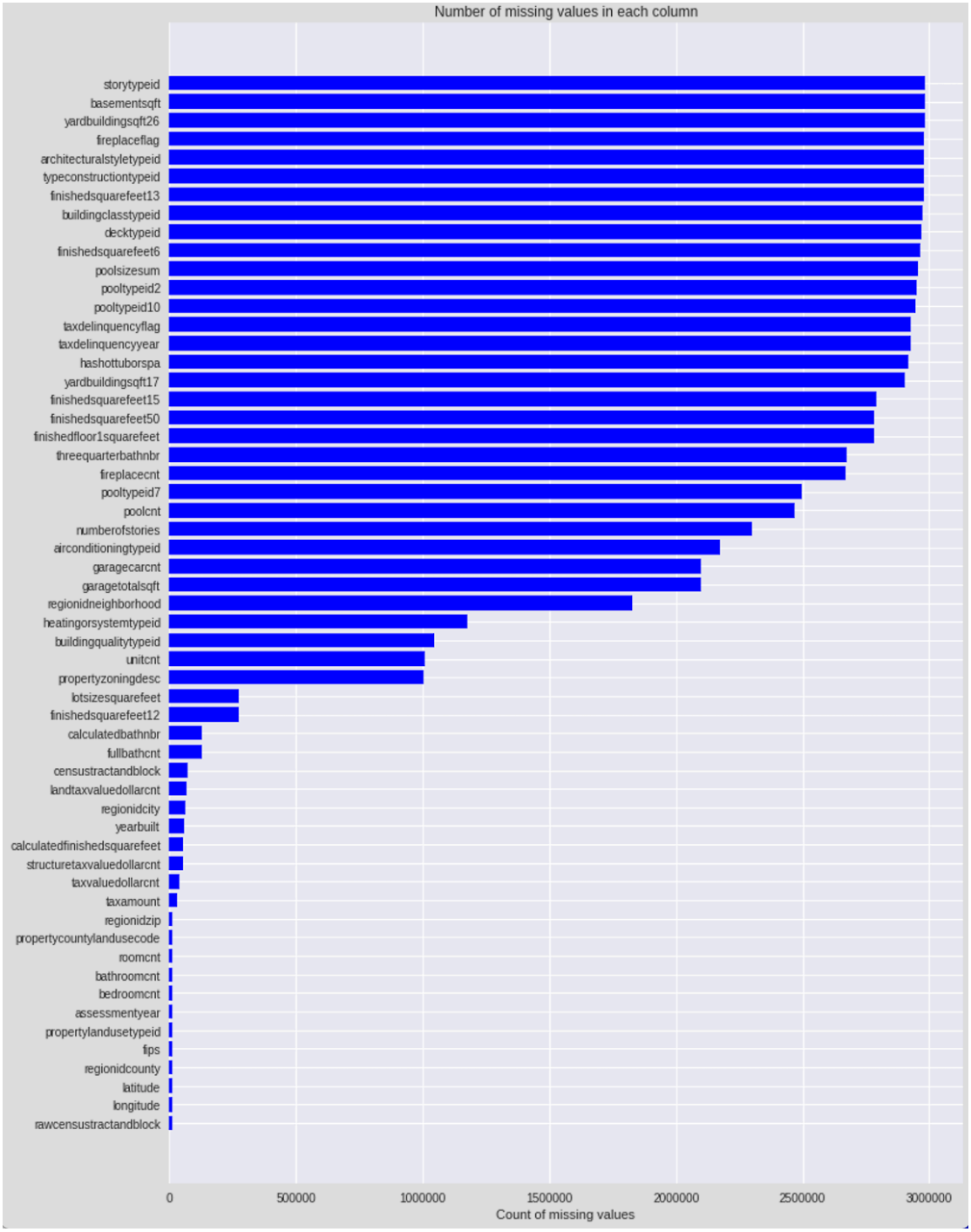
```
prop_df.head()
```

	parcelid	airconditioningtypeid	architecturalstyletypeid	basementsqft	bathroomcnt	bedroomcnt	buildingclasstypeid	buildingquali
0	10754147	NaN	NaN	NaN	0.0	0.0	NaN	NaN
1	10759547	NaN	NaN	NaN	0.0	0.0	NaN	NaN
2	10843547	NaN	NaN	NaN	0.0	0.0	NaN	NaN
3	10859147	NaN	NaN	NaN	0.0	0.0	3.0	7.0
4	10879947	NaN	NaN	NaN	0.0	0.0	4.0	NaN

We find that there are many NaN values in the dataset, so we do some

exploration on that one.

```
missing_df = prop_df.isnull().sum(axis=0).reset_index()
missing_df.columns = ['column_name', 'missing_count']
missing_df = missing_df.ix[missing_df['missing_count']>0]
missing_df = missing_df.sort_values(by='missing_count')
ind = np.arange(missing_df.shape[0])
width = 0.9
fig, ax = plt.subplots(figsize=(12,18))
rects = ax.barh(ind, missing_df.missing_count.values, color='bl
ue')
ax.set_yticks(ind)
ax.set_yticklabels(missing_df.column_name.values, rotation='hor
izontal')
ax.set_xlabel("Count of missing values")
ax.set_title("Number of missing values in each column")
plt.show()
```



We have about 90,811 rows in train but we have about 2,985,217 rows in properties file, so we merge the two files and then carry out our analysis.

```
train_df=pd.merge(train_df, prop_df, on='parcelid', how='left')
train_df.head()
```

	parcelid	logerror	transactiondate	transaction_month	airconditioningtypeid	architecturalstyletypeid	basementsqft	bathroomcnt
0	11016594	0.0276	2016-01-01	1	1.0	NaN	NaN	2.0
1	14366692	-0.1684	2016-01-01	1	NaN	NaN	NaN	3.5
2	12098116	-0.0040	2016-01-01	1	1.0	NaN	NaN	3.0
3	12643413	0.0218	2016-01-02	1	1.0	NaN	NaN	2.0
4	14432541	-0.0050	2016-01-02	1	NaN	NaN	NaN	2.5

We checked the dtypes of different types of variable

```
pd.options.display.max_rows = 65
dtype_df = train_df.dtypes.reset_index()
dtype_df.columns = ["Count", "Column Type"]
dtype_df
dtype_df.groupby("Column Type").aggregate('count').reset_index
()
```

	Column Type	Count
0	int64	2
1	float64	53
2	datetime64[ns]	1
3	object	5

Then we check the number of Nulls in this new merged dataset.

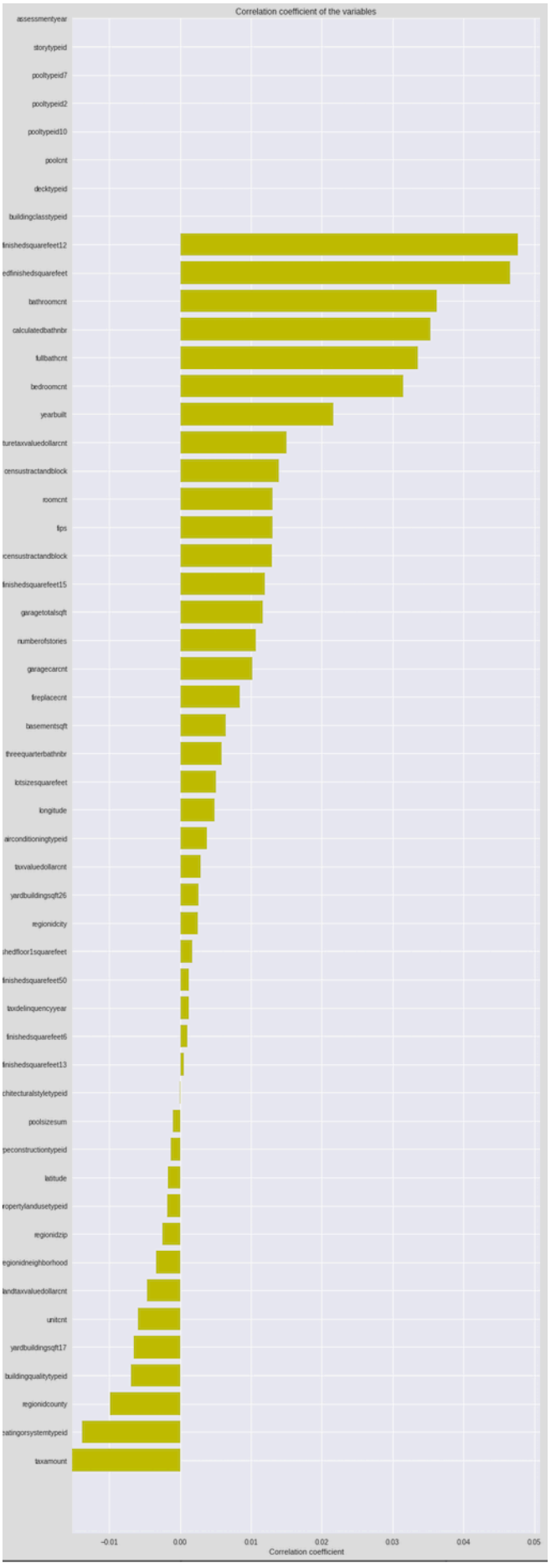
```
missing_df = train_df.isnull().sum(axis=0).reset_index()
missing_df.columns = ['column_name', 'missing_count']
missing_df['missing_ratio'] = missing_df['missing_count'] / tra
in_df.shape[0]
missing_df.ix[missing_df['missing_ratio']>0.999]
```

	column_name	missing_count	missing_ratio
6	basementsqft	90768	0.999526
9	buildingclasstypeid	90795	0.999824
16	finishedsquarefeet13	90778	0.999637
44	storytypeid	90768	0.999526

Several columns have missing values 99.9% of the times.

Since there are so many variables, we first take the 'float' variables alone and then get the correlation with the target variable to see how they are related.

```
mean_values = train_df.mean(axis=0)
train_df_new = train_df.fillna(mean_values, inplace=True)
x_cols = [col for col in train_df_new.columns if col not in ['lo
gerror'] if train_df_new[col].dtype=='float64']
labels = []
values = []
for col in x_cols:
    labels.append(col)
    values.append(np.corrcoef(train_df_new[col].values, train_d
f_new.logerror.values)[0,1])
corr_df = pd.DataFrame({'col_labels':labels, 'corr_values':valu
es})
corr_df = corr_df.sort_values(by='corr_values')
ind = np.arange(len(labels))
width = 0.9
fig, ax = plt.subplots(figsize=(12,40))
rects = ax.barh(ind, np.array(corr_df.corr_values.values), colo
r='y')
ax.set_yticks(ind)
ax.set_yticklabels(corr_df.col_labels.values, rotation='horizon
tal')
ax.set_xlabel("Correlation coefficient")
ax.set_title("Correlation coefficient of the variables")
plt.show()
```


Correlation coefficient of the variables
assessmentyear
storytypeid
pooltypeid7
pooltypeid2
pooltypeid10
poolcnt
decktypeid
buildingclasstypeid
finishedsquarefeet12
edfinishedsquarefeet
bathroomcnt
calculatedbathnbr
fullbathcnt
bedroomcnt
yearbuilt
turetaxvaluedollarcnt
censustractandblock
roomcnt
fips
censustractandblock
finishedsquarefeet15
garagetotalsqft
numberofstories
garagecarcnt
fireplacecnt
basementsqft
threequarterbathnbr
lotsizesquarefeet
longitude
airconditioningtypeid
taxvaluedollarcnt
yardbuildingsqft26
regionidcity
shedfloor1squarefeet
finishedsquarefeet50
taxdelinquencyyear
finishedsquarefeet6
finishedsquarefeet13
chitecturalstyletypeid
poolsizesum
peconstructiontypeid
latitude
ropertylandusetypeid
regionidzip
egionidneighborhood
landtaxvaluedollarcnt
unitcnt
yardbuildingsqft17
buildingqualitytypeid
regionidcounty
eatingorsystemtypeid
taxamount
-0.01
0.00
0.01
0.02
0.03
0.04
0.05
Correlation coefficient

The correlation of the target variable with the given set of variables are low overall.

There are few variables at the top of this graph without any correlation values. They have only one unique value and hence no correlation value. We confirm the same.

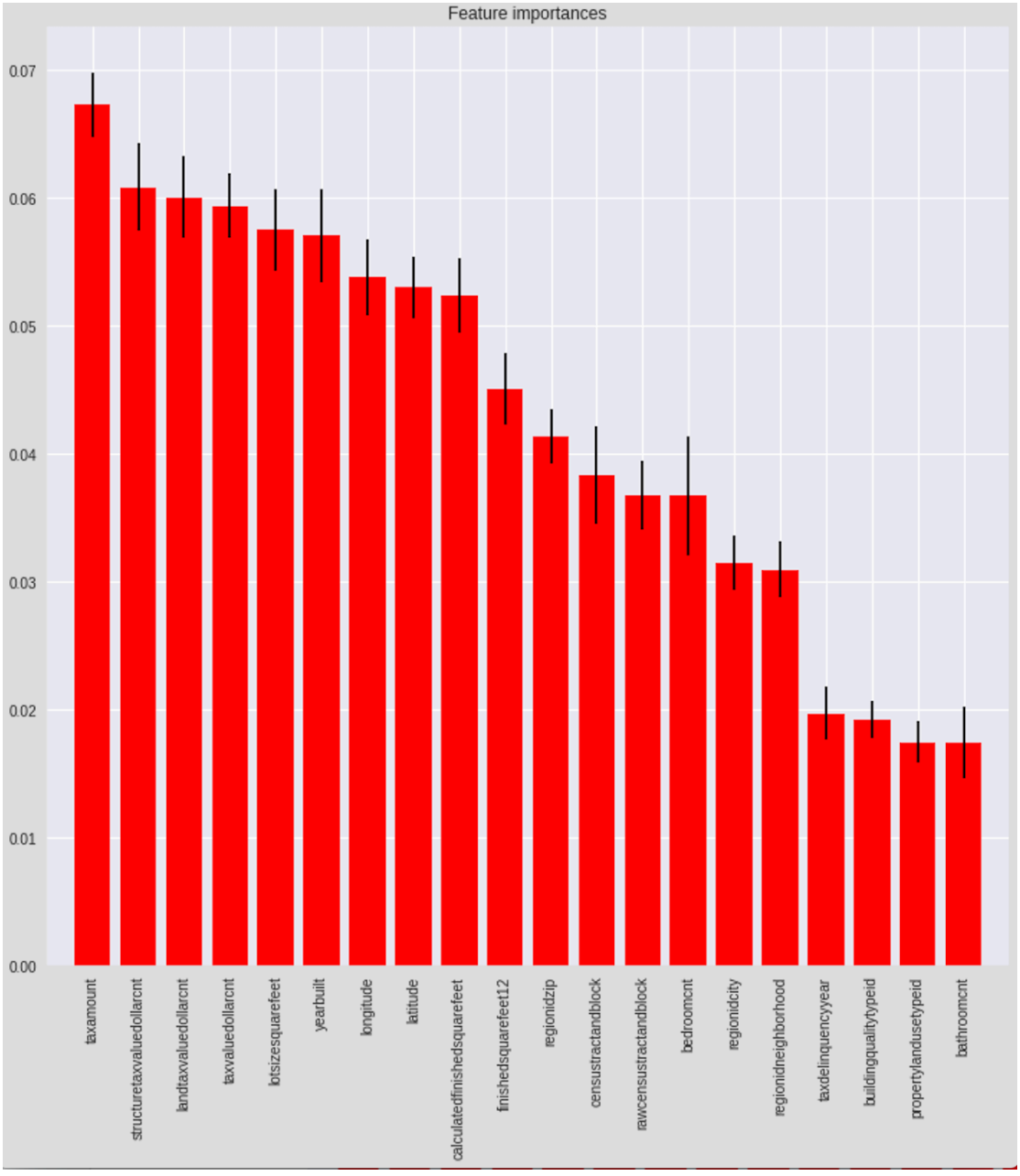
```
corr_zero_cols = ['assessmentyear', 'storytypeid', 'pooltypeid2
', 'pooltypeid7', 'pooltypeid10', 'poolcnt', 'decktypeid', 'bui
ldingclasstypeid']
for col in corr_zero_cols:
    print(col, len(train_df_new[col].unique()))
```

```
assessmentyear 1
storytypeid 1
pooltypeid2 1
pooltypeid7 1
pooltypeid10 1
poolcnt 1
decktypeid 1
buildingclasstypeid 1
```

We had an understanding of important variables from the univariate analysis. But this is on a stand alone basis and also we have linearity assumption. Now let us build a non-linear model to get the important variables by building Extra Trees model.

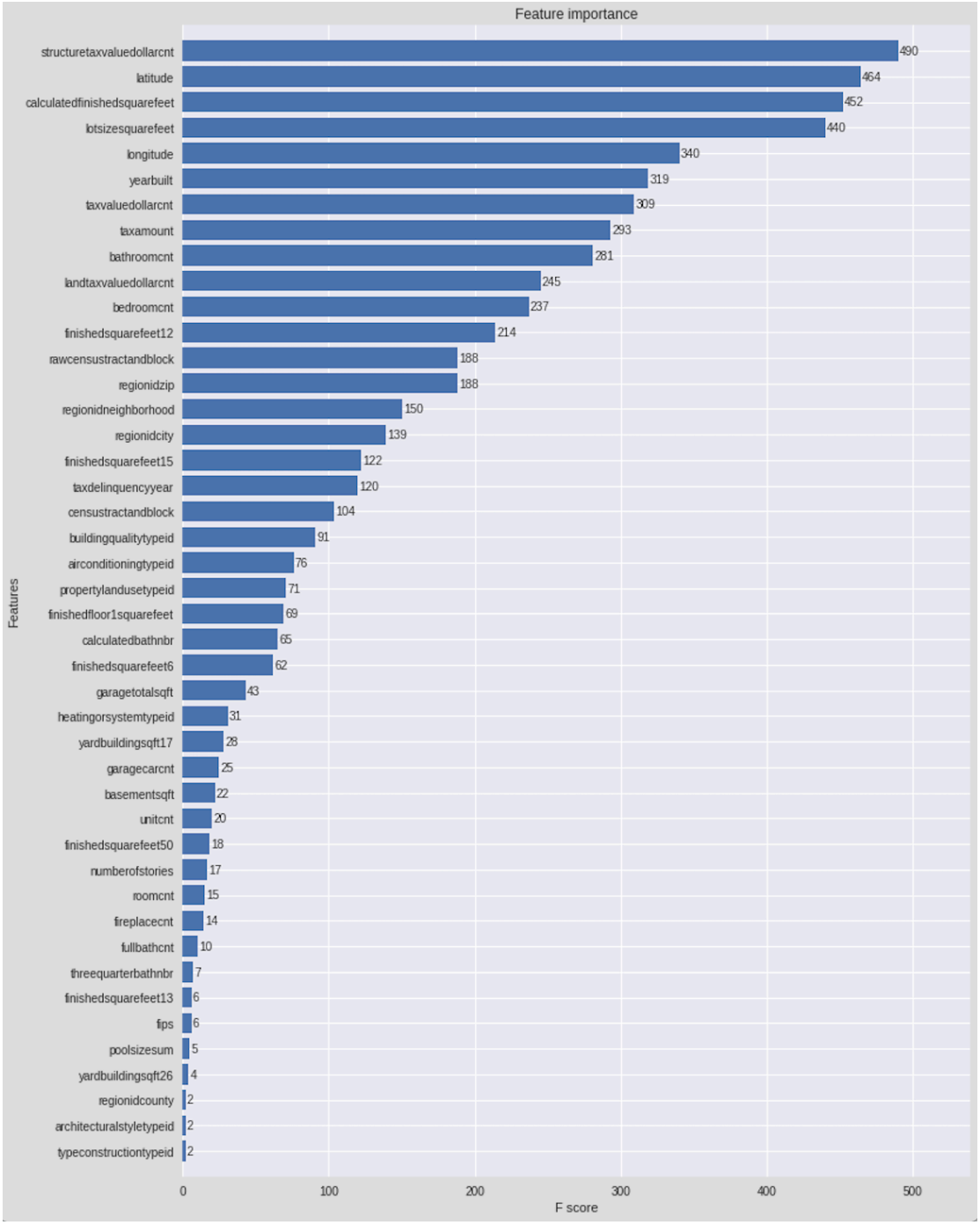
```
train_y = train_df['logerror'].values
cat_cols = ["hashottuborspa", "propertycountylandusecode", "pro
pertyzoningdesc", "fireplaceflag", "taxdelinquencyflag"]
train_df = train_df.drop(['parcelid', 'logerror', 'transactiond
ate', 'transaction_month']+cat_cols, axis=1)
feat_names = train_df.columns.values
from sklearn import ensemble
model = ensemble.ExtraTreesRegressor(n_estimators=25, max_depth
=30, max_features=0.3, n_jobs=-1, random_state=0)
model.fit(train_df, train_y)
importances = model.feature_importances_
std = np.std([tree.feature_importances_ for tree in model.estim
ators_], axis=0)
```

```
indices = np.argsort(importances)[::-1][:20]
plt.figure(figsize=(12,12))
plt.title("Feature importances")
plt.bar(range(len(indices)), importances[indices], color="r", y
err=std[indices], align="center")
plt.xticks(range(len(indices)), feat_names[indices], rotation='
vertical')
plt.xlim([-1, len(indices)])
plt.show()
```



Seems "tax amount" is the most important variable followed by "structure tax value dollar count" and "land tax value dollar count"

```
import xgboost as xgb
xgb_params = {
    'eta': 0.05,
    'max_depth': 8,
    'subsample': 0.7,
    'colsample_bytree': 0.7,
    'objective': 'reg:linear',
    'silent': 1,
    'seed' : 0
}
dtrain = xgb.DMatrix(train_df, train_y, feature_names=train_df.
columns.values)
model = xgb.train(dict(xgb_params, silent=0), dtrain, num_boost
_round=50)
fig, ax = plt.subplots(figsize=(12,18))
xgb.plot_importance(model, max_num_features=50, height=0.8, ax=
ax)
plt.show()
```



As shown in the above picture, we get the importance of each of the attribute in descending order. After discussion, we decided to remove features with F score under 50, and keep the rest of the features as the input data.

4. Proposed solution and methods

As we see before，the dataset given us has 58 features, each of them doesn't have too much relation with each other. Even though we pre-processing the data and throw some useless data away, it still hard to build model with regular regression model.

In this case, we use a very powerful algorithm: Gradient Boosting Algorithm and we use two very popular framework: XGBoost and LightGBM to deal with this project.

LightGBM based on decision tree algorithms, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise.

So when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms.

XGBoost implements parallel processing and is blazingly faster as compared to GBM.XGBoost allows user to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run. They both run very fast, LightGBM runs even faster than XGBoost.

Although it is easy to build model with these two algorithms, the tuning of parameter and improve the accuracy is the hard, as them have so many parameters.

We try to set the parameter such as max_depth, min_leaves, etc set to the value which works well from experience (not default). The learning parameter controls the magnitude of this change in the estimates. Lower values are

generally preferred as they make the model robust to the specific characteristics of tree and thus allowing it to generalize well. We set the learning rate to 0.002.

5. Experimental results and analysis

Xgboost version1:

params['eta'] = 0.02

params['objective'] = 'reg:linear'

params['eval_metric'] = 'mae'

params['max_depth'] = 4

params['silent'] = 1

Xgboost version2:

params['eta'] = 0.02

params['objective'] = 'reg:linear'

params['eval_metric'] = 'mae'

params['max_depth'] = 5

params['silent'] = 1

Xgboost version3:

params['eta'] = 0.02

params['objective'] = 'reg:linear'

params['eval_metric'] = 'mae'

params['max_depth'] = 4

params['silent'] = 2

Name	Submitted	Wait time	Execution time	Score
prediction.csv	a few seconds ago	19 seconds	21 seconds	0.0652777

LightGBM version1:

params['learning_rate'] = 0.002

params['boosting_type'] = 'gbdt'

params['objective'] = 'regression'

params['metric'] = 'mae'

params['sub_feature'] = 0.5

params['num_leaves'] = 30

params['min_data'] = 500

params['min_hessian'] = 1

Name	Submitted	Wait time	Execution time	Score
lgb_starter.csv	2 hours ago	16 seconds	22 seconds	0.0646697

LightGBM version2：Number of leaves set to 30(which is smaller than 2^Max_depth) to avoid overfitting.

params['learning_rate'] = 0.002

params['boosting_type'] = 'gbdt'

params['objective'] = 'regression'

params['metric'] = 'mae'

params['sub_feature'] = 0.5

params['num_leaves'] = 30

params['min_data'] = 500

params['min_hessian'] = 1

Name	Submitted	Wait time	Execution time	Score
lgb_starter.csv	2 minutes ago	21 seconds	22 seconds	0.0646927

LightGBM version3：To avoid overfitting, we set min_data to 1000, and set the learning rate to 0.001.

```
params['learning_rate'] = 0.001
params['boosting_type'] = 'gbdt'
params['objective'] = 'regression'
params['metric'] = 'mae'
params['sub_feature'] = 0.5
params['num_leaves'] = 60
params['min_data'] = 1000
params['min_hessian'] = 1
```

Name	Submitted	Wait time	Execution time	Score
lgb_starter2.csv	a minute ago	15 seconds	22 seconds	0.0548080

6. Conclusion

Through tuning of parameter, we find learning_rate(eta for tree booster) need to be set a low level like 0.02 to help improving the accuracy. And also it not a good idea to set the max_depth less than 5.

Overall, using LightGBM get a better score than using XGBoost， and LightGBM perform faster than XGBoost. But overall, the lightGBM is more likely have the problem of overfitting as the numbers in the leaves and learning rate is settled in a low level.

7. Contribution of team members

Team member Netid: lxq160030, hxl164530, fxz160630

Introduction and problem description, Related work parts are contributed by lxq160030 and hxl164530

Dataset description, Pre-processing techniques parts are contributed by lxq160030 and hxl164530.

Proposed solution and methods part are contributed by fxz160630 and lxq160030.

Experimental results and analysis, Conclusion parts are contributed by fxz160630 and hxl164530.

8. Reference

http://xgboost.readthedocs.io/en/latest/python/python_api.html
http://xgboost.readthedocs.io/en/latest/get_started/
https://github.com/Microsoft/LightGBM
https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/