Housing Price Prediction Model: Stacking Approach

Shuto Araki, Taras Tataryn

Department of Computer Science, DePauw University
Greencastle, IN 46135, U.S.A.

shutoaraki_2020@depauw.edu
tarastataryn_2018@depauw.edu

Abstract

This paper presents on the general steps and process taken to predict the price of residential housings in Ames, Iowa. Preprocessing of the data involved removing/filling missing data, feature engineering and detecting outliers. On top of preprocessing, several predictive models were explored including Gradient Boosting Regressor, Random Forest Regressor, Lasso Regression, etc. The best result was obtained by using the ensemble technique called stacking, which integrates different predictions from different models.
Keywords: Feature Engineering, Housing Price, Ensemble Technique, Stacking Regressor, Gradient Boosting

1 INTRODUCTION

This paper is based on a project conducted for our Data Mining course and a Kaggle.com Competition titled *"House Price: Advanced Regression Techniques."* It utilizes the *Ames Housing Dataset* compiled by Dean de Cock, a more modernized and expanded version of the Boston Housing Dataset. The dataset contains 79 explanatory variables describing essentially every aspect/attribute of residential homes in Ames, Iowa [2]. Overall, our goal is to as least erroneous as possible, predict the final price of each residential home listed in the dataset. The steps taken are described in depth in the following sections, but prior to doing anything with our model, we first found it prudent to look at the variables in the dataset to better understand their meaning and relevance to the dataset and sale price.

We started by fully reading the description of the project as well as a description of each housing attribute as provided in the "data_description.txt" file. Next, in order to visualize and locate the variables we initially expect to perform well, we created a spreadsheet containing the following columns: the variable name, variable type (nominal, ordinal, ratio, interval), building segment, expectation, and a overall conclusion/comment on that variable. The building segment gives us further insight into what the variable is describing, with 3 potential options including building (physical characteristics), space (space properties – eg. TotalBsmtSF) or location (i.e. the neighborhood's proximity to the city). Furthermore, the expectation column will allow us to predetermine what variables we think are either highly correlated with the sale price or, generally speaking, if we think the variable has a high importance on the price of a house. Variables that immediately jumped out to us included GrLivArea, OverallQual and YearBuilt. More in-depth analysis ensued.

In conjunction with our own opinion to the "Expectation" of how important these variables are, we decided to also look at the attributes' correlation with each other, and also to the dependent variable, sale price. This was accomplished via *Seaborn's* correlation matrix function, as used by Pedro Marcelino in his Kaggle Kernel [7], which is displayed in the style of a heat map. Immediate findings were that several variables were highly correlated with each other, in addition to several attributes being highly correlated with sale price. We could further weed out which variables were the most highly correlated with sale price by limiting the number of variables to include in the heatmap to 10 (accomplished by sorting by the 10 largest to the variable SalePrice) as seen in Figure 1. We gained further insight into correlation between variables by looking at the "zoomed heatmap." For instance, GarageCars and GarageArea are very highly correlated, meaning they are accomplishing nearly the same thing. For this reason, we dropped the GarageArea variable to prohibit one of those variables from messing with our regression.

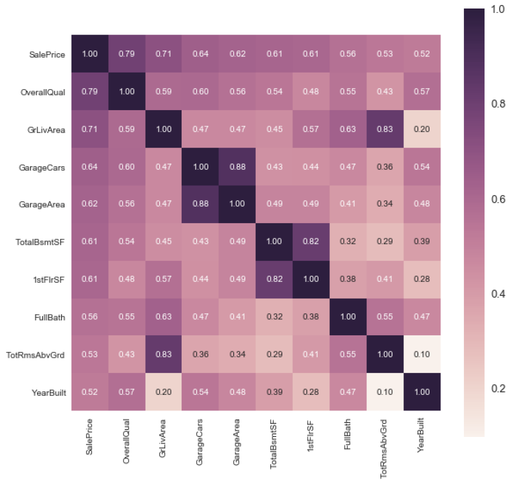


Figure 1: Zoomed Heatmap: Ten most correlated Values

Other variables we labeled as "High" in terms of importance include among others: MSZoning, LotArea, Neighborhood (how close it is to the city), Overall Qual (contains a .79 correlation to Sale Price and is inherently important), and FullBath (a surprising one). With all of this information and analysis, we have a good basis and knowledge of our data, and are ready to begin to preprocess the data to prepare for modeling and testing.

2 PREPROCESSING

The most important component prior to testing our data is transforming or "preprocessing" of the data. In general, raw datasets are "dirty," meaning they are incomplete (missing data, lack of attributes, etc.), contain errors or outliers, and inconsistent data (can have discrepancies in names or type of value). This is definitely relevant in the housing dataset, as several columns contain missing values, irrelevant data, and outliers. So overall, if there isn't quality data there won't be quality predictive analysis. The following sections contain the different steps we took to pre-process the data.

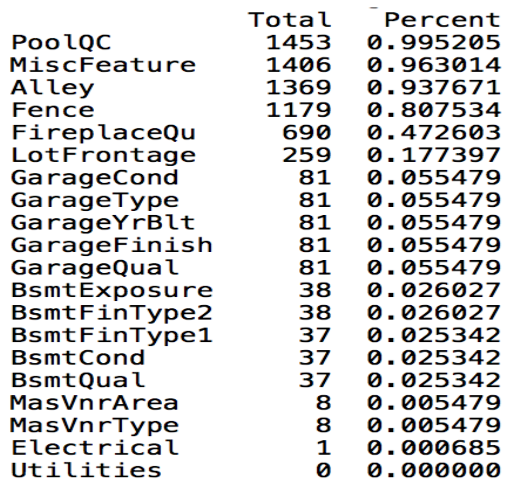
2.1 Missing Data

Removing or altering missing data is an important first step to take in preprocessing of our data. However, we have to tread lightly with this, as removing too much data will drastically reduce the sample size and could hurt our predictions. To first visualize what kinds of variables and how much data is actually missing, we printed out the top 20 variables containing the most missing values. The results weren't totally surprising, but a couple of variables had a lot of missing values. For instance, PoolQC (pool quality), MiscFeature (other features such as elevator, 2nd garage, etc.), and Alley (type of alley access) all contain more than 90% of missing values. Considering that barely .05% of PoolQC's data is existent, we found it fair to drop this variable with minimal effect on the resulting predictions. Additionally, the MiscFeature only has about 4% of its values in the dataset, so we also decided it would be prudent to drop this variable. This isn't to mention that the irrelevance of whether a house contains an elevator or 2nd garage in its sale price (in our opinion). Finally, we removed the Alley variable as it also contains a large amount of missing data.

For other variables with some missing values, the scikit-learn module *Imputer* was utilized to fill in the missing values. By default, the *Imputer* uses a mean value to approximate the missing parts of the data. We can change this strategy to median or mode as well, which will be explored in experiments later.

2.2 Feature Engineering

Feature Engineering is a process in which new attributes are revised. In this project, three new attributes were created: YearsOld, YardSize, and combinedSF. Additionally, label-encoding is utilized to encode categorical values and



```
                Total   Percent
PoolQC           1453  0.995205
MiscFeature      1406  0.963014
Alley            1369  0.937671
Fence            1179  0.807534
FireplaceQu       690  0.472603
LotFrontage       259  0.177397
GarageCond         81  0.055479
GarageType         81  0.055479
GarageYrBlt        81  0.055479
GarageFinish       81  0.055479
GarageQual         81  0.055479
BsmtExposure       38  0.026027
BsmtFinType2       38  0.026027
BsmtFinType1       37  0.025342
BsmtCond           37  0.025342
BsmtQual           37  0.025342
MasVnrArea          8  0.005479
MasVnrType          8  0.005479
Electrical          1  0.000685
Utilities           0  0.000000
```

Figure 2: Top 20 Variables with Most Missing Values

normalize the attribute labels. Finally, one-hot encoding is used to further filter the dataset.

YearsOld attribute is created by subtracting YearsBuilt value by 2008 because the data was collected between 2006 and 2010. YardSize attribute was calculated by using two other variables, LotArea and GrLivArea. Subtracting GrLivArea by LotArea approximates the area of yard in each house. CombinedSF is simply combining the 1st and 2nd floor square foot attributes into one. Similarly, there are infinite number of possibilities in devising new attributes that better explain the housing prices. However, doing so for a plethora of variables would add unnecessary dimensionality to our dataset and would confound our results.

Moreover, label encoding was included on some of our attributes to get better analysis. For instance, the *KitchenQual* variable has five labels, ranging from Excellent to Typical/Average to Poor. However, when we use it in our analysis there isn't a way to know if Excellent is better than Typical/Average and vice-versa. Alongside the *KitchenQual* attributes we decided to keep it consistent and also label encode the attributes which have similar values. This includes *GarageQual, BsmtQual, BsmtCond, ExterQual, GarageCond, OverallCond, FireplaceQu, ExterCond, HeatingQC, BsmtFinType1, and BsmtFinType2*. Thus, label encoding gives specific meaning to these orders and we don't need to introduce any new variables as with one-hot encoding to avoid dimensionality.

Using the `get_dummies` function from `pandas` module, it allows us to one-hot encode attributes. While it may add dimensionality to our dataset, it has an advantage in that is more suitable for machine-learning as labels are independent to each other. With one-hot encoding, essentially a new binary variable is introduced for each unique integer value. We attempted to use Principal Component Analysis in order to reduce dimensionality in data, but it did not improve the overall performance and rather deteriorated the outcome.

2.3 Outliers

Documentation of the Ames Housing Dataset puts forth that outliers do exist, specifically when looking at a plot of GrLivArea and Sale Price. Creating a subplot of those two quickly confirmed this; there were two data points that were very far separated from the others. These two were points were nearly 2000 square feet more than its nearest neighbor as you can see from Figure 3, and definitely would hurt our prediction. Thus, these two points were identified by their unique IDs and subsequently dropped.

Also, the two points with their sale price being above $700,000 could also be considered outliers. Even though they follow the general trend, as suggested by Dean De Cock himself, these two points should also be dropped.

Another point that should be dropped is the one in Figure 4: the house that is sold as about $470,000 with over 110 years old. This house is clearly an outlier and therefore dropped.

We also examined the attribute *LotArea* when considering outliers. We went through the same process as above and found 4 specific points that were clearly outliers with lot sizes well over 100,000 square feet. These 4 points were very distant from any nearest neighbor, and we came to the conclusion that these would only hurt our predictions. Thus, we found the IDs of these four points and subsequently dropped them. The same process was repeated for similar attributes including BedroomAbvGr, KitchenAbvGr, totalBsmtSF, and LotFrontage which we thought were significant outliers. However, excluding these 4 houses from the dataset decreased the overall score at the end. Thus, we decided to include these four dataset.

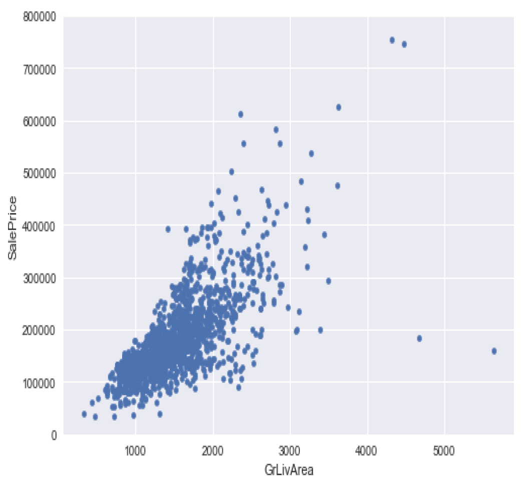


Figure 3: Subplot of GrLiv Area and Sale Price

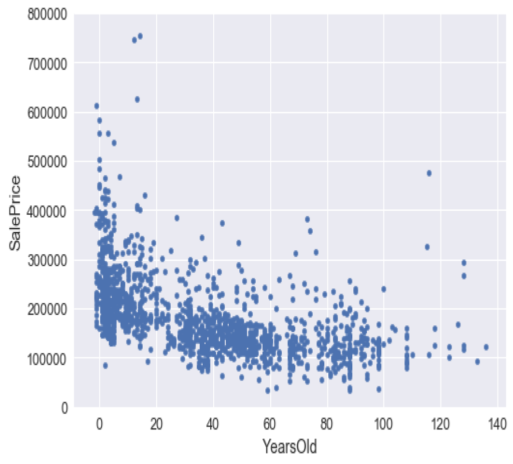


Figure 4: Subplot of YearsOld and Sale Price

3 ALGORITHMS

The performance of each model is measured by Root-Mean-Squared-Error (RMSE) with the y variable in log form as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i}^{n}(\log(h(\vec{x}^{(i)})+1) - \log(y^{(i)}+1))^2}$$

where $h(\vec{x}^{(i)})$ and $y^{(i)}$ denotes the predicted and actual price of the i^{th} house, respectively.

The competition organizer uses this metric to evaluate and rank the users' models. The cross-validation is done with 10 folds. All the presented data are preprocessed as mentioned above.

For any algorithm, the `random_state` variable is set to 0 in order to eliminate the random element in evaluating algorithm performance. All the algorithms except for Extreme Gradient Boosting Regressor (XGBoost), Light Gradient Boosting Regressor (LGB), and Stacking Regressor are implemented by scikit-learn module. We utilized the 44 CPU Virtual Machine with 120GB of memory provided by XSEDE allocation TG-CIE180012 on some of the parameter tuning algorithms.

3.1 Gradient Boosting Regressor

The Gradient Boosting Regressor makes use of ensemble techniques to combine different decision trees to minimize the error of the model prediction. It combines a set of weak learners to attain better prediction. With the default parameter setting, the RMSE was about 0.12076 on local validation and 0.13769 on Kaggle leaderboard.

3.1.1 Parameter Tuning

Using the aforementioned XSEDE cloud computing, the optimal parameters were calculated through GridSearchCV in sklearn. Within the given range, GridSearchCV exhaustively tries every combination of parameters, comparing the cross validation scores to find the best parameter combination. Thus, there is a trade-off between best tuning and computing time. It is sometimes optimal to tune one or a few parameters at a time to reduce the computing time.

GradientBoostingRegressor Parameters	
`n_estimators`	`1000`
`learning_rate`	`0.01`
`min_samples_split`	`17`
`min_samples_leaf`	`7`
`max_depth`	`5`
`max_features`	`'sqrt'`
`subsample`	`0.8`

According to the official scikit-learn documentation[8], the parameter `n_estimator` represents the number of boosting stages that is set as 100 by default. The higher the value, the better the model performs[3]. Just like a smaller `learning_rate` takes more time to compute with better performance, the larger `n_estimator` costs higher computing power. Thus, these two parameters are tuned depending on how much computing power and time are available. We chose a relatively high `n_estimator` and low `learning_rate` because we could afford large computational power.

The `min_samples_split` parameter ”defines the number of observations which are required in a node to be considered for splitting” [3]. A higher value leads to under-fitting and there is no intuitive way to choose the value. Thus, we used GridSearchCV to optimize this parameter and obtained 17 as a result. Meanwhile, the `min_samples_leaf` parameter specifies the number of samples to be required at the leaf node, guaranteeing the minimum number of samples. Like the `min_samples_split`, the `min_samples_leaf` parameter is used to control over-fitting. GridSearchCV was also used to optimize this parameter, and 7 was obtained as a result.

The `max_depth` parameter, quite simply, specifies the maximum depth of a tree. Having a model with high-depth could confound results, as it would allow for for it to learn relations that are specific to an individual sample. Like the max depth parameter, the `max_features` parameter intuitively selects the number of features, done randomly, when searching for a best split. As Aarshay Jain describes in his article, higher values in this can lead to over-fitting, but it also depends on a case to case basis [3].

Finally, the `subsample` parameter will select the fraction of observations that are chosen for each tree and fitting the individual base models. While, a value of .8 means that there may be a small increase in bias, this parameter has a trade-off in that a larger value would lead to an increase in variance. GridSearchCV found that .8 was our sweet spot.

The following algorithms were all tuned in a similar manner using GridSearchCV.

3.2 Extreme Gradient Boosting Regressor

While the Extreme Gradient Boosting Regressor (XGBoost) follows the same principles in gradient boosting as Gradient Boosting, there are some key differences. For one, the XGBoost uses a more regularized model formalization that controls for over-fitting. Additionally, XGBoost implements parallel processing, meaning it is considerably faster than the Gradient Boosting model. Other advantages of XGBoost as mentioned by Aarshay Jain in his article include the in-house ability to handle missing values, higher flexibility (customized objectives and evaluation criteria), tree pruning, built-in cross validation, and the ability to continue on an existing model [4].

3.3 Light Gradient Boosting Regressor

Light Gradient Boosting is a relatively new gradient boosting framework started by Microsoft. Like XGBoost, it too is based on decision tree algorithm, however, the main feature that sets it apart is that it splits the leaf-wise rather than complete level or depth wise (all leaves) as seen in Figure 5. Leaf-wise splitting can be efficient, however can also lead to increased complexity and overfitting. However, as previously explained, this can be controlled by using the max_depth parameter just as it is used in XGBoost.

Some advantages of LGB as provided by Pranjal Khandewal, include: faster training speed (histogram-based algorithm which increases efficiency and quickens training procedure), lower memory usage, better accuracy (because of the leaf-wise approach involving more complex trees), compatibility with larger datasets, and finally, parallel learning is also supported.[6]

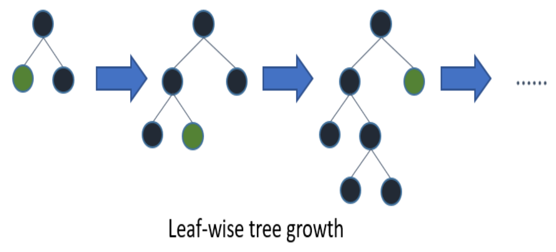


Figure 5: Leaf-wise tree growth in LGB from [6]

LGB was implemented using `lightgbm` module's object `LGBMRegressor`

3.4 Random Forest Regressor

The Random Forest Regressor is a type of model that makes predictions by combining decisions from a sequence of different base models. While the technique of ensembling uses multiple different models for better predictions and performance, in random forests, the base models are constructed independently using a different subsample of the data [10]. Simplified, the model essentially works as follows: each tree gives a classification and the tree votes for that particular class. Then, based on most votes, it chooses that classification and takes the average of all the outputs by different trees.

Random Forest is known to be a good model because it can handle larger datasets (i.e. with greater dimensionality). Along with this, it also can output feature importances. Additionally, like the gradient boosting models, it can also missing and unlabeled data. The main issue with the Random Forest model, is that it we unfortunately do not have much control over what the model does. For this reason, along with poor performance when testing the model and tuning parameters, we decided to exclude it from our algorithm with stacking.

3.5 Lasso Regressor

LASSO, which stands for Least Absolute Shrinkage and Selector Operator, is an algorithm that performs both variable selection and regularization. It is essentially an adjusted version of Linear Regression. Specifically, it performs *L1 Regularization*, which in short essentially "adds a factor of sum of absolute value of coefficient in the optimization objective." [5] In other words, it penalizes the absolute size of the regression coefficients which shrinks the parameter estimates towards zero. The regularization prevents the model from overfitting the training dataset. As the complexity of the model increases, the magnitude of coefficients increase to catch gory details, which are often times just the noise, in the data. The cost function is as follows:

$$J(\vec{\theta}) = \frac{1}{2m}\sum_{i=0}^{m}(\vec{\theta}\cdot x^{(i)} - y^{(i)})^2 + \underbrace{\alpha\sum_{j=0}^{n}|\theta_j|}_{\textit{L1 Regularizer}}$$

The first term is the Sum of Squared Error, which is used in Linear Regression. The second term is the L1 Regularization term. The algorithm tries to minimize this cost function value by changing the coefficients $\vec{\theta}$.

Furthermore, a crucial parameter, `alpha`, controls the scaler of the L1 regularization, or the amount of shrinkage. We experimented with different values of `alpha`, however came to the same conclusion as documentation would say; when Alpha is zero, the estimate would be similar to what linear regression results would yield. As the `alpha` increases, more and more coefficients are eliminated or set to zero, however if the value is too high it will increase variance and bias. [5] The sweet spot we found through GridSearchCV was a value of 0.00009, which yielded a better RMSE as a result.

3.6 Ridge Regressor

Ridge Regression is very similar to Lasso Regression. The only difference is the regularization term in its cost function:

$$J(\vec{\theta}) = \frac{1}{2m}\sum_{i=0}^{m}(\vec{\theta}\cdot x^{(i)} - y^{(i)})^2 + \underbrace{\alpha\sum_{j=0}^{n}|\theta_j|^2}_{\textit{L2 Regularizer}}$$

It penalizes the squared value of the regression coefficients. This technique is called *L2 Regularization*. Since the coefficients are tend to be smaller than 1, their squared values become even smaller, which leads to smaller penalty than L1 Regularization. Thus, the optimal `alpha` value is often far larger than that of Lasso Regression. Using GridSearchCV, the optimal `alpha` turned out to be 12.

3.7 Weighted Averaged Models

This simple approach averages all the base models that we select for our algorithm. This is done by creating a new class called `AveragingModels` that takes an array of models and a list of weights of the same length. Each model produce its predictions and this model aggregates those predictions by calculating the weighted average. It is a simple stacking approach, but was very effective and yielded some of our better results. The optimal weight was calculated through GridSearchCV:

$$0.24\texttt{GBR} + 0.07\texttt{XGB} + 0.14\texttt{LGB} + 0.46\texttt{Lasso} + 0.09\texttt{Ridge}$$

Random Forest Regressor was excluded because it did not perform well.

3.8 Stacking Regressor

Stacking regressor is a more complicated approach that, like `AveragingModels`, combines with several different models to form a better prediction. Stacking is equivalent to building a successful human team or company; every team member has their own niche and specialized skills and will make their own contributions. Moreover, their individual weaknesses and biases will be offset by the strengths of other members in the team. [1]

In short, in a stacking approach the predictions generated by our base models are used as inputs for a second layer learning algorithm. The second layer algorithm is further trained to optimally combine the predictions previously made to form a higher-level learning algorithm utilized called the meta-model which is depicted in 6. We found that using LGB and Lasso Regression as base models and Ridge Regression as a meta model yielded the best result on the Kaggle competition leaderboard. Thus, in this case, the training dataset was split into two, one for LGB and the other for Lasso Regression. These two parts of prediction data were fed to the meta model, Ridge Regression, and it produces the final prediction.

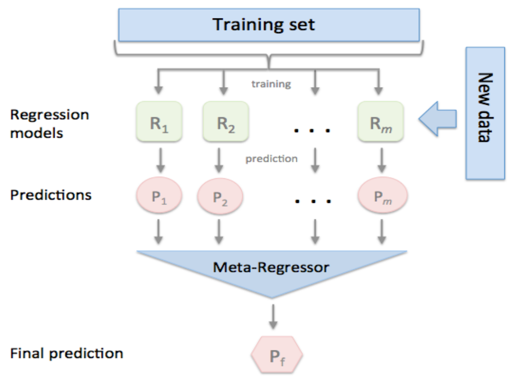


Figure 6: Stacking Regression Overview from [9]

This algorithm was implemented by using mlxtend module's `StackingRegressor` object.

4 RESULTS

Our best score on Kaggle was 0.12175 and ranked as 1019th in the leaderboard among 5343 (and counting) users and teams that have submitted their predictions to this competition. It is about top 19% score in the leaderboard.

Algorithms	Local RMSE	Kaggle RMSE
GBR	0.11226	0.13337
XGB	0.11413	0.13294
LGB	0.11222	0.12211
Random Forest	0.16025	0.18778
Lasso	0.10848	0.15546
Ridge	0.11342	0.15492
Weighted Average	0.10535	0.12981
Stacking	0.11308	**0.12175**

As seen in the table above, it is interesting to see that a great performer like LGB performs even better by stacking with not-so-great Lasso Regressor and Ridge Regressor.

The weighted average of several regressions might have overfitted the training dataset because it yielded the best score locally but did not perform best on the testing dataset.

5 FUTURE WORK

More sophisticated algorithms such as Elastic Net Regression and Deep Neural Network can be explored as our time and knowledge were very limited. Additionally other feature engineering techniques and pre-processing might be able to improve our score. Finally, more `Imputer` strategies can also be explored. Instead of just using mean, other statistical metrics such as median and mode could be utilized.

6 CONCLUSION

This paper explores and demonstrates our different techniques used and the process involved in predicting housing prices for the Ames Housing Dataset. The dataset provided has several attributes that contained missing values, erroneous data, and some irrelevant data. Thus, we took multiple different approaches and steps in cleaning the data. For instance, feature engineering, removing missing data, dropping outliers, etc. were all key elements of our pre-processing and helped to improve our predictions. Additionally, we experimented with numerous different algorithms, including Gradient Boosting Regressor, LightGBM, XGBoost, Lasso, Random Forest, K-Nearest Neighbor, and Ridge Regression. After numerous different tests, we eventually found that that using the stacking approach with the LightGBM, Lasso, and Ridge as our meta-model produced our best performance. The results highlight the effectiveness of utilizing the stacking approach, however, our worse results also show that it's not as simple as throwing in every good model into the algorithm.

References

[1] Leo Breiman. Stacked regressions. *Machine Learning*, 24(1):49–64, Jul 1996.

[2] Dean. De Cock. Ames, iowa: Alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3), 2011.

[3] Aarshay. Jain. Complete guide to parameter tuning in gradient boosting (gbm) in python. "`https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/`", 2016.

[4] Aarshay. Jain. Complete guide to parameter tuning in xgboost (with codes in python). `https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/`", 2016.

[5] Aarshay. Jain. A complete tutorial on ridge and lasso regression in python. "`https://www.analyticsvidhya.com/blog/2016/01/complete-tutorial-ridge-lasso-regression-python/`", 2016.

[6] Pranjal. Khandewal. Which algorithm takes the crown: Light gbm vs xgboost? "`https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/`", 2017.

[7] Pedro. Marcelino. Comprehensive data exploration with python. "`https://www.kaggle.com/pmarcelino/comprehensive-data-exploration-with-python`", 2017. Accessed: 2018-04-16.

[8] White Scott. Louppe Gilles. Olivetti Emanuele. Joly Arnaud. Prettenhofer, Peter. and Jacob. Schreiber. 3.2.4.3.6. sklearn.ensemble.gradientboostingregressor. "`http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html`", 2017. Accessed: 2018-04-16.

[9] Sebastian Raschka. Stackingregressor. "`https://rasbt.github.io/mlxtend/user_guide/regressor/StackingRegressor/`", 2018.

[10] Turi.com. Random forest regression. "`https://turi.com/learn/userguide/supervised-learning/random_forest_regression.html`", 2018.