Machine Learning: Predicting House Prices in Ames, IA

Philippe Heitzmann

Posted on Sep 24, 2020

2 Shares Share Tweet Share

Link to Github

The objective of this third project was to build machine learning models to predict house prices as part of the Kaggle competition *House Prices: Advanced Regression Techniques*. The dataset used in this competition was the Ames Housing dataset,

View Posts by Categories

ALL POSTS	1960 posts
ALUMNI	55 posts
APIS	38 posts
AWS	12 posts
BIG DATA	42 posts
CAPSTONE	149 posts
CAREER EDUCATION	4 posts
COMMUNITY	69 posts
DATA SCIENCE NEWS AND S	
DATA VISUALIZATION	21 posts

which was compiled by Professor Dean De Cock of Truman State University and which describes the sale of residential properties in Ames, Iowa during the period 2006 - 2010. The dataset contains a total of 2919 observations split across 80 different explanatory variables (23 nominal, 23 ordinal, 14 discrete, and 20 continuous) that impact home values, captured in the *SalePrice* variable in this dataset. The data is split equally between a training dataset containing 1460 observations and a testing dataset containing 1459 observations. The steps taken to create a highly accurate model were as follows:

1. Data exploration
2. Data cleaning
3. Feature engineering
4. Feature selection on level 0 models
5. Hyper-parameter tuning on level 0 models
6. Stacking the level 0 models into a level 1 model

Data Exploration

The first step in this analysis was checking for outliers in the dataset that could potentially skew our model predictions. Given that the *GrLivArea* variable, which captures above grade (ground) living area in square feet, exhibited a 0.71 positive correlation with *SalePrice* in the training dataset, this gave me a high degree of confidence that it would offer strong predictive power later on and would by extension provide a good screen for potential outliers. Plotting the *SalePrice* dependent variable vs *GrLivArea* in our training dataset revealed two seeming outliers exceeding 4,500 sqft in total area:

EVENTS	3 posts
FEATURED	37 posts
HADOOP	13 posts
MACHINE LEARNING	255 posts
MEETUP	102 posts
PYTHON	280 posts
R	347 posts
R SHINY	494 posts
R VISUALIZATION	390 posts
SPARK	18 posts
STUDENT WORKS	1419 posts
TABLEAU	9 posts
TENSORFLOW	1 posts
WEB SCRAPING	441 posts

Search For

Our Recent Popular Posts

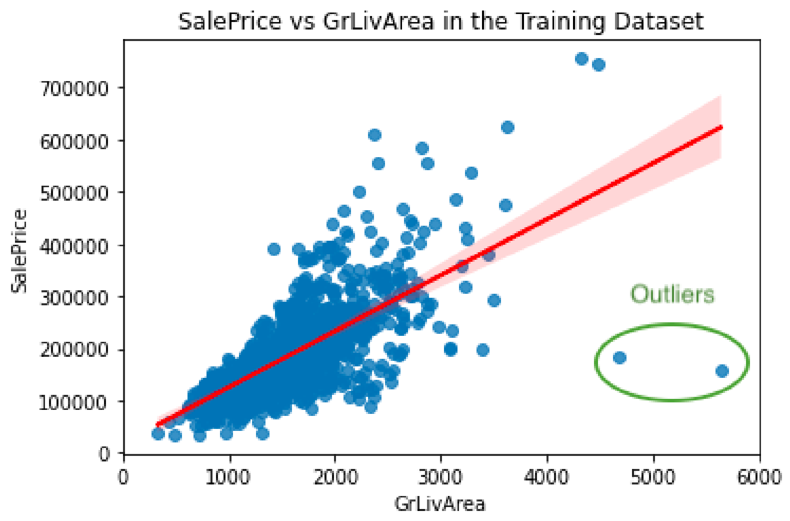
Meet Your Mentors: Kyle Gallatin

by Vivian Zhang

Nov 4, 2020

Predicting NICU Admissions and CCHD

by Paul Lee, Aron Berke, Be
Bettina Meier and Ira Villar



Jan 7, 2020

Why is R a Must-Learn for Data Scientists?

by Aiko Liu, Hanqing Zhang and Pranjali Galgali

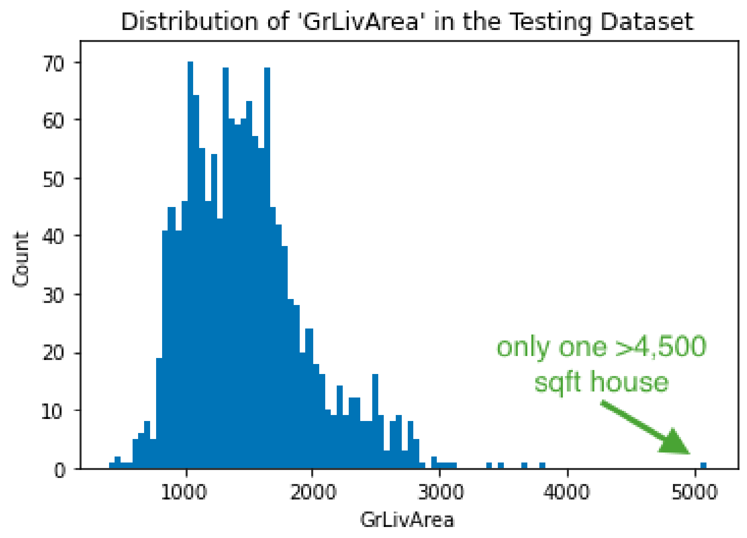
Dec 4, 2019

Checking the distribution of the *GrLivArea* variable in the testing dataset confirmed these two outlier points could potentially end up skewing our predictions, as only one house in the testing dataset exceeds >4,500 sqft in total area. The two previously highlighted points in the training dataset were therefore removed.

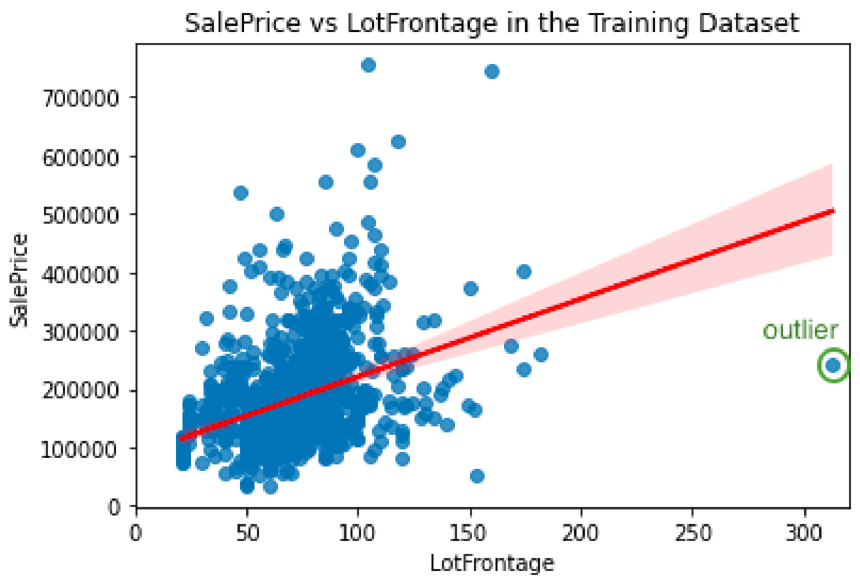
View Posts by Tags

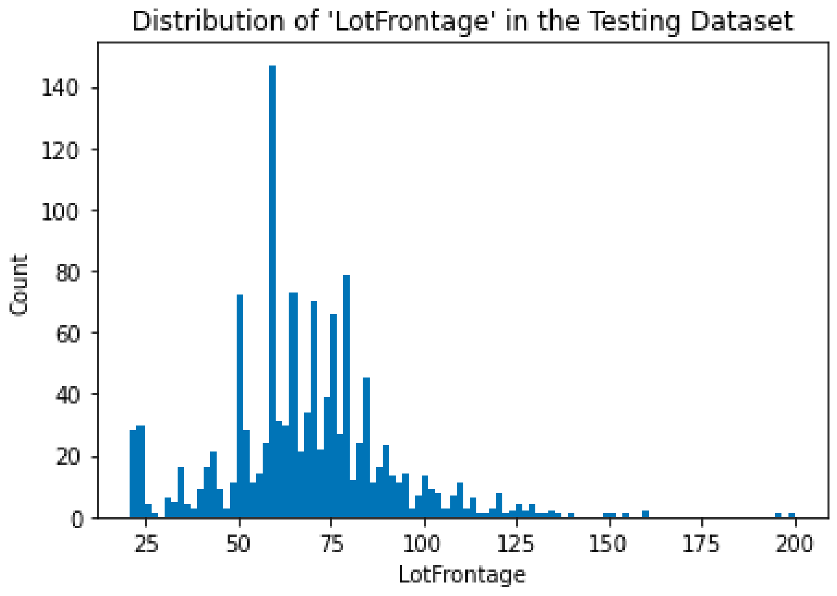
#python #trainwithnycdsa 2019 airbnb Alex Baransky alumni

Show more



The same process was performed with *LotFrontage*. One outlier with >300ft *LotFrontage* was found in the training dataset and was subsequently removed as the testing dataset did not contain any observations with *LotFrontage* > 200ft:

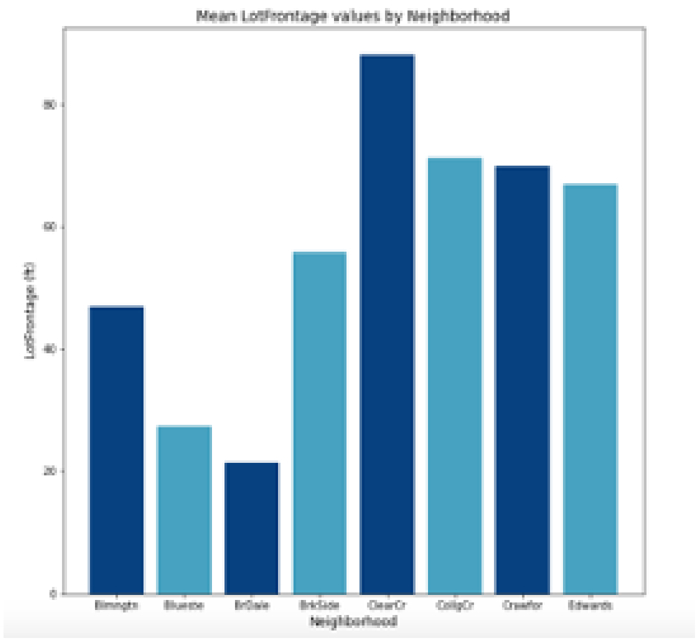




Data Cleaning

Careful reading of the supporting documentation on each variable informed the imputation process. Most of the imputation process dealt with variables having ‘NA’ values that corresponded to the absence of a feature. These ‘NA’ values were replaced by str(None) or 0 values depending on the type of the variable. For instance, the 80 ‘NA’ values in the qualitative feature *BsmtFinType2* meant ‘No Basement’ and were therefore mapped to str(None).

For numerical variables, imputation was performed using either the mean or the mode, whichever was most appropriate. For instance, the 486 missing values in the *LotFrontage* variable were imputed using a groupby + aggregate of mean *LotFrontage* values for each *Neighborhood* (no NA's), as it was observed that LotFrontage values for each neighborhood were tightly distributed. On the other hand, the one missing value in the continuous variable *GarageArea* was imputed to zero as it was assumed that the absence of a value most likely entailed the absence of a garage. Some numerical variables, such as *MoSold*, were converted to string type in order for proper dummification to take place later on.



For categorical variables, values were imputed using the mode of that feature set. The one missing value in the *Exterior1st* variable, for example, was imputed with the mode *VinylSd,* which corresponded to a house with a Vynil siding exterior. Other categorical variables were dropped entirely from the dataset when one value dominated the distribution, such as

RoofMatl, which was dropped as *CompShg* accounted for 98% of values.

Feature Engineering

For categorical variables with string values, these were converted to ordinal variables using dictionaries mapping the string values to integers. For instance, the *KitchenQual* variable with categories *Excellent*, *Good*, *Typical*, *Fair*, and *Poor*, was mapped to {'Po':1, 'Fa':2, 'TA':3, 'Gd':4, 'Ex':5} once its one missing value was imputed to the mode (*Typical* in this case). A total of fifteen variables were transformed using this methodology.

In addition, certain binary variables were created to capture the presence or absence of certain features. The *Has2ndFloor* binary variable, for instance, took on a value of 1 if *2ndFlrSF* > 0, indicating that a house had a second floor. Likewise, the *YearsSinceRemodel* variable, capturing how many years it had been since the house was remodeled, was computed by taking the difference of *YrSold* and *YearRemodAdd.* Some of the other features that followed similar methodologies were:

1. *Remodeled* – Binary variable that takes 1 if *YearBuilt* != *YearRemodAdd*
2. *RecentRemodel* – Binary variable that takes 1 if *YearRemodAdd* = *YrSold*
3. *SqYearBuilt* – Continuous variable constructed by exponentiating *YearBuilt* to the second power
4. *YearSoldYearBuilt* – Binary variable that takes 1 if *YearBuilt* = *YrSold*
5. *Age* – Continuous variable constructed by subtracting *YrBuilt* from *YrSold*

6. *CentralAir* - Binary variable that takes 1 if *CentralAir* = ‘Y’
7. *Area1st2nd* – Continuous variable constructed by summing *1stFlrSF* and *2ndFlrSF*
8. *HasFireplace* - Binary variable that takes 1 if *Fireplaces* > 0
9. *HasShed* – Binary variable that takes 1 if *MiscFeature* = Shed
10. *RR_Proximity* – Binary variable that takes 1 if *Condition1* = 'RRNn', 'RRAn', 'RRNe', 'RRAe'

Feature Selection on Level 0 models

Given the large number of features in the dataset that could cause simple linear models to overfit, I decided to skip Linear Regression and to start directly with penalized linear models that would tend to overfit the data less. The five models I was therefore interested in testing were Ridge, Lasso, Random Forests Regressor, Gradient Boosting Regressor and XGBoost:

Table A. Different Machine Learning Trained on the Data

Model Category	**Type**	**Python**
Linear Models	Ridge	(scikit-learn) Ridge
	Lasso	(scikit-learn) Lasso
Tree-based Models	Gradient Boosting	*(scikit-learn) GradientBoostRegressor*
	Random Forest	*(scikit-learn) RandomForestRegressor*
	XGBoost	*XGBoost) XGBoostRegressor*

Feature selection was approached two separate ways to test which worked best for different models, the first using correlation values and the second using Recursive Feature Elimination ("RFE"). The first approach was to select features based on the absolute value of their Pearson correlation coefficient with *SalePrice* and to fit each of the models on these different feature subsets. In order to speed up this process and to standardize the way selections were made by correlation coefficient value, I wrote a Python script using object-oriented programming that selected these features based on different input correlation cutoff points. All models were tested using cross validation with 2 train folds and 1 test folds each containing 33.3% of the data. The preliminary results following this methodology for each of the models with no hyperparameter tuning can be seen below:

Table B. Feature Selections and CV RMSE & Kaggle Scores using Pearson Correlation Values

Model	Variables Used	CV Neg RMSE	Kaggle Score
Ridge	99	-766k	0.1458
Lasso	98	-780k	0.1403
Gradient Boosting	All variables	-548k	0.1367
Random Forest	All variables	-714k	0.1395
XGBoost	151	-687k	0.1388

The second approach was to use Recursive Feature Elimination to select variables by recursively considering smaller and smaller sets of features. RFE works by repeatedly building models and selecting the best or worst performing feature at any given point, setting this feature aside and repeating the process with the rest of the features. RFE computes which features should be selected-out at every stage by looking at the feature's weight coefficients for linear models and feature importances for tree-based algorithms. Once these features selected-out, RFEcv calculates the new root mean squared error score on the validation data, and the subset of features left at the step that gives the highest score on the validation data is considered to be "the best n_features" subset. Scikit-learn's RFEcv library was used to implement this recursive feature selection. The following graph shows trends in RMSE vs number of features for each of our models:

Table C. Feature Selections and CV RMSE & Kaggle Scores using RFE

Model	**Variables Used**	**CV Neg RMSE**	**Kaggle Score**
Ridge	211	-697k	0.1385
Lasso	222	-723k	0.1394
Gradient Boosting	210	-571k	0.1320
Gradom Forest	32	0874k	0.1464
XGBoost	46	-519k	0.1333

As can be seen by comparing the two tables, all models except for Random Forests saw cross-validated RMSE score improvements using the RFE-selected feature subsets. The RFE-selected feature subsets were therefore used for these models while the Random Forests model was trained using all variables going forward.

The feature importance rankings of the RFE-selected feature subsets provided some interesting preliminary insights, such as that the overall material and finish of the house was a strong predictor of *SalePrice* in our XGBoost model:

Hyper-parameter Tuning

Once the data processed and the features selected, the next step in the process was finding the best hyperparameters for each of our models to ensure high model accuracy. Hyper-parameter tuning is important to make sure that models are not either overfitting or underfitting and to control for the bias-variance tradeoff. Hyperparameters such as *n_estimators* or *learning_rate* for tree-based models can also significantly impact model run time and can in certain cases make the added computational expense not worth the added accuracy benefit they yield. The GridSearchCV package in Python sklearn was used to complete an exhaustive search of the available hyperparameters and yielded higher accuracy scores for each of our models after tuning took place. The different model predictions and how

they compared to the true *SalePrice* values
represented by the straight line can be seen below:

Stacking the Models

The last step in the process was stacking the models. Model stacking is a method that uses the predictions of different Level 0 models to train a Level 1 meta-model that predicts the same y variable and can oftentimes produce an even more accurate prediction of the target variable. In this way, the Level 1 model can weigh the predictive strengths and weaknesses of the different models and capture different kinds of relationships in the data that an individual model may miss.

The tradeoffs for this potentially higher accuracy are higher computational expense as well as lower interpretability, as individual features can no longer be examined separately to better understand what each contributes to the model. This tradeoff of lower interpretability is addressed in the conclusion. For the purposes of this project, a Level 1 model was trained on top of our Ridge, Lasso, RandomForests, XGBoost, and Gradient Boosting Models and as expected outperformed all individual model components, yielding a final Kaggle score of 0.13072.

Conclusion

Given our stacked Level 1 model trains on the predictions of the Level 0 models, I wanted to take a step back to analyze the feature importances in our Level 0 models. The following are some of the insights gleaned from the outputted feature importances of each model as well as some recommendations for residential real estate buyers and sellers:

XGBoost: Qualitative variables such as *OverallQual* and *ExterQual* are most important

- Sellers: Consider investing more into quality of the external finish of a house to fetch higher prices
- Buyers: Consider buying houses with no/low-quality basements and excavating/enlarging/improving these yourself if cheaper in the aggregate

Random Forests: *OverallQual*, *TotalArea*, *Area1stArea2nd*, *LotFrontage* are most important

- Sellers: Consider improving fireplace quality to fetch higher prices
- Buyers: Consider buying houses with no garage and building one yourself if cheaper in the aggregate

Gradient Boosting Regressor: *OverallQual*, *LotArea*, *BsmtQual & YearBuilt* are most important

- Sellers: Consider investing in Central Air conditioning to fetch higher prices
- Buyers: Consider buying older houses and remodeling these yourself

Linear Models: *LotArea*, *Neighborhood* variables are most important

- Sellers: Consider building an open porch to increase house value
- Buyers: Consider building an open porch yourself to save money if cheaper in the aggregate

Thanks for reading!

2
Shares

Share Tweet Share

About Author

Philippe Heitzmann

Philippe is an aspiring data scientist with a track record of using data to drive significant and tangible business results in the Sales & Trading and financial advisory fields. He has hands on experience in R and Python...

View all posts by Philippe Heitzmann >

Related Articles

STUDENT WORKS

Data Science Courses on Udemy: Comparative Analysis

DATA VISUALIZATION

Retention-Driven Marketing for Music Apps

CAPSTONE

Identifying Provider Fraud For Healthcare Insurers

PYTHON

Manhattan Food Trends

PYTHON

Maximum Element In A Stack Hack

Leave a Comment

No comments found.