Project 1: Predict the Housing Prices in Ames

STAT 542: Statistical Learning
Oct.18, 2021

Yu-Wei Lai (yuwei6)

1. Introduction

The project aimed to predict the Ames Housing Data with two kinds of models. The models included Linear Models and Tree-Based Models. For tree-based models, I chose to use the XGBoosting models, while for linear models, I tried Lasso, Ridge, and Elastic Net models. The performance targets are 0.125 for the first five and 0.135 for the last five in log term of RMSE for the pre-specified 10-folds dataset from the Housing Prices in Ames. I mainly used Python for this project.

2. Process

The training and testing datasets were pre-specified by rules, so we should do data preprocessing separately for both sets. Then I fitted two kinds of models: first, I tried the XG Boost models. Secondly, I tried three types of linear models with different hyperparameters and their combinations. Also, I did data processing separately for two requirements. Finally, tested 10tenifferent datasets and output their results of RMSE.

3. Models

Before fitting any model, I used a function train_pre_process to pass the original training set and required the x_train (independent variables) and y_train (target variables). Our target in the project is the variable: "Sale_Price."

3-1. Tree-Based Models: XG Boost

- Data Processing:

  I tried a model without any processing (Raw Model), and a model removed some of the redundant variables which were highly scaled or could be explained by other variables.

  For variables removable, I followed the professor's suggestion that remove highly imbalanced categorical variables and variables which might be explainable or unexplainable. For the former, I removed: 'Street', 'Condition_2', 'Roof_Matl', 'Utilities', 'Heating', 'Pool_QC', 'Pool_Area', 'Misc_Feature', 'Electrical', 'Bsmt_Half_Bath', and 'Low_Qual_Fin_SF'. For the latter, I removed: 'MS_SubClass', 'Longitude', and 'Latitude'.

  For those categorical variables, I first try the method "get_dummies" built-in pandas. It is very efficient and easy to implement. However, we should process the training set and the testing set separately. Using this method would face a severe problem. If there are some factors in the testing group but not in the training groups, then it is difficult for us to create these dummies. This would cause a problem when fitting models.

  Therefore, I choose to use the one-hot encoding method for dealing with non-numerical variables. The technique "OneHotEncoder" from scikit-learn provided an option for handing the unknown factors in the new data frame. It is memory-based that can apply to another data frame after fitting.

- Model Building

For XG Boosting tree models, I used "XGBRegressor" from the package "xgboost." The important hyperparameters in XGBRegressor are:

objective	reg:squarederror; For the regression question, the learning objective should choose to be squarederror.
learning_rate	It is the size of shrinkage of every stage, which is used for preventing overfitting. The default is 0.3. Followed with the instruction, for this model, we should choose a smaller value of 0.05 to make the model more stable.
max_depth	The maximum number of nodes that is, how deep a leaf can be far from the root. The default value is 6. I chose not to change this value.
alpha	Just like the shrinkage penalty term of Lasso, which can make the model more stable. Default is zero.
n_estimators	How many trees are included in the model. The default value is 100. I try the values: 100, 300, 500, 1000, and 5000. The larger will take more time, which makes the model more complex. Both 500, 1000, and 5000 can have good performance. However, it might tempt to be overfitting and huge resources consumption for a large value. Finally, I used 500 for the final model.

The raw model with simple setting {"objective":"reg:squarederror", 'colsample_bytree': 0.5,'learning_rate': 0.05, 'max_depth': 6, 'alpha': 10, 'n_estimators': 500} could get a good performance with the first five at the average 0.1168, and the last five at the average 0.1307. However, the 7th set's RMSE is 0.135824, which is below our target performance.

Therefore, I did data preprocessing, which was mentioned above, for reliable performance. The model fitted by processed data performed better with the same hyperparameters, with 0.116524 on average for the first five sets and 0.129292 on average for the last five sets.

3-2. Linear Models:

- Data Processing:

Followed the same procedure above, remove imbalanced variables and redundant variables. Also, I followed the instruction to deal with extreme values by the Winsorization method. That is, treat the outer value of 95% of the central value to become the same value of minimum or maximum of the interval. The method I used was "winsorize," which is in the package "scipy"."The method apply to these variables: ["Lot_Frontage", "Lot_Area", "Mas_Vnr_Area", "BsmtFin_SF_2", "Bsmt_Unf_SF", "Total_Bsmt_SF", "Second_Flr_SF", 'First_Flr_SF', "Gr_Liv_Area", "Garage_Area", "Wood_Deck_SF", "Open_Porch_SF", "Enclosed_Porch", "Three_season_porch", "Screen_Porch", "Misc_Val"] For non-numerical variables, I used the same one-hot encoding method in XG Boosting.

- Model Building:

I mainly used the scikit-learn package for model building since I faced several issues when using the glmnet_pyhton. I tried the three models with cross-validation processes separately, resulting in the value being far more different than the targets. The single Lasso performed 0.1462 on average; The Elastic Net performed 0.1519 on average; The Ridge performed the worst, the average at 0.5510.

Followed the instructions, I tried to combine the model of Lasso and Elastic Net. First, I fitted the Lasso with Cross-Validation to get the best lambda value (I use min-lambada). Also, we can drop the variables which were set to be 0 by the Lasso algorithms. Then, we fitted the Elastic Net model with the smaller data frame.

The result of the final model was slightly better, with the first five values with an average of 0.1407 and the last five values with an average of 0.147462; however, it still not met the target.

4. Performance

The RMSE of testing results from the ten training and testing split into two methods:

	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6	Set 7	Set 8	Set 9	Set 10
Tree-Based Model	0.118643	0.118217	0.118375	0.116144	0.111241	0.129088	0.131514	0.131300	0.133264	0.121293
Linear Regression Models	0.155230	0.132206	0.136328	0.151149	0.128619	0.160753	0.140808	0.133618	0.162208	0.139922

- Running Time

Using a 2020 model of MacBook Air (3.2GHz M1 chip with 8GB RAM), the program proceeded with the final model of XG Boost for ten testing procedures with 43.6947 seconds. For the regression model, the program consumed 35.11618 seconds.

5. Conclusion

The combined linear model did not perform well. Only one value meets the target. Maybe applying the glmnet_python package can have a better performance or by using 1se-lambda instead of the min-lambda. (After an update for OS, the package "glmnet_python" will cause an error on the importing stage. I still couldn't figure out the solution.) Secondly, the methods of dealing with categorical variables could be improved. Maybe encode factors for each variable could pick some complex variables at the first stage.

While between my results, the difference of time consumption between train/test for XG Boost and Linear models is not that significant. However, XG Boost can give us a better-fitted model even without being carefully trained, and it still performed better than those linear regression models. On the other hand, since we are trying to beat the target performance, our models might be overfitted for the dataset.