Donia Zaheri

PROJECT 1
Predict the Housing Prices in Ames

Introduction and Goal

For this project, we would like to predict the housing price for a house in Ames, Iowa, based on the provided dataset of the Ames sold houses from 2006 to 2010 with their sale price (target values) and many characteristics of the sold house (features).

Preprocessing the Data

Training and test data were pre-processed separately. In the first step, I stored “PID” and “Sale_Price” columns in individual variables and dropped them from the training data (same process was done for the test data in regards to “PID”), since the prior feature is not a predicting feature, but just the parcel ID and the second is our target.

Log Transformation

The models chosen for this project assume normal distribution of data. In the next step, I found highly skewed features (abs(skewness) > 0.5) and replaced them with their logarithm to transform them into normal distribution. The highly skewed features were:

{Sale_Price, Lot_Area, Year_Built, Mas_Vnr_Area, BsmtFin_SF_2, Bsmt_Unf_SF, Total_Bsmt_SF, First_Flr_SF, Second_Flr_SF, Low_Qual_Fin_SF, Gr_Liv_Area, Bsmt_Full_Bath, Bsmt_Half_Bath, Half_Bath, Kitchen_AbvGr, TotRms_AbvGrd, Fireplaces, Garage_Yr_Blt, Wood_Deck_SF, Open_Porch_SF, Enclosed_Porch, Three_season_porch, Screen_Porch, Pool_Area, Misc_Val, Latitude}

Variables Correlation

Then I explored correlation between variables. I expected features like "Overall Quality", "Neighborhood", "Lot size", "garage quality" to have high correlation with the house price. Also, some features such as "Overall Quality" and "Overall Condition" or "Garage Quality" and "Garage Condition" sounded similar, so must have high correlation. I plotted a heatmap plot to find the highest correlations (correlations above > 05). Correlation heatmap showed that these features were highly correlated: “garageCars" and "GarageArea" (0.9), "TotalBsmtSF" and "1stFlrSF" (0.8), "GrLivArea" and "TotRmsAbvGrd" (0.8). Therefore, only one feature in each pair needed to be used for developing the model. Between "GrLivArea" and TotRmsAbvGrd" I chose to keep "GrLivArea" since it has a higher correlation with "Sale_Price". The other two pairs have the same correlation value, so there was no preference over which one to keep.

Handling Missing Values

The only feature with missing values was “Garage_Yr_Built”. Missing values for this numerical feature were replaced with zero.

Removing Unimportant Features

Features {Fireplace_Qu, Garage_Cars, Total_Bsmt_SF, TotRms_AbvGrd, Utilities, Condition_2, Roof_Matl, Heating, Pool_QC, Misc_Feature, Low_Qual_Fin_SF, Pool_Area, Longitude, and Latitude} were removed. These features were either duplicate of the other features or had a very low correlation with the sale price.

Adding a New Column

Based on personal experience and exploring Zillow and Redfin listings, I know that the total area of a house is an essential feature and the variable used to find price/sf. This dataset had individual areas per floor, yet not the total area. I added this feature column to the data using the individual area measures.

Converting Categorical Variables to Numerical

Categorical variables need to be converted to numericals to be able to be used in the models. This goal was achieved by creating dummy variables. For each level of a categorical variable, a dummy variable was created. The test data missed some values for categorical values that the training data had. Hence, the number of dummy feature columns created for the test data was fewer than the training data. XGBOOST function needs the test data to have the same shape as the training data. To address this issue, I included another step to add those missing columns to the test data with "0" values.

Applying Models

Tree-Based Gradient Boosting

For the tree-based model, the xgboost.XGBRegressor function was used. Hyperparameter tuning for this function was done by a few rounds of try and error to achieve an RMSE with satisfactory running time. The hyper parameters below were tuned for this function:

Parameter	**Description**	**Value**
gamma	Minimum loss reduction used for making a leaf node partition. The larger gamma is, the more conservative the algorithm will be.	0.025
colsample_bytree	The subsample ratio of columns when constructing each tree. For a good RMSE result a subsample of around 0.5 is used	0.42
learning_rate	Step size shrinkage used in updates to prevent overfitting. Learning rate shrinks the feature weights to make the boosting process more conservative.	0.005
max_depth	Maximum depth of a tree. Larger max_depth will make the model more likely to overfit and more complex	6
n_estimators	Number of estimators used in building the tree	3800
reg_alpha	L1 regularization term on weights. Increasing alpha value will make model more conservative	0.1
reg_lambda	L2 regularization term on weights. Increasing lambda will make model more conservative	0.95
subsample	The ratio of the training instances used for subsampling.For a good RMSE result a subsample around 0.5 is used to prevent overfitting, meaning the model will randomly sample half of the training data prior to growing trees	0.55
random_state	Random number seed	9549

Linear Model – Lasso

For linear regression models such as Lasso it is important to have the data scaled, since these models are very sensitive to feature weights. Lasso function from sklearn.linear_model was used to fit the linear regression model. The only specifying parameter for the Lasso function to be tuned was "alpha". Alpha is the parameter to control how much regularization we want on our model and prevents overfitting the training data with adding penalty for including more features. To find out the best alpha in terms of the least RMSE, I used GridSearchCV. Alpha = 0.00038 was selected as a result.

Performance

The main dataset was divided into 10 different training and test sets using the provided project1_testIDs.dat file. Each training/test set was run individually and their RMSE and running time was recorded. Below are the RMSE and running time errors for the 10 sets for both models:

	XGBoost		Linear Regression	
Test Set	Time	RMSE	Time	RMSE
1	29.92594	0.12316	0.120483	0.125876
2	29.99997	0.109623	0.109133	0.12009
3	29.80449	0.122041	0.101127	0.129478
4	29.87296	0.126849	0.093477	0.140937
5	28.78613	0.131746	0.119433	0.132121
6	29.95526	0.12355	0.123011	0.125787
7	30.27333	0.110561	0.097039	0.120092
8	30.09129	0.123944	0.099251	0.129506
9	30.06791	0.128082	0.096533	0.140937
10	29.60826	0.131259	0.117605	0.13212

Conclusion

Based on the results XGBoost provides slightly better RMSE than linear regression, while the Lasso model is much faster. XGBoost has more parameters to play with and was easier to get the model to an RMSE below threshold compared to Lasso that only has alpha to play with to get a better RMSE. However, linear regression (here Lasso) is a better interpretability model as it is easier to understand the weights (features significance) and the tuning parameter – alpha (reducing number of features).

Platform

Windows 10 (64 bit), Installed memory: 32.0 GB. System: Intel®, Core™ i7-9850H CPU @ 2.60 Hz 2.59 GHz

References

1. https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html
2. https://www.kaggle.com/tomasmontielprieto/keeping-it-simple-eda-lasso-model-rmse-0-126