Project 1: Predict the Housing Prices in Ames
CS598: Practical Statistical Learning

Naomi Bhagat - nbhagat3, Michael Miller - msmille3, Joe May - jemay3

02 October 2023

Assignment Data

Program: MCS-DS Assignment post: campuswire

Team contributions:

Person	Contribution
Naomi Bhagat	Data pre-procesing, traning/testing general process, report Section 1
Michael Miller	Training/testing refinement and finalization, general process
Joe May	Report Section 1 and 2, debugging, and quality control

Section 1: Technical Details

The goal of this project is to predict the price of a home using certain features present in the Ames housing dataset. Importantly, the Sale price is predicted on a logarithmic scale. To achieve this goal, we built two prediction models: one using linear regression and one using a tree model to yield an RSME of 0.125 for the first five splits and 0.135 on the final five splits.

Our implementation begins by looping through the packages 'glmnet', 'randomForest', and 'xgboost', which we install or import as needed. We have a variable for debugging that is FALSE by default. When it's set to TRUE our code runs in debugging mode. After our installs, we initialize vectors for winsorization and omitted variables, followed by functions to be called when we train and evaluate the model.

There were several pre-processing steps necessary for the linear model. The first function 'clean_input' takes a dataframe as an input and returns a cleaned dataframe. First we replace missing values for the year the garage was built with zero. This is the only column missing values.

Second, we remove the columns for variables we've decided to remove. We removed 'Street', 'Utilities', 'Condition_2', 'Roof_Matl', 'Heating', 'Pool_QC', 'Misc_Feature', 'Low_Qual_Fin_SF', 'Pool_Area', 'Longitude', & 'Latitude'. We chose them because most are categorical values with a skewed balance of indicators, which would likely cause over fitting. Latitude and longitude don't lend themselves to interpretable insights, especially since neighborhood does a clearer and better job.

The next pre-processing step was Winsorization, which applies transformations to numerical variables. We perform winsorization on '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'. This identifies values above the the 95% quantile with that value. This creates a cap because there are diminishing returns for considering larger values for these numerical columns.

Finally, we convert any left over categorical variables to 1-hot vectors. Our pre-processing step counts the unique categorical values for any character vector, creates a column for each value, and assigns a one to the column representing a row's category, and sets the rest equal to zero. The last part of our 'clean_input' function puts 'Sale_Price' at the end and returns the cleaned dataframe.

Afterward, we have functions 'force_col_match', 'print_formatted', and 'get_rmse'. The first function is one which pads with columns of zeros to ensure our training and test data have the same number of

columns. The function 'print_formatted' prints our predictions into the two submission files. Also, we have the 'get_rmse' function, which the sum of squares difference in actual and predicted y values, and takes the square root of the sum of squares divided by n.

Next we have a 'train_and_eval' function that takes the test and training data as inputs, processes the data, and creates the models. It begins by cleaning the training data with the common 'clean_input' function. Then we clean the test data with 'clean_input'. Next, the cleaned dataframes are passed into 'force_col_match' so that the dataframes have the same number of columns. Then we format the data as matrices. Finally, we are ready to create our models.

We use 'glmnet' to create a temporary model, with an alpha of one. This model is used to determine the selection of model variables. Again, we use 'glmnet' to create the actual linear model, restricting our data to the variables selected by the temporary model, with alpha set to zero. Our model utilizes a blend of lasso and ridge regression as implied by our different alphas. Our variable selection of 'glmnet' in R utilized a lasso for regression modeling since alpha was zero. For our actual model, we utilized a ridge for regression modeling since we used 'glmnet' in R with an alpha of one. Once the modeling is finished, we make our predictions of the test data.

The largest challenge was dealing with categorical variables. The model would more appropriately deal with categorical variables using 1-hot encoded columns, but this introduced a challenge because not all the same categorical values existed in both datasets. Consequently, we had to force the dataframes to match to deal with the unintended consequence of our preprocessing.

Next our model implements the tree model. We use 'xgboost' in R with a maximum depth of 6, and eta of 0.05, an nthread of 2, 500 n rounds, and set verbose and print every n to zero. Then, we make our predictions on the test data. Lastly, our function evaluates the models.

Last, but not least, we run the entire process. Whether it is in debugging mode or not, we import the appropriate test and training data. If it is, we split the test data into ten folds, iterate through the ten folds, train, evaluate, and print statistics. We print RMSE for the linear and tree model for its performance and the time it took to train that fold. If the code is not in debugging mode, it imports test data, imports training data, trains, and evaluates the models. Finally, it prints the submission files.

Section 2: Performance Metrics

The computer system this was run on was a Dell Inspiron 3501, 1.00 GHz with 8.00 GB of installed RAM for all 10 training/test splits. Below is a summary of the fold #, the RMSEs for the linear and tree models respectively and the time it took to train the models:

Fold	Linear RMSE	Tree RMSE	Linear Time	Tree Time
1	0.1248	0.1171	1.415	6.725
2	0.1211	0.1211	1.919	10.118
3	0.1201	0.1138	1.672	10.393
4	0.1208	0.1181	1.730	9.897
5	0.1140	0.1152	2.155	10.121
6	0.1341	0.1308	1.473	10.330
7	0.1266	0.1340	1.379	10.355
8	0.1192	0.1283	1.756	10.378
9	0.1305	0.1315	1.547	10.226
10	0.1257	0.1269	1.528	10.348