Final Project

House Prices: Advanced Regression Techniques

Abhiram Reddy Kanmatha Reddy
Team members- Siddharth Kilaru, Praneel Naidu Lalam.

Problem Statement

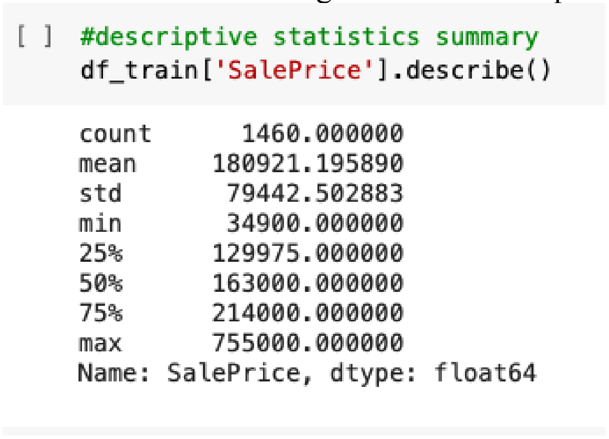
A banking company wants to come up with a model to predict house prices. Using the predictions, they want to assess the risk of lending to a particular borrower. Generally, financial companies offer loans based on collateral. The process usually takes around 7-10 days to give an estimate on the house price. To speed up the process, I was hired by the company to come up with a model to predict the sale prices.

The data is compressed of all the residential homes in Ames, Iowa. It has 79 exploratory variables and the sale prices(Target variable) of the houses.

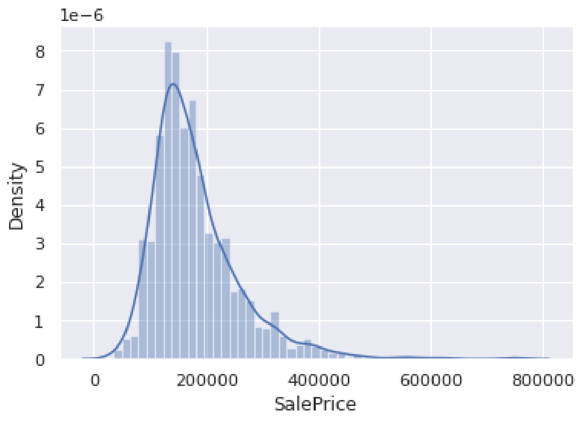
Understanding the data(EDA)

We'll look at each variable and do an exploratory data analysis and identify their meaning and importance for this problem.

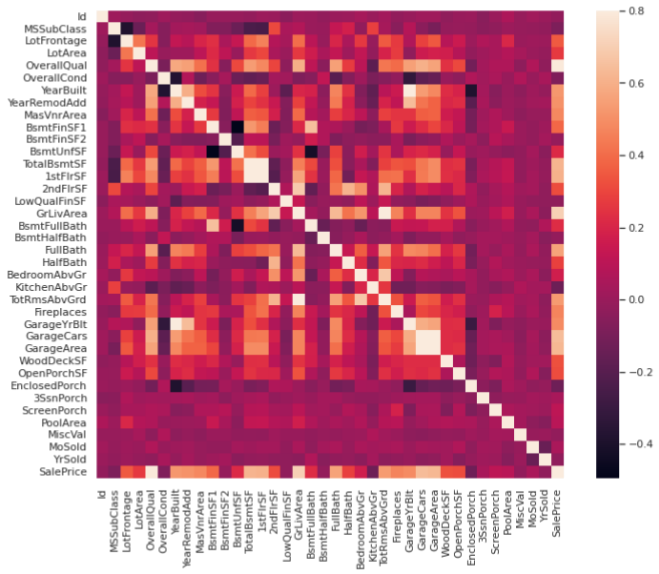
Now let's look at our target column- Sales price.

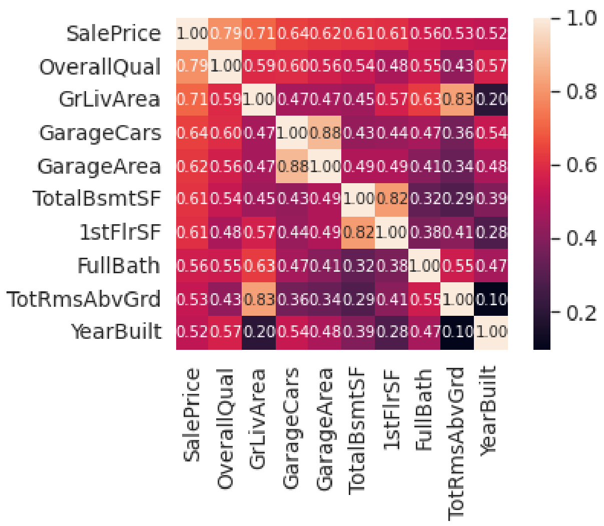


```
[ ]  #descriptive statistics summary
     df_train['SalePrice'].describe()

     count      1460.000000
     mean     180921.195890
     std       79442.502883
     min       34900.000000
     25%      129975.000000
     50%      163000.000000
     75%      214000.000000
     max      755000.000000
     Name: SalePrice, dtype: float64
```



From the distribution plot, we can see that the graph is not normally distributed. It also shows skewness and peakedness in the data. Now we can directly look at the correlation matrix and see the correlated variables.





There is a sign of multicollinearity in the data. For instance, All the GarageX variables are strongly correlated with each other. We'll remove the variables and keep those that have higher correlation with sale price.
Overall, We can say that-

- 'OverallQual', 'GrLivArea' and 'TotalBsmtSF' are strongly correlated with 'SalePrice'.
- 'YearBuilt' is slightly correlated with 'SalePrice'
- 'TotalBsmtSF' and '1stFloor' seems to be correlated with each other so we'll keep 'TotalBmstSF'
- We can remove all the variables that shows multicollinearity

Handling missing data

We will remove the skewness of our target feature and numerical feature if exists. Next, we'll handle the missing data in both numerical and categorical features and perform feature engineering.

Reducing skewness for numerical features

```
data_num_skew = data_num.apply(lambda x: skew(x.dropna()))
data_num_skew = data_num_skew[data_num_skew > .75]
# apply log + 1 transformation for all numeric features with skewnes over .75
data_num[data_num_skew.index] = np.log1p(data_num[data_num_skew.index])
```

Handling missing values in both categorical and numerical features

I've assumed that if there are more than 50 missing values in a column, we'll drop the feature. If there are less than 50 missing values, we'll delete the observation from the data and keep the feature.

```
# drop column if there is more than 50 missing values
    if missing_values > 50:
        print("droping column: {}".format(col))
        data_cat.drop(col, axis = 1)
```

Applying log transformation for the target variable.

```
target = train['SalePrice']
target_log = np.log1p(train['SalePrice'])
```

Modeling

I'm not splitting the data for train and test from the train data set. I decided to use all the training data for training the model and as per the Kaggle evaluations, I'm using the Root Mean Squared Error(RMSE) of the logarithm of predicted values and the logarithm of the observed sale price.

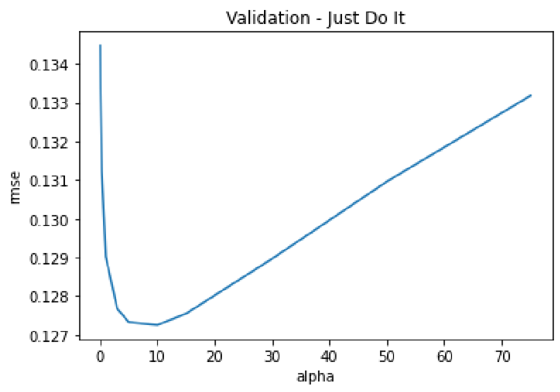
```
# remove Id and target variable
X_train = train[train.columns.values[1:-1]]
y_train = train[train.columns.values[-1]]
# remove Id
X_test = test[test.columns.values[1:]]
```

Let's start with a simple linear regression model and move on to regularized models(Ridge and Lasso penalty).

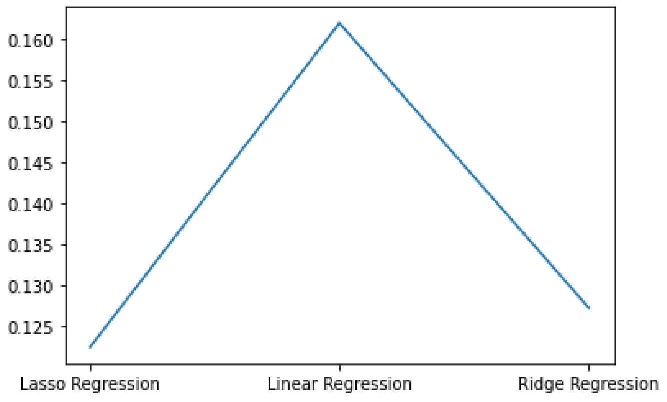
```
# Linear Regression
model_linear=LinearRegression()
model_linear.fit(X_train,y_train)
y_test_pred_log = model_linear.predict(X_test)

# Ridge Regression
Model_ridge=Ridge()
alphas = [0.05, 0.1, 0.3, 1, 3, 5, 10, 15, 30, 50, 75]
cv_ridge = [rmse_cv(Ridge(alpha = alpha)).mean() for alpha in alphas]

# Lasso Regression
model_lasso = LassoCV(alphas = [1, 0.1, 0.001, 0.0005]).fit(X_train,
y_train)
```

For both the ridge and lasso regression, finding the best alpha value is important. We can build the models for all the alpha values as we did with ridge regression and plot a graph, or we can use grid search CV as we did with LassoCV and let it choose the best alpha value.

Choosing the right alpha for ridge

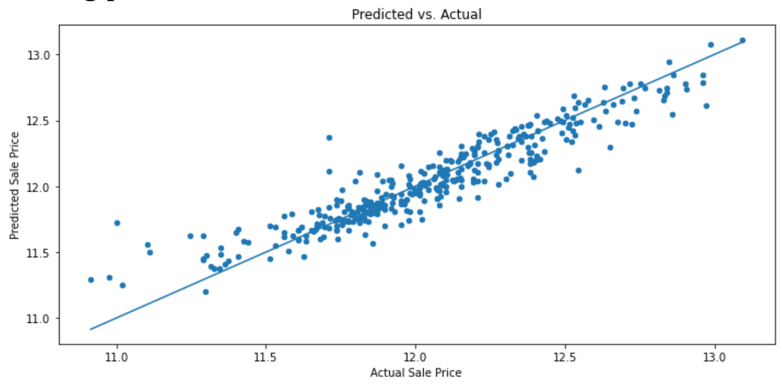


Comparison of the three models



Random Forest

As we know, Random Forest in an ensemble learning method where it combines the predictions of multiple individual models to make the most accurate prediction. the final prediction is made by averaging the predictions of all the trees. It is relatively resistant to overfitting, which is good for making predictions



Boosting (XGBoost and Gradient Boost)

Boosting involves training a series of weak models in sequence, with each model trying to correct the errors made by the previous model. The final prediction is made by combining the predictions of all the individual models. Let’s look at the gradient boosting and XGboosting models.

Tuning the models

```
param_grid = {'max_depth': [2, 4],
        'learning_rate': [0.0001, 0.001, 0.01, 0.1, 0.2, 0.3],
        'min_child_weight': range(1, 10, 2),
        'n_estimators': range(50, 300, 50),
        'objective': ['reg:linear']}
grid_search = GridSearchCV(model, param_grid, n_jobs=-1, cv=kfold, verbose=1, scoring=scorer)
grid_result = grid_search.fit(X_train, y_train)
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

We tune the models using grid search CV to find the best parameters for building the model. In the above code snippet, We provide the parameters such as max_depth and learning_rate, and so on. GridSearchCV will fit all the possible combinations of parameters and find the best possible one.

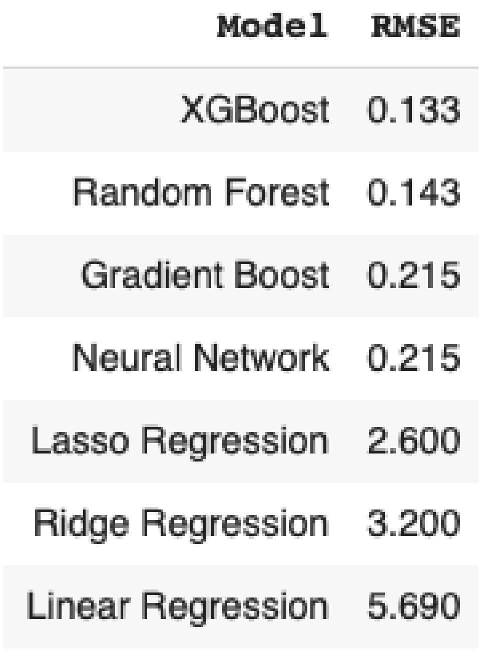
Building a Neural Network

I built a neural network with 3 layers(Input layer, Hidden layer and Output layer). The activation function for the input and hidden layer is ‘relu’ and sigmoid function for the output layer.

```
def build_model():
    model = Sequential()
    #Input layer
    model.add(Dense(20, input_dim=288, activation='relu'))
    #Hidden layer
    model.add(Dense(40, activation='relu'))
    #Output layer
    model.add(Dense(1, activation='sigmoid'))
    # Compile model
    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
    return model
```

Conclusion

Random forest and XGBoost perform the best out of all the models that were compared, it could indicate that these models are well-suited to the task and the data that they were applied to. It could suggest that the flexibility of these models is higher compared to the other models.



Model	RMSE
XGBoost	0.133
Random Forest	0.143
Gradient Boost	0.215
Neural Network	0.215
Lasso Regression	2.600
Ridge Regression	3.200
Linear Regression	5.690