Independent Project Report

1/5/22

Professor Edward Wong

Jaeha Huh

Content

1. Abstract

The goal of this independent project is to predict housing prices using machine learning models on a housing price dataset. More precisely, it is to find a machine learning model that produces high accuracy in predicting housing prices. Since the housing price dataset has a target value (housing price), I decided that it would be more appropriate to use supervised learning models. First, before testing the machine learning model on this dataset, I used visualization techniques to understand the dataset and find the distribution of each feature, outliers in specific columns, and missing data values. After identifying the data structure, I divided the data set into training set and test set at a ratio of 8 to 2. Finally, I selected three supervised learning models, tested each model on a normalized dataset and an unnormalized dataset, and selected the model with the highest accuracy in predicting housing prices.

2. Program Background

To test the supervised learning model, I chose a Scikit-learn library that provides a variety of regressors and decided to proceed with the project on the Jupyter Notebook. In addition to Scikit-learn library, I also used a variety of other libraries in jupyter Notebook. The libraries I used are as follows: sklearn, numpy, matplotlib, seaborn, scipy, pandas, xgboost.

Sklean: I used the library to apply normalization, ridge regressor, and MLP regressor models and divide data into test sets and training sets.

Numpy: It was used for various functions necessary for large-scale multidimensional arrangements and matrix operations.

Matplotib: The library was used to represent data in a two-dimensional coordinate system.

Seaborn: I used the library to visualize the data.

Scipy: I used the library to show the distribution of data.

Pandas: It is a library that can create and process data objects consisting of rows and columns and is used to process large amounts of datasets more reliably.

Xgboots: I used this library to use XGBregresor.

```
import numpy as np
import matplotlib.pyplot as plt  # A library for representing multiple data in␣
 ↪a two-dimensional coordinate system
import seaborn as sns    # statistical data visualization
color = sns.color_palette()
from scipy.stats import norm, skew # A library for standard continuous/
 ↪differential probability distribution and various statistical tests.
from scipy import stats
import warnings
warnings.filterwarnings(action = 'ignore') # turn off the warning
import pandas as pd
from sklearn.model_selection import train_test_split # for dividing the␣
 ↪training and test set
from sklearn.linear_model import RidgeCV
from sklearn.neural_network import MLPRegressor
from xgboost import XGBRegressor
from sklearn import metrics  # A library for result
from sklearn.preprocessing import MinMaxScaler   # normalization tool
pd.options.display.max_rows = 4000   # Set the setting value to 4,000 to see␣
 ↪the entire result

#Code that allows you to see the graph directly from the browser that ran␣
 ↪jupyter notebook.
%matplotlib inline
```

Figure 1. List of libraries

3. Data analysis and visualization

A. Data Structure

The housing price dataset has a total of 81 features and includes factors used to evaluate the house, such as street, house size, building type, house condition, number of toilets, etc. The description of each column is written in link below.

https://github.com/Jeaha31/Independent-project/blob/main/housing_data_description

First, to understand the structure of the data, each column was divided into three categories according to the values in that column. This classification shows how the data is distributed according to the type of each column, and later complements the missing values of each column in a different way according to this classification. Thus, I divided columns into nominal_vars, ranking_vars, and continue_vars, respectively, as shown in the figure below.

```
# nominal vars
nominal_vars = [
    'MSZoning', 'LandContour', 'Utilities',
    'LotConfig', 'Neighborhood', 'Condition1', 'Condition2', 'BldgType',
    'HouseStyle', 'RoofStyle', 'RoofMatl', 'Exterior1st',
    'Exterior2nd', 'MasVnrType', 'Foundation', 'Heating', 'Electrical',
    'GarageType', 'MiscFeature',
    'SaleType', 'SaleCondition'
]
```

```
# ranking vars
ranking_vars = [
    'OverallCond', 'OverallQual', 'ExterQual', 'ExterCond', 'BsmtQual', 'BsmtCond',
    'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2', 'HeatingQC', 'KitchenQual',
    'FireplaceQu', 'GarageQual', 'GarageCond', 'PoolQC', 'Fence', 'Street', 'Alley',
    'LandSlope', 'Functional', 'GarageFinish', 'MoSold', 'YrSold', 'PavedDrive',
    'CentralAir', 'LotShape', 'MSSubClass'
]
```

```
# continuous vars
continue_vars = [
    'LotFrontage', 'LotArea', 'MasVnrArea', 'BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF',
    'TotalBsmtSF', '1stFlrSF', '2ndFlrSF', 'LowQualFinSF', 'GrLivArea', 'BsmtFullBath',
    'BsmtHalfBath', 'FullBath', 'HalfBath', 'TotRmsAbvGrd', 'BedroomAbvGr', 'KitchenAbvGr',
    'Fireplaces', 'GarageCars', 'GarageArea', 'WoodDeckSF', 'OpenPorchSF', 'EnclosedPorch',
    '3SsnPorch', 'ScreenPorch', 'PoolArea', 'MiscVal', 'GarageYrBlt', 'YearBuilt', 'YearRemodAdd'
]
```

Figure 2. Table of each column classified according to the type

i) Ranking_vars

In the case of ranking_vars, the values in the column indicate a string or rank or step. For example, the values in the dataset in the example below (Figure 3) indicate the ranking of house quality. Thus, I classified it as ranking_vars.

	OverallCond
0	5
1	8
2	5
3	5
4	5
5	5
6	5
7	6
8	5
9	6
10	5
11	5
12	6
13	5
14	5

OverallCond: Rates the overall condition of the house

10	Very Excellent
9	Excellent
8	Very Good
7	Good
6	Above Average
5	Average
4	Below Average
3	Fair
2	Poor
1	Very Poor

Figure 3. Rates the overall condition of the house column

Thus, not only OveralCond, but also OverallQual, ExterCond, ExterQual, KitchenQual, etc., which show the ranking or sequence related to the house were classified as running_var.

ii) continue_vars

In the case of continue_vars, the values in the column are numbers. For example, the values in the dataset in the example below (Figure 4) indicate the lot size in square feet. Thus, I classified it as continue_vars.

	LotArea
0	8450
1	9600
2	11250
3	9550
4	14260
5	14115
6	10084
7	10382
8	6120
9	7420
10	11200
11	11924
12	12968
13	10652
14	10920

Figure 4. Size of Lot Area column

As above, LotArea, LotFrontage, TotalBsmtSF, BsmtHalfBath, GarageArea, etc., which contain numerical values such as size and width, were classified as continue_vars

iii) nominal_vars

In the case of nomial_vars, the values in the column are types. As in the example below (Figure 5), columns such as MSZoning, RoofStyle, SaleType, etc., including street, contain values indicating the type. Those columns were classified as nominal_vars.

	Street
0	Pave
1	Pave
2	Pave
3	Pave
4	Pave
5	Pave
6	Pave
7	Pave
8	Pave
9	Pave
10	Pave
11	Pave
12	Pave
13	Pave
14	Pave

Street: Type of road access to property

Grvl	Gravel
Pave	Paved

Figure 5. Street column

The process of classifying these data will later be a guide to what values to put in when replacing missing values in the data.

B. Outlier

According to the chart (Figure 6), House prices are generally in the 200,000 dollars range, with outliers ranging from 34,900 dollars to 755,000 dollars.

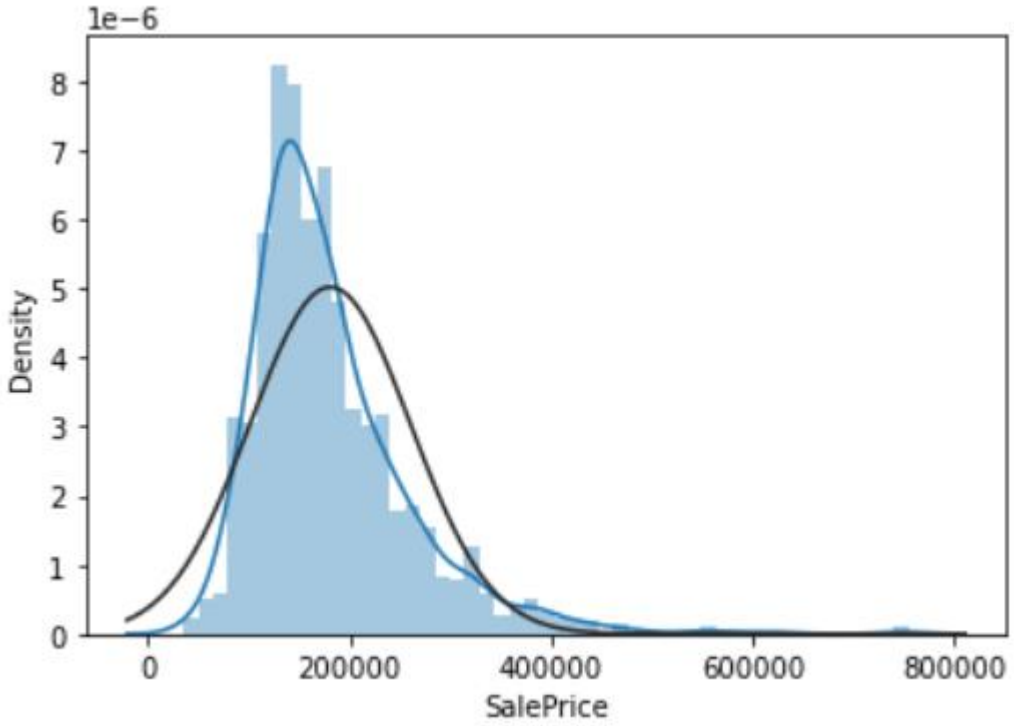

Figure 6. The distribution of housing prices

The below charts (Figure 7) are showing the distributions of each column of continue_vars. Each chart shows a variety of distributions from dense to irregularly distributed columns. However, the reason why the outliers of irregularly distributed graphs cannot be removed or replaced is that they have a great influence on machine learning outcomes.

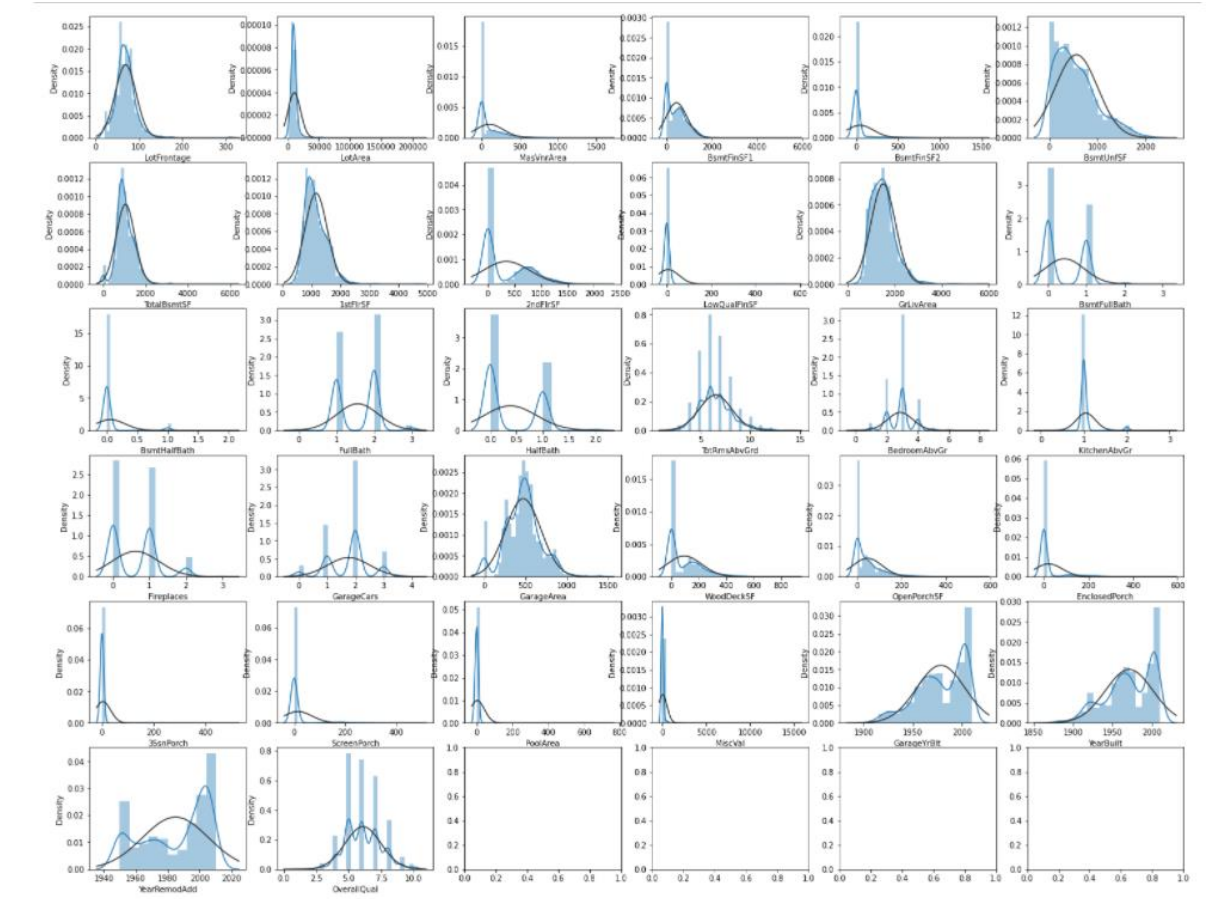

Figure 7. The distributions of continue_vars columns

Thus, although it does not have a significant impact on the machine learning model results, we deleted the outliers in the GrLivArea column to further improve the performance of the model. The chart below (Figure 8) is the distribution chart of the GrLivArea according to the house price. GrLivArea had a high distribution of data at the bottom and a small number of outliers. I removed values of 4000 or more, which are the outliers of this chart.

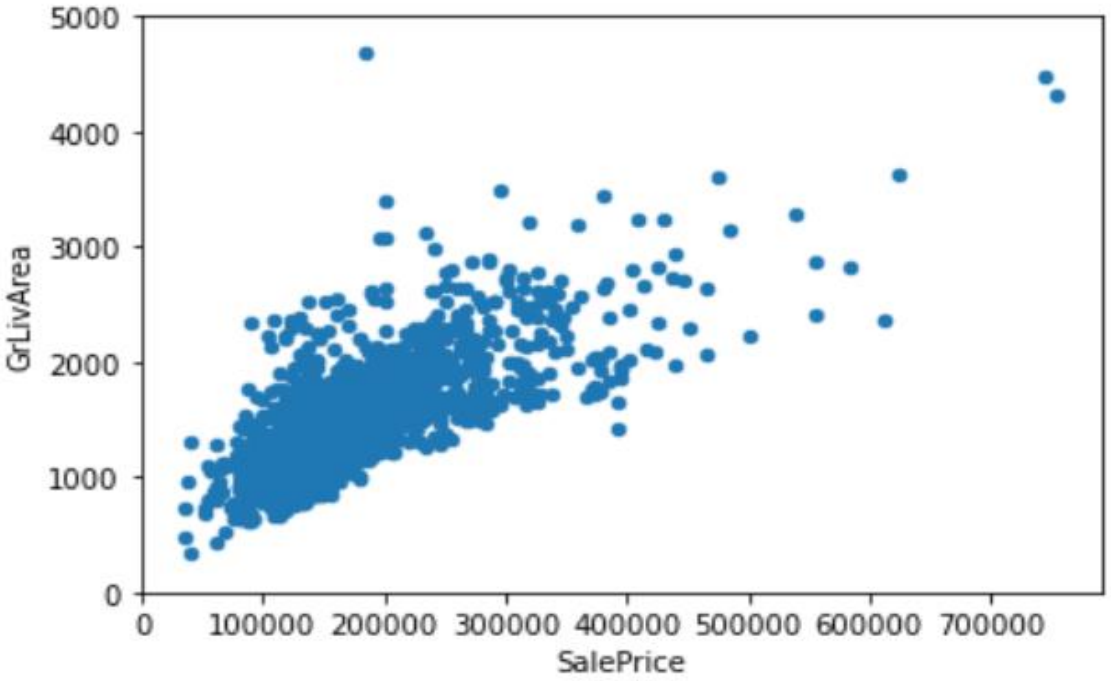

Figure 8. The distribution of GrLivArea values

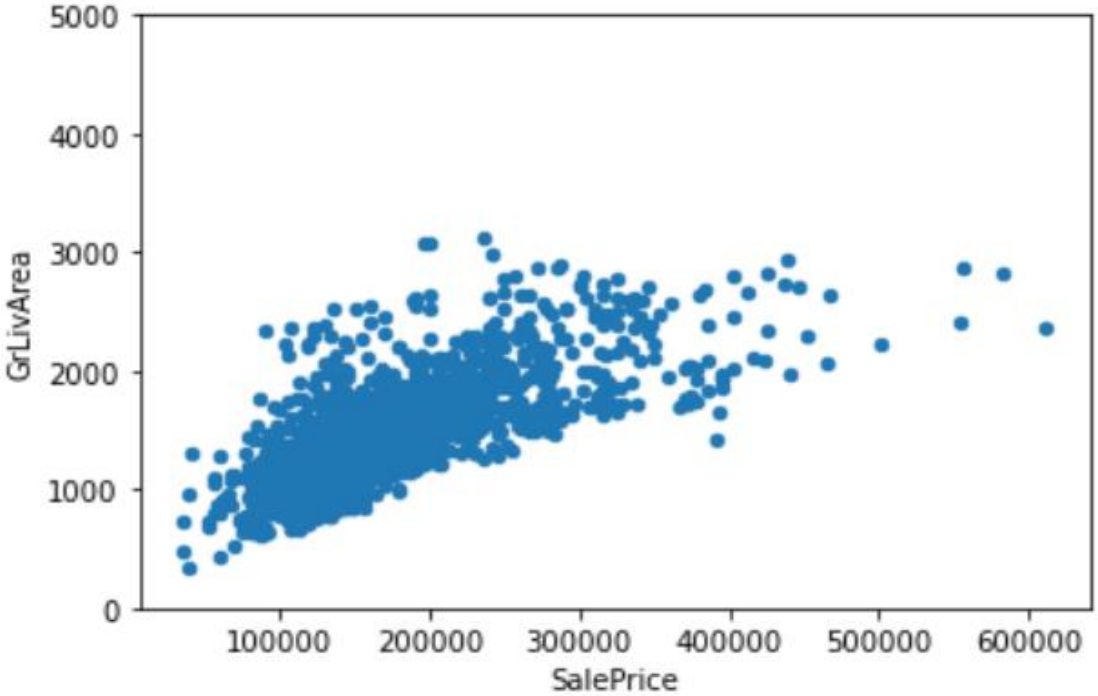

Figure 9. The chart with outliers removed

C. Missing values

Most machine learning algorithms do not work properly with missing values. So, I checked and removed the missing values of the data through data preprocessing and switched to consistent data form.

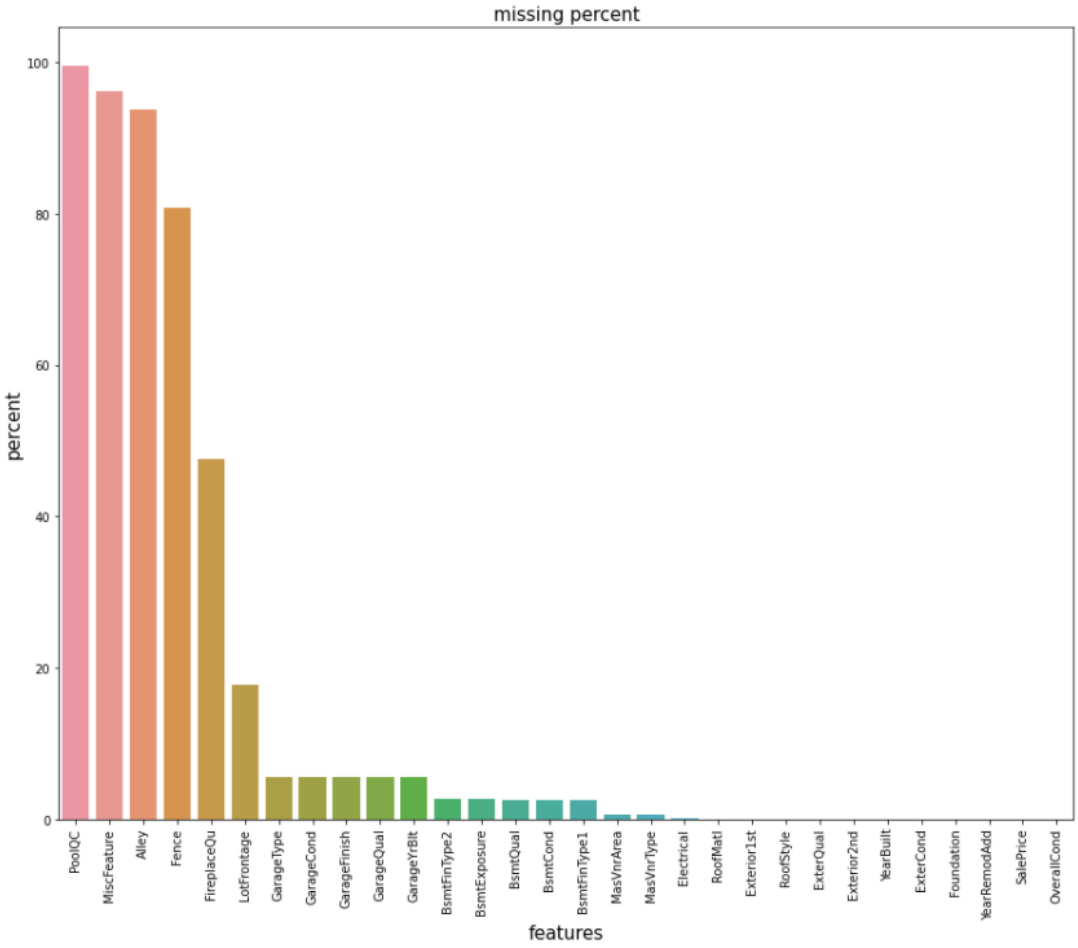

Figure 10. Number of missing values in each column.

As a result of checking for the missing values above, there is a significant amount of missing values in this dataset. Especially, PoolQC, MiscFeature, Alley, Fence columns show an overwhelmingly high rate of missing values. At first, I thought about deleting all the missing values, but I did not use this method because more than 90% of the missing values were found

in some columns. Thus, I filled the missing values using a substitution method suitable for each column according to the data type of each column.

```
db["PoolQC"] = db["PoolQC"].fillna("None")
db["MiscFeature"] = db["MiscFeature"].fillna("None")
db["Alley"] = db["Alley"].fillna("None")
db["Fence"] = db["Fence"].fillna("None")
db["FireplaceQu"] = db["FireplaceQu"].fillna("None")

for col in ('GarageType', 'GarageFinish', 'GarageQual', 'GarageCond'):
    db[col] = db[col].fillna('None')

for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
    db[col] = db[col].fillna(0)

for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'BsmtHalfBath'):
    db[col] = db[col].fillna(0)

for col in ('BsmtQual', 'BsmtCond', 'BsmtExposure', 'BsmtFinType1', 'BsmtFinType2'):
    db[col] = db[col].fillna('None')

db["MasVnrType"] = db["MasVnrType"].fillna("None")
db["MasVnrArea"] = db["MasVnrArea"].fillna(0)
db["Functional"] = db["Functional"].fillna("Typ")
db['MSSubClass'] = db['MSSubClass'].fillna("None")
```

Figure 11. Insert 0 and None values in the missing values

```
# mean value replacements
db['MSZoning'] = db['MSZoning'].fillna(db['MSZoning'].mode()[0])
db['Electrical'] = db['Electrical'].fillna(db['Electrical'].mode()[0])
db['KitchenQual'] = db['KitchenQual'].fillna(db['KitchenQual'].mode()[0])
db['Exterior1st'] = db['Exterior1st'].fillna(db['Exterior1st'].mode()[0])
db['Exterior2nd'] = db['Exterior2nd'].fillna(db['Exterior2nd'].mode()[0])
db['SaleType'] = db['SaleType'].fillna(db['SaleType'].mode()[0])
db['LotFrontage'] = db['LotFrontage'].fillna(db['LotFrontage'].mode()[0])
```

Figure 12. Insert mean value in the missing values

For columns with string data, “None” is inserted for missing values, and 0 is inserted for columns with numeric data. Mean value imputation was used for columns that significantly affect model training when replaced with 0 or None.

```
db.isnull().sum().sum()
```

```
0
```

Figure 13. The number of missing values in the dataset after data insertion is completed

After removing all missing values, I converted the string data into numeric data as the last step of data preprocessing. The reason is that Scikit-learn's machine learning algorithm does not allow string values as input. Thus, the function I used to replace the string with the numeric is get_dummies(). In the Scikit-learn, the method used by get_dummies() is One-Hot-Encoding. One-Hot-Encoding is a method of adding new features according to the type of feature value, displaying 1 only in columns corresponding to unique values, and 0 for the rest.

	LotShape
0	Reg
1	Reg
2	IR1
3	IR1
4	IR1
5	IR1
6	Reg
7	IR1
8	Reg
9	Reg
10	Reg
11	IR1
12	IR2
13	IR1
14	IR1

ID	Reg (Regular)	IR1 (Slightly irregular)	IR2 (Moderately Irregular)	IR3 (Irregular)
0	1	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	1	0	0
4	0	1	0	0
5	0	1	0	0
6	1	0	0	0
7	0	1	0	0
8	1	0	0	0
9	1	0	0	0
10	1	0	0	0
11	0	1	0	0
12	0	0	1	0
13	0	1	0	0
14	0	1	0	0

Figure 14. Example of One-Hot-Encoding (LotShape)

Thus, the number of columns has increased due to the replacement of missing values in the One-Hot-Encoding method. For example, in LotShape above, the number of columns has increased from 1 to 4.

```
db = pd.get_dummies(db)
print(db.shape)
```

```
(1445, 302)
```

Figure 15. Number of columns after using One-Hot-Encoding

4. Testing machine learning models

Three models applied for supervised learning: MLP Regressor, Ridge Regressor, and XGB Regressor. I created two versions of the data before applying the model. First, the original version of the dataset is set to unnormalized data, and the conversion of existing data to normalized data is set as the normalized data version. The normalization method I used is min-max normalization. The reason for using this is that minimal-maximum normalization is the most common way to normalize data. For all features, we convert each minimum value of 0, maximum value of 1, and the others into values between 0 and 1. For example, if the minimum value of a characteristic is 10 and the maximum value is 50, 30 is converted to 0.5 because it is exactly in the middle. Then, I divided the dataset into a training set and a test set at a ratio of 8 to 2.

```
# db is an existing data set, db_normalized is a normalized data set.
target = db['SalePrice']
db = db[db.columns.difference(['SalePrice'])]
scaler = MinMaxScaler(feature_range=(0, 100))
scaler.fit(db)
db_normalized = scaler.transform(db)
```

Figure 16. create normalized data set

```
# Divide the training set and test set through sklearn's train_test_split function.
x_train, x_test, y_train, y_test = train_test_split(db, target, test_size=0.2,
                                                    shuffle=True,  random_state=34)
x_train_nor, x_test_nor, y_train_nor, y_test_nor = train_test_split(db_normalized, target,
                                                                    test_size=0.2, shuffle=True,
                                                                    random_state=34)
```

Figure 17. divide dataset into a training set and a test set

A. MLP Regressor

MLP Regression is short for multi-layer perceptron, which can be described as an upgraded version of logistic regression and artificial neural networks. MLP has the advantage of being able to better divide the division boundary in the form of adding a hidden layer between the input layer and output layer, which are logistic returns. In addition, by using multi-layer perceptron, it is possible to solve the problem in which a proper learning is not possible for nonlinearly separated data that cannot be solved with a single-layer perceptron. Therefore, I decided to use this model.

First, I compared the accuracy score between the normalized and unnormalized datasets.

```
# Unnormalized dataset
regressorMLP = MLPRegressor()
regressorMLP.fit(x_train, y_train)
y_pred = regressorMLP.predict(x_test)
print("Training set score: {:.2f}".format(regressorMLP.score(x_train, y_train)))
print("Test set score: {:.2f}".format(regressorMLP.score(x_test, y_test)))
```

```
Training set score: 0.70
Test set score: 0.69
```

Figure 18. unnormalized datasets score (MLP)

```
# normalized dataset
regressorMLP = MLPRegressor()
regressorMLP.fit(x_train_nor, y_train_nor)
y_pred = regressorMLP.predict(x_test_nor)
print("Training set score: {:.2f}".format(regressorMLP.score(x_train_nor, y_train_nor)))
print("Test set score: {:.2f}".format(regressorMLP.score(x_test_nor, y_test_nor)))
```

```
Training set score: 0.60
Test set score: 0.58
```

Figure 19. normalized datasets score (MLP)

The unnormalized data showed better results, so I proceeded to the unnormalized dataset. Then, I performed the activation parameter test.

```
# activation = identity
Training set score: 0.63
Test set score: 0.61


# activation = logistic
Training set score: -6.04
Test set score: -5.32


# activation = tanh
Training set score: -6.04
Test set score: -5.32


# activation = relu
Training set score: 0.69
Test set score: 0.68
```

Figure 20. Activation parameter test (MLP)

As a result of testing four activation parameter, relu showed the most accurate results, so I set relu as the activation parameter and tested the solver.

```
# solver= lbfgs
Training set score: 0.80
Test set score: 0.80


# solver= sgd
Training set score: -398246593408596574901647103141702983447571735191826427008415233462745601503522037653757939618600938535933608094200761493170792418636905852391026298366525717345172331345372637120991304494125133509833218080599769937967316992.00
Test set score: -327462736949123958976558440279613001886379426634649138752485787456843596352662043551421281208662791396264175857492386187468911933934107022165495315279031724050639116253089294051008871303458292705857587233770469610366173184.00


# solver= adam
Training set score: 0.69
Test set score: 0.69
```

Figure 21. Solver test (MLP)

lbfgs showed the best value, so I fixed it as the solver and tested the alpha value (L2

regularization parameter).

```
# alphas=0.01
Training set score: 0.80
Test set score: 0.80


# alphas=0.1
Training set score: 0.80
Test set score: 0.81


# alphas=1
Training set score: 0.80
Test set score: 0.80


# alphas=10
Training set score: 0.79
Test set score: 0.79


# alphas=100
Training set score: 0.80
Test set score: 0.79


# alphas=1000
Training set score: 0.80
Test set score: 0.80
```

Figure 22. alpha value test(MLP)

After completing the parameter test, I trained the MLP regression model with the best results. (dataset = unnormalized, activation = 'relu', solver = 'lbfgs', alpha = 100)

```
# The model to which the parameters with the best results were applied
regressorMLP = MLPRegressor(activation = 'relu',solver = 'lbfgs', alpha = 100)
regressorMLP.fit(x_train, y_train)
y_pred = regressorMLP.predict(x_test)
```

Figure 23. MLP model with best results of parameters

B. Ridge Regressor

Ridge Regression is a model that uses a normalization method to increase the predictive power of a linear model. Using the basic linear model is very suitable for frequently occurring overfitting, i.e., data, resulting in extremely fluctuating graphs, and the coefficient value of linear regression representing it is large. To prevent this situation, the ridge regression is a random small adjustment of the coefficient by adding a term. Therefore, it is a technique that can expect better results by minimizing errors with a penalizing term. After confirming that there are as many as 80 columns in this dataset and that the coefficients of each value are large, I chose to use the Ridge Regression model.

First, I compared the accuracy score between the normalized and unnormalized datasets.

```
# Unnormalized dataset
alphas = [0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1,0.5, 1, 10, 15, 20, 30, 40, 100]
regressorR = RidgeCV(alphas = alphas)
regressorR.fit(x_train, y_train)
y_pred = regressorR.predict(x_test)
print("Training set score: {:.2f}".format(regressorR.score(x_train, y_train)))
print("Test set score: {:.2f}".format(regressorR.score(x_test, y_test)))
```

```
Training set score: 0.93
Test set score: 0.91
```

Figure 24. unnormalized datasets score (Ridge)

```
# Normalized dataset
alphas = [0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1,0.5, 1, 10, 15, 20, 30, 40, 100]
regressorRR = RidgeCV(alphas = alphas)
regressorRR.fit(x_train_nor, y_train_nor)
y_pred = regressorRR.predict(x_test_nor)
print("Training set score: {:.2f}".format(regressorRR.score(x_train_nor, y_train_nor)))
print("Test set score: {:.2f}".format(regressorRR.score(x_test_nor, y_test_nor)))
```

```
Training set score: 0.94
Test set score: 0.90
```

Figure 25. normalized datasets score (Ridge)

The unnormalized dataset showed better results, so I proceeded with the unnormalized dataset. Then, I performed the alpha value test.

```
# alphas=0.0001
Training set score: 0.88
Test set score: 0.57


# alphas=0.001
Training set score: 0.94
Test set score: 0.90


# alphas=0.01
Training set score: 0.94
Test set score: 0.90


# alphas=0.1
Training set score: 0.94
Test set score: 0.90


# alphas=1
Training set score: 0.94
Test set score: 0.91


# alphas=10
Training set score: 0.93
Test set score: 0.91


# alphas=100
Training set score: 0.92
Test set score: 0.90
```

Figure 26. alpha value test (Ridge)

The alpha values from 0.0001 to 0.001 show relatively low results, while the alpha values from 0.001 to 0.9 show that the range of change in the result is not large and stable. After completing the parameter test, I trained the ridge regression model with the best results. (dataset = unnormalized, alpha = 100)

```
# The model to which the parameters with the best results were applied (unnormalized data and alpha value = 10)
regressorR = RidgeCV(alphas = [10])
regressorR.fit(x_train, y_train)
y_pred = regressorR.predict(x_test)
```

Figure 27. Ridge regression model with best results of parameters

C. XGB Regressor

XGB Regression is an ensemble method based on a decision tree, and among them, it is a machine learning technique based on the boosting method. It is also introduced as a good technique to prevent overfitting and has the advantage of supporting cross validation. So, I chose this model as the model to use for the project.

First, before comparing the accuracy scores of the normalized and unnormalized datasets, the default value of the XGB Regressor model was set as shown in the figure below.

```
#max_depth: Maximum tree depth for base learners (default: 3)
#learning_rate: Decide how much weight to use for each training step (default: 0.1)
#n_estimaters: Number of gradient boosted trees
#reg_alpha: L1 regularization term on weights
#reg_lambda: L2 regularization term on weights
#n_jobs: Number of parallel threads used to run xgboost. (default: -1 for use all cores of the computer)
#min_child_weight: Minimum sum of weights for all observations needed in child .

XGB = XGBRegressor(max_depth = 3,learning_rate = 0.1,n_estimators = 1000,reg_alpha = 0.001,reg_lambda = 0.000001,
                   n_jobs = -1,min_child_weight = 3)
XGB.fit(x_train,y_train)
y_pred = XGB.predict(x_test)
```

Figure 28. default setting for XGB Regressor

```
#Unnormalized dataset
print("Training set score: {:.2f}".format(XGB.score(x_train, y_train)))
print("Test set score: {:.2f}".format(XGB.score(x_test, y_test)))
```

```
Training set score: 1.00
Test set score: 0.92
```

Figure 29. unnormalized datasets score (XGB)

```
#Normalized dataset
print("Training set score: {:.2f}".format(XGB.score(x_train_nor, y_train_nor)))
print("Test set score: {:.2f}".format(XGB.score(x_test_nor, y_test_nor)))
```

```
Training set score: 0.19
Test set score: 0.19
```

Figure 30. normalized datasets score (XGB)

In the case of XGB Regression, normalized data show exceptionally low result values. Therefore, the test was conducted with an unnormalized dataset. And by changing the value of n-estimators, I adjusted the number of boosted trees and found the best value of the XGB model.

```
# n_estimators=10
Training set score: -0.02
Test set score: 0.05


# n_estimators=50
Training set score: 0.95
Test set score: 0.90


# n_estimators=100
Training set score: 0.97
Test set score: 0.91


# n_estimators=300
Training set score: 0.99
Test set score: 0.91


# n_estimators=600
Training set score: 1.00
Test set score: 0.92


# n_estimators=900
Training set score: 1.00
Test set score: 0.92


# n_estimators=1000
Training set score: 1.00
Test set score: 0.92
```

Figure 31. n-estimators (number of boost round) test

When n_estimator is 10, it goes down to an unpredictable level, and the score steadily rises to 100, and then stabilizes from 300. In the case of other parameters, only n_estimator was tested as they did not significantly affect the result value. What I can infer from here is that when learning_rate is low, overfitting is prevented only by increasing n_estimators. After completing the parameter test, I trained the XGB regression model with the best results. (dataset = unnormalized datasets, max_depth = 3, learning_rate = 0.1, n_estimators = 1000, reg_alpha = 0.001, reg_lambda = 0.000001, n_jobs = -1, min_child_weight = 3)

```
XGB = XGBRegressor(max_depth = 3,learning_rate = 0.1,n_estimators = 1000,reg_alpha = 0.001,reg_lambda = 0.000001,
                   n_jobs = -1,min_child_weight = 3)
XGB.fit(x_train,y_train)
y_pred = XGB.predict(x_test)
```

Figure 32. XGB regression model with best results of parameters

5. Conclusion

A. Result Table

	Mean of SalePrice	Mean Absolute Error (MAE)	Mean Squared Error (MSE)	Root Mean Squared Error (RMSE)	R-squared (R2)
MLP Regression	178395.914186851 2	25676.825170743752	1339127823.8277714	36594.095477655566	0.7877078683056064
Ridge Regression	178395.914186851 2	16342.555776968436	539413182.0742222	23225.270333716726	0.9144867485 76945
XGB Regression	178395.914186851 2	14712.978116890139	529691715.21889216	23015.032374926006	0.9160278941 533361

Figure 33. Result Table.

MAE is defined as the average absolute value of the errors that are the difference between the actual and predicted values.

$$\frac{1}{n}\sum_{i=1}^{n}(|y_i - \hat{y}_i|)$$

Figure 34. Formula of MAE

MSE is defined as the square mean of the errors that are the difference between the predicted and actual values.

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Figure 35. Formula of MSE

RMSE is the root of the MSE. Interpretation becomes somewhat easier because the error index is converted back to a unit similar to the actual value.

R2, also called coefficient of determination, is an indicator of how much the independent variable explains the variation of the dependent variable in the regression model.

I found the model with the highest accuracy in predicting housing prices using two of the above five indicators, R2 and MAE. The reason I chose R2 here is that R2 is a good indicator of the performance of the regression model in machine learning. R2 evaluates predictive performance based on variance and the closer the value is to 1, the higher the predictive accuracy. Both MAE and MSE are better models as the value approaches zero. However, MSE is sensitive to outliers because they need to be squared. Therefore, I chose MAE, which is less sensitive to outliers than MSE, as an indicator to compare models.

B. Graph comparison

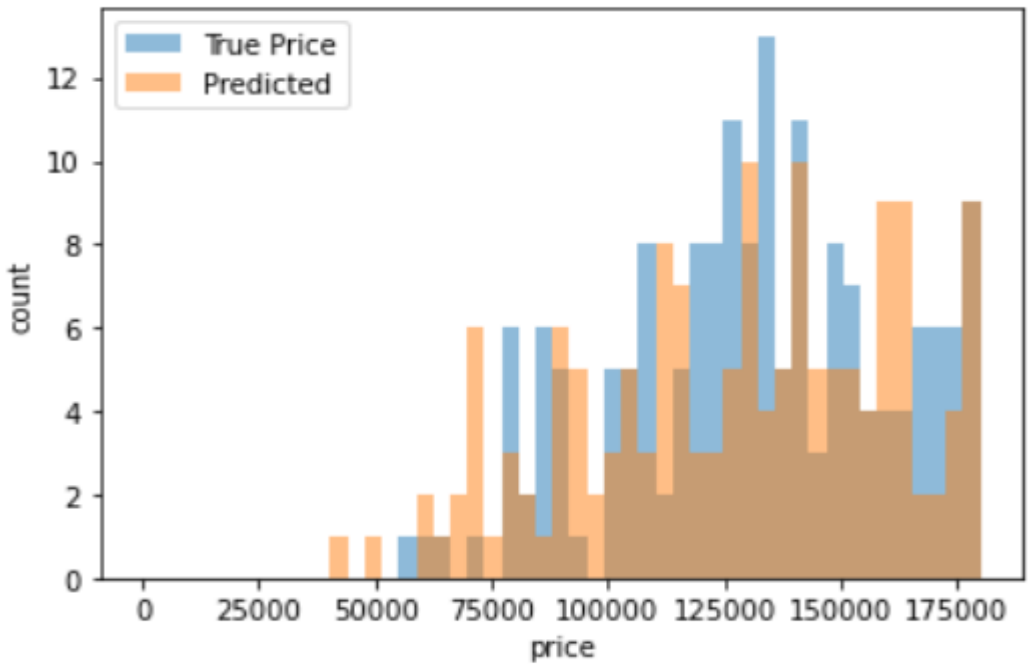

Figure 36. MLP model graph and housing price graph.

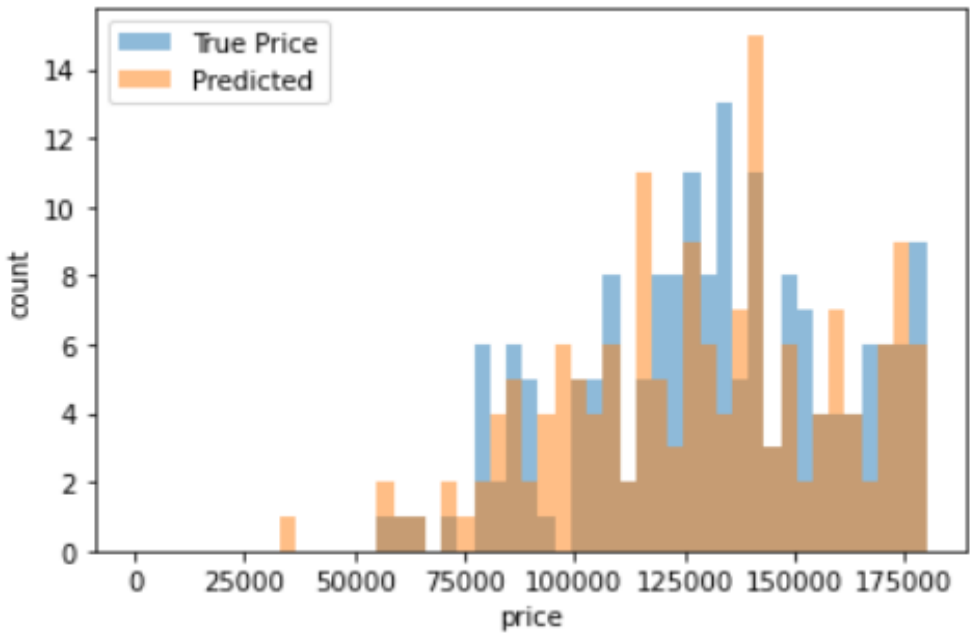

Figure 37. Ridge regression model graph and housing price graph.

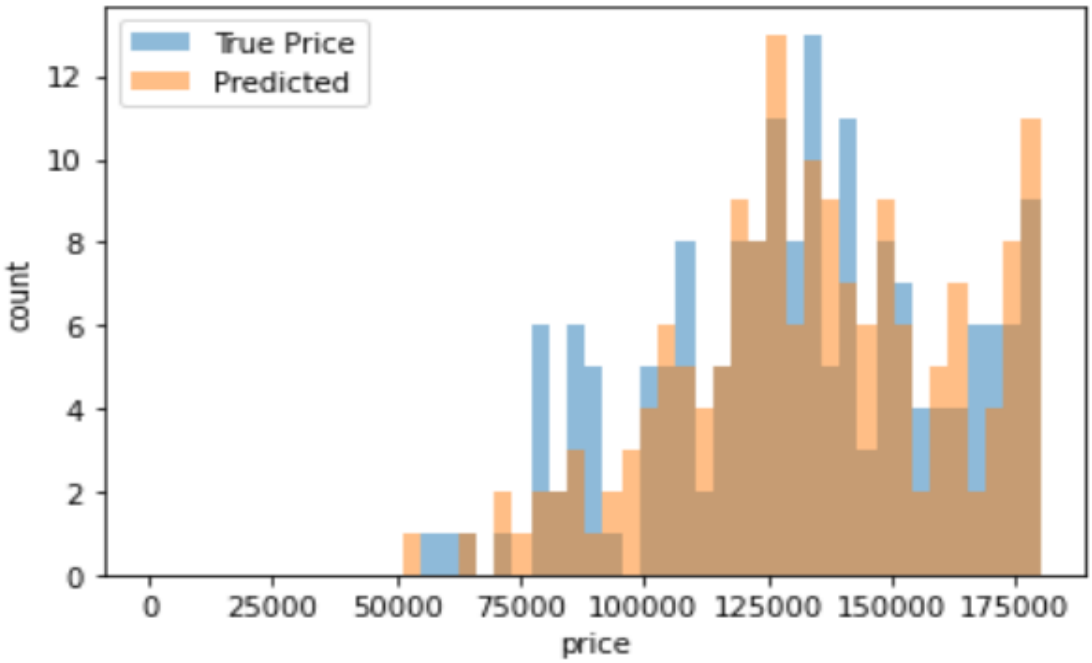

Figure 38. XGB regression model graph and housing price graph.

The graphs above (Figure 36-38) are graphs that visually show the difference between the actual house price and the predicted house price.

What can be seen from the comparison of the above three graphs are that we can find the correlation between the graph and the comparison indicators R2 and MAE of my choice. The graph of the MLP regression model has the least overlap with the graph of the target value among the three models. In contrast, for models Ridge regression and XGB regression with similar R2 and MAEs, they are significantly overlapping similar to the target value graphs.

C. Analysis

MLP regression model, Ridge regression model, and XGB regression model all showed better scores when data was not normalized. At first, I did not understand why that the test score for unnormalized datasets was higher because the goal of normalization was to ensure that all data points are reflected on the same scale. However, I realized that there was a fatal drawback to the min-max normalization I used. It was highly influenced by outliers. For example, there are 100 values, 99 of which are between 0 and 50, and what if the other is 100. All 99 values are then converted into values between 0 and 0.5.

I could have used the z-score normalization to address the issues caused by outliers. With z-score normalization, when the value of feature matches the mean, it will be normalized to 0. Nonetheless, looking at the target population provided by the dataset, the distribution is right-skewed, which indicates that the Gaussian assumption no longer holds and that the z-score normalization is not suitable for this dataset. Moreover, there was not a definite reason to eliminate the outliers other the reason that they were outliers. Unable to use z-score nor min-max normalization due to non-Gaussian distribution with outliers, I decided to proceed with the unnormalized data.

In the case of the MLP regression model, after normalization/non-normalization testing, parameter testing was performed to find a solver under activation, and then the alpha value was tested. The R2 score of the final model is 0.78 and MAE is 25677, which is low considering the average house price is 178,395. In the case of the Ridge regression model, alpha parameter testing was performed, and weight comparison was performed through the chart. An R2 score of 0.91 MAE is 16343 showed improved results compared to the MLP model. In the XGB model, you can see that a lot of parameters are applied, but the difference

in parameter values except for n_estimator does not have a big effect on the result, so only n_estimators were performed. R2 score of 0.91 showed the same results as Ridge, and MAE showed better results than Ridge regression model.

Among the three models, MLP regression model showed the lowest performance, and the XGB regression model showed the best score compared to Ridge regression model, although by a little. Thus, XGB regressor, which has the highest R2 value and the lowest MAE value, was selected as a model with high accuracy in predicting the house value.