House Price Prediction Using Regression - Report

An end-to-end analysis and modelling approach.

by: LIN XIAO & MIRUTHULA SIVAKUMAR

INTRODUCTION

The main goal is to predict the final sale price of houses.

- This involves understanding **the data, cleaning and transforming** it, and applying machine learning techniques.
- **Regression analysis** helps us find the relationships between features and the target variable (Sale Price).
- **Classification modelling** to categorize houses as expensive or affordable based on the median sale price.

Data Loading

- Used pandas to read both training and testing datasets from CSV files.
- Training data is used to build and validate the model.
- Testing data is used to evaluate model performance on unseen data.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
[ ] # Read test and training data from csv file
    test_data = pd.read_csv('test.csv')
    train_data = pd.read_csv('train.csv')
```

Part I: Regression: Predict sale price

1. Split Features and Target

- Prepares the training feature set X by excluding the target variable.

- We separate the input features (X_test) from the target (y_test)
- We apply a log transformation to SalePrice to normalize it
- House prices are often right-skewed. Taking the log makes the distribution more normal, which helps improve regression model performance.

```
# set read test Data set
X_test= test_data.drop('SalePrice', axis=1)
y_test = np.log(test_data['SalePrice'])
```

```
[ ] X_test
```

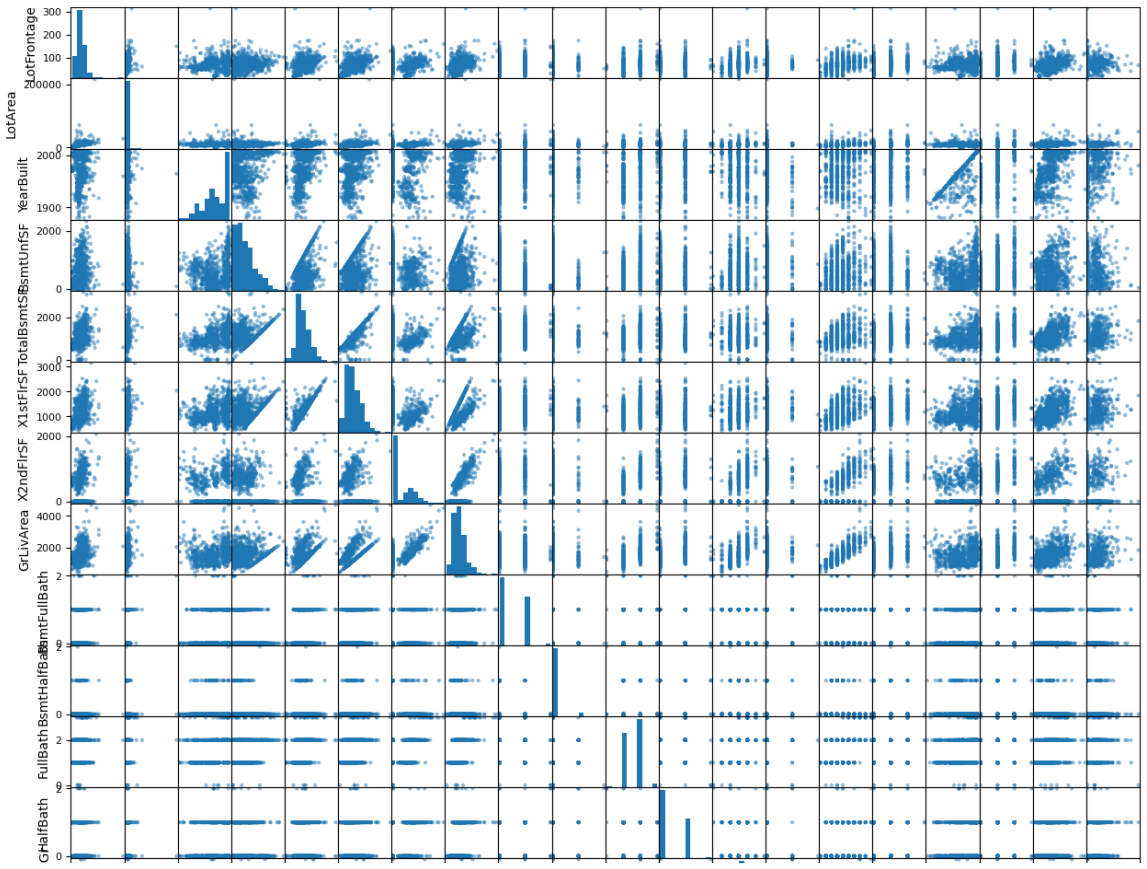
	LotFrontage	LotArea	Street	LotShape	YearBuilt	BsmtUnfSF	TotalBsmtSF	CentralAir	X1stFlrSF	X2ndFlrSF	...	FullBath	HalfBath	BedroomAbvGr	Kitchen
0	68	11250	Pave	IR1	2001	434	920	Y	920	866	...	2	1	3	
1	85	14115	Pave	IR1	1993	64	796	Y	796	566	...	1	1	1	
2	66	13695	Pave	Reg	2004	468	1114	Y	1114	0	...	1	1	3	
3	70	7560	Pave	Reg	1958	525	1029	Y	1339	0	...	1	0	3	
4	50	8500	Pave	Reg	1920	649	649	N	649	668	...	1	0	3	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
221	80	10721	Pave	IR1	1959	1252	1252	Y	1252	0	...	1	0	3	
222	60	21930	Pave	IR3	2005	732	732	Y	734	1104	...	2	1	4	
223	80	8400	Pave	Reg	1962	1319	1319	Y	1537	0	...	1	1	3	
224	70	8400	Pave	Reg	1966	0	814	Y	913	0	...	1	0	3	
225	80	10000	Pave	Reg	1995	141	1220	Y	1220	870	...	2	1	3	

226 rows x 23 columns

2. Scatter Matrix Visualization

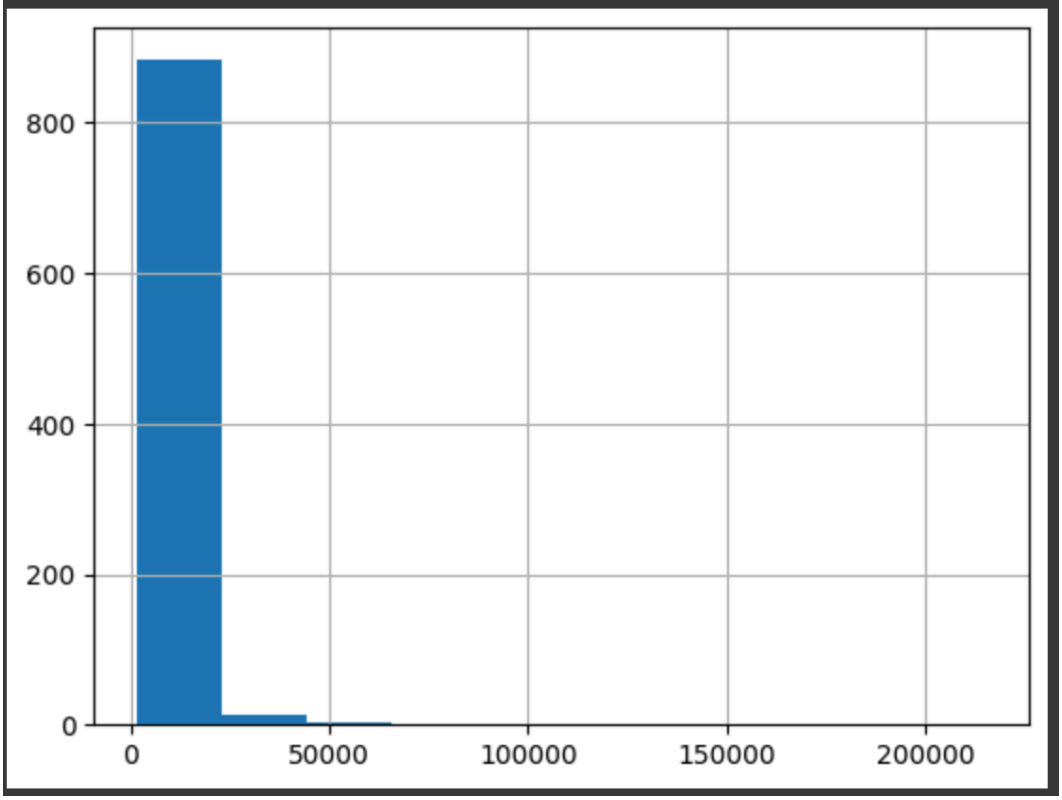
- Plots pairwise relationships between numeric features.
- Useful for identifying correlations, clusters, and potential outliers.

```
[5]  from pandas.plotting import scatter_matrix

     scatter_matrix(X, figsize=(15, 20))
     plt.show()
```



3. Histogram to Check Outliers in LotArea

```
# find the outliner data for LotArea
X['LotArea'].hist()

plt.show()
```



Displays a histogram of the LotArea feature to detect distribution and outlier

Count Outliers

- Prints how many entries have very large LotArea values.
- These outliers might need removal to avoid skewing the model.

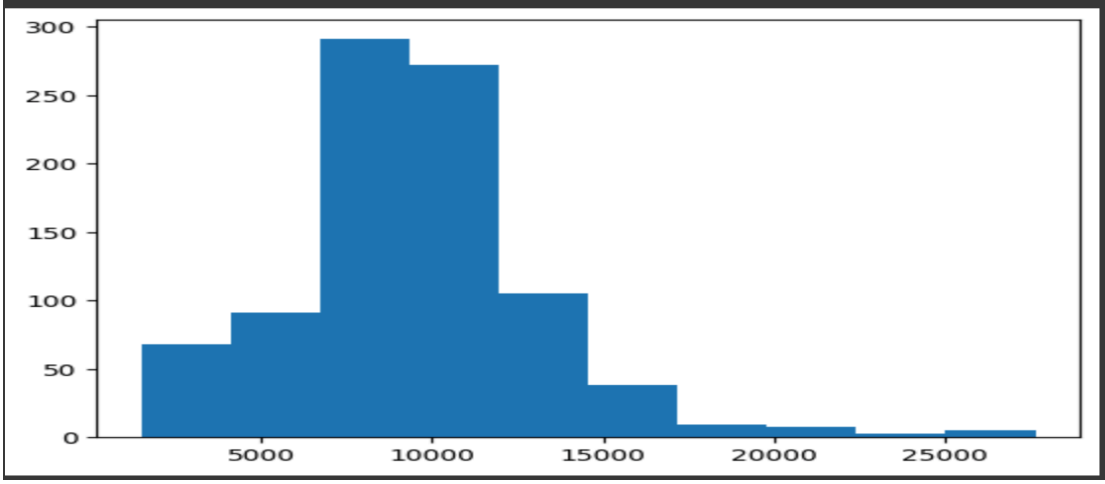
```
# Get and analyize  the data of LotArea

# check the data
print(f"The number of data LotArea that are greater than 50000 is {sum(X['LotArea']>50000)}")

print(f"The number of data LotArea that are greater than 30000 is {sum(X['LotArea']>30000)}")

train_data_New = train_data.copy()
```

```
The number of data LotArea that are greater than 50000 is 4
The number of data LotArea that are greater than 30000 is 11
```

Remove Outliers

- Makes a copy of the training dataset.Filters out rows where LotArea is greater than 50,000 to remove extreme outliers.

```
# keep the data row while LotArea less than 30000
train_data_New = train_data_New[train_data_New['LotArea']< 30000]
X_train = train_data_New.drop('SalePrice', axis=1)
y_train = np.log(train_data_New['SalePrice'])

plt.hist(X_train['LotArea'])
plt.show()
```



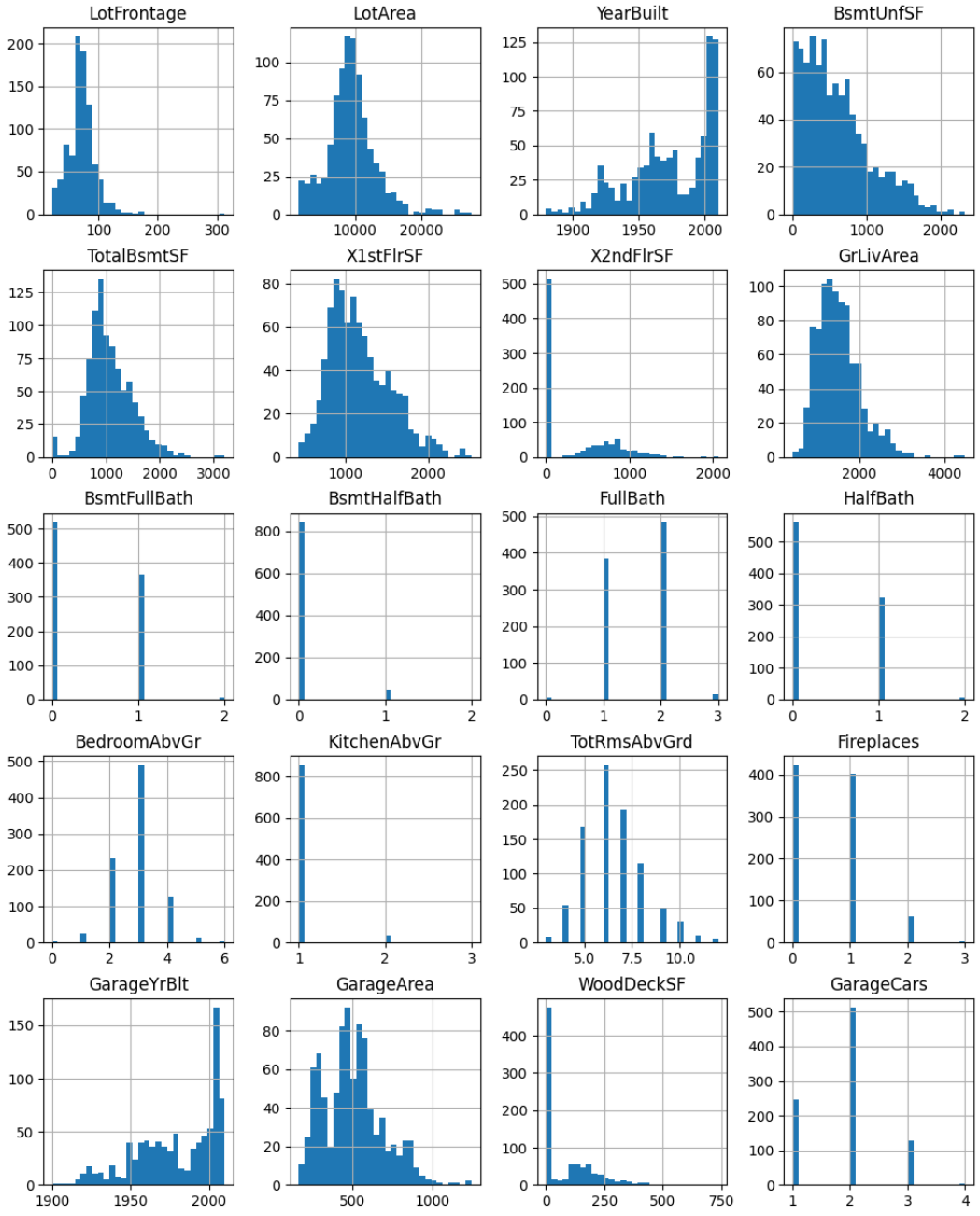
Splitting features into numeric and categorical helps in applying different preprocessing techniques

Use different preprocessing techniques:

- Standardization for numbers
- Encoding for categories
  This separation is essential for building pipelines

```
# prepare data and data standardize
# number_feature = X_train.select_dtypes(include=['int64','float64']).columns.tolist()
# cat_feature = X_train.select_dtypes(include=['object']).columns.tolist()

number_feature = ['LotFrontage', 'LotArea', 'YearBuilt', 'BsmtUnfSF', 'TotalBsmtSF',
    'X1stFlrSF', 'X2ndFlrSF', 'GrLivArea', 'BsmtFullBath', 'BsmtHalfBath', 'FullBath',
    'HalfBath', 'BedroomAbvGr', 'KitchenAbvGr', 'TotRmsAbvGrd', 'Fireplaces',
    'GarageYrBlt','GarageArea', 'WoodDeckSF',  'GarageCars']

cat_feature = ['Street','LotShape', 'CentralAir']


#plot all number feature
X_train[number_feature].hist(bins=30, figsize=(12,15))
```

- Many features like **LotArea**, **GrLivArea**, and **GarageArea** are **right-skewed** and may need **log transformation**for modeling.
- Features such as **KitchenAbvGr**, **BsmtHalfBath**, and **Fireplaces** show **categorical-like behavior** with few distinct values.
- Some features like **2ndFlrSF** and **WoodDeckSF** have a **large number of zeroes**, suggesting a good case for creating **binary flags** (e.g., "Has_2nd_Floor")



We use this to visually inspect which features are **skewed** or have **abnormal distributions**. It helps decide which columns need transformation, like taking the logarithm.

Apply Log Transformation to Skewed Numeric Features

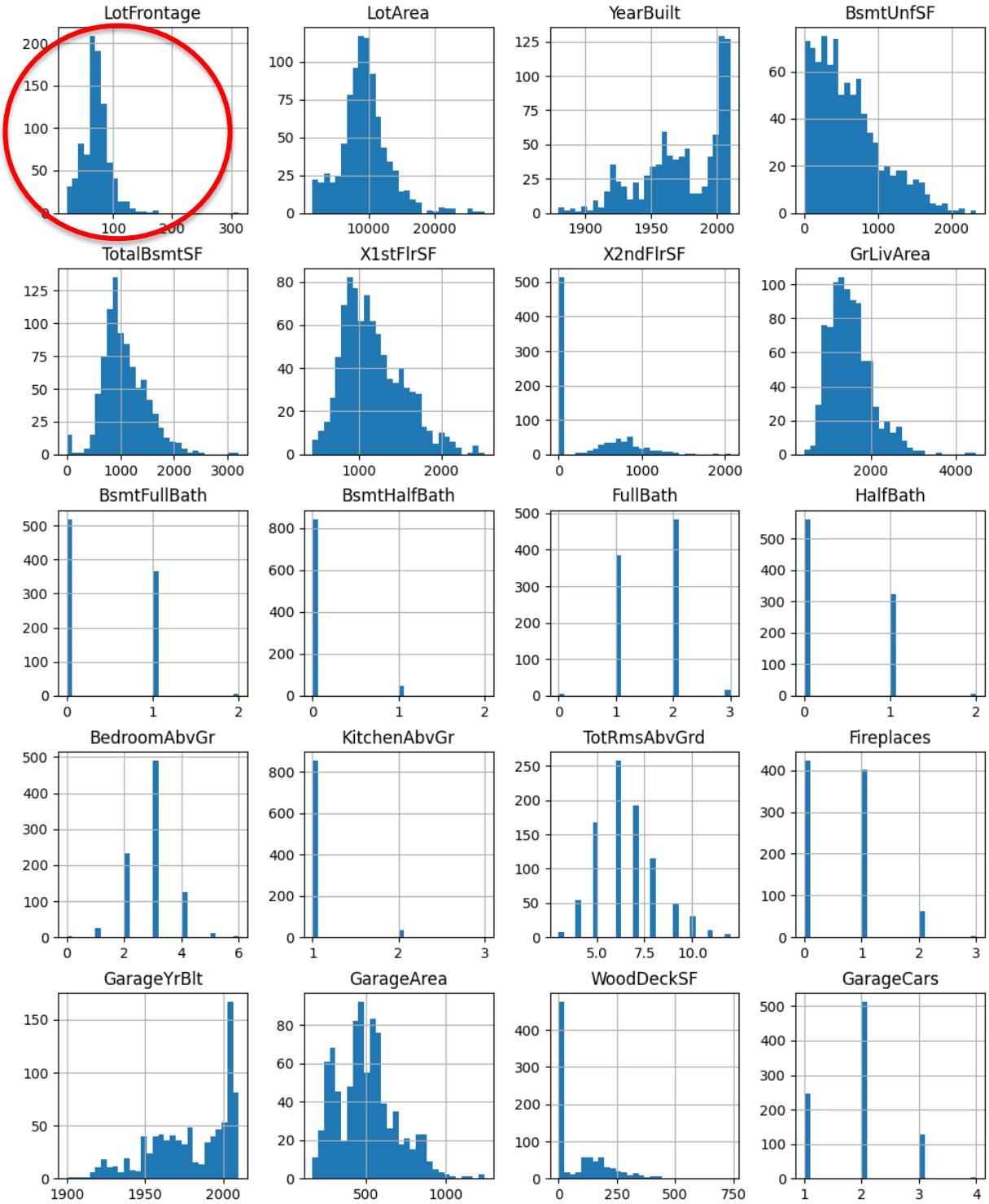
- Log transformation reduces the effect of **extreme values** and **skewness** in the data.
- It makes features more **normal-like**, which is important for linear models.

```
# list features need to log
log_features = [
    'LotFrontage', 'LotArea', 'BsmtUnfSF', 'TotalBsmtSF', 'YearBuilt',
    'X1stFlrSF', 'GarageYrBlt', 'GrLivArea', 'GarageArea', 'WoodDeckSF','X2ndFlrSF'
]
#  X_train  X_test
for col in log_features:
    X_train[col] = np.log1p(X_train[col])
    X_test[col] = np.log1p(X_test[col])
```

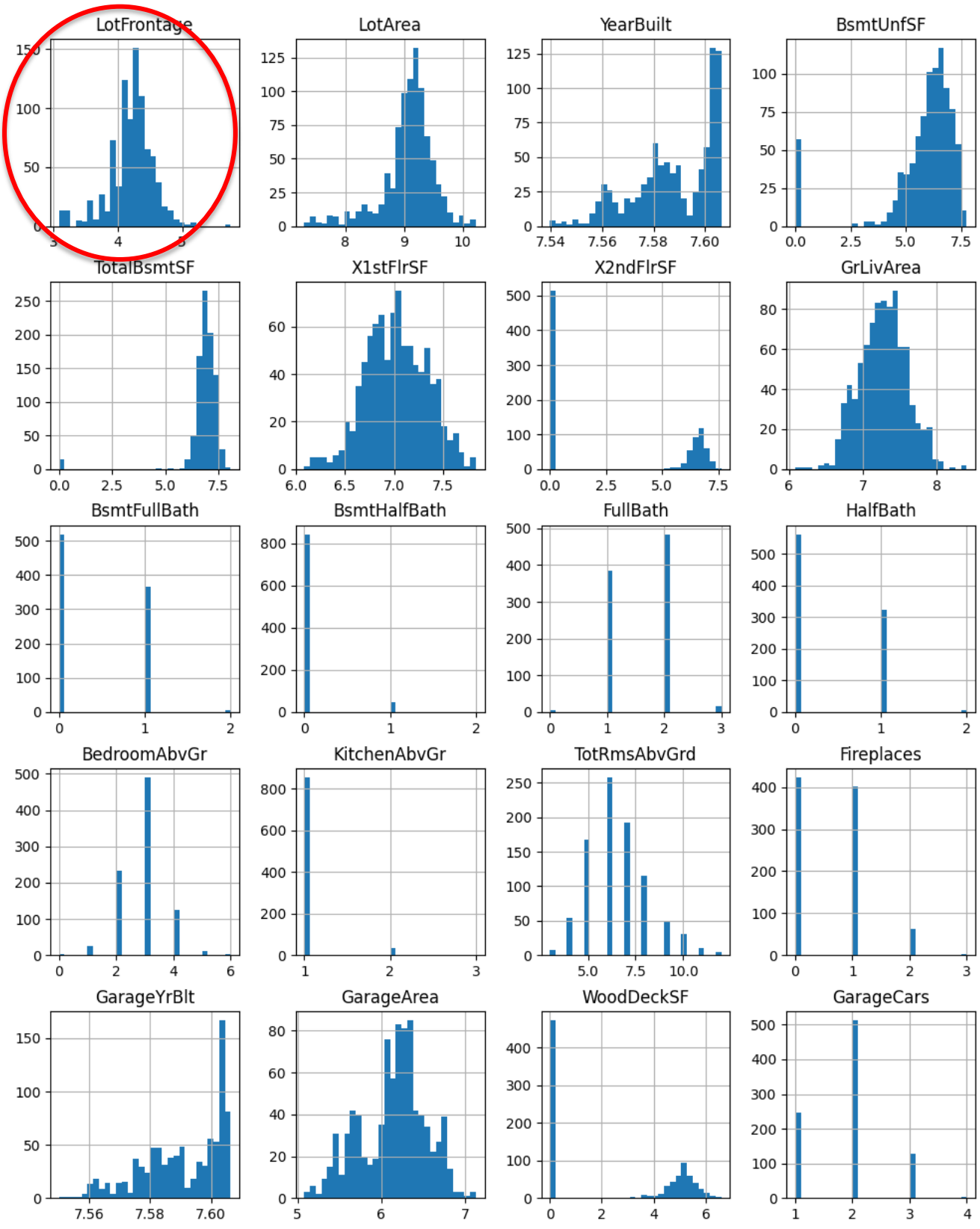
This block log-transforms selected features using `np.log1p()` ($\log(1 + x)$).

Recheck Histograms After Transformation

BEFORE Transformation



Updated Histograms After Transformation



Preprocessing Pipelines

- Numerical Pipeline : Fills missing values with the mean.
- Categorical Pipeline: Fills missing values with the most frequent value.
- One-hot encodes categorical features.

- Column Transformer combines both and is applied to both training and test sets.

```
# preprocessor pipeline
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# numbers
num_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
     ('scaler', StandardScaler())
])

# object
cat_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown = 'ignore'))
])

preprocessor = ColumnTransformer(transformers = [
     ('num', num_transformer, number_feature),
    ('cat', cat_transformer, cat_feature),
])

x_prepared = preprocessor.fit_transform(X_train)
test_x_prepare = preprocessor.transform(X_test)
```

Different types of features need different handling. Pipelines allow us to **automate preprocessing** in a clean and modular way.

x_prepared = preprocessor.fit_transform(X_train)

test_x_prepare = preprocessor.transform(X_test)

We must ensure the model gets the **same type of preprocessed input** from both datasets. This keeps things consistent.

II. (Model-0) Use linear regression to fit the training data, then calculate the training and test MSE.

Linear regression is a baseline model. It assumes a straight-line relationship between features and the target. Simple, yet powerful for comparison.

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

linear_reg = LinearRegression()
linear_reg.fit(x_prepared, y_train)

# training MSE
y_train_pred_lin = linear_reg.predict(x_prepared)
lin_train_mse = np.sqrt(mean_squared_error(np.exp(y_train), np.exp(y_train_pred_lin)))
lin_train_logmse = np.sqrt(mean_squared_error(y_train, y_train_pred_lin))

# Test MSE
y_test_pred_lin = linear_reg.predict(test_x_prepare)
lin_test_mse = np.sqrt(mean_squared_error(np.exp(y_test), np.exp(y_test_pred_lin)))
lin_test_logmse = np.sqrt(mean_squared_error(y_test, y_test_pred_lin))

print(f"Training RMSE(SalePrice): ${lin_train_mse:.2f}")
print(f"Training RMSE(log(SalePrice)): ${lin_train_logmse:.4f}")

print(f"Testing RMSE(SalePrice): ${lin_test_mse:.2f}")
print(f"Testing RMSE(log(SalePrice)): ${lin_test_logmse:.4f}")

print(f"Model configerition:{linear_reg.coef_}")
```

```
Training RMSE(SalePrice): $32038.94
Training RMSE(log(SalePrice)): $0.1511
Testing RMSE(SalePrice): $48915.20
Testing RMSE(log(SalePrice)): $0.1891
Model configerition:[ 0.0217955   0.02141211  0.05378825 -0.01666757  0.05743106  0.01176066
  0.01661411  0.14870989  0.02939151  0.00701735  0.03579094  0.01266448
 -0.06262359 -0.06329481  0.0421845   0.02937875  0.02816089 -0.01115766
  0.00618491  0.05944416 -0.20892593  0.20892593 -0.00215666  0.0633647
 -0.03052133 -0.03068671 -0.05017822  0.05017822]
```

- The provided code fits a Linear Regression model to predict house prices using log-transformed SalePrice as the target variable to account for skewness.
- After training on the prepared feature set, the model's performance is evaluated using Root Mean Squared Error (RMSE) on both the original SalePrice scale and the log-transformed scale.
- The training RMSE values indicate a reasonably good fit, while the higher test RMSE values suggest mild overfitting.
- This analysis helps assess model generalization and the benefits of working in the log scale for more stable and meaningful predictions in real estate price modeling.

Lasso Regression with Grid Search

- Lasso Regression introduces regularization by penalizing large coefficients. This helps prevent overfitting and performs automatic feature selection by zeroing out less important predictors. Grid search optimizes the regularization strength (alpha), fine-tuning the model for performance.
- Trains a Lasso model with L1_ratio regularization, which performs feature selection by shrinking less important feature weights to zero.
- Uses GridSearchCV to find the optimal alpha value.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Lasso

# creat Lsso model
lasso = Lasso(max_iter = 10000)

# params = {"alpha": np.logspace(-4, 1, 30)}
params = {"alpha": np.linspace(0.01, 1, 10)}

# put cv as integer no shuffle
grid_lasso = GridSearchCV(lasso, params, cv =5)

grid_lasso.fit(x_prepared, y_train)

best_alpha = grid_lasso.best_params_['alpha']
best_lasso = grid_lasso.best_estimator_

print(f"Best alpha : {best_alpha:.5f}")
```

```
Best alpha : 0.01000
```

- Evaluated using Root Mean Squared Error (RMSE) on both the original SalePrice and its log-transformed version.

```
# training and test RMSE
y_train_pred_lasso = best_lasso.predict(x_prepared)
lasso_train_logmse = np.sqrt(mean_squared_error(y_train, y_train_pred_lasso))
lasso_train_mse = np.sqrt(mean_squared_error(np.exp(y_train), np.exp(y_train_pred_lasso)))

y_test_pred_lasso = best_lasso.predict(test_x_prepare)
lasso_test_logmse = np.sqrt(mean_squared_error(y_test, y_test_pred_lasso))
lasso_test_mse = np.sqrt(mean_squared_error(np.exp(y_test), np.exp(y_test_pred_lasso)))

print(f"Training RMSE(SalePrice): ${lasso_train_mse:.2f}")
print(f"Training RMSE(log(SalePrice)): ${lasso_train_logmse:.4f}")

print(f"Testing RMSE(SalePrice): ${lasso_test_mse:.2f}")
print(f"Testing RMSE(log(SalePrice)): ${lasso_test_logmse:.4f}")
```

```
Training RMSE(SalePrice): $35350.12
Training RMSE(log(SalePrice)): $0.1591
Testing RMSE(SalePrice): $46456.64
Testing RMSE(log(SalePrice)): $0.1882
```

- On the training data, the model achieves an RMSE of $33,550.12, while the log-transformed RMSE is 0.1591, indicating a good fit.
- when tested on unseen data, the RMSE increases to $46,456.64, and the log-transformed RMSE rises to 0.1882.
- This increase suggests a slight overfitting, where the model performs better on training data than on the test set.
- The log RMSE values remain relatively low, indicating the model generalizes reasonably well.
- Further improvements might be achieved through hyperparameter tuning, advanced feature engineering, or experimenting with more complex models like ensemble methods.

Use the Elastic Net model to fit the training data. Then, calculate the training and test MSE.

```
from sklearn.linear_model import ElasticNet

# creat Elastic Net model

elstNet = ElasticNet()

params1 = {
    'alpha': np.linspace(0.01, 1, 30),
    'l1_ratio': np.linspace(0.01, 0.99, 10)
}


# put cv as integer no shuffle
grid_eNet = GridSearchCV(elstNet, params1, cv =5)

grid_eNet.fit(x_prepared, y_train)

best_alpha = grid_eNet.best_params_['alpha']
best_ratio = grid_eNet.best_params_['l1_ratio']
best_alpha, best_ratio
```

```
(np.float64(0.04413793103448276), np.float64(0.01))
```

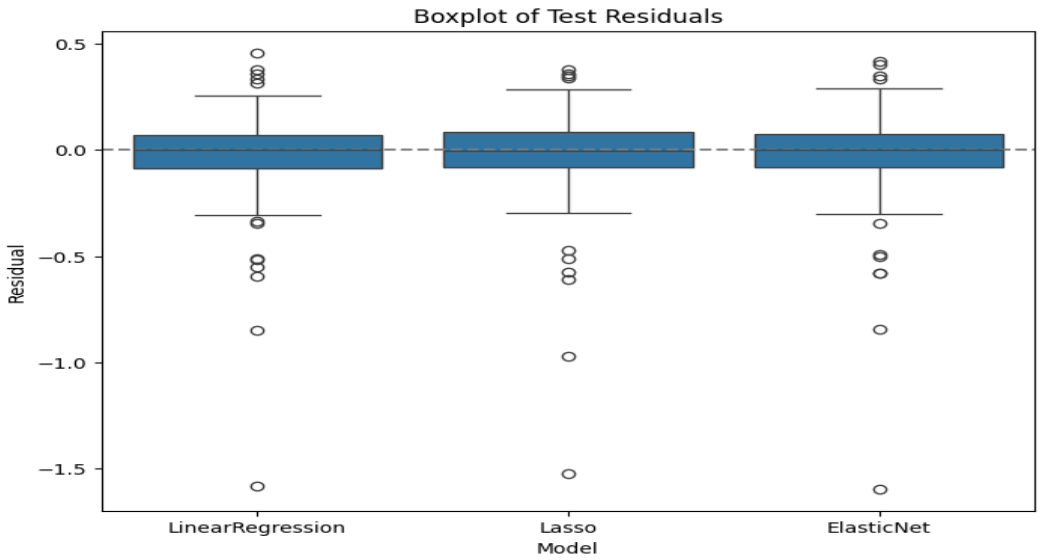
- Trains an ElasticNet model using L1_ratio regularization. It blends the strengths of both Lasso and Ridge Regression.

Performance of the three models

- To assess model effectiveness, residuals are compared visually using a boxplot. Narrow boxes and fewer outliers indicate more stable models.
- This analysis highlights which model generalizes best on unseen data.
- Lasso often strikes a good balance between bias and variance.

```
import seaborn as sns
# DataFrame
residuals_df = pd.DataFrame({
    'LinearRegression': y_test - y_test_pred_lin,
    'Lasso': y_test - y_test_pred_lasso,
    'ElasticNet': y_test - y_test_pred_eNet
})

residuals_long = residuals_df.melt(var_name='Model', value_name='Residual')

# plot
plt.figure(figsize=(8, 6))
sns.boxplot(x='Model', y='Residual', data=residuals_long)
plt.title("Boxplot of Test Residuals")
plt.axhline(0, color='gray', linestyle='--')
plt.show()
```

- Gathers and compares the prediction errors (residuals) from all three models visualised using a boxplot.



- This allows for a visual comparison of model performance.

- A tighter box with fewer outliers means a more stable and accurate model.

Part II: Classification (Model-3))

- Create a binary response by determining whether SalePrice is greater than its median
- Converting SalePrice into a binary target (above or below median) transforms the problem into a classification task.
- It simplifies decision-making for buyers and sellers alike.
- 1 if the price is above the median
- 0 if it's below

```
# set test and training y values
# X_test_cls = x_prepared.copy()
y_train_cls = (train_data_New['SalePrice'] > np.median(train_data_New['SalePrice'])).astype(int)

# X_train_cls = test_x_prepare
y_test_cls = (test_data['SalePrice'] > np.median(test_data['SalePrice'])).astype(int)
```

Use Logistic regression model to fit the training data, then calculate the confusion matrix, precision, and recall for training and test data.

- Fits a Logistic Regression model using the same features as before, but with the new classification target.

```
# build and fit model
from sklearn.linear_model import LogisticRegression


log_reg = LogisticRegression()
log_reg.fit(x_prepared, y_train_cls)
```

Predict and Evaluate with Confusion Matrix

```
from sklearn.metrics import confusion_matrix

# confusion metrix for training set
y_train_pred_lr = log_reg.predict(x_prepared)

matrix_train_lr = confusion_matrix(y_train_cls, y_train_pred_lr)

# confusion metrix for testing set
y_test_pred_lr = log_reg.predict(test_x_prepare)

matrix_test_lr =confusion_matrix(y_test_cls, y_test_pred_lr)
matrix_train_lr,matrix_test_lr
```

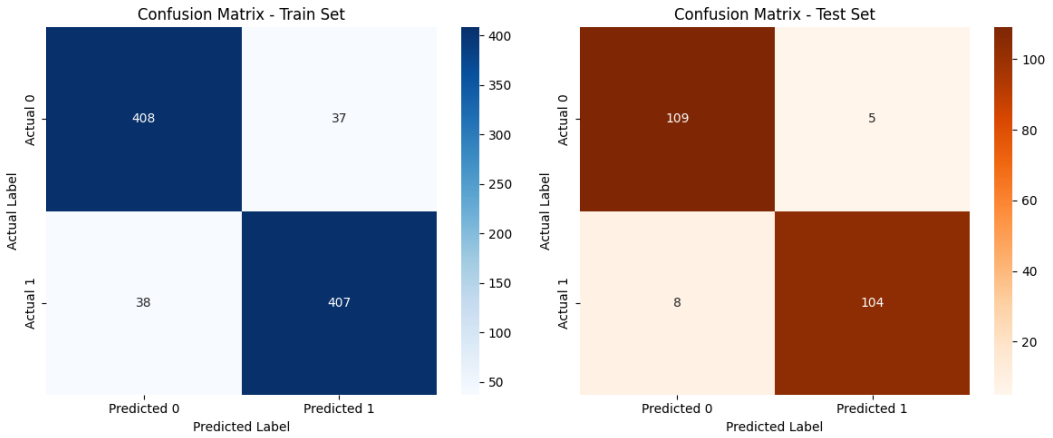
```
(array([[408,  37],
        [ 38, 407]]),
 array([[109,   5],
        [  8, 104]]))
```

A confusion matrix shows how many predictions were correct or incorrect:

- True Positives (TP) False Positives (FP) ,
- True Negatives (TN) False Negatives (FN)

Uses heatmaps to visually present the training and test confusion matrices.

- This makes it easier to interpret how well the model is classifying expensive vs. affordable homes.



Calculate Precision and Recall

- Calculates two classification performance metrics:
- **Precision:** How many predicted “expensive” houses are expensive
- **Recall:** How many of the truly expensive houses were correctly identified

```
# precision and recall
from sklearn.metrics import precision_score, recall_score

precision_test_lr = precision_score(y_test_cls, y_test_pred_lr)
recall_test_lr = recall_score(y_test_cls, y_test_pred_lr)

precision_train_lr = precision_score(y_train_cls, y_train_pred_lr)
recall_train_lr = recall_score(y_train_cls, y_train_pred_lr)

print(f"The precision for training is {precision_train_lr}and test data is {precision_test_lr}.")
print(f"The recall for training is {recall_train_lr}and test data is {recall_test_lr}.")
```

```
The precision for training is 0.9166666666666666and test data is 0.9541284403669725.
The recall for training is 0.9146067415730337and test data is 0.9285714285714286.
```

III. (Model-4) Use Logistic regression model with regularization (C parameter) to fit the training data

- This logistic regression includes L2 regularization, with a specific C value controlling the regularization strength.
- Performs a grid search to find the best value for C using 5-fold cross-validation.
- Selecting the optimal regularization parameter improves generalization and reduces errors on unseen data.

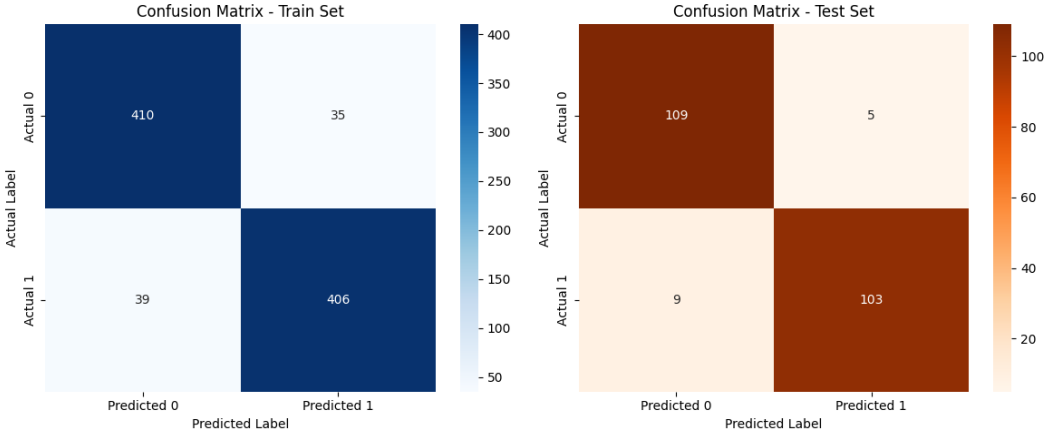
```
log_reg_C = LogisticRegression(C = 0.01, max_iter = 10000)
log_reg_C.fit(x_prepared, y_train_cls)
```

```
LogisticRegression(C=0.01, max_iter=10000)
```

```
# params = {"C": np.logspace(-3, 1, 10)}  # 0.001 ~ 10
params = {"C": np.linspace(0.01, 2, 10)}

# put cv as integer no shuffle
grid_logregC = GridSearchCV(log_reg_C, params, cv =5)

grid_logregC.fit(x_prepared, y_train_cls)

best_C = grid_logregC.best_params_['C']
best_logregC = grid_logregC.best_estimator_

print(f"Best C : {best_C:.5f}")
```

```
Best C : 0.23111
```

Calculate the confusion matrix, precision:



- The visual presents the **confusion matrices** for both the training and testing sets using a classification model (likely Logistic Regression).
- The training confusion matrix shows high accuracy, with the model correctly classifying 410 instances of class 0 and 406 instances of class 1, with only minor misclassifications (35 and 39).
- The testing matrix also displays strong predictive performance, with 109 and 103 correct predictions for class 0 and class 1, respectively, and minimal errors (5 and 9).
- These results indicate that the model generalizes well from training to testing, maintaining balanced classification performance across both datasets.

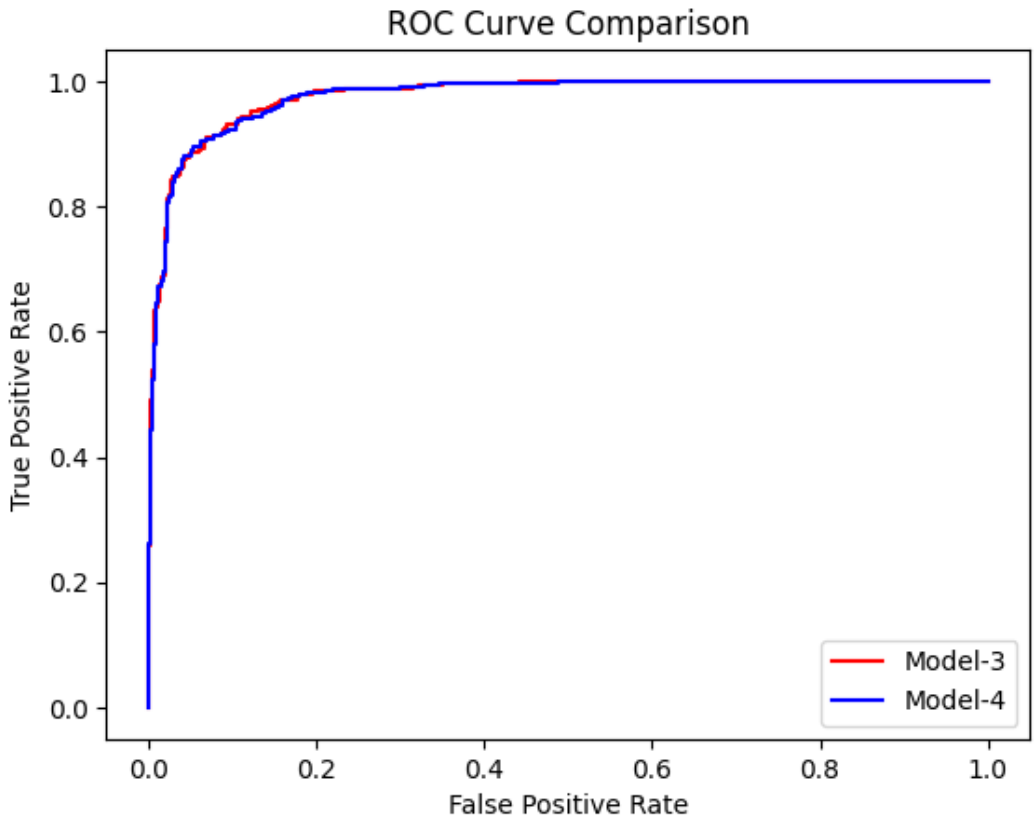
Precision and recall metrics

```
# precision and recall
precision_test_lrc = precision_score(y_test_cls, y_test_pred_lrc)
recall_test_lrc = recall_score(y_test_cls, y_test_pred_lrc)

precision_train_lrc = precision_score(y_train_cls, y_train_pred_lrc)
recall_train_lrc = recall_score(y_train_cls, y_train_pred_lrc)

recall_train_lrc, precision_train_lrc, recall_train_lrc, precision_train_lrc

print(f"The precision for training is {precision_train_lrc}and test data is {precision_test_lrc}.")
print(f"The recall for training is {recall_train_lrc}and test data is {recall_test_lrc}.")
```

```
The precision for training is 0.9206349206349206and test data is 0.9537037037037037.
The recall for training is 0.9123595505617977and test data is 0.9196428571428571.
```

- **Precision and recall metrics** for the classification model are calculated for both training and test sets.
- The **training precision** is approximately **0.9206**, and the **training recall** is **0.9124**, suggesting that the model is accurate in its predictions and also captures most relevant instances during training.
- On the **test set**, precision increases to **0.9537**, and recall remains high at **0.9196**, which indicates that the model maintains excellent performance on unseen data, with very few false positives and false negatives.
- These metrics demonstrate that the classifier is both **reliable and effective**, making it a well-performing model with minimal signs of overfitting.

ROC Curve Comparison



- This ROC curve compares the performance of two classification models—Model-3 in red and Model-4 in blue.
- Both curves rise steeply towards the top-left corner, which is a good sign because it means the models are achieving a high true positive rate while keeping the false positive rate low.
- When looked closely at the curves, noticed that they're almost overlapping, which tells both models are performing similarly.
- Model-4 (the blue one) appears to be just a bit more consistent or slightly better in some areas.
- Overall, both models are strong classifiers, but Model-4 may have a small advantage in terms of prediction accuracy depending on their AUC scores.

Conclusion

This project successfully demonstrates a complete machine learning pipeline for both **regression** and **classification** tasks using real-world housing data.

We started by performing data cleaning, handling missing values, and applying transformations like log-scaling to normalize skewed features. Using pipelines, we effectively standardized numeric features and encoded categorical ones, ensuring the data was ready for modeling.

For **regression**, we trained and evaluated models such as:

- **Linear Regression** (baseline),
- **Lasso Regression** (for automatic feature selection), and
- **ElasticNet Regression** (to balance bias and variance).

These models were evaluated using **Root Mean Squared Error (RMSE)**. The ElasticNet model offered strong generalization, while Lasso reduced overfitting by eliminating less important features.

For **classification**, we reframed the problem to predict whether a house was **expensive or affordable** based on the median price. We used **Logistic Regression** and improved it further through **hyperparameter tuning** with regularization. The models were evaluated using:

- **Confusion Matrices**
- **Precision & Recall Scores**
- **ROC Curves**

Overall, the project provided valuable insights into the housing dataset and showcased how different machine learning techniques can be applied to both predict exact prices and categorize homes. It highlights the importance of data preprocessing, model evaluation, and choosing the right algorithm based on the task at hand.