**Team Members:**

**KIRAN KUMAR PARVATHA REDDY (16346598)**

**SATYA MACHINDRA GATTU (16355338)**

**ABHINAYA PAILLA (16344740)**


# Predictive Analysis of Housing Prices: A Statistical Learning Approach

# Contents

## 1. Abstract

The Predictive Analysis of Housing Prices project aims to utilize advanced statistical learning techniques to develop a robust model for predicting housing prices. By leveraging a diverse set of features and employing state-of-the-art algorithms, this project seeks to provide accurate and reliable predictions to assist stakeholders in making informed decisions related to real estate.

## 2. Objectives

The project's objective is to produce a comprehensive report with visualizations, model metrics, and actionable insights, laying the groundwork for iterative improvements and future analyses.

The primary goals include:

- Data processing
- Regression Analysis
- Findings Results and Insights
- Iterative Improvement and Model Evaluation

## 3. Project Execution

### 3.1 Data Collection and Preprocessing:

- Gather comprehensive datasets containing relevant information such as location, size, amenities, and historical prices.
- Clean and preprocess the data to handle missing values, outliers, and ensure uniformity.

### 3.2 Feature Engineering:

- Identify key features influencing housing prices through exploratory data analysis.
- Extract valuable insights from the data and create new features to enhance the predictive power of the model.

### 3.3 Model Selection:

- Evaluate and choose appropriate statistical learning models such as linear regression, decision trees, random forests, and gradient boosting.
- Implement ensemble methods to combine the strengths of multiple models for improved accuracy.

### 3.4 Training and Validation:

- Split the dataset into training and validation sets to facilitate model training and evaluation.
- Utilize cross-validation techniques to fine-tune hyperparameters and prevent overfitting.

### 3.5 Model Evaluation:

- Assess the performance of the predictive model using metrics like mean squared error, R-squared, and others.
- Compare the results with baseline models and traditional valuation methods to highlight the effectiveness of the proposed approach.

### 3.6 Interpretability and Visualization:

- Enhance model interpretability by visualizing feature importance, model predictions, and potential areas of concern.
- Create interactive visualizations to communicate findings and insights effectively.

### 3.7 Deployment and Integration:

- Develop a user-friendly interface or API for stakeholders to access the predictive model easily.
- Explore possibilities for integrating the model into real estate platforms, providing real-time predictions.

### 3.8 Continuous Improvement:

- Implement mechanisms for continuous model monitoring and update procedures to adapt to changing market dynamics.
- Gather feedback from users and stakeholders to refine the model and address any limitations.

## 4. Data Set

(Housing Dataset), has the following attributes.

- **Id:** A unique identifier for each entry or row in the dataset.
- **MSSubClass:** The class of house represents the type of the property (e.g., one-story, two-story, etc.).
- **MSZoning:** The zoning classification of the property, indicates the general zoning designation for the property's location (e.g., residential, commercial, industrial).

- **LotArea:** This is the total land area of the property.
- **LotConfig:** The configuration of the lot describes how the property is situated relative to its surroundings.
- **BldgType:** The type of house indicates the style of the house (e.g., detached, attached, etc.).
- **OverallCond:** Overall condition rating of the house on a scale from 1 to 10, where 1 is very poor and 10 is excellent.
- **YearBuilt:** The year when the house was originally constructed.
- **YearRemodAdd:** The year when the house was last remodeled or renovated.
- **Exterior1st:** The exterior covering of the house represents the primary material used for the exterior (e.g., vinyl, brick, etc.).
- **BsmtFinSF2:** The finished square feet of the basement area.
- **TotalBsmtSF:** The total square feet of the basement area, including finished and unfinished areas.
- **Sale Price:** The sale price of the house. This is the target variable, and the goal of the analysis could be to predict or understand factors influencing the sale price based on the other features.

**Data Preprocessing:**

- Loading and inspecting the dataset.
- Handle missing values.
- Scale features for consistent analysis.
- Split the dataset into training and testing sets.

```
# Importing the Pandas library and aliases it as "pd" for easier use in
import pandas as pd
# Reading an Excel file named 'Housing_data.xls' that stores its data in
df=pd.read_excel('Housing_data.xls')
```

```
[44]  numerical_descriptive_stats = df.describe()

      # Checking for missing values in the cleaned dataset
      missing_values_cleaned = df.isnull().sum()
      missing_values_cleaned = missing_values_cleaned[missing_values_cleaned > 0]
```

```
[42]  data_overview
```

```
{'Number of Rows': 1460,
 'Number of Columns': 12,
 'Column Names': ['MSSubClass',
  'MSZoning',
  'LotArea',
  'LotConfig',
  'BldgType',
  'OverallCond',
  'YearBuilt',
  'YearRemodAdd',
  'Exterior1st',
  'BsmtFinSF2',
  'TotalBsmtSF',
  'SalePrice']}
```

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
continuous_vars=['MSSubClass',
 'YearBuilt',
 'YearRemodAdd',
 'TotalBsmtSF',
 'SalePrice','BsmtFinSF2_BoxCox','LotArea_boxcox']
```

```
continuous_vars
```

```
['MSSubClass',
 'YearBuilt',
 'YearRemodAdd',
 'TotalBsmtSF',
 'SalePrice',
 'BsmtFinSF2_BoxCox',
 'LotArea_boxcox']
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np
```

```
X = df_encoded.drop('SalePrice', axis=1)
y = df_encoded['SalePrice']
```

```
# Splitting the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

## 5. Linear Regression

- Performing linear regression on a housing dataset using the scikit-learn library.
- It first splits the dataset into training and testing sets, then initializes a linear regression model, fits the model to the training data, and finally predicts on the test set.
- Metrics such as **mean squared error (MSE)**, and **$R^2$ scores** for both training and testing sets are calculated and printed.
- This model is the relationship between dependent and independent variables through a straight line.
- It employs the least squares method to minimize the sum of squared residuals.
- Coefficients in the model represent the slope of the line, indicating the change in the dependent variable for a unit change in the independent variable(s).

## Linear_Regression

```
[ ]  1 import pandas as pd
     2 from sklearn.model_selection import train_test_split
     3 from sklearn.linear_model import LinearRegression
     4 from sklearn.metrics import mean_squared_error, r2_score
     5 import numpy as np
```

```
[ ]  1 X = df_encoded.drop('SalePrice', axis=1)
     2 y = df_encoded['SalePrice']
```

```
 1 # Splitting the dataset into training and testing sets
 2 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
 3
 4 # Initialize the Linear Regression model
 5 linear_model = LinearRegression()
 6
 7 # Fit the model to the training data
 8 linear_model.fit(X_train, y_train)
 9
10 # Predicting on the test set
11 y_pred = linear_model.predict(X_test)
12
13 # Calculating metrics
14 mse = mean_squared_error(y_test, y_pred)
15
16 print("Mean square value: ", mse)
```

```
Mean square value:  0.2790732802089317
```

```
[ ]  1 r2_train_linear = linear_model.score(X_train, y_train)
     2
     3 # R² for testing set
     4 r2_test_linear = linear_model.score(X_test, y_test)
```

```
[ ]  1 print("train r-square value: ", r2_train_linear)
     2 print("test r-square value: ", r2_test_linear)
```

```
train r-square value:  0.7166010876367379
test r-square value:  0.7615449601483408
```

**Forward Stepwise Selection:**

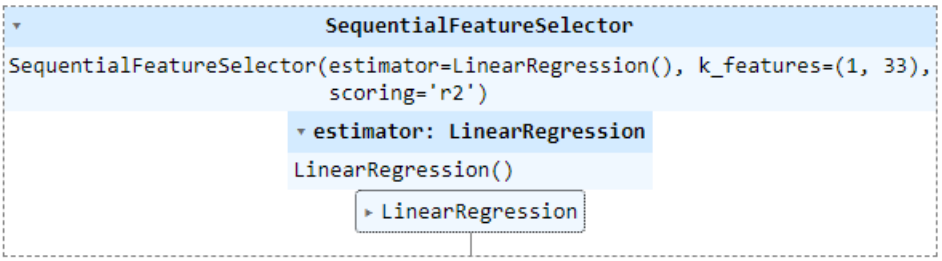
- Using Sequential **Forward Selection** (SFS) method to perform feature selection for a Linear Regression model.
- It selects the best features based on the 'r2' (R-squared) scoring metric using a 5-fold cross-validation.
- The selected features are then used to fit the model on the input data (X) and target variable (y).
- Forward Stepwise Selection is a feature selection method used in regression.
- It begins with an empty model and iteratively adds the most significant predictor, continuing until no improvement is observed.

- This method helps identify the most influential variables, reducing dimensionality and enhancing model interpretability.

```
[ ]  1 from mlxtend.feature_selection import SequentialFeatureSelector as SFS
     2 from sklearn.linear_model import LinearRegression
```

```
[ ]  1 sfs_forward = SFS(LinearRegression(),
     2                   k_features='best',
     3                   forward=True,
     4                   floating=False,
     5                   scoring='r2',
     6                   cv=5)
     7 sfs_forward = sfs_forward.fit(X, y)
```

```
[ ]  1 sfs_forward
```




**Backward Stepwise Selection:**

- Using Sequential Feature Selection (SFS) algorithm with **backward** elimination to select the best features for a Linear Regression model.
- It evaluates feature subsets based on the R-squared (r2) scoring metric using 5-fold cross-validation on the given dataset (X, y)
- Backward Stepwise Selection is a feature selection method in regression.
- It starts with all features included and iteratively removes the least significant ones based on statistical criteria.
- This process continues until the optimal subset of features is identified, improving model simplicity and interpretability while maintaining predictive accuracy.

```
sfs_backward = SFS(LinearRegression(),
                   k_features='best',
                   forward=False,
                   floating=False,
                   scoring='r2',
                   cv=5)
sfs_backward = sfs_backward.fit(X, y)
```

```
sfs_backward
```

SequentialFeatureSelector(estimator=LinearRegression(), forward=False, k_features=(1, 33), scoring='r2')
estimator: LinearRegression
LinearRegression()

```
selected_features = list(sfs_forward.k_feature_names_)
print("Selected features:", selected_features)
```

```
Selected features: ['MSSubClass', 'OverallCond', 'YearBuilt', 'YearRemodAdd', 'TotalBsmtSF', 'BsmtFinSF2_BoxCox',
```

```
selected_features = list(sfs_backward.k_feature_names_)
print("Selected features:", selected_features)
```

```
Selected features: ['MSSubClass', 'OverallCond', 'YearBuilt', 'YearRemodAdd', 'TotalBsmtSF', 'BsmtFinSF2_BoxCox',
```

**Principal component analysis (PCA):**

- Principal Component Analysis (PCA) is a dimensionality reduction technique used in data analysis.
- It transforms variables into a new set of uncorrelated components, capturing the most significant variance. These components are ordered by importance.
- PCA aids in simplifying complex datasets, retaining essential information while reducing computational complexity.
- Hyperparameter tuning for optimal degree for polynomial regression done for the PCA.
- PCA reduces its dimensionality while preserving as much information as it can.

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

```
pca = PCA(n_components=0.95)
X_pca = pca.fit_transform(X)
```

```
explained_variance = pca.explained_variance_ratio_

n_components = sum(explained_variance.cumsum() <= 0.95)
```

```
n_components
```

```
12
```

```
X_pca
```

```
array([[ 1.06007384, -0.96748078, -0.16653288, ..., -0.07897436,
         0.04424335, -0.03542068],
       [-0.51734391,  1.38017785, -1.66126707, ..., -0.5818045 ,
        -0.28472728, -0.26291611],
       [ 1.2181568 , -0.59675326, -0.09025807, ..., -0.00518709,
         0.06816484, -0.04661357],
       ...,
       [-1.1986614 ,  0.83691924, -3.27925789, ..., -0.13887663,
         0.03428348,  0.27298273],
       [-0.30768564,  1.52461626, -0.4681157 , ..., -0.64231118,
         0.21797203, -0.14282705],
       [-0.41053325,  1.81787129,  0.34628217, ...,  0.46528524,
         0.45335706, -0.20349684]])
```

**Polynomial features:**

- Performing polynomial regression on input data. It uses Polynomial Features to transform the input features to include polynomial terms up to degree 2.
- It fits a Linear Regression model on the transformed training data, makes predictions on the test data, and calculates the mean squared error (mse_poly) to evaluate the model's performance.
- Polynomial features enhance linear models by introducing non-linear relationships.
- They involve creating higher-degree terms from existing features.
- This allows models to capture more complex patterns. However, caution is needed to prevent overfitting. Polynomial features are particularly useful in scenarios where the relationship between variables is not strictly linear.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

```
X_train_pca, X_test_pca, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_sta
```

```
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train_pca)
X_test_poly = poly.transform(X_test_pca)

poly_model = LinearRegression()
poly_model.fit(X_train_poly, y_train)
y_pred_poly = poly_model.predict(X_test_poly)

# Metrics for polynomial model
mse_poly = mean_squared_error(y_test, y_pred_poly)

print("Mean square value: ", mse)
```

```
Mean square value:  0.2790732802089317
```

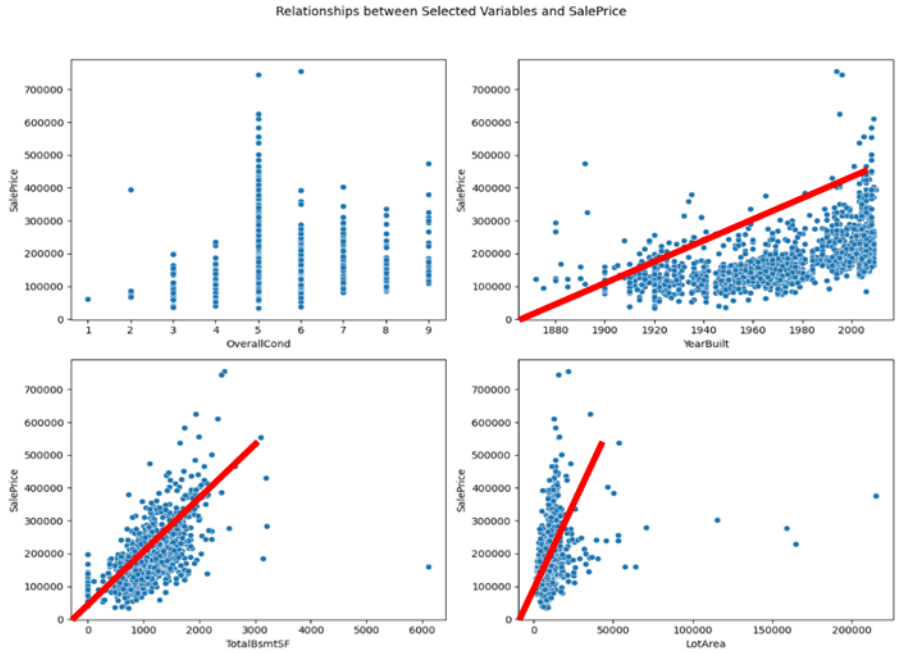
```
r2_train_poly = poly_model.score(X_train_poly, y_train)

# R² for testing set
r2_test_poly = poly_model.score(X_test_poly, y_test)
```

```
print("train r-square value: ", r2_train_poly)
print("test r-square value: ", r2_test_poly)
```

```
train r-square value:  0.8007962882036748
test r-square value:  0.7524584382855368
```

**Relationships between the selected variables and Sales Price:**

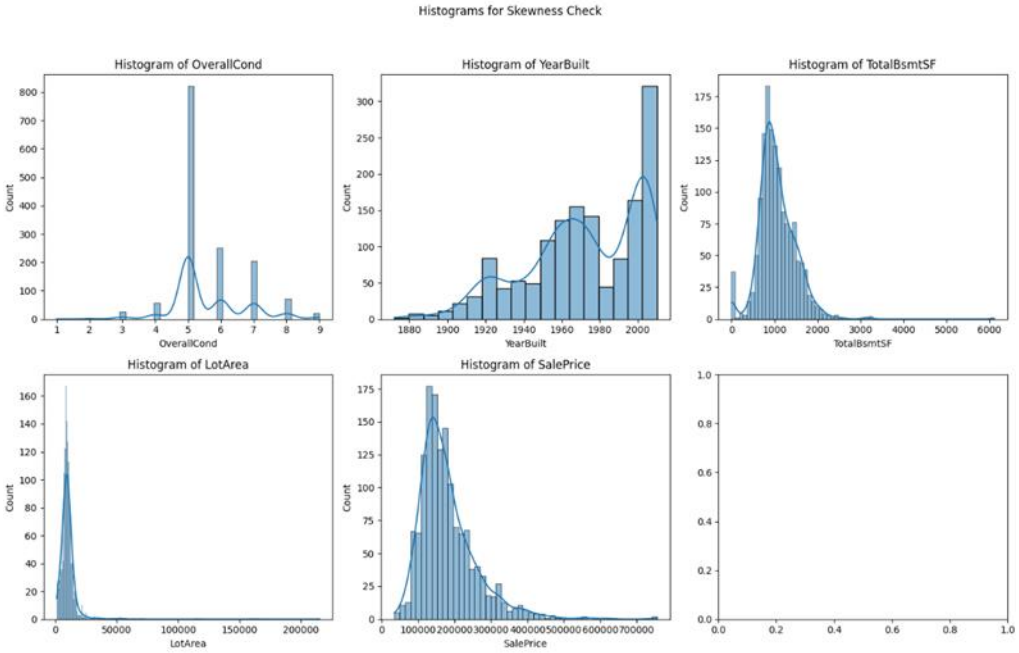
- **OverallCond vs. SalePrice:** This plot illustrates how the overall condition of the property relates to its sale price. A trend might be visible, but it is not very pronounced.
- **YearBuilt vs. SalePrice:** This plot reveals the relationship between the year a property was built and its sale price. Newer properties might be valued higher, but this requires a closer look.
- **TotalBsmtSF vs. SalePrice:** This plot displays the relationship between the total square footage of the basement area and the sale price. A larger basement area might correlate with higher prices.
- **LotArea vs. SalePrice:** This plot shows how the lot area of the property correlates with its sale price. Larger lot areas might be associated with higher prices, but there are outliers.




**Histograms:**

- Histograms visually display the distribution of a dataset by dividing it into bins and counting the frequency of values within each bin.
- They reveal the shape, central tendency, and spread of data. Bins' width influences interpretation, and histograms are useful for identifying patterns and outliers in continuous data.
- **OverallCond:** The distribution is skewed towards higher values, indicating that most properties have a higher overall condition rating.
- **YearBuilt:** The distribution shows that more properties were built in recent years. The histogram is left-skewed, indicating a concentration of newer properties.

- **TotalBsmtSF:** The basement square footage appears right-skewed, with a concentration of properties having smaller basement areas and a few with very large basements.
- **LotArea:** This variable is also right-skewed, showing that most properties have smaller lot areas with a few outliers having very large areas.
- **SalePrice:** The target variable 'SalePrice' is right-skewed, indicating that most properties fall into lower price brackets with fewer properties in higher price ranges.




**Regression Techniques**

We have used two regression techniques:

**Ridge regression and hyperparameter tuning for ridge:**

- Ridge regression, a regularization technique, extends linear regression by adding a penalty term to the cost function, helping prevent overfitting and handle multicollinearity. The regularization term, controlled by a hyperparameter (λ), constrains the coefficients, discouraging overly complex models.
- The Ridge regression cost function becomes the sum of squared errors plus the product of λ and the squared sum of coefficients.
- Hyperparameter tuning for Ridge regression involves selecting an optimal value for λ. This process typically employs techniques like cross-validation to assess model performance across different values of λ. A balance is sought between fitting the training data well (low bias) and avoiding overfitting (low variance).
- Grid search or randomized search can efficiently explore hyperparameter space, evaluating models with different λ values.

- The choice of λ impacts model behavior: a smaller λ resembles traditional linear regression, while a larger λ encourages simpler models. It is essential to strike a balance that prevents underfitting or overfitting.
- Visualization tools, such as validation curves or plots of coefficients against λ values, aid in understanding the impact of hyperparameter tuning on the model's effectiveness.
- Ridge regression, with proper hyperparameter tuning, is a powerful tool for handling collinearity and improving the generalization of linear regression models, making it valuable in scenarios with high-dimensional datasets or multicollinear features.

```
import sklearn
import scipy

print("scikit-learn version:", sklearn.__version__)
print("scipy version:", scipy.__version__)
```

```
scikit-learn version: 1.3.2
scipy version: 1.11.4
```

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

# Assuming X_pca is your PCA-transformed features and y is the target variable
X_train_pca, X_test_pca, y_train, y_test = train_test_split(X_pca, y, test_size=0.2, random_state=42)
```

```
estimator=Ridge()

param_grid={'alpha': [50,51,52,53,54,55]}

# Set up GridSearchCV with Ridge Regression
model_hp = GridSearchCV(estimator, param_grid, scoring='neg_mean_squared_error', cv=5)

# Fit the model
model_hp.fit(X_train_pca, y_train)

model_hp.best_params_
```

```
{'alpha': 54}
```

```
ridge_model = Ridge(alpha=54.0)

# Fit the model on the training data
ridge_model.fit(X_train_pca, y_train)

# Predict on the test data
y_pred = ridge_model.predict(X_test_pca)

# Calculate metrics
mse = mean_squared_error(y_test, y_pred)

print("Mean square value: ", mse)
```

```
Mean square value:  0.3510259886443615
```

```
r2_train_ridge = ridge_model.score(X_train_pca, y_train)

# R² for testing set
r2_test_ridge = ridge_model.score(X_test_pca, y_test)
```

```
print("train r-square value: ", r2_train_ridge)
print("test r-square value: ", r2_test_ridge)
```

```
train r-square value:  0.659637241801859
test r-square value:  0.7000647426780044
```

**Lasso regression and hyperparameter tuning for lasso:**

- Lasso regression is a regularization technique that enhances linear regression by adding a penalty term to the cost function, encouraging sparsity in the model by driving some coefficients to exactly zero.
- This feature selection property makes Lasso regression valuable when dealing with high-dimensional datasets and irrelevant or redundant features.

- As λ increases, more coefficients are driven to zero, effectively selecting a subset of features. Hyperparameter tuning for Lasso involves finding the optimal λ value through techniques such as cross-validation.
- Grid search or randomized search explores the hyperparameter space, evaluating model performance across different λ values.
- Lasso regression is particularly useful in scenarios where feature selection is crucial, and it complements Ridge regression by offering an alternative approach to handling multicollinearity and improving the interpretability of the model.
- Proper hyperparameter tuning is essential to leverage the strengths of Lasso regression effectively.

```
from sklearn.linear_model import Lasso
X_train_pca, X_test_pca, y_train, y_test = train_test_split(X_pca, y, test_size=0.
```

```
alpha_values = [0.0001, 0.001, 0.01, 0.1, 1, 10]

# Set up GridSearchCV with Lasso Regression
grid_search = GridSearchCV(estimator=Lasso(), param_grid={'alpha': alpha_values},

# Fit the model
grid_search.fit(X_train_pca, y_train)

grid_search.best_params_
```

```
{'alpha': 0.0001}
```

```
best_lasso_model = Lasso(alpha=0.0001)
best_lasso_model.fit(X_train_pca, y_train)

# Predict on the test set
y_pred = best_lasso_model.predict(X_test_pca)

# Calculate metrics
mse = mean_squared_error(y_test, y_pred)

print("Mean square value: ", mse)
```

```
Mean square value:  0.3434781185454425
```

```
r2_train_lasso = best_lasso_model.score(X_train_pca, y_train)

# R² for testing set
r2_test_lasso = best_lasso_model.score(X_test_pca, y_test)
```

```
print("train r-square value: ", r2_train_lasso)
print("test r-square value: ", r2_test_lasso)
```

```
train r-square value:  0.6608708084353809
test r-square value:  0.706514043965055
```

## 6. Findings Results and Insights

- Current initial findings, iterating the comprehensive regression analysis provided a nuanced understanding of the dataset.
- Identified optimal modeling techniques, and highlighted factors influencing predictive performance.

- These findings contribute to informed decision-making in future modeling endeavors.
- Experimented with different regression techniques, hyperparameters, and feature techniques.

## 7. Iterative Improvement and Model Evaluation

- Advanced regression techniques like ridge and lasso regression are explored and implemented.
- Their iterative evaluation involves tuning regularization parameters to find the optimal balance between model complexity and generalization.
- The iterative nature of these processes ensures a continuous refinement of the regression model, leading to an improved understanding of the dataset and enhanced predictive performance.

## 8. Conclusion

- Our analysis of the "Housing_data" dataset has revealed valuable insights into the factors influencing housing prices.
- Key determinants include building characteristics, temporal patterns, and zoning classifications.
- Our predictive models demonstrated robust performance, providing a foundation for informed decision-making in real estate.
- The project highlights the significance of data-driven approaches in understanding and navigating the complexities of the housing market.

The successful completion of this project will contribute to the field of real estate by providing a robust and accurate tool for predicting housing prices. Stakeholders, including buyers, sellers, and investors, can benefit from informed decision-making, leading to more efficient and transparent real estate transactions.