Analysis of House Price in Austin, TX

ql2466, yc4180, ys3391, cw3353

STAT 5291

David Rios

April 26

Abstract

The project mainly focuses on the analysis of house prices in the Austin area. Qingci Luo, Yisheng Chen, Chang Wang and Yiran Su work together to solve the problems as a group. We are trying to find what are the main features that influence the real estate sale price in Austin, as the Austin Housing market was one of the hottest markets in 2021, and these listings show how that market has changed over the past couple of years. The project includes the entire process of analyzing the data, cleaning the data, picking relevant features, modeling, and evaluation.

1. Introduction

The human's desire to have a favorable place to live is instinctive and understanding the factors that affect real estate prices has always been a significant concern. This report aims to investigate the relationship between a house's characteristics and its price using Austin, TX House Listings (Eric Pierce,2021). While the dataset offers a rich array of features, it can also be overwhelming to navigate. Therefore, the study will focus on exploring the potential connection between a property's price and factors such as architectural style, condition, size, location, and greenery. Two research questions will be mainly focused and answered based on the data to shed light on this relationship.

2. Research Questions

2.1 Which variables have the strongest relationship with the response variable Price?

2.2 Which model has better performance in predicting the response variable Price?

3. Data Exploration & Analysis

3.1 Data Description

Our dataset, Austin, TX House Listings, was downloaded from Kaggle. The original uncleaned dataset consisted of over 700 columns and Eric made this clean set of features and uploaded them on the website as sources. The dataset contains 15,171 listings with 45 features, one key column, and one image name which references an image in the home Images folder. We create a geographical visualization (Fig.1) to get a general idea of price density in Austin first.

These are some descriptions of variables we used in this project:

- latestPrice: The most recent available price at time of data acquisition.
- hasAssociation: Boolean indicating if there is a Homeowners Association associated with the listing.
- hasCooling: Boolean indicating if the home has a cooling system.
- hasGarage: Boolean indicating if the home has a garage.
- hasHeating: Boolean indicating if the home has a heating system.

- hasSpa: Boolean indicating if the home has a Spa.
- hasView: If the home comes with a view, determined by the property lister.
- zpid: Unique Identifier - Zillow Property ID.
- city: The lowercase name of a city or town in or surrounding Austin, Texas.
- streetAddress: The street address of the listing.
- zipcode: The listing 5-digit ZIP code.
- description: The description of the listing from Zillow.
- latitude/longitude: Latitude/Longitude of the listing.
- propertyTaxRate
- garageSpaces: Number of garage spaces. This is a subset of the ParkingSpaces feature.
- homeImage: The name of the first image from the home listing. This image file can be found in the included homeImages folder.
- latestPriceSource: The party that provided the sale price, includes Agent, broker and other sources.
- latest_saleyear/ latest_salemonth/latest_saledate: The latest sale year/month/date (YYYY-MM-DD).
- homeType: The home type (i.e., Single Family, Townhouse, Apartment).
- parkingSpaces: The number of parking spots that come with a home.
- yearBuilt: The year the property was built
- numPriceChanges: The number of price changes a home has undergone since being listed.
- numOfPhotos: The number of photos in the Zillow listing. Only the first photo is included in the homeImages folder.
- numOfAccessibilityFeatures: The number of unique accessibility features in the Zillow listing.
- numOfAppliances: The number of unique appliances in the Zillow listing.
- numOfParkingFeatures: The number of unique parking features in the Zillow listing
- numOfPatioAndPorchFeatures: The number of unique patio and/or porch features in the Zillow listing.

- numOfWaterfrontFeatures: The number of unique waterfront features in the Zillow listing.
- numOfWindowFeatures: The number of unique window aesthetics in the Zillow listing.
- numOfWindowFeatures: The number of unique window aesthetics in the Zillow listing.
- lotSizeSqFt: The lot size of the property reported in Square Feet.
- livingAreaSqFt: The living area of the property reported in Square Feet.
- numOfPrimarySchools/numOfElementarySchools/numOfMiddleSchools/numOfHighSchools: The number of Primary/Elementary/Middle/High schools listed in the area on the Zillow listing.
- avgSchoolDistance
- The average distance of all school types (i.e., Middle, High) in the Zillow listing.
- avgSchoolRating
- The average school rating of all school types (i.e., Middle, High) in the Zillow listing.
- avgSchoolSize
- The average school size of all school types (i.e., Middle, High) in the Zillow listing.
- MedianStudentsPerTeacher
- The median students per teacher for all schools in the Zillow listing.
- numOfBathrooms/Bedrooms/Stories: The number of bathrooms/bedrooms/stories in a property

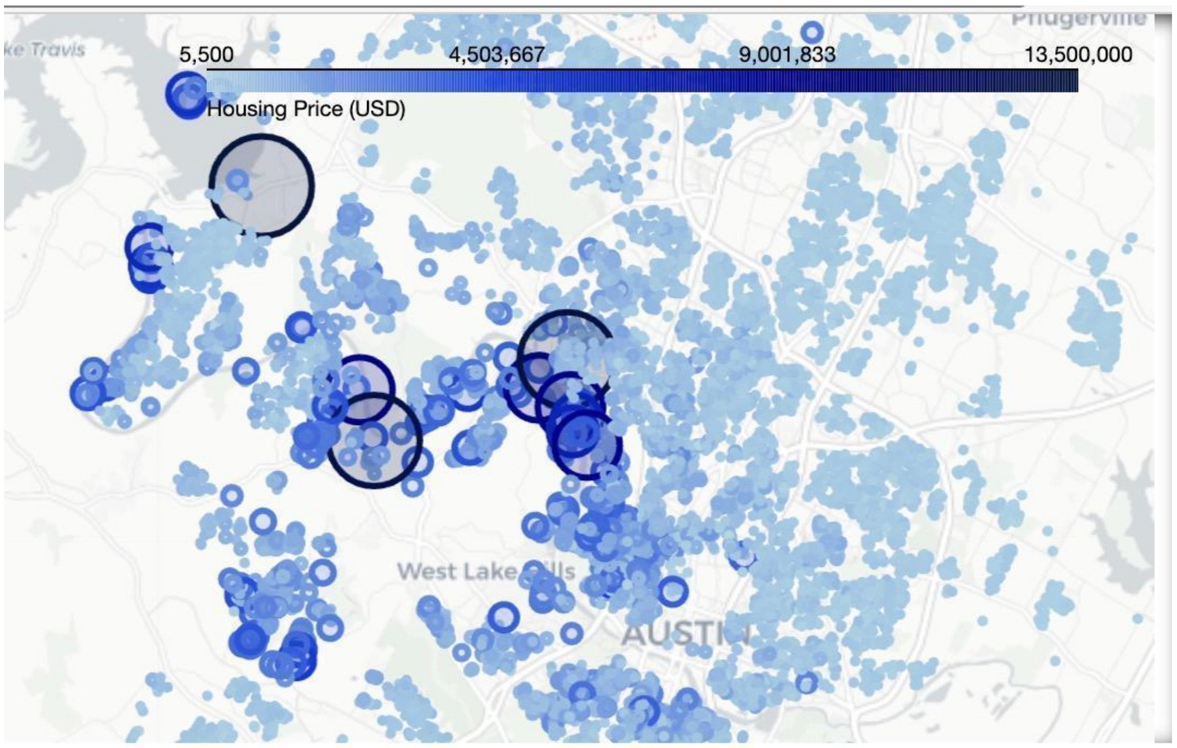


Fig.1: Austin House Price Density Map

3.2 Target Variable

The target variable in the project is the sale price named in the column of ‘latestPrice’, with the mean sale price of $ 512767.7 and the highest sale price of $13500000.

From Fig.2, we can see that the ‘latestPrice’ histogram is positively skewed and has an extremely high peak. When the distribution of the target variable is skewed, it can affect the assumptions of the regression model, especially in linear regression. For example, the normality of the errors, constant variance of the errors, and linearity among variables. Violations will ultimately lead to biased or inefficient parameter estimates and inaccurate predictions, so we need to transform the target variable and address the skewness before applying to models.

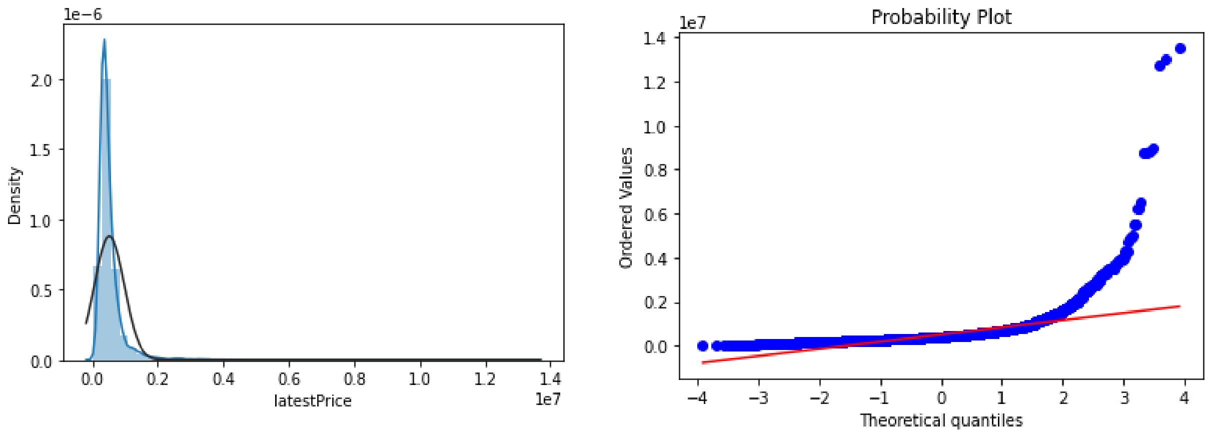


Fig.2: Highly Positively Skewed Target Variable

To address this, a logarithmic transformation can be applied to the original dataset to improve normality. This is because the logarithmic transformation compresses the range of values, making extreme values less influential in the analysis. Overall, the logarithmic transformation can improve the performance and accuracy of statistical models when dealing with heavily skewed data distributions. The resulting data can be visualized in Figure 3.

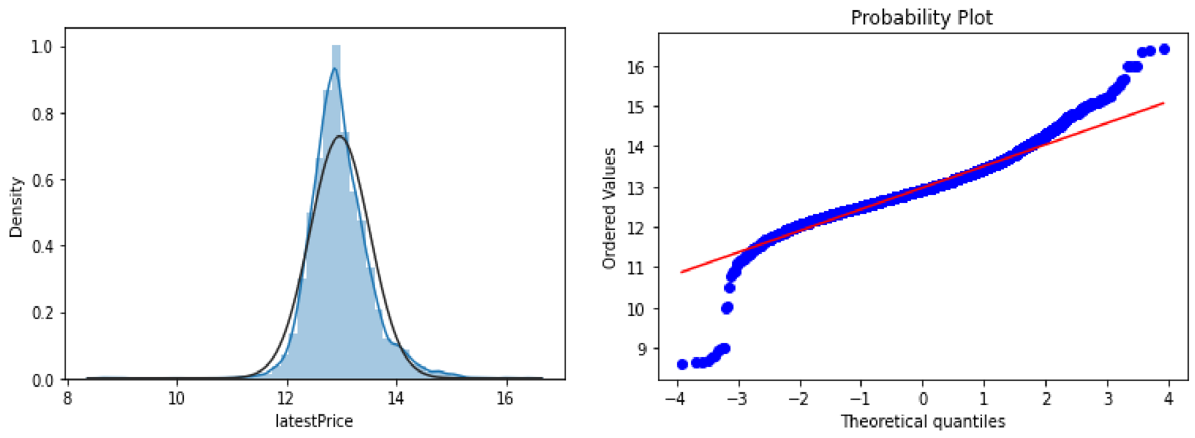


Fig.3: Transformed log-latest price Target Variable

3.3 Data Preprocessing

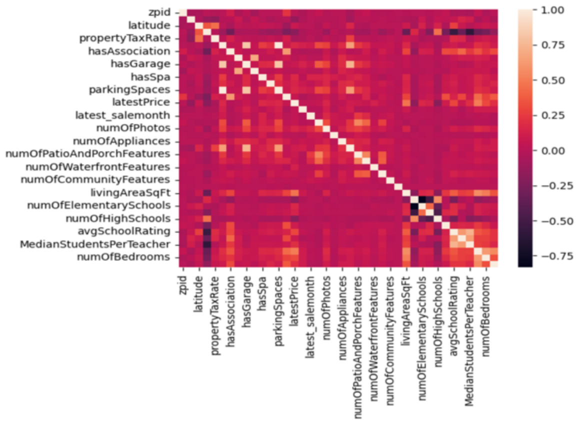
Before setting up the models, we need to do some data cleaning and preprocessing. We first drop all the duplicate and missing value then convert the data type of several boolean columns (‘hasAssociation’, ‘hasCooling’, ‘hasGarage’, ‘hasHeating’, ‘hasSpa’, and ‘hasView’) to integer (1 for True and 0 for False). We perform the one-hot encoding for the categorical variable ‘homeType’, converting it into multiple binary columns (dummy variables) so that it can be used in our later regression analysis or other machine learning algorithms that require numerical inputs.

Our next step is to remove outliers by using the Mahalanobis distance. The Mahalanobis distance is a measure of the distance between each single data point and the mean of the dataset, taking into account the correlations between variables. We calculate the threshold value for identifying outliers based on the chi-squared distribution. The threshold is calculated as the mean of the Mahalanobis distances plus three times the standard deviation. Data points with a Mahalanobis distance greater than the calculated threshold are outliers.

$$D^2 = (x - m)^T \cdot C^{-1} \cdot (x - m)$$

3.4 Feature Selection

The aim of feature selection is to improve the performance of our models by reducing the number of irrelevant or redundant features, which may lead to overfitting and lower accuracy. There are 45 features in the original dataset. While having many features can be beneficial for building with more accuracy, it also presents challenges in terms of overfitting, multicollinearity and reduced interpretability. Thus, we decide to drop some features manually through our first glance.



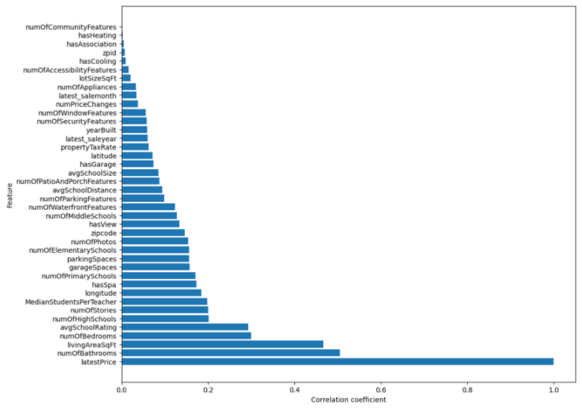


Fig.4&5: Heatmap of Correlation matrix

We Drop:

- ‘Zipcode’, ‘Zpid’ and ‘streetAddress’ since they are only identifiers

- ‘description’, ‘homeImage’ and ‘latestPriceSource’ since they cannot be converted as numeric values.
- ‘latest_saleyear’ and ‘latest_salemonth’ since ‘latest_saledate’ contains exactly the same amount of information.

After the feature selection, there are 38 features left for further exploration. Note that the feature selection process described above is a preliminary step, intended to identify and remove features that could distract our attention from the response variable ‘latestPrice’. Further feature selection will be performed using machine learning techniques such as Lasso regression and Random Forest, which can help to identify the most important features for predicting price and provide more accurate models.

3.5 Feature Representation & Transformation

- Boolean Features (True/False) were converted as Binary Values (1/0)
- Categorical Feature, ‘homeType’, was converted into multiple binary columns (dummy variables) by one-hot encoding so that it can be used in our later regression analysis or other machine learning algorithms that require numerical inputs.
- Continuous numerical features were log-transformed if their skewness>1, which indicate right-skew distribution.

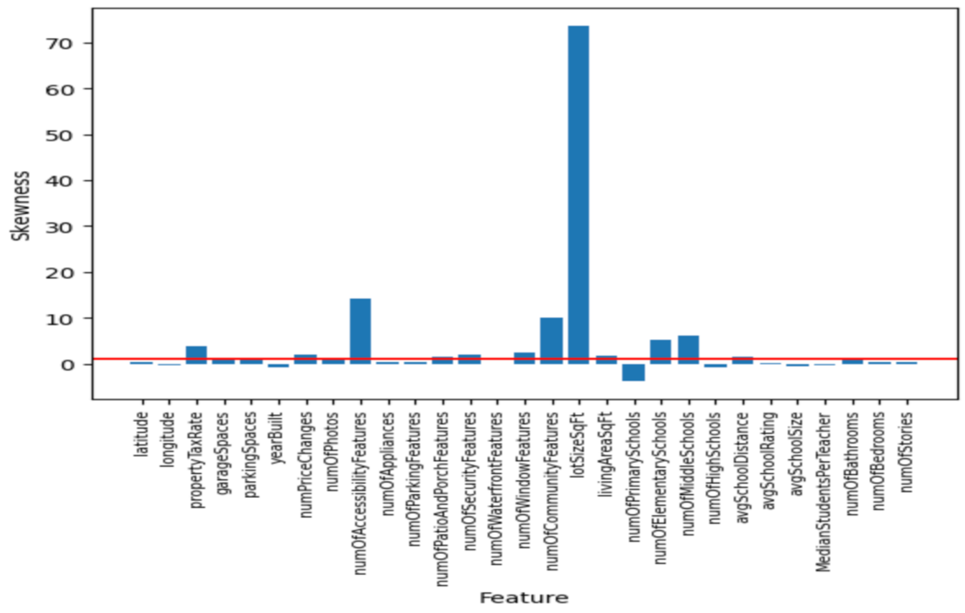


Fig.6: Feature Skewness

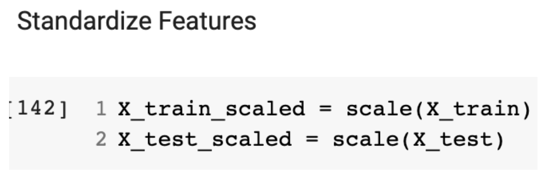
The dataset is now ready to be used for building models.

4. Model Explorations

4.1 Regression Models

Given that the response variable, ‘latestPrice’, is a quantitative variable, it’s reasonable to consider a linear approach to model between multiple predictors and itself. There are three regression models introduced in the section: Linear Regression, Lasso Regression and Principal Component Regression (PCR). The first two were considered as Baseline Models to compare with.

Before constructing models, we first standardize the dataset to center around the mean and to scale to unit variance to make sure each feature is given equal weight in the model and that the relative importance of each feature is not influenced by its scale or unit of measurement.



Standardize Features

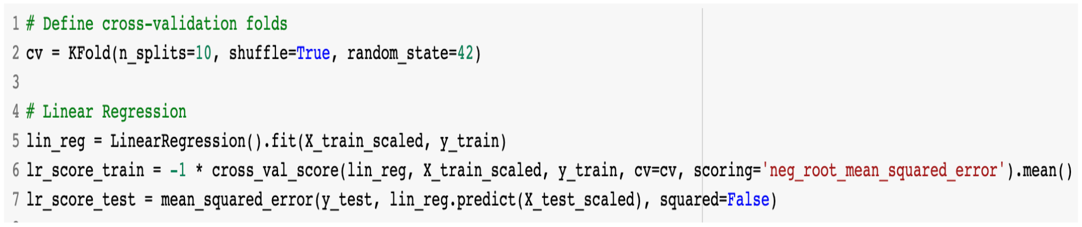
```
[142]  X_train_scaled = scale(X_train)
       X_test_scaled = scale(X_test)
```

To assess the accuracy of the model and to identify any potential issues within the modeling approach, we consider Root Mean Squared Error (RMSE) as the main metric: with a lower RMSE indicating a better fit of the model to the data.

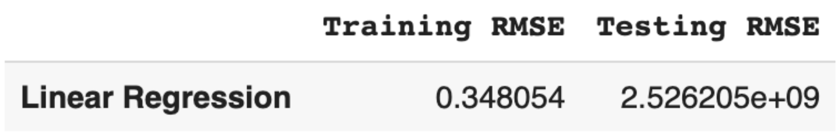
$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} \| y(i) - \hat{y}(i) \|^2}{N}},$$

RMSE Formula

4.1.1 Linear regression model



```
# Define cross-validation folds
cv = KFold(n_splits=10, shuffle=True, random_state=42)

# Linear Regression
lin_reg = LinearRegression().fit(X_train_scaled, y_train)
lr_score_train = -1 * cross_val_score(lin_reg, X_train_scaled, y_train, cv=cv, scoring='neg_root_mean_squared_error').mean()
lr_score_test = mean_squared_error(y_test, lin_reg.predict(X_test_scaled), squared=False)
```



	Training RMSE	Testing RMSE
Linear Regression	0.348054	2.526205e+09

Fig.7: Linear Regression model with its RMSE

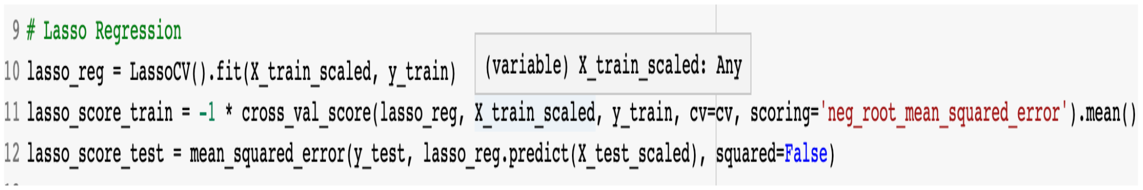
Taking the linear regression model with 10-fold Cross Validation as one of the baselines, we include all predictor variables that exist after preliminary feature selection. Here we have training RMSE = 0.348054 and testing RMSE = 2.526205e+09. When a model has a relatively small training RMSE and a huge testing RMSE, it implies a high accuracy on the training data but poor performance on new or unseen data. The linear regression model containing all features is overfitting to the training data.

4.1.2 Least Absolute Shrinkage and Selection Operator (LASSO)

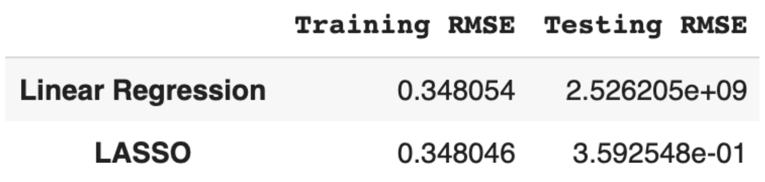
To address overfitting, it may be necessary to simplify the model by reducing the number of features or regularizing the model coefficients. Here we consider LASSO regression as the regularization technique: it adds a penalty that is equal to the absolute value of the magnitude of the coefficient. Some coefficients, with larger penalty, may shrink to zero and get eliminated from the model.

Equation 4-10. Lasso Regression cost function

$$J(\boldsymbol{\theta}) = \mathrm{MSE}(\boldsymbol{\theta}) + \alpha\Sigma_{i=1}^{n}\left|\theta_i\right|$$



```python
# Lasso Regression
lasso_reg = LassoCV().fit(X_train_scaled, y_train)
lasso_score_train = -1 * cross_val_score(lasso_reg, X_train_scaled, y_train, cv=cv, scoring='neg_root_mean_squared_error').mean()
lasso_score_test = mean_squared_error(y_test, lasso_reg.predict(X_test_scaled), squared=False)
```

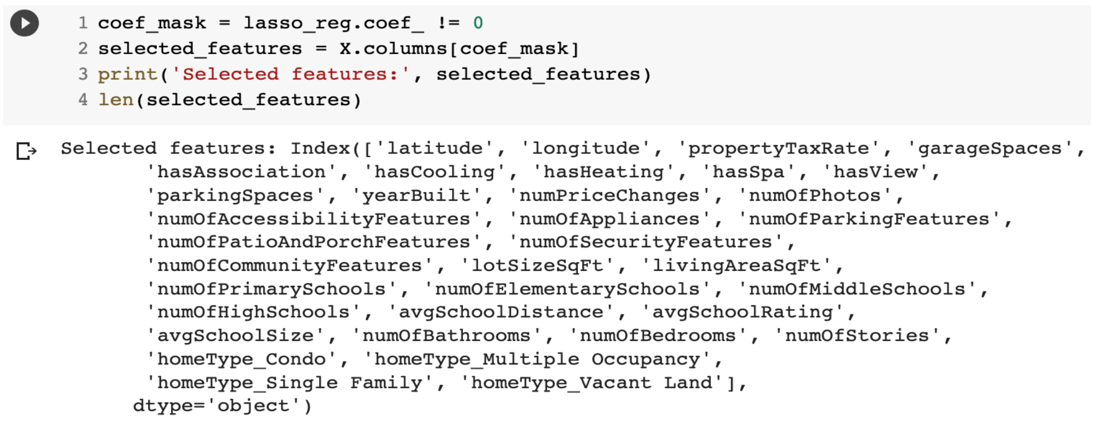


	Training RMSE	Testing RMSE
Linear Regression	0.348054	2.526205e+09
LASSO	0.348046	3.592548e-01

Fig.8: LASSO Regression model with its RMSE

Here we have training RMSE = 0.348046 and testing RMSE = 0.35925. A large decrease of testing RMSE indicates that the LASSO model handles overfitting well and performs much better on new or unseen data than linear regression model.

An important characteristic of LASSO is that it tends to completely remove the weight of the least important features, which means LASSO performs feature selection automatically.



```python
coef_mask = lasso_reg.coef_ != 0
selected_features = X.columns[coef_mask]
print('Selected features:', selected_features)
len(selected_features)
```

```
Selected features: Index(['latitude', 'longitude', 'propertyTaxRate', 'garageSpaces',
       'hasAssociation', 'hasCooling', 'hasHeating', 'hasSpa', 'hasView',
       'parkingSpaces', 'yearBuilt', 'numPriceChanges', 'numOfPhotos',
       'numOfAccessibilityFeatures', 'numOfAppliances', 'numOfParkingFeatures',
       'numOfPatioAndPorchFeatures', 'numOfSecurityFeatures',
       'numOfCommunityFeatures', 'lotSizeSqFt', 'livingAreaSqFt',
       'numOfPrimarySchools', 'numOfElementarySchools', 'numOfMiddleSchools',
       'numOfHighSchools', 'avgSchoolDistance', 'avgSchoolRating',
       'avgSchoolSize', 'numOfBathrooms', 'numOfBedrooms', 'numOfStories',
       'homeType_Condo', 'homeType_Multiple Occupancy',
       'homeType_Single Family', 'homeType_Vacant Land'],
      dtype='object')
```

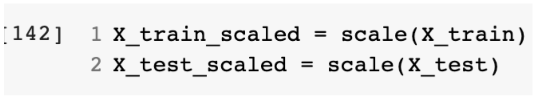
Fig.10: Feature selection by LASSO

Fig.10 shows the output of automatic Feature Selection done by LASSO. It’s true this regularization technique helps to drop some insignificant features, but a model containing 35 features is still hard to interpret.

4.1.3 Principal Component Regression (PCR)

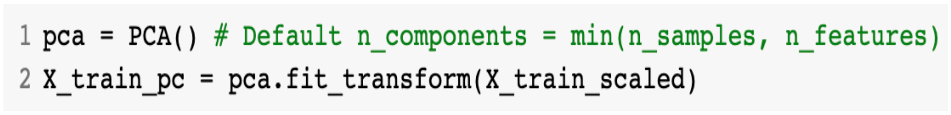
PCR can be useful when dealing with high-dimensional datasets with many predictors, as it can reduce the dimensionality of the data while preserving most of the relevant information.
The technique involves the following steps:

- Standardize the predictors



```
[142]  X_train_scaled = scale(X_train)
       X_test_scaled = scale(X_test)
```

- Calculate the principal components



```
pca = PCA() # Default n_components = min(n_samples, n_features)
X_train_pc = pca.fit_transform(X_train_scaled)
```

- Select the principal components

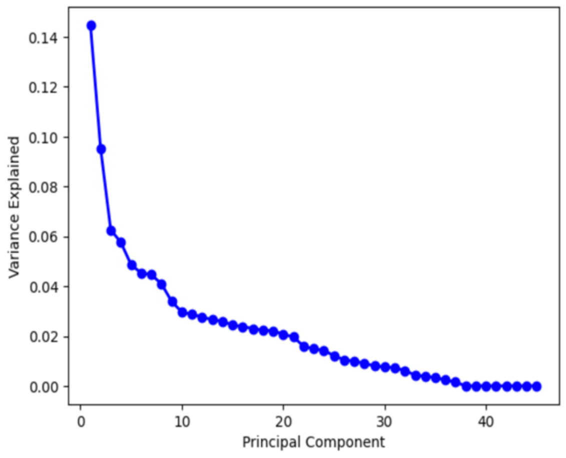


Fig.11: Scree Plot

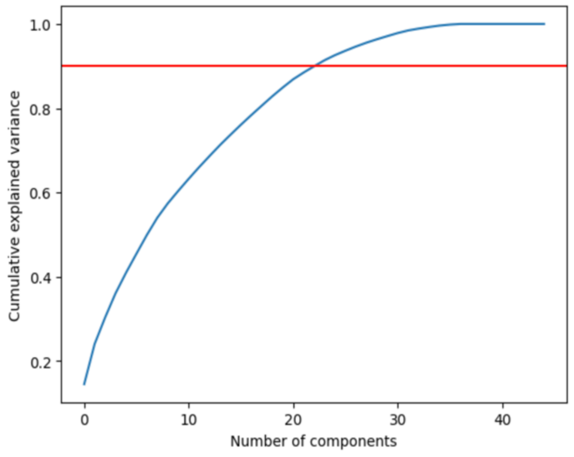
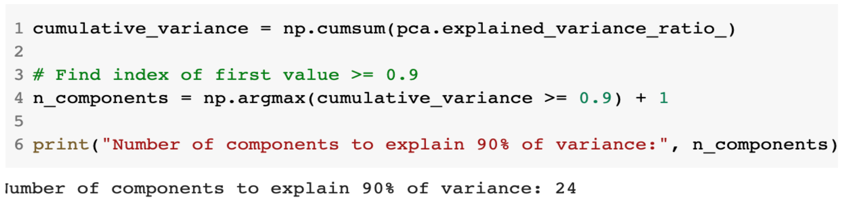


Fig.12: Cumulative explained variance

Usually, we determine the Number of Principal Components by eyeballing the “elbow” in the curve and selecting all components just before the line flattens out, but in our case, the ‘elbow’ is hard to identify as the Fig.11 showed. So, we plot the cumulative explained variance and set the threshold to be 0.9 to make sure the optimal number of PCs being chosen to hold 90% of explained variance.

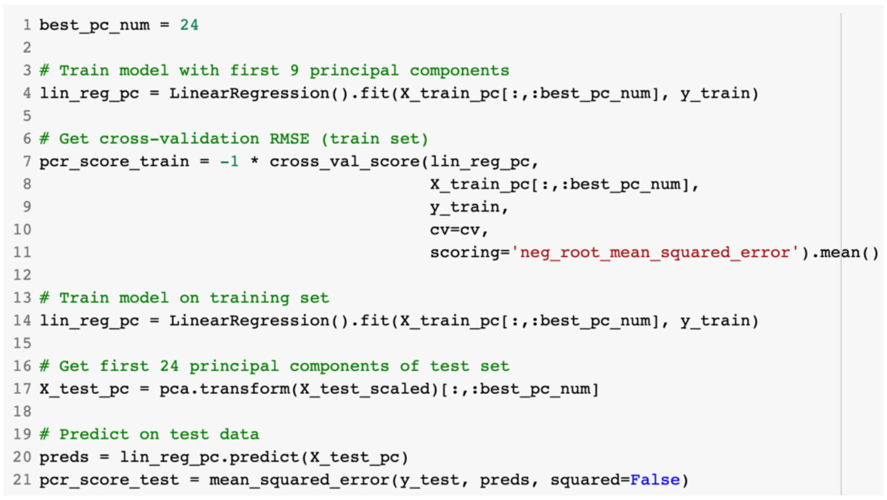


```python
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)

# Find index of first value >= 0.9
n_components = np.argmax(cumulative_variance >= 0.9) + 1

print("Number of components to explain 90% of variance:", n_components)
```

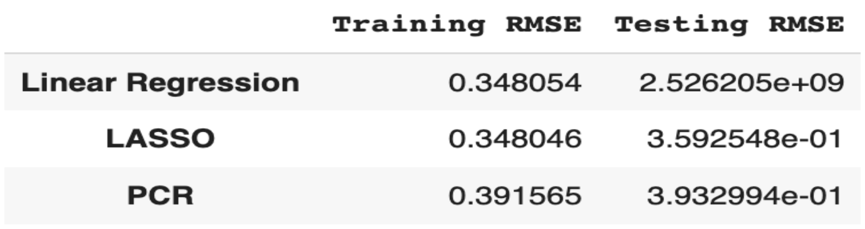
```
umber of components to explain 90% of variance: 24
```

- Fit a linear regression model

  After determining the best number of principal components to use (i.e., N=24), we proceed to run PCR on our test dataset.



```python
best_pc_num = 24

# Train model with first 9 principal components
lin_reg_pc = LinearRegression().fit(X_train_pc[:,:best_pc_num], y_train)

# Get cross-validation RMSE (train set)
pcr_score_train = -1 * cross_val_score(lin_reg_pc,
                                       X_train_pc[:,:best_pc_num],
                                       y_train,
                                       cv=cv,
                                       scoring='neg_root_mean_squared_error').mean()

# Train model on training set
lin_reg_pc = LinearRegression().fit(X_train_pc[:,:best_pc_num], y_train)

# Get first 24 principal components of test set
X_test_pc = pca.transform(X_test_scaled)[:,:best_pc_num]

# Predict on test data
preds = lin_reg_pc.predict(X_test_pc)
pcr_score_test = mean_squared_error(y_test, preds, squared=False)
```



	Training RMSE	Testing RMSE
Linear Regression	0.348054	2.526205e+09
LASSO	0.348046	3.592548e-01
PCR	0.391565	3.932994e-01

Fig.12: PCR with its RMSE

Comparing two RMSE of Linear regression and PCR, one thing which could be pointed out is that PCR indeed helps to reduce the overfitting problems but does not perform better than previous models. A poor fit of PCR may be caused by non-linear data: PCR assumes that the relationship between the predictors and the response variable is linear. If the relationship is not linear, PCR is not able to capture the underlying patterns and relationships in the data, and the resulting model may perform poorly.

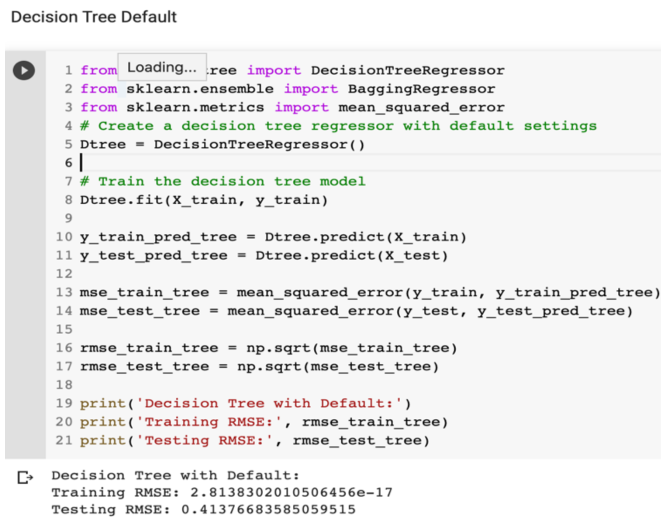
In this case, other modeling techniques like decision trees or random forest may be more appropriate.

4.2 Tree-Based Model & Ensemble Learning

As we can see that the linear regression is still suffering from the problem of overfitting, indicating our model might not be sophisticated enough to interpret the testing set. Hence, we introduce the tree-based models and ensemble learning, like decision trees, bagging & pasting, and boosting. These models can be effective in handling many features because they naturally perform feature selection during the process of constructing the tree. They tend to work well with high-dimensional data because they can capture complex interactions between features. Below we tried various tree models with hyperparameter tuning using cross-validations.

4.2.1 Decision Tree

We started by implementing a Decision Tree model with default settings and seeing how it goes.



Decision Tree Default

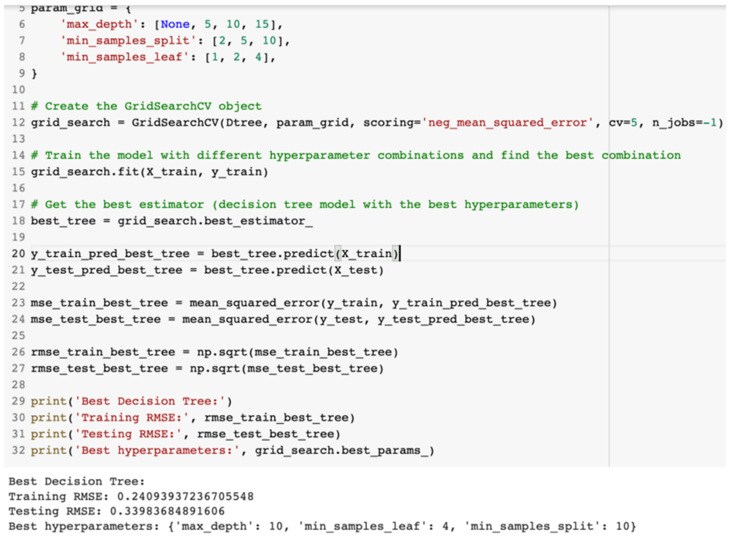
```python
from Loading... ree import DecisionTreeRegressor
from sklearn.ensemble import BaggingRegressor
from sklearn.metrics import mean_squared_error
# Create a decision tree regressor with default settings
Dtree = DecisionTreeRegressor()

# Train the decision tree model
Dtree.fit(X_train, y_train)

y_train_pred_tree = Dtree.predict(X_train)
y_test_pred_tree = Dtree.predict(X_test)

mse_train_tree = mean_squared_error(y_train, y_train_pred_tree)
mse_test_tree = mean_squared_error(y_test, y_test_pred_tree)

rmse_train_tree = np.sqrt(mse_train_tree)
rmse_test_tree = np.sqrt(mse_test_tree)

print('Decision Tree with Default:')
print('Training RMSE:', rmse_train_tree)
print('Testing RMSE:', rmse_test_tree)
```

```
Decision Tree with Default:
Training RMSE: 2.8138302010506456e-17
Testing RMSE: 0.413766835850595l5
```

Fig.13: Decision Tree with its RMSE

The Decision Tree with default parameters achieved an almost perfect fit on the training data, as indicated by the extremely low RMSE of 2.813e -17. However, the model seems too overfit, as the testing RMSE is significantly higher at 0.413.

Acknowledging the overfitting issue in the initial decision tree model, we decided to perform hyperparameter tuning to address this problem.



```
param_grid = {
    'max_depth': [None, 5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}

# Create the GridSearchCV object
grid_search = GridSearchCV(Dtree, param_grid, scoring='neg_mean_squared_error', cv=5, n_jobs=-1)

# Train the model with different hyperparameter combinations and find the best combination
grid_search.fit(X_train, y_train)

# Get the best estimator (decision tree model with the best hyperparameters)
best_tree = grid_search.best_estimator_

y_train_pred_best_tree = best_tree.predict(X_train)
y_test_pred_best_tree = best_tree.predict(X_test)

mse_train_best_tree = mean_squared_error(y_train, y_train_pred_best_tree)
mse_test_best_tree = mean_squared_error(y_test, y_test_pred_best_tree)

rmse_train_best_tree = np.sqrt(mse_train_best_tree)
rmse_test_best_tree = np.sqrt(mse_test_best_tree)

print('Best Decision Tree:')
print('Training RMSE:', rmse_train_best_tree)
print('Testing RMSE:', rmse_test_best_tree)
print('Best hyperparameters:', grid_search.best_params_)
```

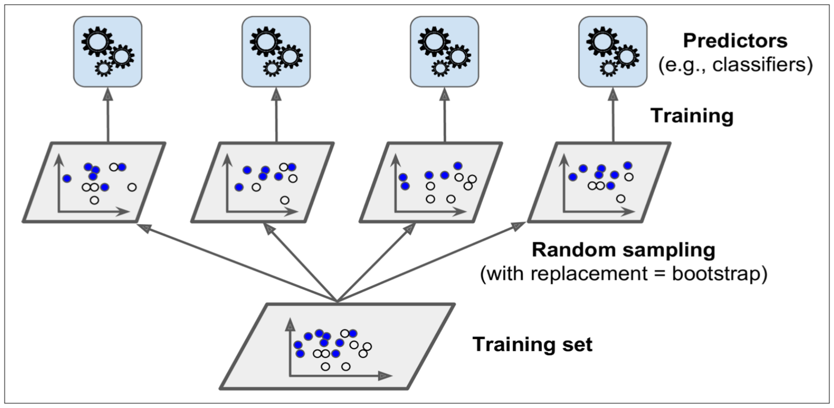
```
Best Decision Tree:
Training RMSE: 0.24093937236705548
Testing RMSE: 0.33983684891606
Best hyperparameters: {'max_depth': 10, 'min_samples_leaf': 4, 'min_samples_split': 10}
```

Fig.14: Decision Tree Hyperparameter Tuning with its RMSE

After tuning, we observed a significant reduction in overfitting compared to the default settings. The optimized decision tree achieved a training RMSE of 0.2409 and a testing RMSE of 0.3398 using the best hyperparameters: ‘max_depth’=10, ‘min_samples_leaf’=4, and ‘min_samples_split’=10. This indicates a better generalization of the model to unseen data.

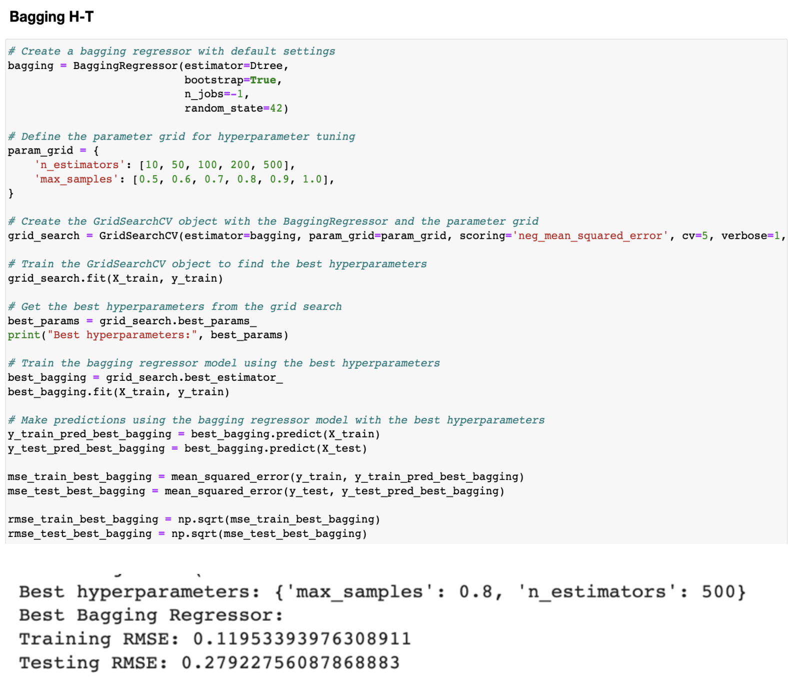
4.2.2 Bagging & Pasting

Bagging and pasting are ensemble learning techniques that use the same training algorithm for all predictors but train them on different random subsets of the data. Bagging samples with replacement, while pasting samples without replacement.



The ensemble's predictions are aggregated by using the average for regression. Despite individual predictors having higher bias, the ensemble tends to have similar bias, but lower variance compared to a single predictor trained on the original dataset.

After observing the limitations of a single Decision Tree model and to address potential overfitting and improve the model's performance, we decided to implement a **Bagging Regressor & Pasting Regressor**, an ensemble learning technique that trains multiple base models on random subsets of the data and then aggregates their predictions.



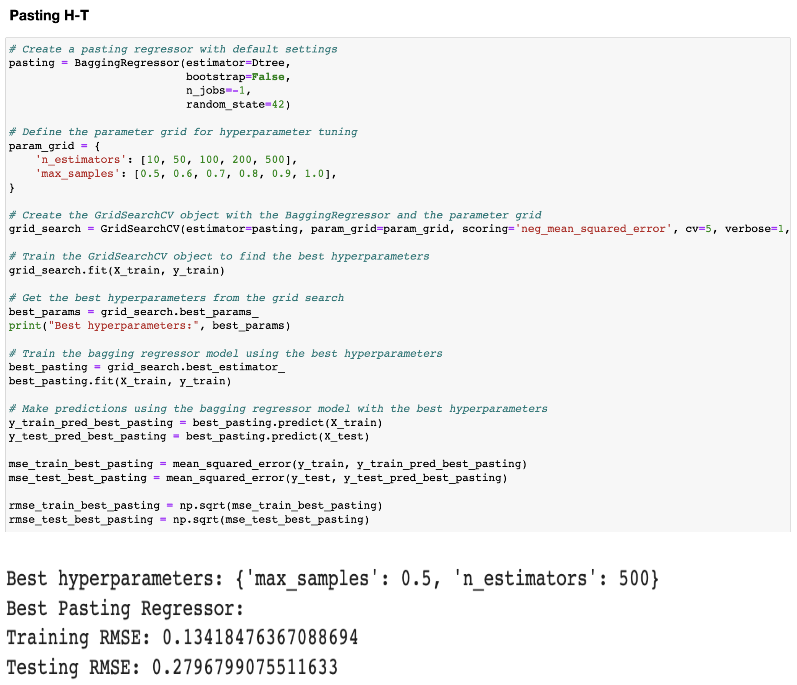
Bagging H-T

```
# Create a bagging regressor with default settings
bagging = BaggingRegressor(estimator=Dtree,
                           bootstrap=True,
                           n_jobs=-1,
                           random_state=42)

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [10, 50, 100, 200, 500],
    'max_samples': [0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
}

# Create the GridSearchCV object with the BaggingRegressor and the parameter grid
grid_search = GridSearchCV(estimator=bagging, param_grid=param_grid, scoring='neg_mean_squared_error', cv=5, verbose=1,

# Train the GridSearchCV object to find the best hyperparameters
grid_search.fit(X_train, y_train)

# Get the best hyperparameters from the grid search
best_params = grid_search.best_params_
print("Best hyperparameters:", best_params)

# Train the bagging regressor model using the best hyperparameters
best_bagging = grid_search.best_estimator_
best_bagging.fit(X_train, y_train)

# Make predictions using the bagging regressor model with the best hyperparameters
y_train_pred_best_bagging = best_bagging.predict(X_train)
y_test_pred_best_bagging = best_bagging.predict(X_test)

mse_train_best_bagging = mean_squared_error(y_train, y_train_pred_best_bagging)
mse_test_best_bagging = mean_squared_error(y_test, y_test_pred_best_bagging)

rmse_train_best_bagging = np.sqrt(mse_train_best_bagging)
rmse_test_best_bagging = np.sqrt(mse_test_best_bagging)
```

```
Best hyperparameters: {'max_samples': 0.8, 'n_estimators': 500}
Best Bagging Regressor:
Training RMSE: 0.11953393976308911
Testing RMSE: 0.27922756087868883
```

Fig.15: Bagging H-T with its RMSE

The Bagging Regressor achieved a training RMSE of 0.1195 and a testing RMSE of 0.2792 with the best hyperparameters: ‘max_samples’=0.8 and ‘n_estimators’=500.



Pasting H-T

```
# Create a pasting regressor with default settings
pasting = BaggingRegressor(estimator=Dtree,
                           bootstrap=False,
                           n_jobs=-1,
                           random_state=42)

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [10, 50, 100, 200, 500],
    'max_samples': [0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
}

# Create the GridSearchCV object with the BaggingRegressor and the parameter grid
grid_search = GridSearchCV(estimator=pasting, param_grid=param_grid, scoring='neg_mean_squared_error', cv=5, verbose=1,

# Train the GridSearchCV object to find the best hyperparameters
grid_search.fit(X_train, y_train)

# Get the best hyperparameters from the grid search
best_params = grid_search.best_params_
print("Best hyperparameters:", best_params)

# Train the bagging regressor model using the best hyperparameters
best_pasting = grid_search.best_estimator_
best_pasting.fit(X_train, y_train)

# Make predictions using the bagging regressor model with the best hyperparameters
y_train_pred_best_pasting = best_pasting.predict(X_train)
y_test_pred_best_pasting = best_pasting.predict(X_test)

mse_train_best_pasting = mean_squared_error(y_train, y_train_pred_best_pasting)
mse_test_best_pasting = mean_squared_error(y_test, y_test_pred_best_pasting)

rmse_train_best_pasting = np.sqrt(mse_train_best_pasting)
rmse_test_best_pasting = np.sqrt(mse_test_best_pasting)
```

```
Best hyperparameters: {'max_samples': 0.5, 'n_estimators': 500}
Best Pasting Regressor:
Training RMSE: 0.13418476367088694
Testing RMSE: 0.2796799075511633
```

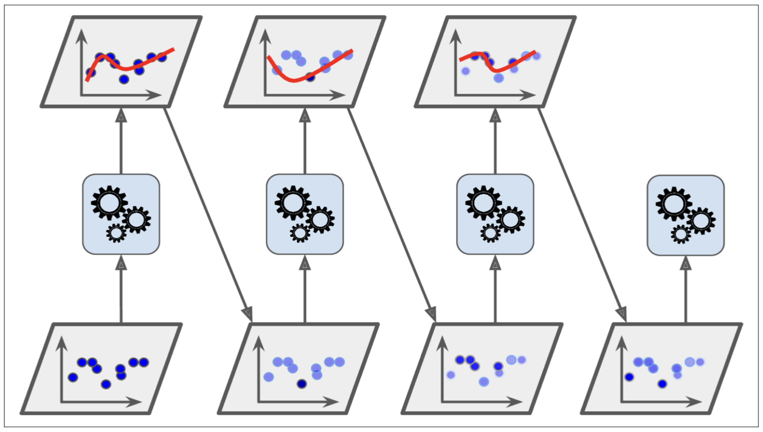
Fig.16: Pasting H-T with its RMSE

The Pasting regressor resulted in a training RMSE of 0.13418 and a testing RMSE of 0.27967, which indicates improved performance compared to the single Decision Tree model, although the performance is slightly less robust than the Bagging Regressor.

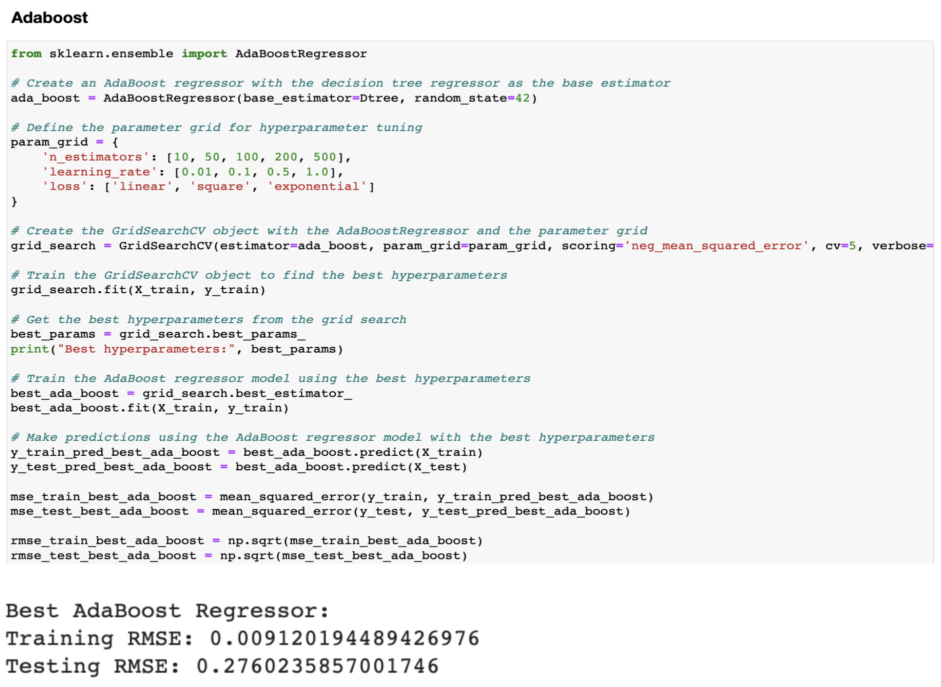
4.2.3 Boosting (Adaboost & Gradient Boost)

Following the exploration of bagging and pasting methods, we decided to experiment with another two ensemble learning techniques that focuses on boosting weak learners, AdaBoost and Gradient Boost.

AdaBoost stands for Adaptive Boosting, and it works by training subsequent predictors to focus more on the misclassified instances of the preceding predictors, ultimately adapting to the errors and improving overall performance.



For example, to build an AdaBoost regressor, a first base regressor (such as a Decision Tree) is trained and used to make predictions on the training set. The residuals (differences between the predicted values and the actual values) are then computed. A second regressor is trained on these residuals, and it attempts to correct the errors made by the first regressor. Once the second regressor has been trained, it makes predictions on the training set, new residuals are computed, and the process continues. This is done for a predefined number of iterations or until the residuals can no longer be effectively reduced.



Adaboost

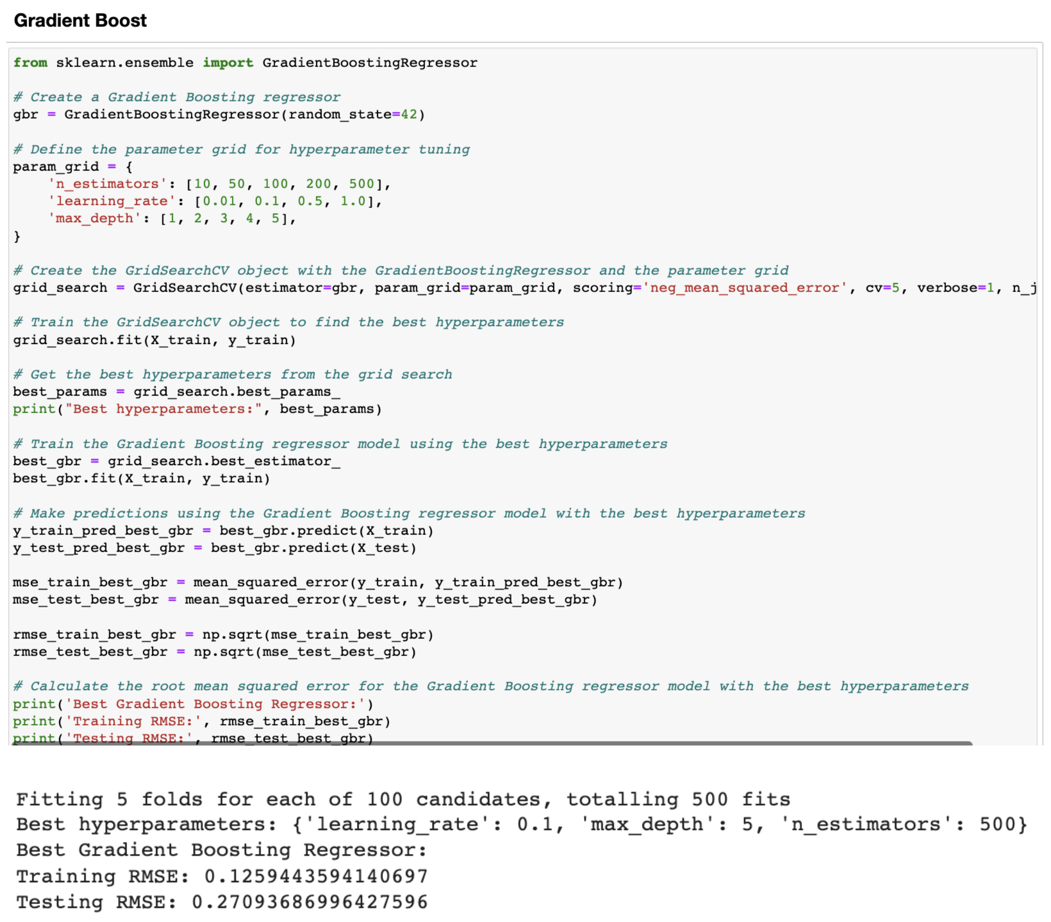
```python
from sklearn.ensemble import AdaBoostRegressor

# Create an AdaBoost regressor with the decision tree regressor as the base estimator
ada_boost = AdaBoostRegressor(base_estimator=Dtree, random_state=42)

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [10, 50, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.5, 1.0],
    'loss': ['linear', 'square', 'exponential']
}

# Create the GridSearchCV object with the AdaBoostRegressor and the parameter grid
grid_search = GridSearchCV(estimator=ada_boost, param_grid=param_grid, scoring='neg_mean_squared_error', cv=5, verbose=

# Train the GridSearchCV object to find the best hyperparameters
grid_search.fit(X_train, y_train)

# Get the best hyperparameters from the grid search
best_params = grid_search.best_params_
print("Best hyperparameters:", best_params)

# Train the AdaBoost regressor model using the best hyperparameters
best_ada_boost = grid_search.best_estimator_
best_ada_boost.fit(X_train, y_train)

# Make predictions using the AdaBoost regressor model with the best hyperparameters
y_train_pred_best_ada_boost = best_ada_boost.predict(X_train)
y_test_pred_best_ada_boost = best_ada_boost.predict(X_test)

mse_train_best_ada_boost = mean_squared_error(y_train, y_train_pred_best_ada_boost)
mse_test_best_ada_boost = mean_squared_error(y_test, y_test_pred_best_ada_boost)

rmse_train_best_ada_boost = np.sqrt(mse_train_best_ada_boost)
rmse_test_best_ada_boost = np.sqrt(mse_test_best_ada_boost)
```

```
Best AdaBoost Regressor:
Training RMSE: 0.009120194489426976
Testing RMSE: 0.2760235857001746
```

Fig.17: Adaboosting with its RMSE

In this case, we used GridSearchCV for hyperparameter tuning and found the best parameters to be ‘learning_rate’: 1.0, ‘loss’: ‘linear’, and ‘n_estimators’: 500. The AdaBoost regressor achieved a training RMSE of 0.00912 and a testing RMSE of 0.276. These results indicate a significant improvement in training performance compared to the single Decision Tree, Bagging, and Pasting models, and a slightly better testing performance than the Bagging and Pasting models. This highlights the potential of AdaBoost as an effective ensemble learning method in reducing both bias and variance in model predictions.

Building upon the exploration of ensemble learning techniques, we proceeded to experiment with **Gradient Boosting**. Gradient Boosting is another powerful ensemble method that works by training predictors sequentially, with each subsequent predictor correcting the errors made by its predecessor. This approach is known to produce highly accurate models by combining the strengths of multiple weak learners.



Gradient Boost

```
from sklearn.ensemble import GradientBoostingRegressor

# Create a Gradient Boosting regressor
gbr = GradientBoostingRegressor(random_state=42)

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'n_estimators': [10, 50, 100, 200, 500],
    'learning_rate': [0.01, 0.1, 0.5, 1.0],
    'max_depth': [1, 2, 3, 4, 5],
}

# Create the GridSearchCV object with the GradientBoostingRegressor and the parameter grid
grid_search = GridSearchCV(estimator=gbr, param_grid=param_grid, scoring='neg_mean_squared_error', cv=5, verbose=1, n_j

# Train the GridSearchCV object to find the best hyperparameters
grid_search.fit(X_train, y_train)

# Get the best hyperparameters from the grid search
best_params = grid_search.best_params_
print("Best hyperparameters:", best_params)

# Train the Gradient Boosting regressor model using the best hyperparameters
best_gbr = grid_search.best_estimator_
best_gbr.fit(X_train, y_train)

# Make predictions using the Gradient Boosting regressor model with the best hyperparameters
y_train_pred_best_gbr = best_gbr.predict(X_train)
y_test_pred_best_gbr = best_gbr.predict(X_test)

mse_train_best_gbr = mean_squared_error(y_train, y_train_pred_best_gbr)
mse_test_best_gbr = mean_squared_error(y_test, y_test_pred_best_gbr)

rmse_train_best_gbr = np.sqrt(mse_train_best_gbr)
rmse_test_best_gbr = np.sqrt(mse_test_best_gbr)

# Calculate the root mean squared error for the Gradient Boosting regressor model with the best hyperparameters
print('Best Gradient Boosting Regressor:')
print('Training RMSE:', rmse_train_best_gbr)
print('Testing RMSE:', rmse_test_best_gbr)
```

```
Fitting 5 folds for each of 100 candidates, totalling 500 fits
Best hyperparameters: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 500}
Best Gradient Boosting Regressor:
Training RMSE: 0.1259443594140697
Testing RMSE: 0.27093686996427596
```

Fig.18: Gradient Boost with its RMSE

The Gradient Boosting regressor yielded a training RMSE of 0.1259 and a testing RMSE of 0.2709. These results demonstrate that Gradient Boosting outperforms the single Decision Tree, Bagging, Pasting, and AdaBoost models in terms of testing performance. The training performance is also considerably better than the single Decision Tree, Bagging, and Pasting models, but slightly worse than the AdaBoost model. This indicates that Gradient Boosting can strike a balance between reducing bias and variance while delivering improved overall model accuracy. The success of Gradient Boosting in this project highlights its potential as an effective ensemble learning method for complex modeling tasks.

4.2.3.1 Feature importance Analysis

After training the Gradient Boosting Regressor model, we can analyze the importance of the features in predicting the target variable. To do this, we can use the 'feature_importances'_ attribute of the model object and create a bar plot to visualize the results.

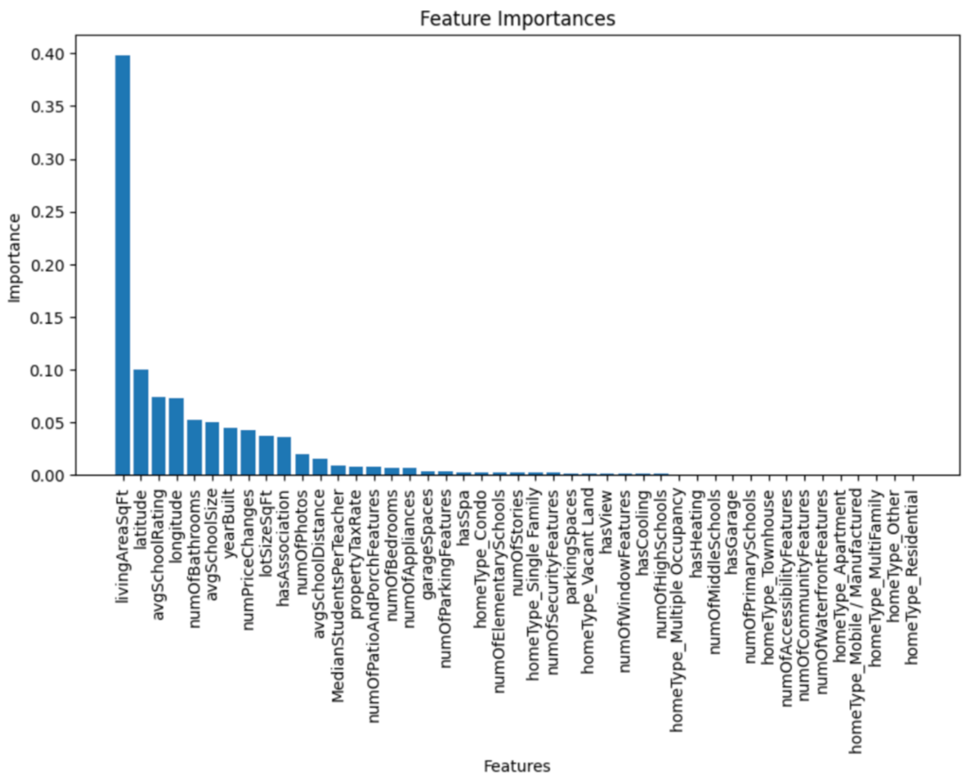


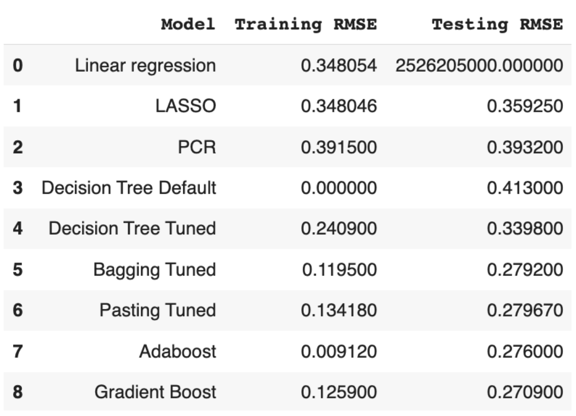
Fig.19: Feature importance based on Gradient Boost

Based on the Feature Importance, we can see that feature ‘living AreasqFt’ has the highest importance, followed by feature ‘latitude’ and ‘avgSchoolRating’. This information can help us gain insights into the relationship between the features and the housing price: ‘living AreasqFt’ is a common predictor of housing prices, as larger homes generally cost more than smaller ones. Similarly, the location of the property, represented by 'latitude', can have a significant impact on housing prices, as certain neighborhoods or areas may be more desirable and therefore command higher prices. ‘avgSchoolRating’ may also be an important predictor, as homes in areas with better schools may be more valuable and therefore have higher prices.

5. Conclusion

In this project, our group is trying to identify features related to Sales Price and construct the model with higher accuracy to better predict the house price. In the data preprocessing stage, we performed several essential tasks to clean and prepare the data for further analysis and modeling. This process involved handling missing values, removing outliers, and transforming features as needed. We identified and removed outliers to minimize their potential impact on the model's performance and transformed features, using techniques such as normalization and one-hot encoding, to ensure compatibility with the machine learning algorithms used later in the process. This thorough preprocessing allowed us to create a high-quality dataset for our modeling efforts. Throughout the project, we visualized our results using various plotting techniques to better understand and provide a clear relationship between the features and the target variable. These visualizations helped us identify potential outliers, trends, and patterns within the data, which helps us continue developing our models.

Our models include linear regression, LASSO and PCR. We also experimented with ensemble methods, such as pasting and bagging, AdaBoost and Gradient Boost to further improve the performance of our model. By combining multiple models, we were able to achieve a more robust prediction with reduced overfitting. We constantly evaluated our models using metric root mean squared error (RMSE) to assess the model's accuracy and performance. We also performed 10-fold cross-validation to ensure the stability of our results.



	Model	Training RMSE	Testing RMSE
0	Linear regression	0.348054	2526205000.000000
1	LASSO	0.348046	0.359250
2	PCR	0.391500	0.393200
3	Decision Tree Default	0.000000	0.413000
4	Decision Tree Tuned	0.240900	0.339800
5	Bagging Tuned	0.119500	0.279200
6	Pasting Tuned	0.134180	0.279670
7	Adaboost	0.009120	0.276000
8	Gradient Boost	0.125900	0.270900

Fig.20: RMSE result

In conclusion, by combining various machine learning techniques and feature selection methods, we were able to create a robust model that can effectively predict house prices in the Austin area. This model can be a valuable tool for real estate investors, homebuyers, and policymakers to make informed decisions in the housing market since the comprehensive and rigorous data preprocessing to ensure that the model is making predictions or classifications based on relevant and accurate information. One potential limitation would be that the model was trained and tested on a specific dataset from a particular region, and its performance may be influenced by the unique characteristics and patterns of that region.

To further expand the potential applications of our work and to apply to other region's dataset,there might be some considerations on other external factors, such as the local economy, job market, and neighborhood amenities, to provide a comprehensive prediction of house prices.

References

1. Leung, K. (2022, September 14). Principal component regression - clearly explained and implemented. Medium. Retrieved May 3, 2023, from https://towardsdatascience.com/principal-component-regression-clearly-explained-and-implemented-608471530a2f
2. Pierce, E. (2021, April 12). Austin, TX House listings. Kaggle. Retrieved May 3, 2023, from https://www.kaggle.com/datasets/ericpierce/austinhousingprices
3. Géron, A. (2020). Hands-on machine learning with scikit-learn, Keras, and tensorflow: Concepts, tools, and techniques to build Intelligent Systems. O'Reilly.