Regression

Tan Wen Tao Bryan
2214449
DAAA/FT/2A/01

Project Objective

- To build a regression model to predict the housing price in US based on a few various factors such as city, house area, No. of bedrooms and toilets, renovation status, etc.

Step 1: EDA

In [4]:
```
print(housing_ds.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 545 entries, 0 to 544
Data columns (total 8 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   House ID           545 non-null    int64
 1   City               545 non-null    object
 2   House Area (sqm)   545 non-null    float64
 3   No. of Bedrooms    545 non-null    int64
 4   No. of Toilets     545 non-null    int64
 5   Stories            545 non-null    int64
 6   Renovation Status  545 non-null    object
 7   Price ($)          545 non-null    int64
dtypes: float64(1), int64(5), object(2)
memory usage: 34.2+ KB
None
```

Observations:
- 545 rows and 8 columns
- 2 categorical columns, 5 numerical columns
- All columns has an equal number of observations and no null values
- House ID has a unique value for every row so it can be removed later
- Price ($) is the target variable as it fulfils the project objective

Unique Values for Columns with Categorical Data

In [5]:
```
#cols with categorical data
categorical_col = housing_ds.dtypes[housing_ds.dtypes=='object']
print(categorical_col)
print()

#prints unique categorical values
for cat_val in categorical_col.index:
    print(f'{cat_val} - {housing_ds[cat_val].unique()}')
```

```
City                object
Renovation Status   object
dtype: object

City - ['Chicago' 'Denver' 'Seattle' 'New York' 'Boston']
Renovation Status - ['furnished' 'semi-furnished' 'unfurnished']
```

Missing Values

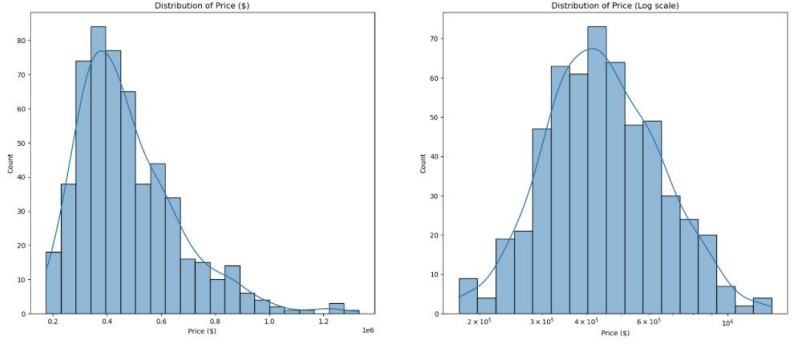
In [6]:
```
#check for missing values in the dataframe
print(housing_ds.isna().sum().sort_values())
```

```
House ID            0
City                0
House Area (sqm)    0
No. of Bedrooms     0
No. of Toilets      0
Stories             0
Renovation Status   0
Price ($)           0
dtype: int64
```

Observations:
- No missing values for the dataset so there is no need for imputation

Target Variable

In [8]:
```
#plot a histograph to show distribution of target variable (price)
fig, ax = plt.subplots(1,2, figsize=(20,8))
sns.histplot(data=housing_ds, x="Price ($)", kde=True, ax=ax[0])
ax[0].set_title("Distribution of Price ($)")
sns.histplot(data=housing_ds, x="Price ($)", kde=True, log_scale=True, ax=ax[1])
ax[1].set_title("Distribution of Price (Log scale)")
plt.show()
```

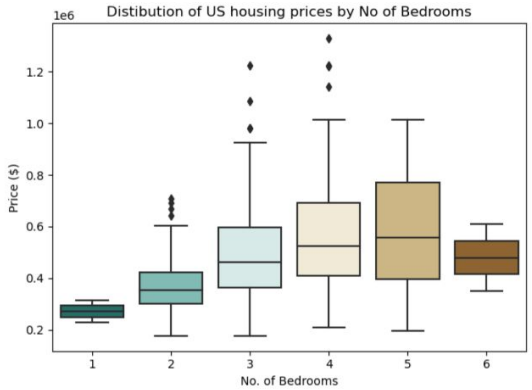


Observations:
- Histogram shows that the distribution of housing prices in the US is positively skewed, mostly costing between $ 300 000 & $400 000
- Using a log transformation will deskew the data, making it normalised
- Shows that US housing prices follow a log normal distribution

Step 1: EDA (Numerical Variables)

Number of Bedrooms

```
# Distribution of Houses Prices by No. Of Bedrooms
sns.boxplot(data=housing_ds, x="No. of Bedrooms", y="Price ($)", palette='BrBG_r')
plt.title("Distibution of US housing prices by No of Bedrooms")
plt.tight_layout()
plt.show()
```



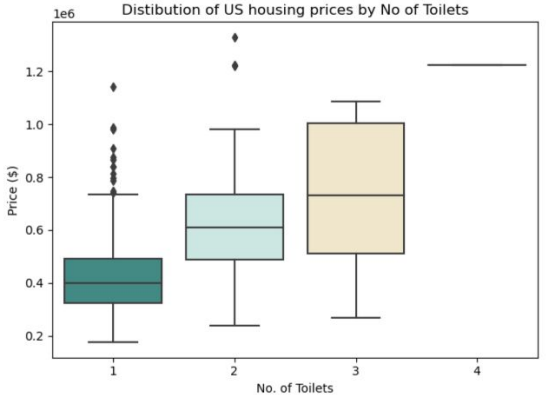
Observations:
- Generally, the greater the number of bedrooms, the greater the price of the houses in US
- An exception would be a 6-room house in the US is cheaper than a 4-room house
- Prices of a 6-room house is also more consistent than 3-5 room house
- Distribution is quite symmetrical for all variables

Number of Stories

```
# Distribution of Houses Prices by No. Of Stories
sns.boxplot(data=housing_ds, x="Stories", y="Price ($)", palette='BrBG_r')
plt.title("Distibution of US housing prices by No of Stories")
plt.tight_layout()
plt.show()
```

Number of Toilets

```
# Distribution of Houses Prices by No. Of Toilets
sns.boxplot(data=housing_ds, x="No. of Toilets", y="Price ($)", palette='BrBG_r')
plt.title("Distibution of US housing prices by No of Toilets")
plt.tight_layout()
plt.show()
```




```
#cols with categorical data
Toilets4 = housing_ds[housing_ds['No. of Toilets']==4]
display(Toilets4)
print(len(Toilets4))
```

	House ID	City	House Area (sqm)	No. of Bedrooms	No. of Toilets	Stories	Renovation Status	Price ($)
1	1	Denver	896.0	4	4	4	furnished	1225000

1

Observations:
- Generally, the greater the number of toilets, the greater the price of houses in US
- There is only 1 house in the dataset with 4 toilets hence the IQR of houses with 4 toilets is constant
- Distribution is quite symmetrical in general.

Step 1: EDA (Bivariate Relationship)

In [14]:
```
plt.figure(figsize=(15,8))
sns.heatmap(housing_ds.corr(), annot=True)
plt.title("Relationship between numerical variables")
plt.show()
```

```
C:\Users\bryan\AppData\Local\Temp\ipykernel_38232\422641813.py:2: FutureWarning: The default value of numeric_only in DataFram
e.corr is deprecated. In a future version, it will default to False. Select only valid columns or specify the value of numeric_
only to silence this warning.
  sns.heatmap(housing_ds.corr(), annot=True)
```

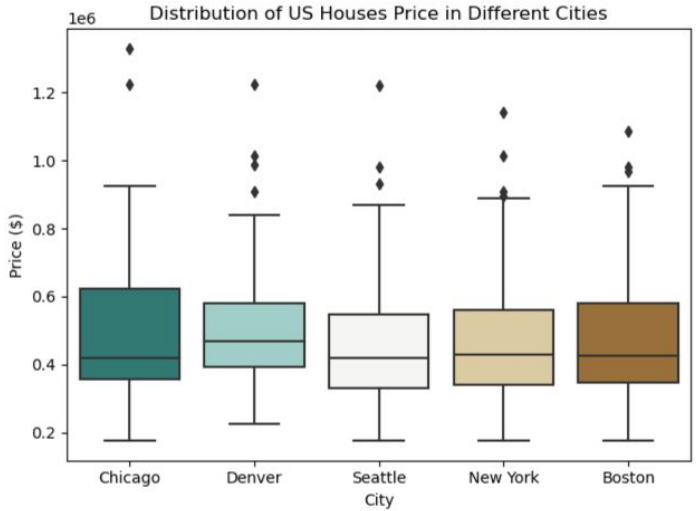
Observations:

- Positive moderate linear relationship between House Area & Price of houses in US

Step 1: EDA (Categorical Variables)

Cities

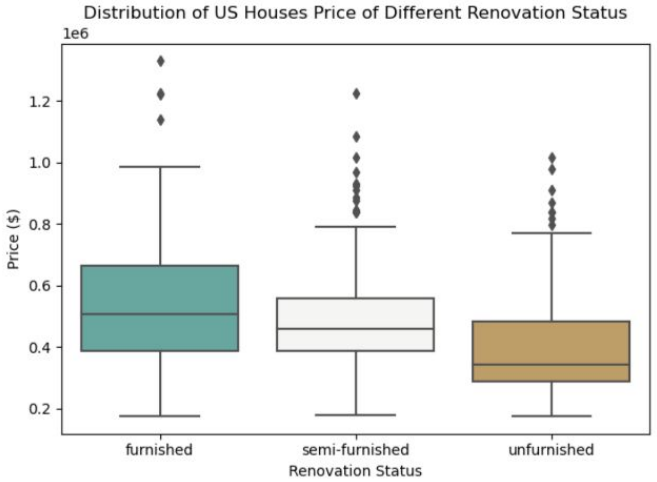
In [15]:
```
# Distribution of US Houses in Different Cities
sns.boxplot(data=housing_ds, x="City", y="Price ($)", palette='BrBG_r')
plt.title("Distribution of US Houses Price in Different Cities")
plt.tight_layout()
plt.show()
```



Observations:
- Houses in Denver seems to be the most expensive among the other cities.
- Houses in other cities are quite similar in price.

Renovation Status

In [16]:
```
# Distribution of US Houses Price of Different Renovation Status
sns.boxplot(data=housing_ds, x="Renovation Status", y="Price ($)", palette='BrBG_r')
plt.title("Distribution of US Houses Price of Different Renovation Status")
plt.tight_layout()
plt.show()
```



Observations:
- Furnished houses are the most expensive, followed by semi-furnished then unfurnished.
- Semi-furnished houses are the most consistent in the prices sold.

Step 2: Data Cleaning/Feature Engineering

Drop Unwanted Columns

- Drop House ID as it is unique for every row

```
housing_df.drop("House ID", axis=1, inplace=True)
display(housing_df.head())
```

	City	House Area (sqm)	No. of Bedrooms	No. of Toilets	Stories	Renovation Status	Price ($)
0	Chicago	742.0	4	2	3	furnished	1330000
1	Denver	896.0	4	4	4	furnished	1225000
2	Chicago	996.0	3	2	2	semi-furnished	1225000
3	Seattle	750.0	4	2	2	furnished	1221500
4	New York	742.0	4	1	2	furnished	1141000

Separate Features & Target labels

```
X, y = housing_df.drop("Price ($)", axis=1), housing_df["Price ($)"]
```

Split Testing & Training data

- Split data 80/20 training/testing data to train and test the model

```
#Split test and training data for features and target label
#random_state sets a seed to the random generator
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, shuffle=True, random_state=42)
print(X_train.shape)
print(X_test.shape)
```

```
(436, 6)
(109, 6)
```

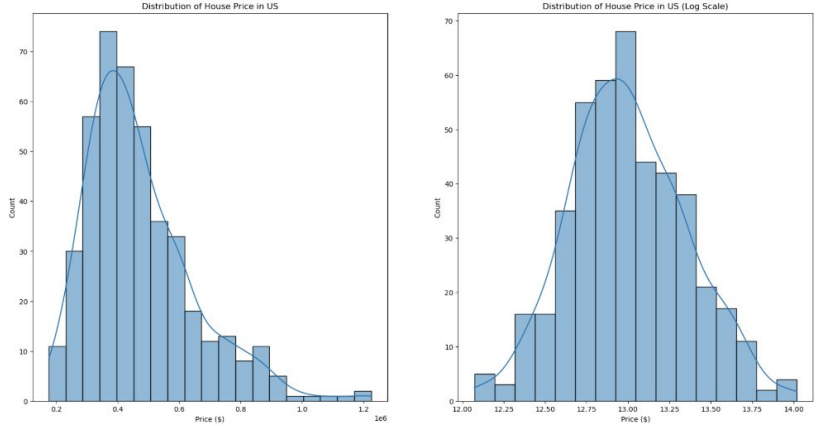
Step 2: Data Cleaning/Feature Engineering

Price (Target)

- From the histogram, we can see that the distribution of house price in US is positively skewed.
- To make the target variable a normal distribution, we shall apply logarithm scale onto the target variable by using np.log1p
- Need to add 1 to X because if X=0, there will be no value

$$newX = log(1 + X)$$

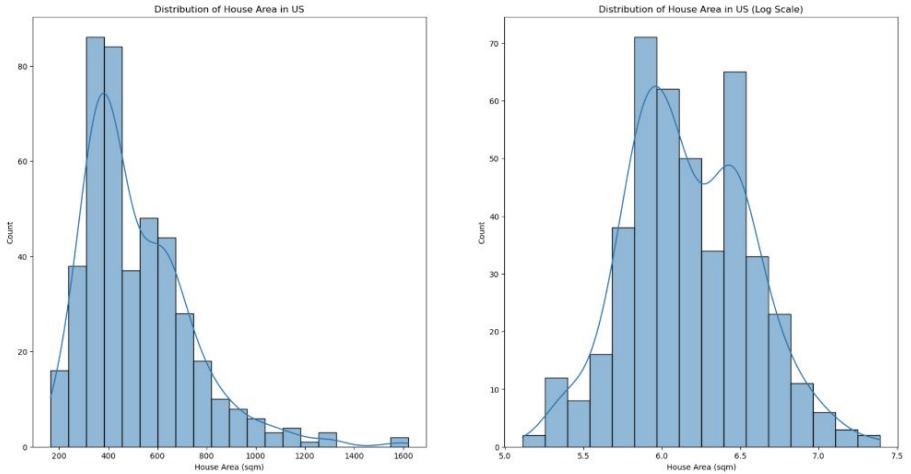
```
fig, ax = plt.subplots(1, 2, figsize=(20,10))
sns.histplot(y_train, kde=True, ax=ax[0])
ax[0].set_title("Distribution of House Price in US")
sns.histplot(y_train.apply(np.log1p), kde=True, ax=ax[1])
ax[1].set_title("Distribution of House Price in US (Log Scale)")
plt.show()
```



House Area

- Apply logscale to House Area too

```
fig, ax = plt.subplots(1, 2, figsize=(20,10))
sns.histplot(X_train["House Area (sqm)"], kde=True, ax=ax[0])
ax[0].set_title("Distribution of House Area in US")
sns.histplot(X_train["House Area (sqm)"].apply(np.log1p), kde=True, ax=ax[1])
ax[1].set_title("Distribution of House Area in US (Log Scale)")
plt.show()
```



- Use sklearn TransformedTargetRegressor to transform target variable Price distribution into normal distribution through using np.log1p

Step 2: Feature Engineering (Pipeline)

Feature Engineering

- Engineer new features to train the model to explain new patterns

Base Area

- Shows the 2D area of the house

$$BaseArea = \frac{HouseArea}{No.OfStories}$$

```
housing_ds["House Base Area"] = housing_ds["House Area (sqm)"]/housing_ds["Stories"]
display(housing_ds.head())

X_trainSubset = X_train.copy()
```

	House ID	City	House Area (sqm)	No. of Bedrooms	No. of Toilets	Stories	Renovation Status	Price ($)	House Base Area
0	0	Chicago	742.0	4	2	3	furnished	1330000	247.333333
1	1	Denver	896.0	4	4	4	furnished	1225000	224.000000
2	2	Chicago	996.0	3	2	2	semi-furnished	1225000	498.000000
3	3	Seattle	750.0	4	2	2	furnished	1221500	375.000000
4	4	New York	742.0	4	1	2	furnished	1141000	371.000000

Area per Toilet/Area per Bedroom

- Tried new different features but it made the scores worse

$$AreaPerToilet = \frac{HouseArea}{No.OfToilets}$$

$$AreaPerBedroom = \frac{HouseArea}{No.OfBedrooms}$$

```
#     housing_ds["Area per Toilet"] = housing_ds["House Area (sqm)"]/housing_ds["No. of Toilets"]
#     housing_ds["Area per Bedroom"] = housing_ds["House Area (sqm)"]/housing_ds["No. of Bedrooms"]
```

Pipeline

- To link the steps of data engineering and the model implementation together

```
#Feature Engineering
def feature_engineering(df):
    df=pd.DataFrame(df.reset_index(drop=True))
    df["House Base Area"] = df["House Area (sqm)"]/df["Stories"]
    return df

#Logscale
def apply_log(df):
    df["House Area (sqm)"]= np.log1p(df["House Area (sqm)"])
    return df

#Categorical transformer (OneHotEncode + OrdinalEncode)
categoricalTransformer = ColumnTransformer(
    [
        ('oneHotEnc', OneHotEncoder(categories='auto', sparse_output=False), ['City']),
        ('ordinalEnc', OrdinalEncoder(categories=[['furnished','semi-furnished', 'unfurnished']]), ['Renovation Status'])
    ],
    remainder='passthrough'
)

#Numerical transformer
num_cols = ['House Area (sqm)', 'No. of Bedrooms','No. of Toilets','Stories','House Base Area']
numericalTransformer = ColumnTransformer([
    ("standardise", StandardScaler(), num_cols)
],
remainder="passthrough")

#Combining two transformers into preprocessor
preprocessor=ColumnTransformer([
    ("categorical", categoricalTransformer, ["City","Renovation Status"]),
    ("numerical", numericalTransformer, num_cols)
],
remainder="passthrough")

#Building the steps
steps = [
    ("featureEngineering", FunctionTransformer(feature_engineering)),
    ("logScale", FunctionTransformer(apply_log)),
    ("preprocessing", preprocessor),
    ("model")
]

model_step=len(steps)-1
```

LogScale: House Area
OneHotEncode: City
OrdinalEncode: Renovation Status
StandardScaler: All numerical variables

Step 3: Model Selection

- Tried out 9 models and chose **top 3 models** based on the **highest R2 and RMSE** score

Gradient Boosting Regressor

- Ensemble learning method that combines multiple weak or base learners in a stage wise function where each learner is trained to minimise the errors of the previous model
- Uses mean squarred error (MSE) function for regression
- Uses more complex weak learners

Random Forest Regressor

- Make use of bagging where the output of each decision tree averaged to make the final prediction
- Can capture non linear relationships between input features and target feature
- Reduces risk of overfitting
- Robust to noisy data, irrelevant features

Ridge

- Adds "squared magnitude" of coefficient as penalty term to the loss function
- If lambda is very large, it will add too much weight and result in underfitting

$$\hat{\beta}^{ridge} = \arg\min_{\beta} \left(\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right)$$

```
#Defining the models used with default hyperparameters
models = [("LinearRegression", LinearRegression()),
          ("AdaBoostRegressor", AdaBoostRegressor(random_state=42)),
          ("GradientBoostingRegressor", GradientBoostingRegressor(random_state=42)),
          ("RandomForestRegressor", RandomForestRegressor(random_state=42)),
          ("KNeighborsRegressor", KNeighborsRegressor()),
          ("DecisionTreeRegressor", DecisionTreeRegressor(random_state=42)),
          ("SGDRegressor", SGDRegressor(random_state=42)),
          ("Ridge", Ridge(random_state=42)),
          ("Lasso", Lasso(random_state=42))
]
```

- 6 other models chosen were **Linear Regression, AdaBoost, KNN, Decision Tree, Lasso & SGD**

Step 4: Model Evaluation

4) Model Evaluation

As such, the evaluation metrics below are the ones that I have chosen to evaluate my models

- Prioritise R2 & RMSE

R2

- Coefficient of Determination
- Measures the proportion of the variance in the dependent variable that can be explained by the independent variables in a regression model

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

RMSE

- Square root of the mean of the square of all the error
- RMSE is used more instead of MSE as MSE values are too big to compare with

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

MAE

- Measures the average magnitude of errors in the regression model

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

MAPE

- Measures the average magnitude of errors in the regression model as a percentage
- Measure of predicted accuracy

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \times 100$$

MSE

- Measures the performance of regression models
- By squaring the differences, MSE gives more weight to larger errors
- The result is a non-negative value, with lower values indicating better model performance
- Use the root squared version for a better way to measure

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

Step 4: Model Evaluation

Models Selection

In [32]:
```
#Function for the model
def evaluate_Model(X_train, y_train, models, scoring):
    avgModelScores=[]
    #Iterate through the models
    for name, model in models:
        steps[model_step]=('model', TransformedTargetRegressor(regressor = model, func=np.log1p, inverse_func=np.expm1))
        score=cross_validate(
            Pipeline(steps=steps),
            X_train,
            y_train,
            scoring=scoring,
            cv=10,
            return_train_score=True
        )
        avgModelScores.append(pd.Series(score, name=name).apply(np.mean))
    return pd.DataFrame(avgModelScores).sort_values(by=[
        "test_r2",
        "test_neg_root_mean_squared_error"
    ],ascending=False)

model_Scores=evaluate_Model(X_train, y_train, models, score_metrics)
```

In [33]:
```
model_Scores
```

Out[33]:

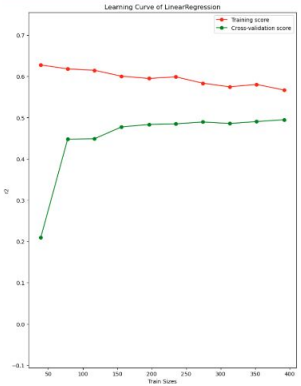
	fit_time	score_time	test_r2	train_r2	test_neg_root_mean_squared_error	train_neg_root_mean_squared_error	st_neg_mea
GradientBoostingRegressor	0.048992	0.004413	0.517955	0.805744	-115499.825667	-77327.579417	
RandomForestRegressor	0.127123	0.009313	0.499724	0.922848	-118228.676578	-48725.617698	
Ridge	0.017093	0.007359	0.495915	0.567285	-118042.812982	-115437.768785	
LinearRegression	0.012493	0.003805	0.495148	0.567126	-118113.848542	-115459.018218	
AdaBoostRegressor	0.040467	0.005372	0.475672	0.635777	-121222.763402	-105862.359030	
KNeighborsRegressor	0.021252	0.014230	0.472768	0.641031	-120590.431710	-105126.490673	
SGDRegressor	0.021871	0.012998	0.421715	0.491480	-126887.786508	-125147.866100	
DecisionTreeRegressor	0.020513	0.012262	0.020001	0.998573	-158908.952846	-6575.846630	
Lasso	0.019345	0.012904	-0.063704	-0.027097	-174682.841189	-177875.427506	

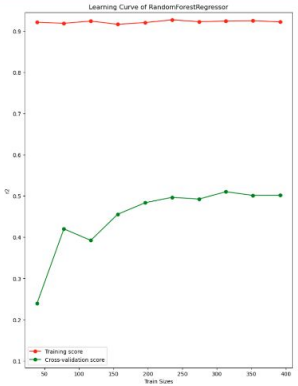
- Gradient Boosting
- Random Forest
- Ridge

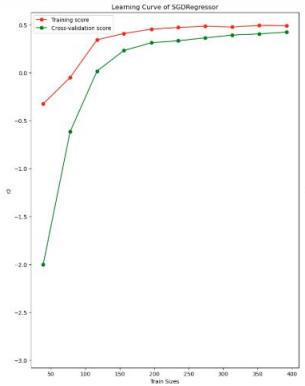
Observations:

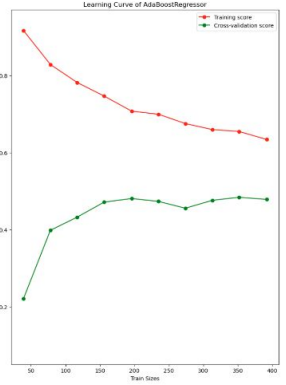
- Due to insufficient data, the r2 score of all the models are not higher than 0.56 which shows that the relationship of the data fitted to the model is only moderate.
- Top best 3 models based on test r2 score and negative root mean squared error are GradientBoostingRegressor, RandomForestRegressor & Ridge.
- Lasso did not do as well as it might have removed some features altogether
- For Decision Tree, the data overfits the model as the train r2 score of 0.99 can drop to 0.02 in the train r2 score

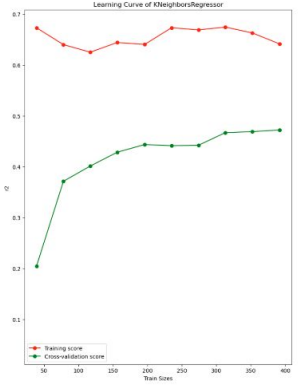
Step 4: Model Evaluation (Learning Curve)

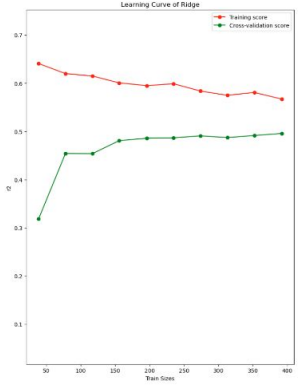




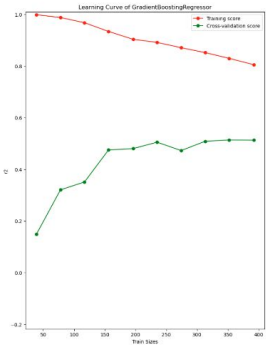


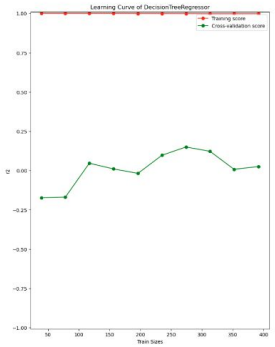


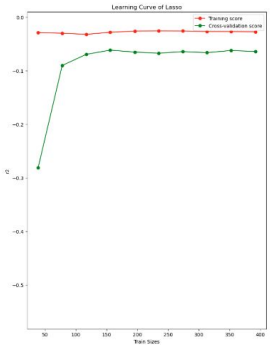




- Shows **relationship of a learner's performance and number of attempts** on a task







Observations:

- Generally, as more data is fitted on to the models, their cross-validation scores increased.
- Negative scores for Lasso show that the data is not suitable for the model as it shrinks the less important features coeffiecient to 0, which will remove these features.
- Decision Tree training score has a constant value of 1.00 which shows that the data has been overfitted to the model

Step 4: Model Evaluation (Dummy Regressor)

- Comparing top 3 best models against Dummy Regressor
- Also had used scores of DummyRegressor (see in Jupyter)

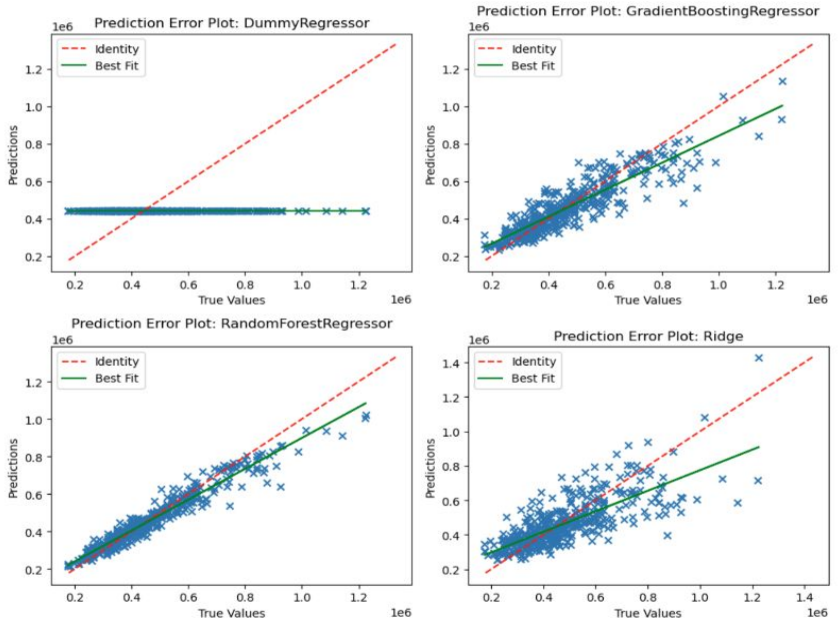
Prediction Error Plot

- Determine the performance of training data for each regressor based on the best-fit line plotted with the data points

```
#Building function to build a prediction error plot
def plot_predictionErrorPlot(model, X, y, ax=None):
    #Store the model name
    try:
        model_name = type(model[-1].regressor).__name__
    except:
        try:
            model_name = type(model[-1]).__name__
        except:
            model_name = type(model).__name__
    y_pred = model.predict(X)
    if ax is None:
        fig, ax = plt.subplots(figsize=(15,15))
    ax.scatter(y, y_pred, marker="x")
    l1=max(max(y_pred), max(y_test))
    l2=min(min(y_pred), min(y_test))
    ax.plot([l1, l2], [l1, l2], "r--", label="Identity")
    a, b = np.polyfit(y, y_pred, 1)
    ax.plot(y, a * y + b, "g-", label="Best Fit")
    ax.legend()
    ax.set_xlabel("True Values")
    ax.set_ylabel("Predictions")
    ax.set_title(f"Prediction Error Plot: {model_name}")
    print(f"{model_name}'s R2: {r2_score(y,y_pred)}")
    return ax
```

```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(10,8))
plot_predictionErrorPlot(dummy, X_train, y_train, ax1)
plot_predictionErrorPlot(gradientBoost_reg, X_train, y_train, ax2)
plot_predictionErrorPlot(randomForest_reg, X_train, y_train, ax3)
plot_predictionErrorPlot(ridge_reg, X_train, y_train, ax4)
plt.tight_layout()
plt.show()
```

```
DummyRegressor's R2: -0.027103635063244536
GradientBoostingRegressor's R2: 0.7916794523111546
RandomForestRegressor's R2: 0.9225784675742272
Ridge's R2: 0.5662274861035446
```



Observations:

- DummyRegressor Best Fit line is completely off the Identity line, with the R2 score as -0.03 so the model does not follow the trend of the housing prices in US.
- GradientBoostingRegressor, RandomForestRegressor Best Fit line is slightly below the Identity line which shows that the model follows the trend of the housing prices in US at a strong positive linear rate.
- Further supported by the R2 score of 0.84 and 0.93 respectively
- Ridge best fit line is slightly below the Identity line which suggests that the housing prices in US is increasing at a moderate positive rate.

Step 5: Model Improvement (GridSearchCV)

5) Model Improvement

Hyperparameter Tuning (GridSearchCV)

Run through the parameters to see which parameters give the best r2 score

Parameters to be tuned (GradientBoostingRegressor):

1. max_depth - maximum depth of the tree
2. max_leaf_nodes - maximum number of leaves nodes
3. n_estimators - numbers of trees in the forest
4. criterion - measure the quality of a split {"friedman_mse",'squared_error'}

Parameters to be tuned (RandomForestRegressor):

1. max_depth - maximum depth of the tree
2. max_leaf_nodes - maximum number of leaves nodes
3. n_estimators - numbers of trees in the forest
4. criterion - measure the quality of a split {"friedman_mse",'squared_error'}

Parameters to be tuned (Ridge):

1. alpha - controls the amount of regularization applied to the model (Adds a penalty term to the loss function, which is proportional to the sum of the squared coefficients multiplied by the alpha parameter)

- As alpha value increase, regularization increases, helps to reduce overfitting and can improve generalization on unseen data
- As alpha value decrease, regularization decreases, allowing model to fit the training data more closely, potentially increasing the risk of overfitting.

2. solver - Solver used in computation {'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga', 'lbfgs'}

Why GridSearchCV and not RandomizedSearchCV?

- GridSearchCV exhaustively considers all hyperparameter combination but
- RandomizedSearchCV can sample a given number of candidates from a hyperparameter space with a specified distribution.

Step 5: Model Improvement (GridSearchCV)

Gradient Boosting Regressor

```
list(GradientBoostingRegressor().get_params().keys())
```

```
['alpha',
 'ccp_alpha',
 'criterion',
 'init',
 'learning_rate',
 'loss',
 'max_depth',
 'max_features',
 'max_leaf_nodes',
 'min_impurity_decrease',
 'min_samples_leaf',
 'min_samples_split',
 'min_weight_fraction_leaf',
 'n_estimators',
 'n_iter_no_change',
 'random_state',
 'subsample',
 'tol',
 'validation_fraction',
 'verbose',
 'warm_start']
```

```
# Create the parameter grid
params_grid = {
    "max_depth": [5,10,20,30,40,50,60,70,80,90,100],
    "max_leaf_nodes": np.arange(10, 16),
    "n_estimators": np.arange(50,201,50),
    "criterion": ["friedman_mse",'squared_error']
}

# Creating a model based on the pipeline
steps[model_step] = (
    "hyper",
    GridSearchCV(
        GradientBoostingRegressor(min_samples_split=2, min_samples_leaf=1, random_state=42),
        params_grid,
        cv=10,
        verbose=1,
        n_jobs=-1,
        scoring="r2"
    )
)

gradientBoost_search = Pipeline(steps=steps)
# Fitting Model
gradientBoost_search.fit(X_train, y_train)

print(gradientBoost_search.named_steps["hyper"].best_estimator_)
print(gradientBoost_search.named_steps["hyper"].best_params_)
print(gradientBoost_search.named_steps["hyper"].best_score_)
```

```
Fitting 10 folds for each of 528 candidates, totalling 5280 fits
GradientBoostingRegressor(criterion='squared_error', max_depth=10,
                          max_leaf_nodes=11, n_estimators=50, random_state=42)
{'criterion': 'squared_error', 'max_depth': 10, 'max_leaf_nodes': 11, 'n_estimators': 50}
0.5229922671920416
```

Random Forest Regressor

```
list(RandomForestRegressor().get_params().keys())
```

```
['bootstrap',
 'ccp_alpha',
 'criterion',
 'max_depth',
 'max_features',
 'max_leaf_nodes',
 'max_samples',
 'min_impurity_decrease',
 'min_samples_leaf',
 'min_samples_split',
 'min_weight_fraction_leaf',
 'n_estimators',
 'n_jobs',
 'oob_score',
 'random_state',
 'verbose',
 'warm_start']
```

```
# Create the parameter grid
params_grid = {
    "max_depth": [5,10,20,30,40,50,60,70,80,90,100],
    "max_leaf_nodes": np.arange(10, 16),
    "n_estimators": np.arange(50,501,50),
    "criterion": ["friedman_mse",'squared_error']
}

# Creating a model based on the pipeline
steps[model_step] = (
    "hyper",
    GridSearchCV(
        RandomForestRegressor(
            min_samples_split=2, min_samples_leaf=1, random_state=42
        ),
        params_grid,
        cv=10,
        verbose=1,
        n_jobs=-1,
        scoring="r2"
    ),
)

random_forest_search = Pipeline(steps=steps)
# Fitting Model
random_forest_search.fit(X_train, y_train)
print(random_forest_search.named_steps["hyper"].best_estimator_)
print(random_forest_search.named_steps["hyper"].best_params_)
print(random_forest_search.named_steps["hyper"].best_score_)
```

```
Fitting 10 folds for each of 1320 candidates, totalling 13200 fits
RandomForestRegressor(criterion='friedman_mse', max_depth=5, max_leaf_nodes=15,
                      n_estimators=250, random_state=42)
{'criterion': 'friedman_mse', 'max_depth': 5, 'max_leaf_nodes': 15, 'n_estimators': 250}
0.5150807894037903
```

Ridge

```
list(Ridge().get_params().keys())
```

```
['alpha',
 'copy_X',
 'fit_intercept',
 'max_iter',
 'positive',
 'random_state',
 'solver',
 'tol']
```

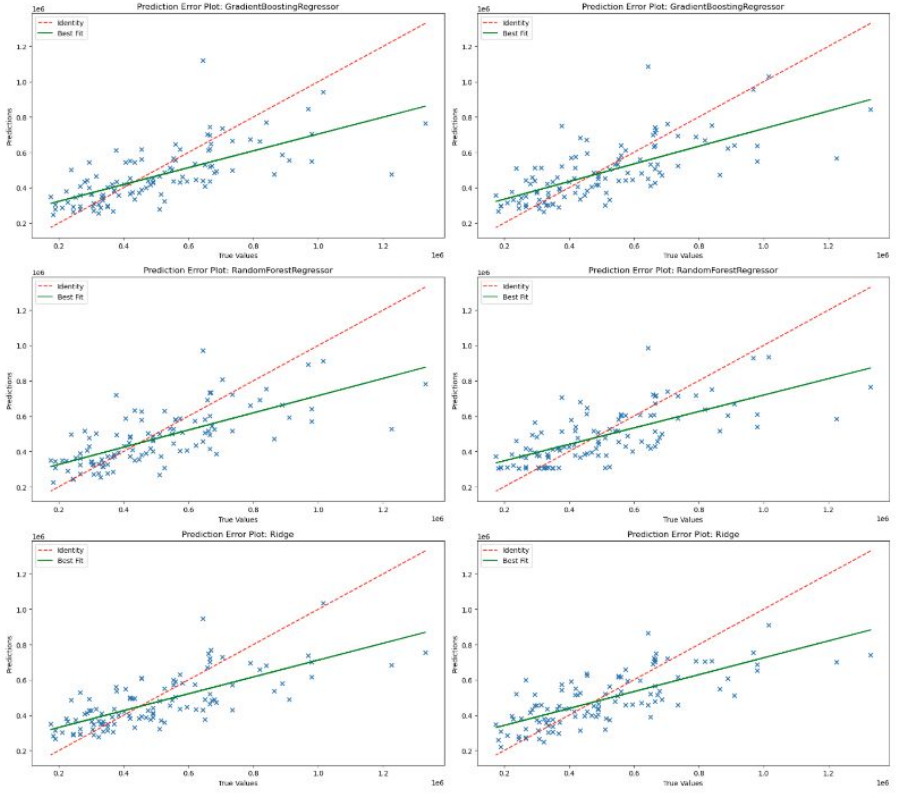
```
import warnings
warnings.filterwarnings('ignore')

# Create the parameter grid
params_grid = {
    "alpha": [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1],
    "solver": ['auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga', 'lbfgs'],
    "max_iter": np.arange(1000, 10001, 1000)
}

# Creating a model based on the pipeline
steps[model_step] = (
    "hyper",
    GridSearchCV(
        Ridge(
            random_state=42
        ),
        params_grid,
        cv=10,
        verbose=1,
        n_jobs=-1,
        scoring="r2"
    ),
)

ridge_search = Pipeline(steps=steps)
# Fitting Model
ridge_search.fit(X_train, y_train)
print(ridge_search.named_steps["hyper"].best_estimator_)
print(ridge_search.named_steps["hyper"].best_params_)
print(ridge_search.named_steps["hyper"].best_score_)
```

```
Fitting 10 folds for each of 720 candidates, totalling 7200 fits
Ridge(alpha=1, max_iter=1000, random_state=42, solver='saga')
{'alpha': 1, 'max_iter': 1000, 'solver': 'saga'}
0.500480463385858
```

Step 5: Evaluate Improved Model (Prediction Error Plot)

```
: #Prediction Error Plot
fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6)) = plt.subplots(3, 2, figsize= (18, 16))

#Untuned
print("Untuned Models")
plot_predictionErrorPlot(gradientBoost_reg, X_test, y_test, ax1)
plot_predictionErrorPlot(randomForest_reg, X_test, y_test, ax3)
plot_predictionErrorPlot(ridge_reg, X_test, y_test, ax5)
print()

#Tuned
print("Tuned Models")
plot_predictionErrorPlot(tuned_gradientBoost_reg, X_test, y_test, ax2)
plot_predictionErrorPlot(tuned_random_forest_reg, X_test, y_test, ax4)
plot_predictionErrorPlot(tuned_ridge_reg, X_test, y_test, ax6)

plt.tight_layout()
plt.show()
```

```
Untuned Models
GradientBoostingRegressor's R2: 0.45742997984449085
RandomForestRegressor's R2: 0.4756221163558043
Ridge's R2: 0.5077743849097727

Tuned Models
GradientBoostingRegressor's R2: 0.47266638957875795
RandomForestRegressor's R2: 0.4770359335769375
Ridge's R2: 0.5280843173397289
```



Observations:

- Generally, the best fit line is placed slightly below the Identity line, hence shows a moderate positive linear relationship.
- Based on the test data r2 score, tuned models all performed better than the untuned models from what I have selected.
- Looking at only test data r2 score, the best model is Ridge.

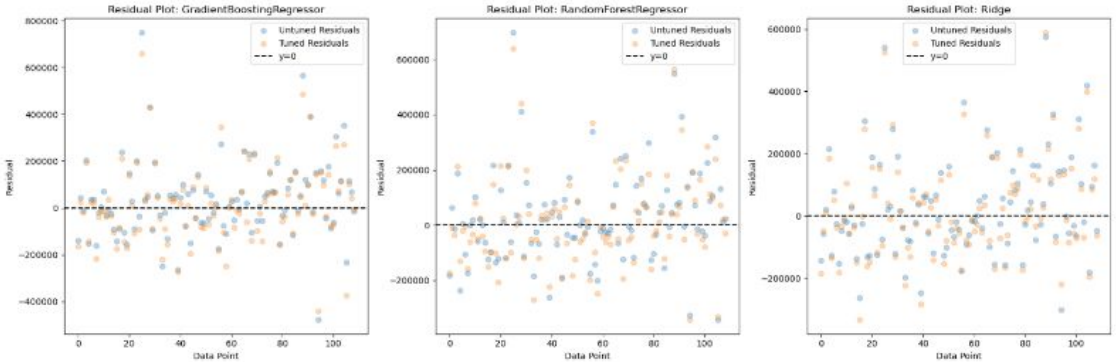
Step 5: Evaluate Improved Model (Residual Plot)

Residual Plot

- Shows the difference between predicted and actual data points

```
def plot_residual_plot(model, X, y, ax=None):
    try:
        model_name = type(model[-1].regressor).__name__
    except:
        try:
            model_name = type(model[-1]).__name__
        except:
            model_name = type(model).__name__
    y_pred = model.predict(X)
    if ax is None:
        fig, ax = plt.subplots(figsize=(15, 15))
    residual = y - y_pred
    ax.scatter(range(len(residual)), residual, alpha=0.3)
    ax.set_xlabel("Data Point")
    ax.set_ylabel("Residual")
    ax.set_title(f"Residual Plot: {model_name}")
    return ax
```

```
#Residual Plot
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(18,6))
plot_residual_plot(gradientBoost_reg, X_test, y_test,ax=ax1)
plot_residual_plot(tuned_gradientBoost_reg, X_test, y_test,ax=ax1)
plot_residual_plot(randomForest_reg, X_test, y_test,ax=ax2)
plot_residual_plot(tuned_random_forest_reg, X_test, y_test,ax=ax2)
plot_residual_plot(ridge_reg, X_test, y_test,ax=ax3)
plot_residual_plot(tuned_ridge_reg, X_test, y_test,ax=ax3)
ax1.axhline(y=0, linestyle="--", color='black')
ax2.axhline(y=0, linestyle="--", color='black')
ax3.axhline(y=0, linestyle="--", color='black')

ax1.legend(["Untuned Residuals", "Tuned Residuals", "y=0"],loc="best")
ax2.legend(["Untuned Residuals", "Tuned Residuals", "y=0"],loc="best")
ax3.legend(["Untuned Residuals", "Tuned Residuals", "y=0"],loc="best")
plt.tight_layout()
plt.show()
```



Observations:

- Based on outliers alone, Ridge does not have outliers that exceed 600000 whereas Random Forest and Gradient Boosting have outliers that exceed 600000.
- This implies that the actual and predicted values for Ridge do not have such a big difference compared to the other 2 models which shows that Ridge is the best model.

Step 5: Evaluate Improved Model (Using test data)

```
#Show the RMSE & R2 values of test data

pd.DataFrame([[
    mean_squared_error(y_test, gradientBoosting_pred, squared=False),
    mean_squared_error(y_test, random_forest_pred, squared=False),
    mean_squared_error(y_test, ridge_predict, squared=False)],
    [r2_score(y_test, gradientBoosting_pred),
     r2_score(y_test, random_forest_pred),
     r2_score(y_test, ridge_predict)]],
    columns=["GradientBoostingRegressor", "RandomForestRegressor", "Ridge"],
    index=["Root Mean Squared Error","R2"],
)
```

	GradientBoostingRegressor	RandomForestRegressor	Ridge
Root Mean Squared Error	163261.899326	162584.089322	154445.167240
R2	0.472666	0.477036	0.528084

Observations:

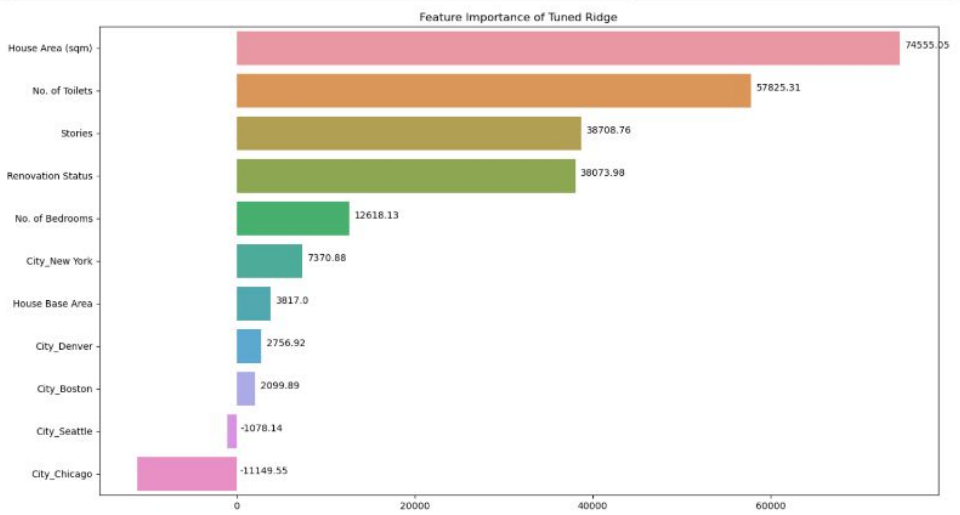
- Based on the root mean squared error, Ridge has the lowest value which implies smaller prediction errors and better performance of the model in capturing the variability of the data.
- Based on the R2 value, Ridge has the best r2 values which implies that it has the best performance

Step 5: Evaluate Improved Model (Feature Importance)

Feature Importance

- Find out which features tuned Ridge considers important (Best Model)

```
feature_names=["City_Boston","City_Chicago","City_Denver","City_New York","City_Seattle","Renovation Status",'House Area (sqm)',

#Create feature importance and sort the features based on the coefficient
importance = pd.Series(
tuned_ridge_reg.named_steps["model"].coef_, index=feature_names
).sort_values(ascending=False)
fig, ax = plt.subplots(figsize=(16, 9))
sns.barplot(
    x=importance.values, y=importance.index, ax=ax
)
ax.set_title("Feature Importance of Tuned {}".format("Ridge"))
for i, v in zip(np.arange(0, len(importance)), importance.values):
    if v >=0:
        ax.text(x=v+600, y=i, s=round(v, 2))
    elif v>= -1500 and v<0:
        ax.text(x=v+1500, y=i, s=round(v, 2))
    elif v<= -12000:
        ax.text(x=v+38200, y=i, s=round(v, 2))
    else:
        ax.text(x=v+11500, y=i, s=round(v, 2))
plt.show()
```



Observations:

- Top 3 features the Ridge has considered most important are House Area (sqm), No. of Toilets and Stories
- City the model considers most important is New York

Summary

- Ridge is my **best model** based on the **r2 score & RMSE** score metrics.
- Due to the **lack of data**, the r2 score of all models used to **predict the price of the houses in US are all quite moderate (0.5+) range**.
- Data only has 545 rows which shows there is **not enough housing data** to **represent the houses in the mentioned cities of the US**.
- For the model to **fully predict** the price of the houses in US, there must be **more data collected** on the houses in **each city of US** including **more features of the houses** like the housing type.

Thank You!

References

1. Gomez, J., 2022. 8 critical factors that influence a home's value. [online]. Opendoor.
Available at: https://www.opendoor.com/articles/factors-that-influence-home-value [Accessed at 4 Jun 2022].

2. sklearn. Plotting Learning Curves and Checking Models'Scalability. [online]. Available at
https://towardsdatascience.com/understanding-feature-importance-and-how-to-implement-it-in-python-ff0287b20285 [Accessed at 9 Jun 2023]

3. Classical ML Equations in LaTeX.[online] GitHub. https://blmoistawinde.github.io/ml_equations_latex/ [Accessed at 9 Jun 2023].

4. Shin, T., 2023. Understanding Feature Importance and How to Implement it in Python. [online]. Medium. Available from:
https://towardsdatascience.com/understanding-feature-importance-and-how-to-implement-it-in-python-ff0287b20285 [Accessed at 9 Jun 2023]

5. Yellowbrick., 2019. Prediction Error Plot. [online]. Available from:
https://www.scikit-yb.org/en/latest/api/regressor/peplot.html#:~:text=A%20prediction%20error%20plot%20shows,variance%20is%20in%20the%20model.
[Accessed at 9 Jun 2023].

6. Interpreting Residual Plots to Improve Your Regression. [online]. Available from:
https://www.qualtrics.com/support/stats-iq/analyses/regression-guides/interpreting-residual-plots-improve-regression/ [Accessed at 9 Jun 2023]

Sklearn References

1. https://scikit-learn.org/stable/modules/generated/sklearn.compose.TransformedTargetRegressor.html
2. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
3. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html
4. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html