Vancouver House Price Forecast

Based on analysis of housing sales from 2019 to 2020

5001 Final Project -Wanyi Jiang

Foreword

- As an international student in Vancouver, I've grown deeply fond of this city and wish to establish my roots here. My aspirations led me to consider investing in my first property, a significant step for a young individual. However, navigating the Vancouver real estate market is daunting, primarily due to the dominance of intermediaries, creating a noticeable information gap and obstructing access to direct, reliable market insights.

- Motivated to overcome these challenges, I write a project aimed at quantifying real estate sales data to develop a predictive model. Utilizing data science and machine learning, this project seeks to transform extensive real estate data into an accessible predict tool for potential buyers. My goal is to provide a clear understanding of property values, enabling well-informed decisions that align with financial and personal goals.

- Next I will introduce in detail how I completed this project using python

Dataset

To begin the analyze, here’s a dataset I found for this project

Vancouver Home Price Analysis - Regression by Darian Ghorbanian

https://www.kaggle.com/datasets/darianghorbanian/vancouver-home-price-analysis-regression

This data set contains the following topics: Address, List Date, Price, Days on market, Total floor area, Year Built, Age, Lot Size.

Since the address data is temporarily not quantifiable, I excluded it and looked at the impact of other factors on the price.

Number	Address	List Date	Price	Days on market	Total floor area	Year Built	Age	Lot Size
1	GRAVELEY STREET	5/8/2020	1500000	18	2447	1946	74	5674.0
2	E 28TH AVENUE	1/22/2020	1300000	7	2146	1982	38	3631.98

Version 1.0

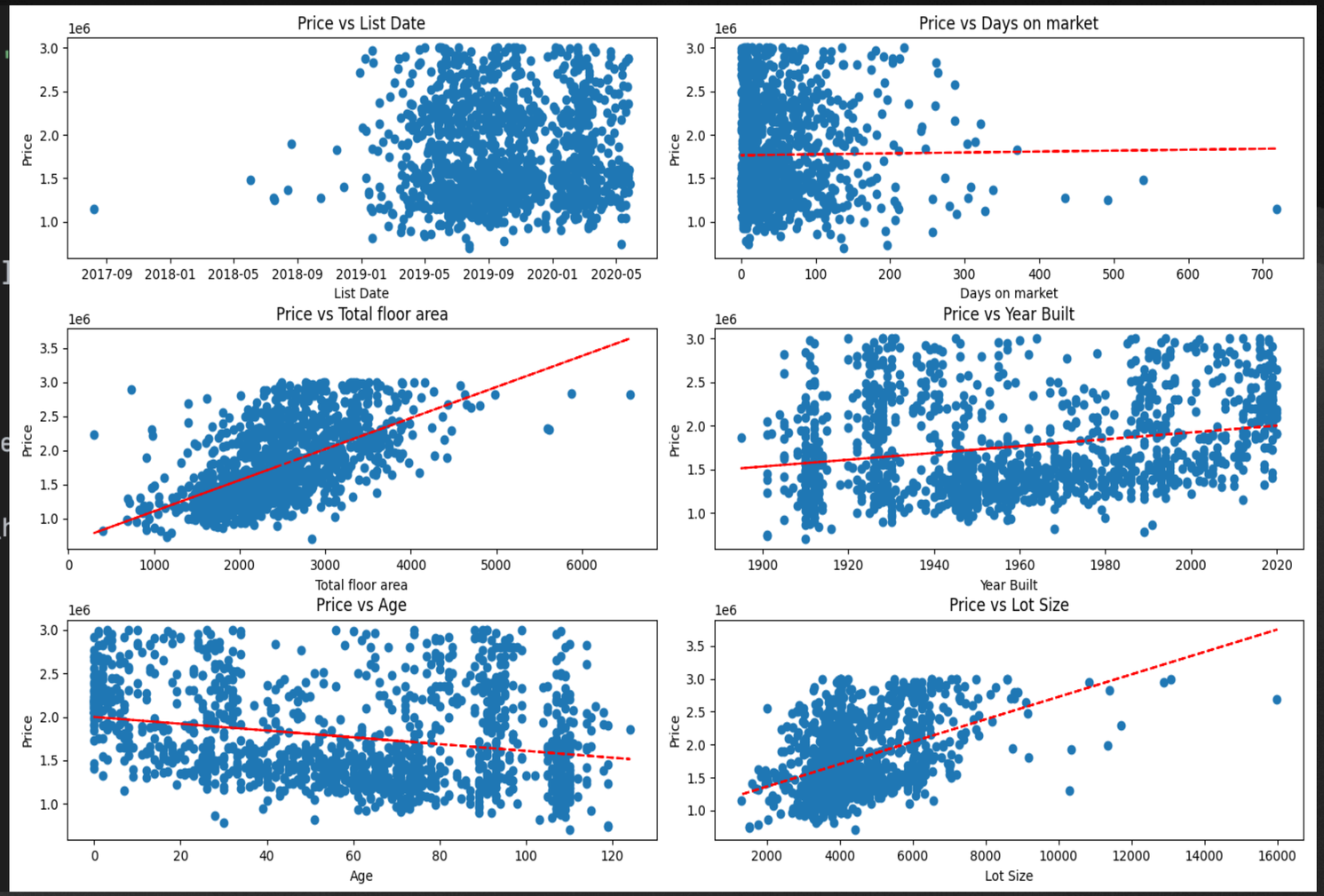
linear regression model—see 1.0linear_regression

```
# List of columns to compare with 'Price'
compare_columns = ['List Date', 'Days on market', '

# Plotting each comparison in a subplot
for i, column in enumerate(compare_columns, 1):
    plt.subplot( *args: 3, 2, i)
    plt.scatter(df_house[column], df_house['Price']
    plt.xlabel(column)
    plt.ylabel('Price')
    if column != 'List Date':
        fit = np.polyfit(df_house[column], df_house
        fit_x = np.poly1d(fit)
        plt.plot( *args: df_house[column], fit_x(df_h
    plt.title(f'Price vs {column}')

# Adjust layout
plt.tight_layout()

# Display plots
plt.show()
plt.close()
```



Version 1.0

linear regression model—see 1.0linear_regression

- From the above figure we can draw the following conclusions:
- 1. There is no obvious linear relationship between price and list days/days on market and can be eliminated.
- 2. Year build and age are essentially the same factor. Here we only select age.
- 3. There is an obvious linear relationship between total floor size, lot size, age and price, and relevant data can be analyzed.
- After determining the elements to be considered, we train and evaluate the model according to the 3:7 principle:
-

```
# building models
df_x = df_house[['Total floor area', 'Age', 'Lot Size']]
df_y = df_house['Price']
x_train, x_test, y_train, y_test = model_selection.train_test_split( *arrays: df_x, df_y, test_size=0.3)

# model training
lr = linear_model.LinearRegression()
lr.fit(x_train, y_train)
```

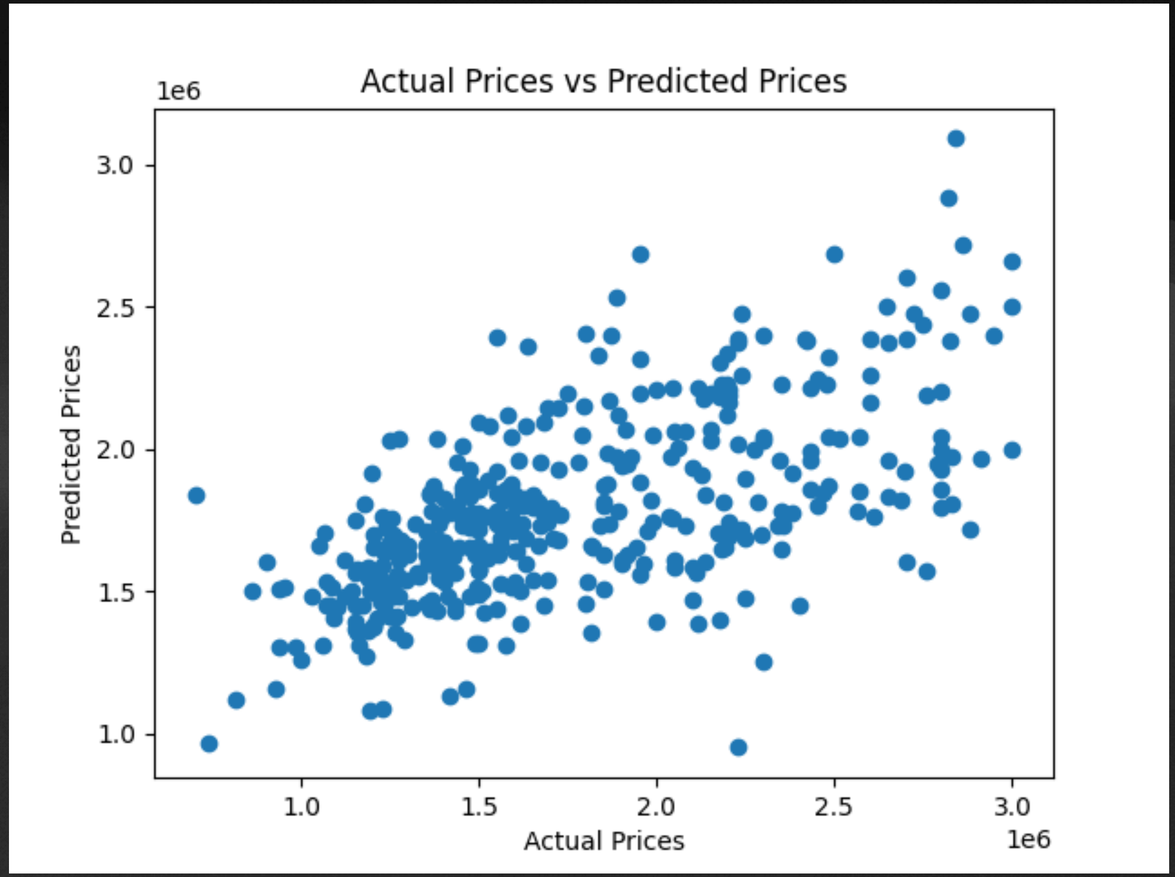
Version 1.0

linear regression model—see 1.0linear_regression

```
# Model evaluation
print("Coefficients:", lr.coef_)
print("Intercept:", lr.intercept_)
print("R^2 Score:", metrics.r2_score(y_test, y_pred))
print("Mean Squared Error:", metrics.mean_squared_error(y_test, y_pred))
print("Mean Absolute Error:", metrics.mean_absolute_error(y_test, y_pred))

# Visualization of actual vs predicted values
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.title('Actual Prices vs Predicted Prices')
plt.show()
```

Here’s the result I got:👉



Version 1.0

linear regression model—see 1.0linear_regression

```
Coefficients: [  373.0835384  -1390.49348318    86.77351381]
Intercept: 562810.9197822509
R^2 Score: 0.3700022087932018
Mean Squared Error: 180556375591.1476
Mean Absolute Error: 345442.73371804494
```

The error using the linear regression model is too large. In order to reduce the error, I decided to use more complex models such as Random Forest which might capture nonlinear relationships better.👇

- 1. Coefficients: The coefficients represent the relationship between each independent variable and the dependent variable (Price).
- - For every one unit increase in 'Total floor area', the price increases by approximately 373.08 units.
- - For every one unit increase in 'Age', the price decreases by approximately 1390.49 units.
- - For every one unit increase in 'Lot Size', the price increases by approximately 86.77 units.
- 2. Intercept: The intercept is the expected mean value of Price when all the predictors (features) are set to zero.
- 3. R^2 Score: The R^2 score indicates that about 37% of the variability in the housing prices is explained by the model. While this is not particularly high, it does suggest that the model has captured some relationship between the features and the housing prices.
- 4. Mean Squared Error (MSE): The MSE of 180556375591.1476 is quite high, indicating that the model's predictions are, on average, a significant distance away from the actual values. This suggests room for improvement in model accuracy.
- 5. Mean Absolute Error (MAE): The MAE of 345442.73371804494 tells us that, on average, the absolute error of the predictions is around 345442.73 units.

Version 2.0

Random forest model - see 2.0random_forest

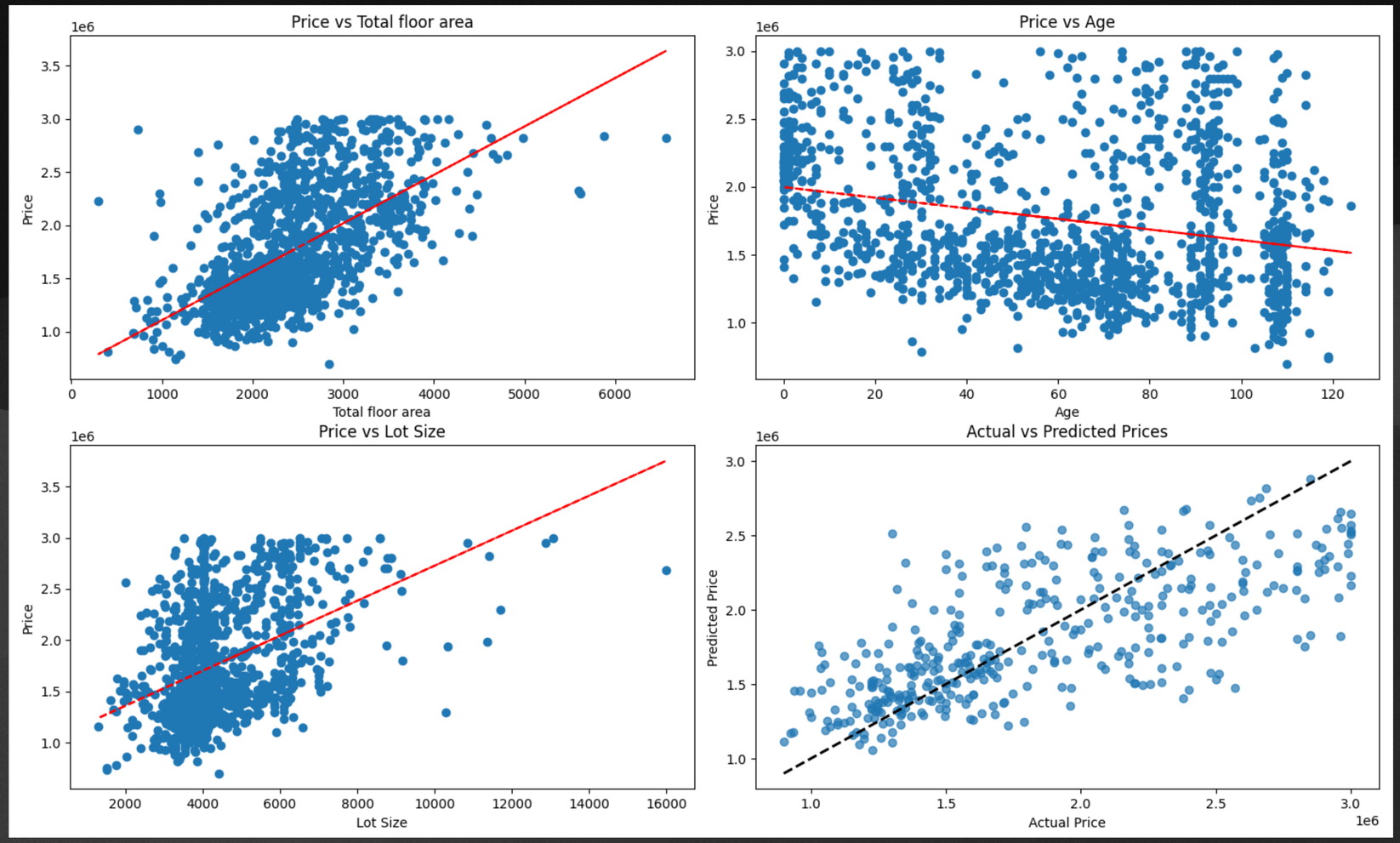
```
# Creating and training the Random Forest model
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
```

- n_estimators=100 specifies the number of trees in the forest. In this case, 100 decision trees will be built and used in the Random Forest.
- random_state=42 sets the seed for the random number generator. This is used for reproducibility of results. Using the same random state ensures that the model behaves in the same way each time it is trained with the same data.

- **The Random Forest model works by creating multiple decision trees (100 in this case), each capable of producing its own predictions.**
- **When making predictions, each tree in the forest votes, and the average of all the tree predictions is the final output of the Random Forest model.**
- **This ensemble approach tends to yield more robust and accurate predictions compared to using a single decision tree.**

Version 2.0

Random forest model - see 2.0random_forest




```
Random Forest Model Evaluation:
Mean Squared Error: 156629942127.8226
Mean Absolute Error: 309107.2527645232
R^2 Score: 0.48347474676381297
```

- The R^2 score about 48.35%, suggests that my model has moderate predictive power.

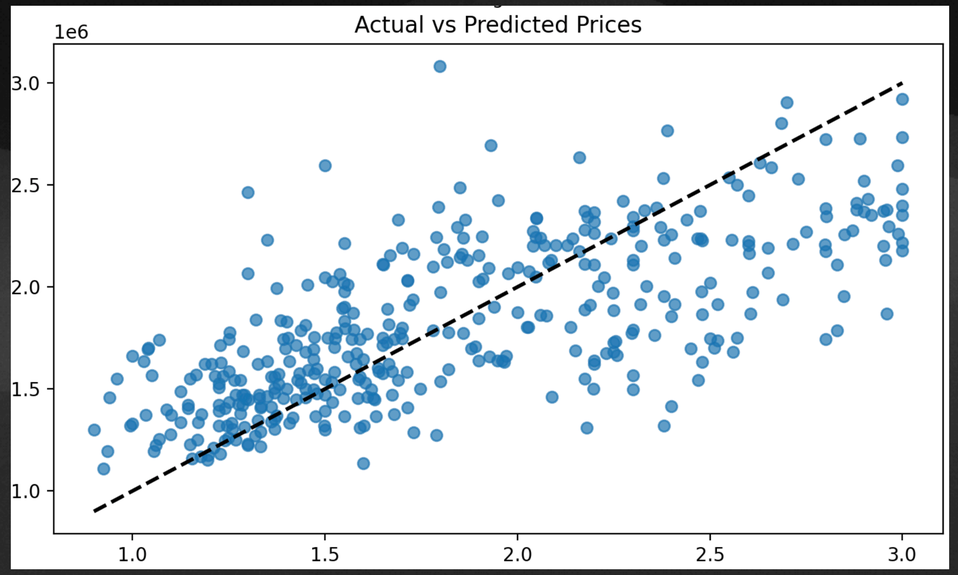
- **The accuracy is much improved than the linear regression model, but it still does not meet my expectations. Let's try another model. ⬇️**

Version 3.0

Gradient boosting model - see 3.0gradient_ boosting

```
# Creating and training the Gradient Boosting model
gb_model = GradientBoostingRegressor(n_estimators=100, random_state=42)
gb_model.fit(X_train, y_train)
```

- This code creates a Gradient Boosting Regressor with 100 sequential trees and then trains this model on the provided training data (`X_train` and `y_train`).
- During training, the model learns how changes in the predictors (features in `X_train`) are associated with changes in the target variable (`y_train`).
- Gradient Boosting is particularly known for its effectiveness in handling various types of data and its ability to improve prediction accuracy by reducing both bias and variance compared to simpler models.



Here’s the result I got:☝️

Version 3.0

Gradient boosting model - see 3.0gradient_ boosting

```
Gradient Boosting Model Evaluation:
Mean Squared Error: 154841541162.06387
Mean Absolute Error: 305271.11812346213
R^2 Score: 0.4893724330502096
```

- This is the highest initial accuracy model I have tried so far. So I decided to take Hyperparameter tuning to optimize the performance of this model.
- Here’s some approach hyperparameter tuning for the Gradient Boosting model:
- 1.Key Hyperparameters to Tune👉
- 2.Grid Search with Cross-Validation
- 3.Random Search Cross-Validation
- Since it is difficult to determine whether the best results are obtained with manual adjustment, I tried 2 and 3, with the goal of achieving a prediction accuracy of more than 50%.

- Key Hyperparameters to Tune:
  - 1.n_estimators: Number of boosting stages to be run (i.e., the number of trees in the forest).
  - 2.learning_rate: Rate at which the model adapts over each boost iteration.
  - 3.max_depth: Maximum depth of the individual regression estimators.
  - 4.min_samples_split: Minimum number of samples required to split an internal node.
  - 5.min_samples_leaf: Minimum number of samp

Version 3.0.1

Gradient boosting model_Grid Search with Cross-Validation

- Use Grid Search to exhaustively search over a specified parameter grid.
- Combine this with Cross-Validation to evaluate the performance of each parameter combination.
- Here's how I did it:

```
# Grid Search Cross-Validation
param_grid = {
    "n_estimators": [100, 200, 300],
    "max_depth": [3, 4, 5, None],
    "min_samples_split": [2, 4, 6],
    "min_samples_leaf": [1, 3, 5],
    "learning_rate": [0.01, 0.1, 0.2]
}
```

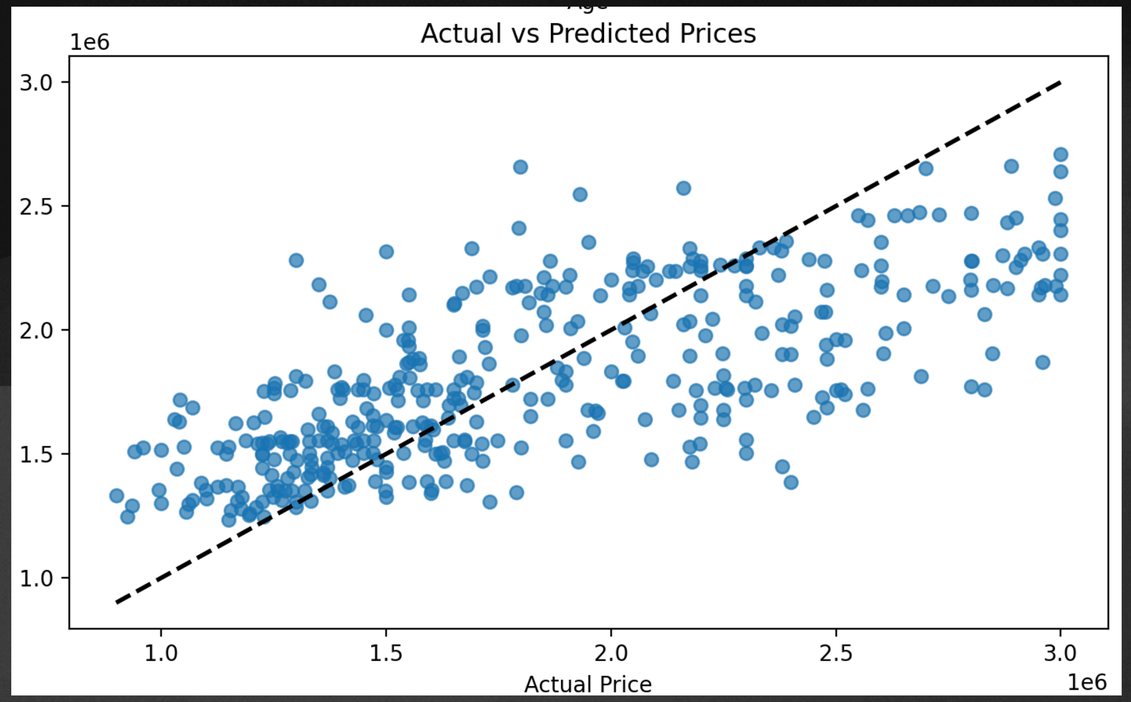
👈n_estimators: This means the grid search will try models with 100, 200, and 300 trees.
👈max_depth: Depths of `3, 4, 5`, and `None` (unlimited) will be tested.
👈min_samples_split: Trying with 2, 4, and 6 samples.
👈min_samples_leaf: The values are 1, 3, and 5.
👈learning_rate: he values `[0.01, 0.1, 0.2]` are to be tried.

```
grid_search = GridSearchCV(GradientBoostingRegressor(random_state=42), param_grid=param_grid,
                           cv=5, verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)
```

-random_state=42: The base model to which the hyperparameters will be applied.
-param_grid=param_grid: The grid of hyperparameters to be tested.
-cv=5: the data is split into 5 parts, each part being used as a test set once.
-verbose=1: The verbosity level.
-n_jobs=-1: This parameter tells the algorithm to use all available CPU cores for parallel processing.

Version 3.0.1

Gradient boosting model_Grid Search with Cross-Validation




```
Fitting 5 folds for each of 324 candidates, totalling 1620 fits
Optimized Gradient Boosting Model Evaluation:
Mean Squared Error: 153791617093.2207
Mean Absolute Error: 312540.71961438336
R^2 Score: 0.49283481251719186
```

- Compared with not using any search method, the accuracy rate has improved, but it still does not reach the 50% expectation.
- Let’s try with Random search instead 👇

Version 3.0.2

Gradient boosting model_Grid Search with Cross-Validation

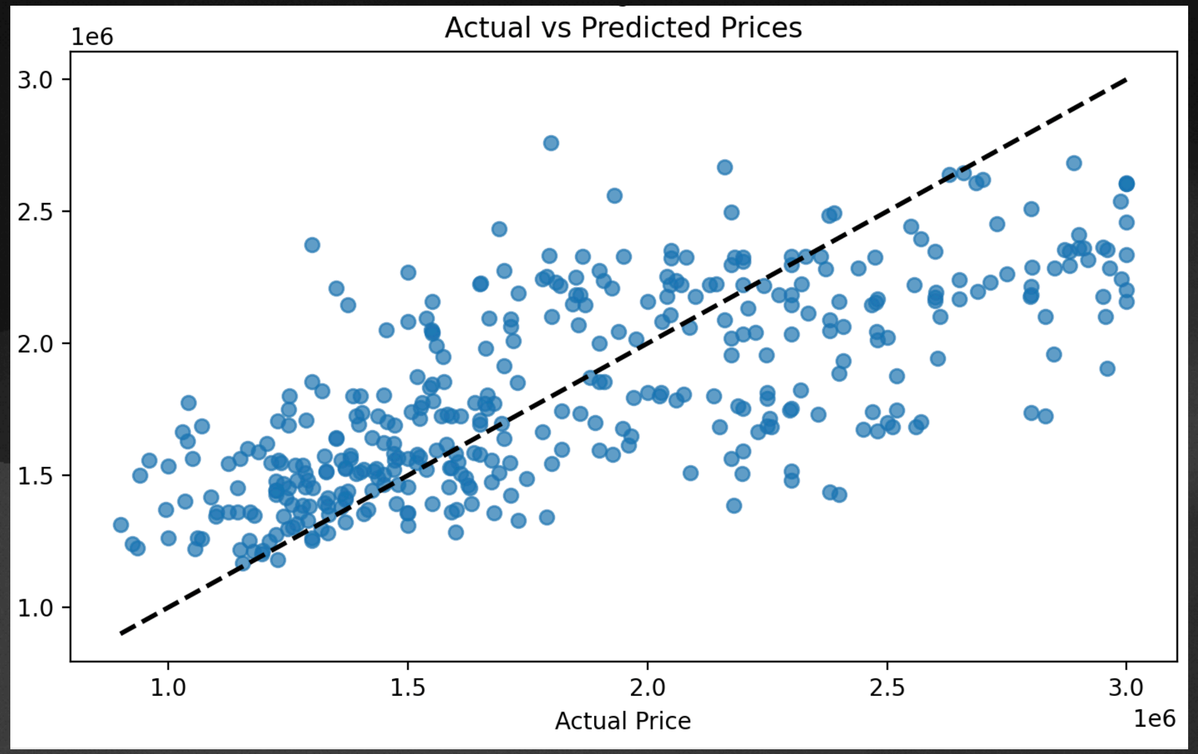
Random Search selects random combinations of parameter values.
This can be more efficient than Grid Search, especially when dealing with a large hyperparameter space

```
# Random Search Cross-Validation
param_dist = {
    "n_estimators": sp_randint(100, 500),
    "max_depth": sp_randint(3, 10),
    "min_samples_split": sp_randint(2, 11),
    "min_samples_leaf": sp_randint(1, 11),
    "learning_rate": [0.01, 0.05, 0.1, 0.2]
}
random_search = RandomizedSearchCV(GradientBoostingRegressor(random_state=42), param_distributions=param_dist,
                                   n_iter=10, cv=5, verbose=1, random_state=42, n_jobs=-1)
random_search.fit(X_train, y_train)
```

- Defining the Parameter Distributions (`param_dist`):
- 👈`param_dist` is a dictionary where keys are the names of hyperparameters, and values are distributions or lists of settings to sample from.
- 👈`n_estimators`, `max_depth`, `min_samples_split`, and `min_samples_leaf` are defined using `sp_randint`. `sp_randint` generates a distribution for random integers in the given range. For example, `sp_randint(100, 500)` will generate random integers between 100 and 500.
- 👈`learning_rate` is a list of possible values. Avalue will be randomly selected from this list.
- ***The use of distributions for most parameters allows Random Search to explore a wider and more varied range of values than Grid Search.***

Version 3.0.2

Gradient boosting model_Grid Search with Cross-Validation




```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Optimized Gradient Boosting Model Evaluation:
Mean Squared Error: 151113611007.16895
Mean Absolute Error: 307650.17891712394
R^2 Score: 0.5016661876232178
```

- By improving our model, we finally reached a prediction rate of 50%.🎉

Postscript

More ways to improve prediction rates

- So far the R^2 score shows that the model has captured some relationship between the features and the house prices, but there's still a significant portion of the variance unexplained.
- The high values of MSE and MAE suggest that the model's predictions can be quite far off for some of the houses. This might be due to the complexity of the housing market, where prices can be influenced by many factors not included in the model.

- 1.Advanced Models: Experiment with more advanced models or ensemble methods, such as Support Vector Regression (SVR) and Stacking Models as they can capture more complex patterns in the data.
- 2.Data Quality: Ensure the data is clean and well-preprocessed. Outliers or incorrect data can significantly affect model performance.
- 3.Model Interpretability: Analyze feature importances to understand what features are most influential in predicting house prices. This can provide insights into the dynamics of the housing market in Vancouver.
- 4.Feature Engineering: Investigate if more relevant features can be added to the model, such as neighborhood quality, proximity to amenities, or economic indicators.

5001 was very challenging for me at first, but I gradually got used to the intensity. I am so happy finally completed my final project, proud of what I've done so far.

Though not perfect, this final project made me realize the endless possibilities of applying what I learned to real life and how efficient it is. It's exciting to think that I might be able to use these models in future work reports.