UPPA Anglet - Project evaluation machine learning - House

For this project we chose to work on the House dataset (house.csv).

This dataset contains house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015. It's a great dataset for evaluating simple regression models for predicting the price of a house depending on its characteristics. **Output variable :** price (continuous)

Steps

1. Clean the dataset
2. Load the dataset
3. Use Machine learning to determine the output (price)

Prequisites

Importing python librairies

We are going to import the Python librairies we are going to use to build our machine learning models.

- pandas
- sklearn
- matplotlib

```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error,
r2_score
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
import seaborn as sns

#For our neural network, we are using keras from tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```

Loading the dataset

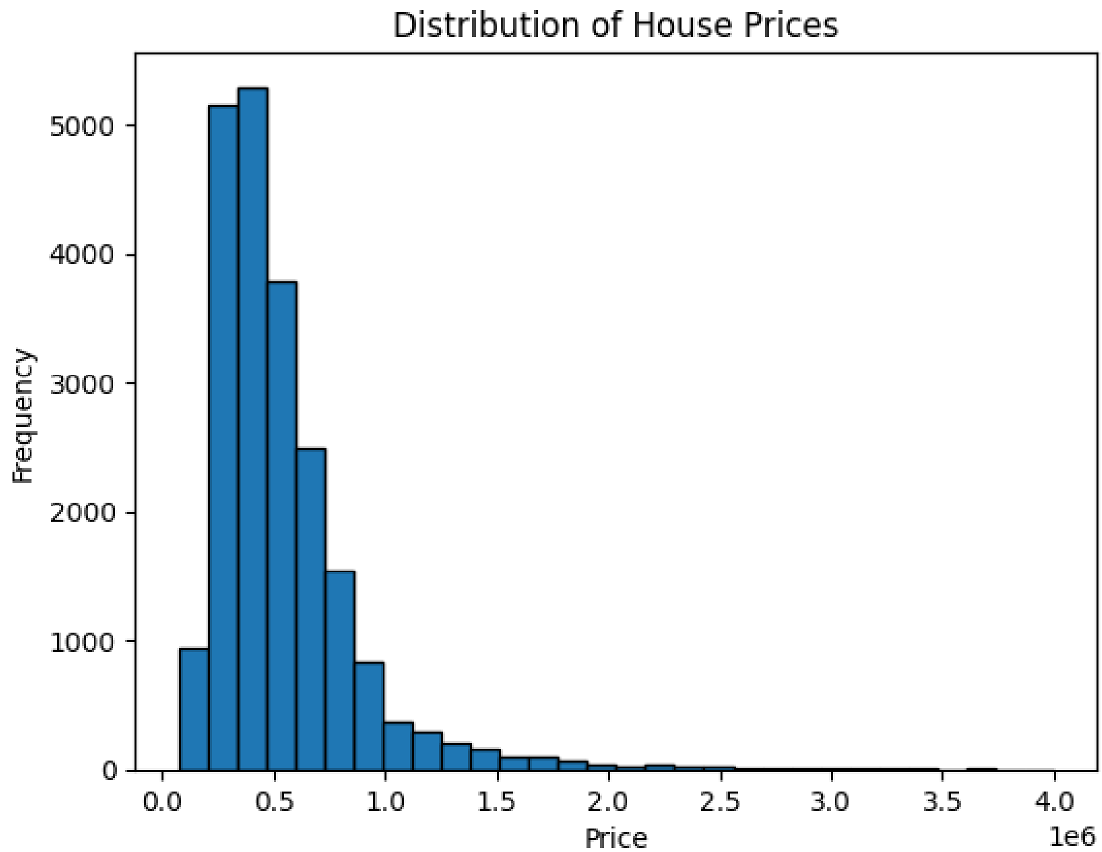
Then we are going to load our data into the code lab.

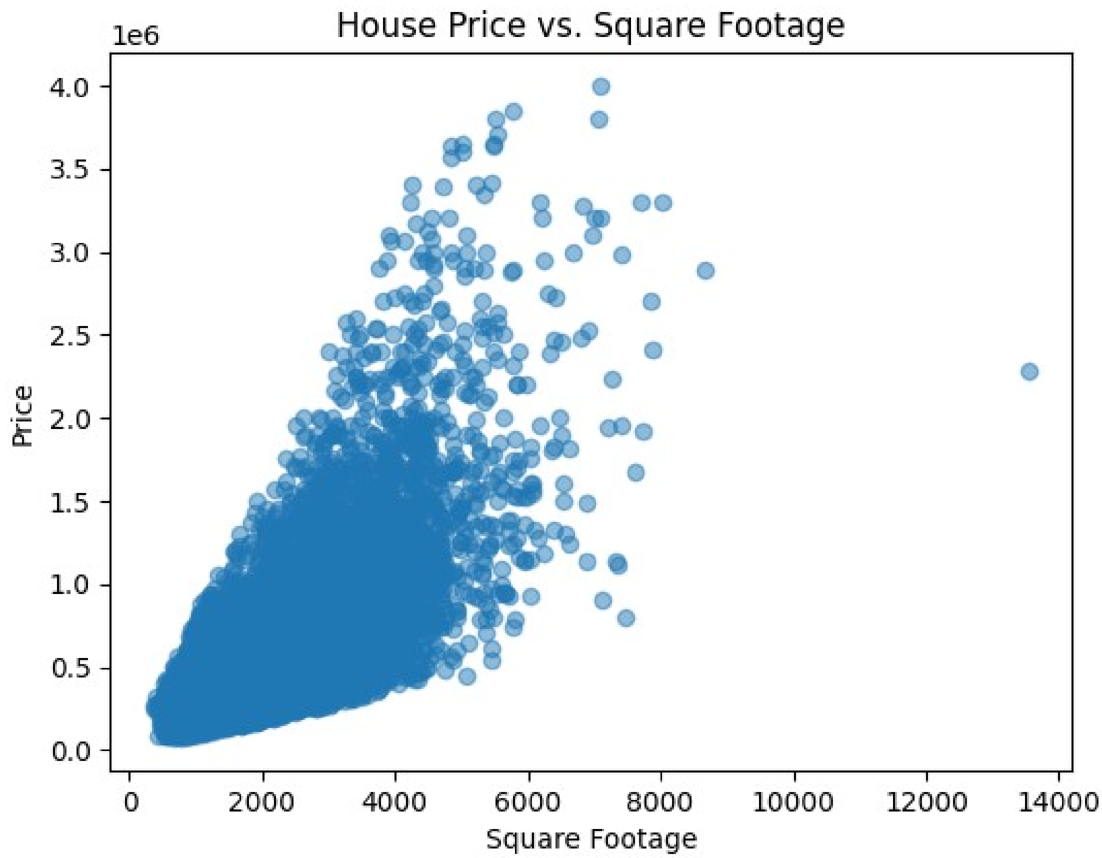
```
# Load the dataset
file_path = './house.csv'
data = pd.read_csv(file_path)
```

Ploting the raw data

The second step in the process for us is to visualize the raw data before working on it in order to spot outliers and potential problems with the dataset.

```
# Data visualization
# This histogram shows the distribution of house prices
plt.hist(data['price'], bins=30, edgecolor='black')
plt.title('Distribution of House Prices')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()

# Scatter plot to visualize the relationship between sqft_living and
price
plt.scatter(data['sqft_living'], data['price'], alpha=0.5)
plt.title('House Price vs. Square Footage')
plt.xlabel('Square Footage')
plt.ylabel('Price')
plt.show()
```


Distribution of House Prices
Frequency
0
1000
2000
3000
4000
5000
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Price
1e6



Analysis conclusion

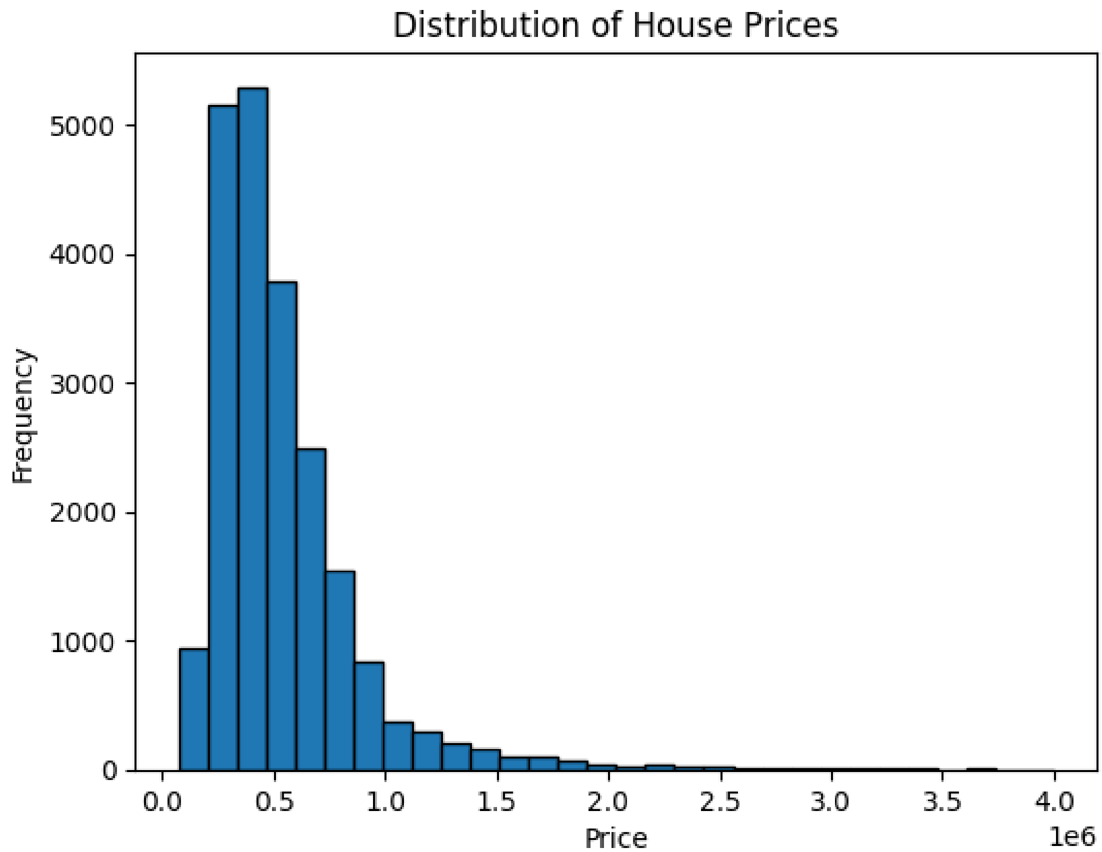
For this first analysis of our raw dataset, we can spot multiple outliers that we could easely fix by cleaning our dataset using simple filters. On the first graph, we can see the repartition is not good for our study and we could easely remove houses with a price > 4 million without causing too much trouble on our dataset for our future machine. The second graph shows us some outliers on the price depending on the square foot of living. By fitlering houses with a price of 4 million as said above, we can clean this outliers out of our dataset.

Cleaning the dataset

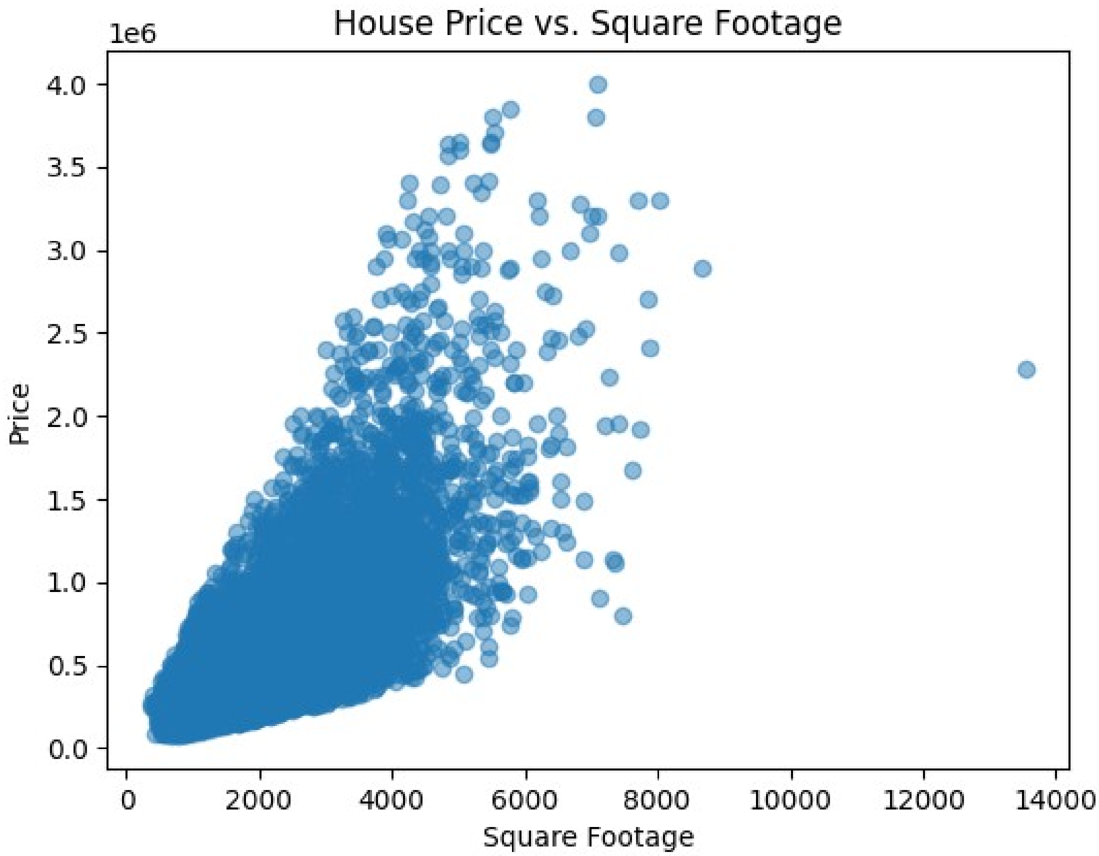
We will clean the dataset to remove outliers. We have studied that by removing all housing prices over 4 000 000 to clean our dataset. We will plot our new dataset following the cleaned dataset.

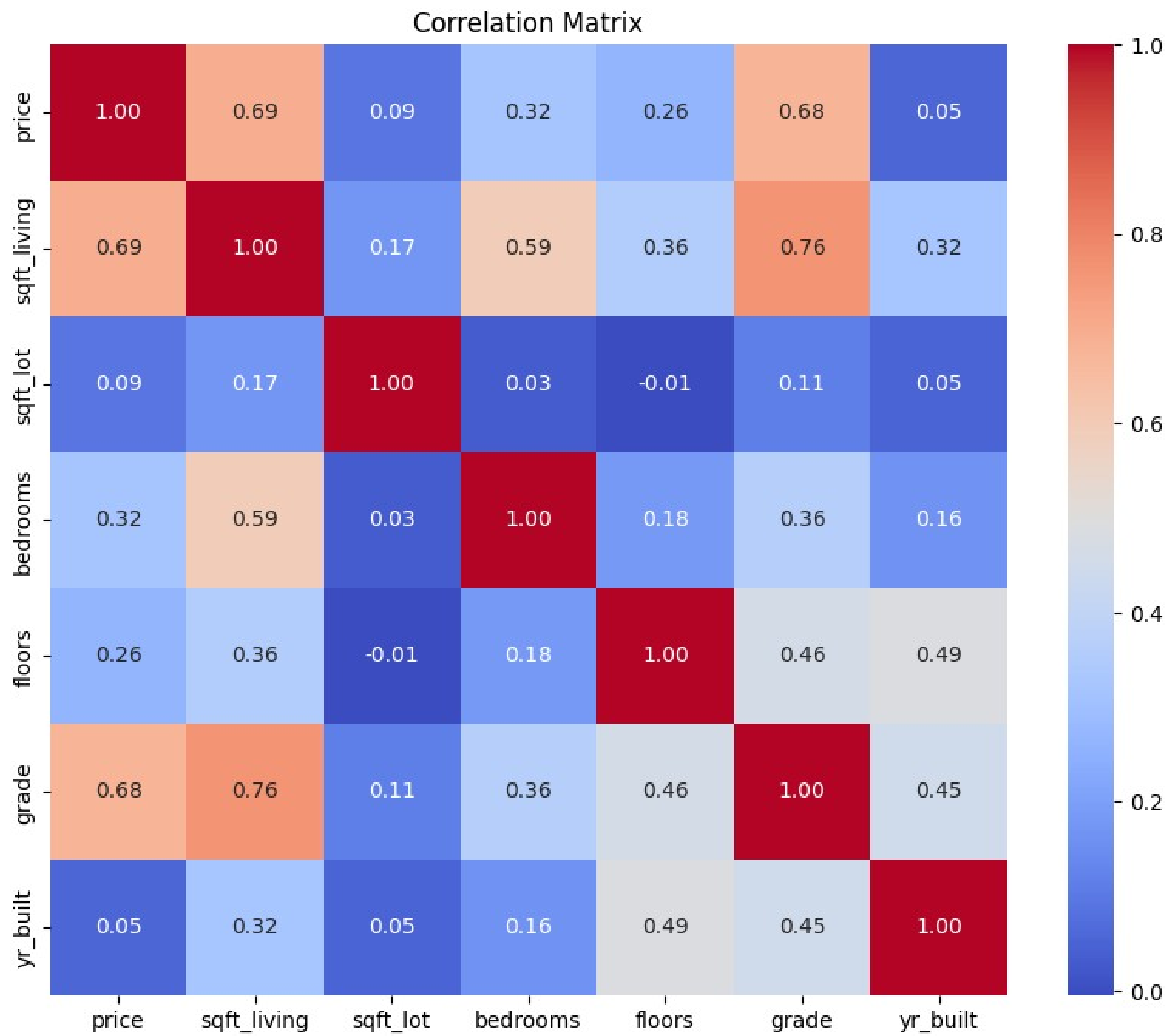
```
# Filtering out houses with prices above $4 million
data = data[data['price'] <= 4e6]
data = data[data['bedrooms'] > 0]
data = data[data['bedrooms'] < 33]

# Data visualization
# This histogram shows the distribution of house prices
plt.hist(data['price'], bins=30, edgecolor='black')
```

```
plt.title('Distribution of House Prices')
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()

# Scatter plot to visualize the relationship between sqft_living and
price
plt.scatter(data['sqft_living'], data['price'], alpha=0.5)
plt.title('House Price vs. Square Footage')
plt.xlabel('Square Footage')
plt.ylabel('Price')
plt.show()

# Selecting the relevant columns
selected_columns = data[['price',
'sqft_living','sqft_lot','bedrooms','floors','grade','yr_built']]

# Calculating the correlation matrix
correlation_matrix = selected_columns.corr()

# Plotting the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
fmt=".2f")
plt.title('Correlation Matrix')
plt.show()
```



Distribution of House Prices
Frequency
5000
4000
3000
2000
1000
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Price
1e6


House Price vs. Square Footage
1e6
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Price
0
2000
4000
6000
8000
10000
12000
14000
Square Footage



Cleaning conclusion

By only removing houses with a price > 4 million, we already have cleaned a good portion of our dataset. Most of the outliers are now gone and the repartition (first graph) is much better than the original one. We have graphed out a resolution matrix to see which variable of our dataset would impact our price. We found two varaibles we can identify as independent for our machine, the square foot living (**sqft_living**) and the grade for each house (**grade**). We are going ot use both these varaibles to feed our machine.

First model : Linear Regression

Linear regression is a fundamental statistical method in machine learning for predicting a quantitative variable. It establishes a linear relationship between a dependent variable (target) and one or more independent variables (predictors).

The core idea is to find a "best fit line" that minimizes the difference between predicted and actual values. It involves calculating coefficients that represent the slope and intercept of this line.

We use linear regression because it's simple, efficient, and interpretable. It's a straightforward technique for predictive analysis and serves as a foundational method for understanding relationships between variables. Due to its simplicity and interpretability, linear regression is often the first approach in modeling linear relationships in data.

Step 1 : Splitting the dataset and implementing the Linear Regression model

Step 1.1 : Selecting our variables

We select our varaibles. For the dependent variable, we've chose the price (because it is the value we are trying to predict) and for our independent variable we've chose the square foot of living).

```
# Selecting the independent and dependent variables
# sqft_living and grade are chosen as the independent variable and
price as the dependent variable
X = data[['sqft_living','grade']]  # Independent variable
y = data['price']  # Dependent variable
```

Step 1.2

We split our variables sets into a training and testing set.

```
# Splitting the dataset into training and testing sets
# We use 80% of the data for training and 20% for testing
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Step 1.3

We implement the Linear Regression

```
# Creating the linear regression model
model = LinearRegression()

# Training the model using the training set
model.fit(X_train, y_train)

LinearRegression()
```

Step 1.4

We predict our dataset using the new linear regression.

```
# Making predictions using the testing set
y_pred = model.predict(X_test)

# Evaluating the model
mse = mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"Mean Absolute Error: {mae}")
print(f"R^2 score: {r2}")
```

```
Mean Squared Error: 51612958929.280914
Mean Absolute Error: 160012.4272095894
R^2 score: 0.532255172461569
```

Prediction conclusion

We evaluate our models with three different indicators, the mean squared and absolute error and the R^2 result. Here we have :

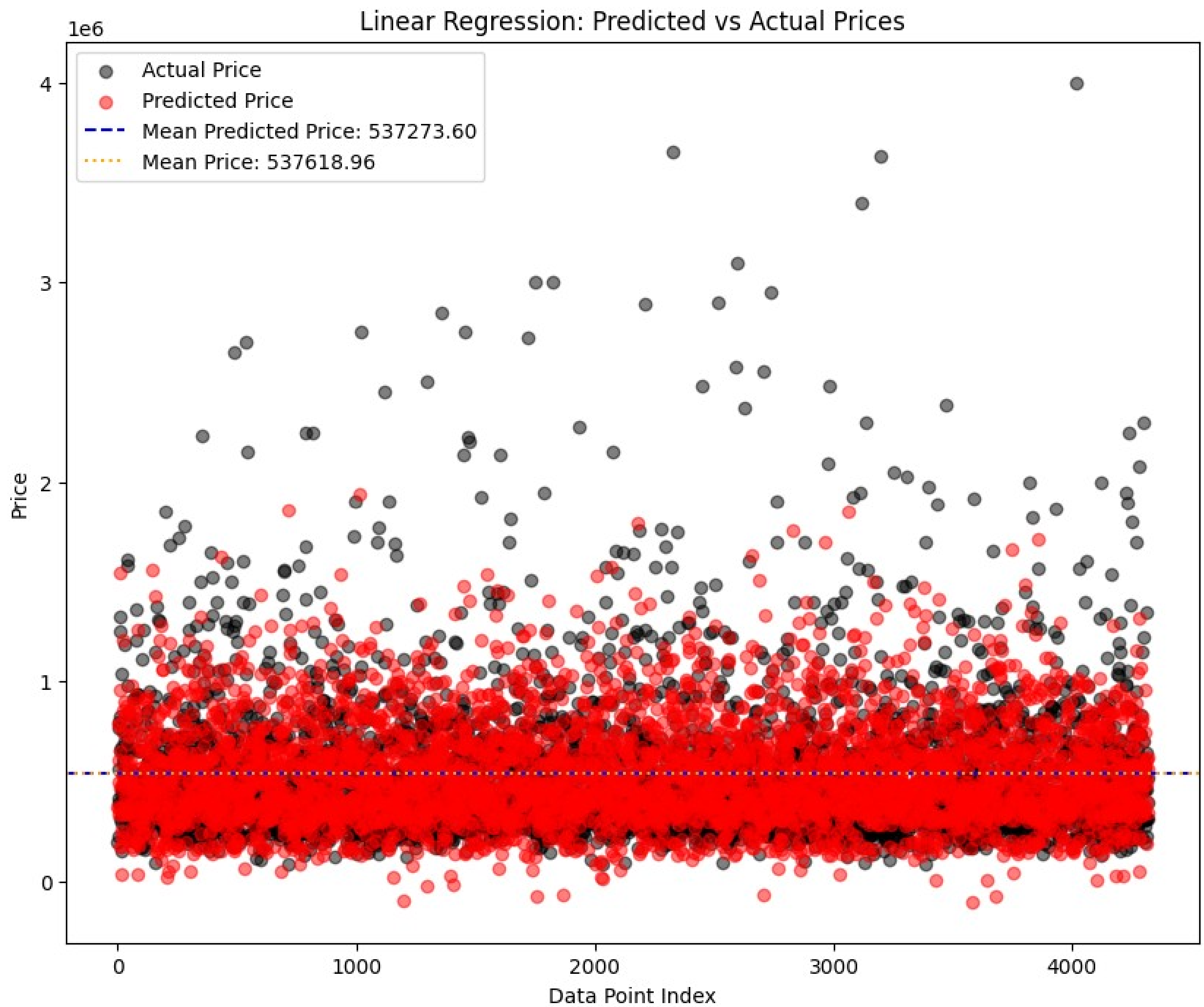
> Mean Squared Error: 51612958929.280914 Mean Absolute Error: 160012.4272095894
> R^2 score: 0.532255172461569

The R^2 score is not bad, indicated a 53% percent of error and for MAE we've got an error marge of 160012.42, which is quite large and must be considered in our final conclusion. However, due to the price range, this is not a bad result (we could have had worse)

Step 2 : Evaluation & result

We will calculate the Mean Squared Error and plot our prediction against the dataset to evaluate our model.

```
mean_predicted_price = y_pred.mean()

# Create the scatter plot for actual and predicted prices
plt.figure(figsize=(10, 8))
plt.scatter(range(len(y_test)), y_test, color='black', alpha=0.5,
label='Actual Price')

# Add a horizontal line for the mean predicted price
plt.axhline(mean_predicted_price, color='blue', linestyle='--',
label=f'Mean Predicted Price: {mean_predicted_price:.2f}')

# Add title and labels
plt.title('Linear Regression: Predicted vs Actual Prices')
plt.xlabel('Data Point Index')
plt.ylabel('Price')
plt.legend()
plt.show()
```



Evaluation conclusion

From the graph above, our model is well representing the actual price for each house depending on their grade and square foot of living. Our mean for the overall prediction is at 537273.60$.

Step 3 : Conclusion

This first model got a R^2 high but not great with only 53%. The model fits our dataset with two correlated variables. We are going to test using the Random Forest Algorithm ---

Second model : Random Forest

Step 1 : Splitting the dataset and implementing the Linear Regression model

Step 1.1 : Selecting our variables

We select our varaibles. For the dependent variable, we've chose the price (because it is the value we are trying to predict) and for our independent variable we've chose the square foot of living).

```
# Selecting the independent and dependent variables
# sqft_living and grade are chosen as the independent variable and
price as the dependent variable
X = data[['sqft_living','grade']]  # Independent variable
y = data['price']  # Dependent variable
```

Step 1.2 : Splitting the data

We split our variables sets into a training and testing set.

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

Step 1.3 : Implementing the random forest algorithm

We implement the Random Forest algorithm and fit it to our dataset from the selected variable.

```
# Scale the feature data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize the Random Forest model
random_forest = RandomForestRegressor(n_estimators=100,
random_state=42)

# Fit the model on the training data
random_forest.fit(X_train_scaled, y_train)

# Make predictions on the test set
predictions_rf = random_forest.predict(X_test_scaled)
```

Step 1.4 : Evaluation

We will calculate the Mean Squared Error and plot our prediction against the dataset to evaluate our model.

```python
# Calculate the metrics
mse_rf = mean_squared_error(y_test, predictions_rf)
mae_rf = mean_absolute_error(y_test, predictions_rf)
r2_rf = r2_score(y_test, predictions_rf)

# Output the performance
print("Random Forest - MSE:", mse_rf)
print("Random Forest - MAE:", mae_rf)
print("Random Forest - R^2:", r2_rf)
```

```
Random Forest - MSE: 58156885271.105965
Random Forest - MAE: 161702.66125672654
Random Forest - R^2: 0.4729505373141223
```

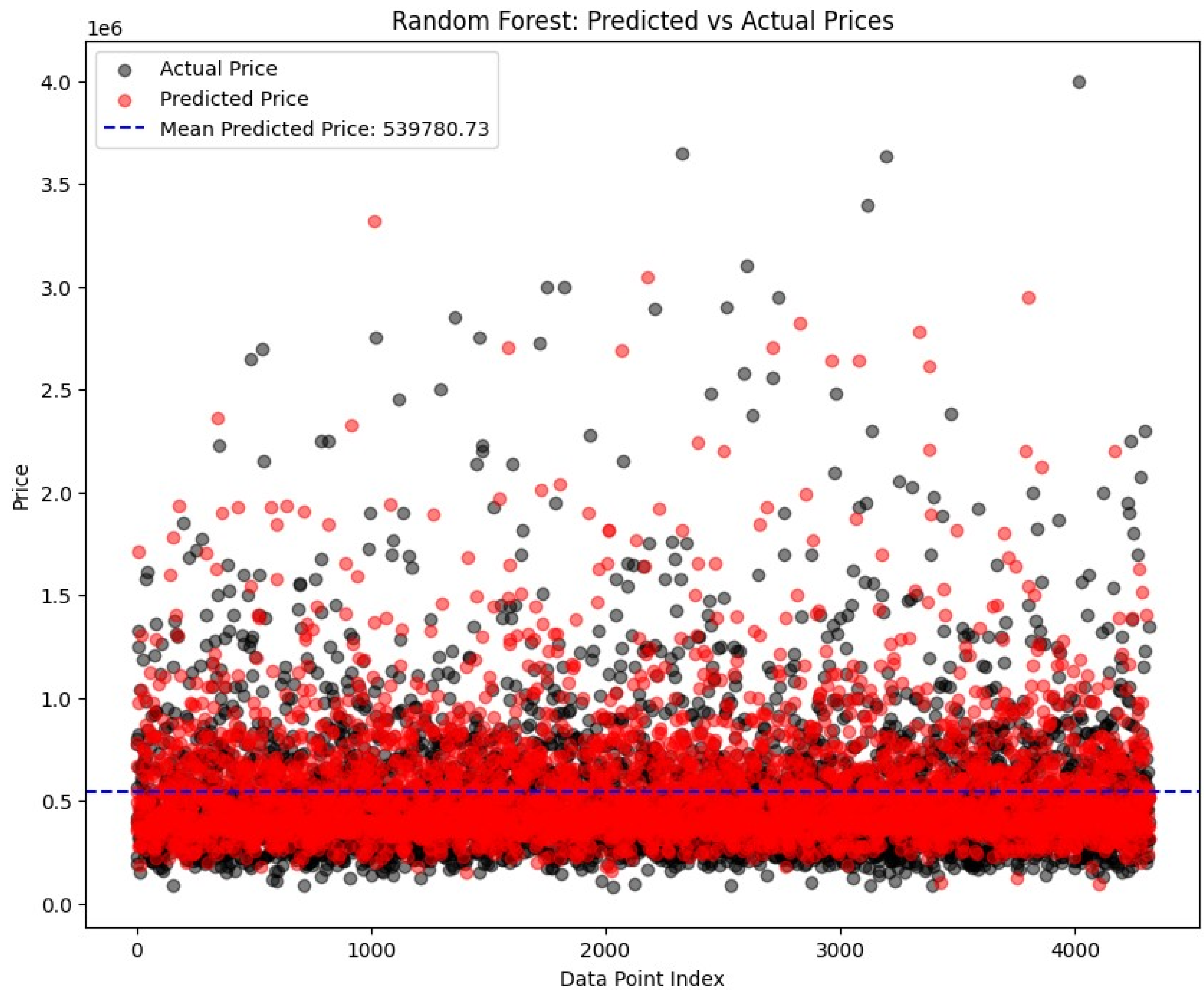
Prediction conclusion

We evaluate our models with three different indicators, the mean squared and absolute error and the R^2 result. Here we have :

> Random Forest - MSE: 58156885271.105965 Random Forest - MAE: 161702.66125672654 Random Forest - R^2: 0.4729505373141223

Our R^2 score for Random Forest is slightly lower than the one we've got for the linear regression. Same for the MAE with an error marge of 161702.66$ on average on the predicted price.

```python
mean_predicted_price = predictions_rf.mean()

# Create the scatter plot for actual and predicted prices
plt.figure(figsize=(10, 8))
plt.scatter(range(len(y_test)), y_test, color='black', alpha=0.5,
label='Actual Price')
plt.scatter(range(len(predictions_rf)), predictions_rf, color='red',
alpha=0.5, label='Predicted Price')

# Add a horizontal line for the mean predicted price
plt.axhline(mean_predicted_price, color='blue', linestyle='--',
label=f'Mean Predicted Price: {mean_predicted_price:.2f}')

# Add title and labels
plt.title('Random Forest: Predicted vs Actual Prices')
plt.xlabel('Data Point Index')
plt.ylabel('Price')
plt.legend()
plt.show()
```



Evaluation conclusion

We can see that our Random Forest, just like our Linear Regression tends to predict the price in a good way. Our predicted mean price is slightly lower at 161702.66$ which is close to the one we've got with the Linear Regression. So it seems both model, using two independant variable, tends to moderatly correlate to the actual dataset and thus predict the data correctly.

Price correlation - House position and price

We are going to plot the position of the houses on the map of Seattle and group housing depending on position and prices. We are going to use a clustering algorithm to achieve this and try to prove their is a correlation between the location of houses and their price. We are logicaly waiting for the price to be higher at the heart of the city and less expensive outside in the countryside.

Price and location using raw data

First, we want to display a heatmap of the housing price depending on the location. Wre are going to use open-stree-map to display this data

```
# Create the heatmap using the density_mapbox function
# Using the "open-street-map" style which does not require an access token
fig = px.density_mapbox(data,
                        lat='lat',
                        lon='long',
                        z='price',  # Data to show in heatmap
                        radius=10,  # Adjust the radius as needed
                        center=dict(lat=data['lat'].mean(), lon=data['long'].mean()),  # Center the map
                        zoom=10,
                        mapbox_style="open-street-map")  # No access token required

fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

As awaited, we do have a correlation between the location of the house and its price. The city heart is way more expensive than the country side.

Clustering the data

We want to cluster our data in order to form different group of prices depending on the location. We are going to group our data into 3 different groups.

1. Expensive
2. Medium
3. Cheap

We will display each group on a map.

```
# Selecting the relevant columns (latitude, longitude, and price)
mdata = data[['lat', 'long', 'price']]

# Performing k-means clustering
k = 3
kmeans = KMeans(n_clusters=k, random_state=0)
clusters = kmeans.fit_predict(mdata[['lat', 'long']])  # Only cluster based on location
data['cluster'] = clusters

# Create the scatter mapbox plot with clusters
fig = px.scatter_mapbox(data,
```

```
                        lat="lat",
                        lon="long",
                        color="cluster",
                        size="price",  # Size points by price

color_discrete_sequence=px.colors.qualitative.Set1,  # Use discrete
color sequence
                        size_max=15,
                        zoom=10,
                        mapbox_style="carto-positron")

# Calculate and draw rectangles for cluster boundaries
for i in range(k):
    cluster_data = data[data['cluster'] == i]
    min_lat, max_lat = cluster_data['lat'].min(),
cluster_data['lat'].max()
    min_long, max_long = cluster_data['long'].min(),
cluster_data['long'].max()

    fig.add_trace(go.Scattermapbox(
        mode = "lines",
        lon = [min_long, max_long, max_long, min_long, min_long],
        lat = [min_lat, min_lat, max_lat, max_lat, min_lat],
        marker = dict(size=1),
        line = dict(width=2, color='black'),
        showlegend=False,
    ))

fig.update_layout(margin={"r":0,"t":0,"l":0,"b":0})
fig.show()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/
_kmeans.py:870: FutureWarning:

The default value of `n_init` will change from 10 to 'auto' in 1.4.
Set the value of `n_init` explicitly to suppress the warning
```

Clustering conclusion

As we were hoping to see, there is a correlation between the different part of the city and the actual price of each house. And, we have been able to conclusively use a clustering algorithm (near K-mean clustering) to graph out the three different part of the city which cost less or the most. Of course, this clustering could be divised in much more groups to output a more granural result over all the city and its region (plotting which stree of the city is considered wealthy or poor.

Neural network to the data

```
# Selecting the specific features and target
features = data[['sqft_living', 'lat', 'long','grade','bedrooms']]  #
Features
target = data['price']  # Target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size=0.2, random_state=0)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Neural network architecture
model = Sequential([
    Dense(64, activation='relu',
input_shape=(X_train_scaled.shape[1],)),
    Dense(64, activation='relu'),
    Dense(1)  # Output layer: 1 neuron for regression
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='mse',
metrics=['mae'])

# Train the model
history = model.fit(X_train_scaled, y_train, validation_split=0.2,
epochs=100, batch_size=32)

# Evaluate the model on the test set
test_loss, test_mae = model.evaluate(X_test_scaled, y_test)

# Predictions
predictions = model.predict(X_test_scaled)

# Output the performance
print(f"Test Loss: {test_loss}, Test MAE: {test_mae}")

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['mae'], label='MAE (training data)')
plt.plot(history.history['val_mae'], label='MAE (validation data)')
plt.title('MAE for House Prices')
plt.ylabel('MAE value')
plt.xlabel('No. epoch')
plt.legend(loc="upper left")
```

```python
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Loss (training data)')
plt.plot(history.history['val_loss'], label='Loss (validation data)')
plt.title('Loss for House Prices')
plt.ylabel('Loss value')
plt.xlabel('No. epoch')
plt.legend(loc="upper right")
plt.show()
```

```
Epoch 1/100
432/432 [==============================] - 2s 3ms/step - loss:
399420850176.0000 - mae: 532708.2500 - val_loss: 438484697088.0000 -
val_mae: 547117.1250
Epoch 2/100
432/432 [==============================] - 1s 3ms/step - loss:
391420674048.0000 - mae: 526635.8750 - val_loss: 421603409920.0000 -
val_mae: 534960.6250
Epoch 3/100
432/432 [==============================] - 1s 2ms/step - loss:
364744769536.0000 - mae: 506215.2500 - val_loss: 381017128960.0000 -
val_mae: 504827.4062
Epoch 4/100
432/432 [==============================] - 1s 2ms/step - loss:
316463218688.0000 - mae: 466809.9375 - val_loss: 319197708288.0000 -
val_mae: 454899.0312
Epoch 5/100
432/432 [==============================] - 1s 2ms/step - loss:
252979838976.0000 - mae: 408756.0938 - val_loss: 247110172672.0000 -
val_mae: 388127.3750
Epoch 6/100
432/432 [==============================] - 1s 2ms/step - loss:
186726170624.0000 - mae: 337838.4688 - val_loss: 178287755264.0000 -
val_mae: 312649.2500
Epoch 7/100
432/432 [==============================] - 1s 2ms/step - loss:
130625732608.0000 - mae: 267333.3125 - val_loss: 125890150400.0000 -
val_mae: 247252.6094
Epoch 8/100
432/432 [==============================] - 1s 3ms/step - loss:
93887004672.0000 - mae: 217935.8438 - val_loss: 95333171200.0000 -
val_mae: 209806.6406
Epoch 9/100
432/432 [==============================] - 2s 4ms/step - loss:
75342962688.0000 - mae: 195365.3438 - val_loss: 80703602688.0000 -
val_mae: 194939.2031
Epoch 10/100
432/432 [==============================] - 2s 4ms/step - loss:
67160432640.0000 - mae: 185574.3438 - val_loss: 73647661056.0000 -
val_mae: 187262.3438
Epoch 11/100
```

```
432/432 [==============================] - 1s 2ms/step - loss:
62717571072.0000 - mae: 179032.8906 - val_loss: 69110546432.0000 -
val_mae: 180667.7812
Epoch 12/100
432/432 [==============================] - 1s 3ms/step - loss:
59353972736.0000 - mae: 173100.4531 - val_loss: 65611137024.0000 -
val_mae: 174293.7500
Epoch 13/100
432/432 [==============================] - 1s 2ms/step - loss:
56388800512.0000 - mae: 166878.5000 - val_loss: 62554419200.0000 -
val_mae: 168171.1406
Epoch 14/100
432/432 [==============================] - 1s 3ms/step - loss:
53686288384.0000 - mae: 160670.0312 - val_loss: 59801436160.0000 -
val_mae: 162456.2500
Epoch 15/100
432/432 [==============================] - 1s 3ms/step - loss:
51258736640.0000 - mae: 155092.6094 - val_loss: 57425272832.0000 -
val_mae: 157231.0156
Epoch 16/100
432/432 [==============================] - 1s 2ms/step - loss:
49146814464.0000 - mae: 149881.5469 - val_loss: 55423111168.0000 -
val_mae: 152545.9375
Epoch 17/100
432/432 [==============================] - 1s 2ms/step - loss:
47365349376.0000 - mae: 145307.7969 - val_loss: 53817294848.0000 -
val_mae: 148500.5469
Epoch 18/100
432/432 [==============================] - 1s 2ms/step - loss:
45906620416.0000 - mae: 141752.1094 - val_loss: 52560900096.0000 -
val_mae: 145108.9688
Epoch 19/100
432/432 [==============================] - 1s 2ms/step - loss:
44737568768.0000 - mae: 138479.7656 - val_loss: 51477184512.0000 -
val_mae: 142508.0469
Epoch 20/100
432/432 [==============================] - 1s 3ms/step - loss:
43826028544.0000 - mae: 136205.8438 - val_loss: 50792435712.0000 -
val_mae: 140256.1406
Epoch 21/100
432/432 [==============================] - 1s 3ms/step - loss:
43118526464.0000 - mae: 134180.4688 - val_loss: 50154119168.0000 -
val_mae: 138738.4375
Epoch 22/100
432/432 [==============================] - 1s 3ms/step - loss:
42573606912.0000 - mae: 132752.9688 - val_loss: 49766477824.0000 -
val_mae: 137279.0781
Epoch 23/100
432/432 [==============================] - 1s 2ms/step - loss:
```

```
42147700736.0000 - mae: 131509.2656 - val_loss: 49405038592.0000 -
val_mae: 136226.4219
Epoch 24/100
432/432 [==============================] - 1s 3ms/step - loss:
41803276288.0000 - mae: 130457.2188 - val_loss: 49010839552.0000 -
val_mae: 135617.3281
Epoch 25/100
432/432 [==============================] - 1s 2ms/step - loss:
41519976448.0000 - mae: 129684.8203 - val_loss: 48727621632.0000 -
val_mae: 134988.5938
Epoch 26/100
432/432 [==============================] - 1s 2ms/step - loss:
41286139904.0000 - mae: 129533.1797 - val_loss: 48662892544.0000 -
val_mae: 134029.5469
Epoch 27/100
432/432 [==============================] - 1s 2ms/step - loss:
41103327232.0000 - mae: 128560.3125 - val_loss: 48353292288.0000 -
val_mae: 133846.6250
Epoch 28/100
432/432 [==============================] - 1s 3ms/step - loss:
40941662208.0000 - mae: 128197.9453 - val_loss: 48176115712.0000 -
val_mae: 133403.6250
Epoch 29/100
432/432 [==============================] - 1s 3ms/step - loss:
40776429568.0000 - mae: 127980.0156 - val_loss: 48115494912.0000 -
val_mae: 132785.6406
Epoch 30/100
432/432 [==============================] - 1s 2ms/step - loss:
40642977792.0000 - mae: 127569.9609 - val_loss: 48003440640.0000 -
val_mae: 132380.2500
Epoch 31/100
432/432 [==============================] - 1s 2ms/step - loss:
40521961472.0000 - mae: 126904.5234 - val_loss: 47755894784.0000 -
val_mae: 132313.0469
Epoch 32/100
432/432 [==============================] - 1s 3ms/step - loss:
40410169344.0000 - mae: 126877.1406 - val_loss: 47564849152.0000 -
val_mae: 132139.4375
Epoch 33/100
432/432 [==============================] - 1s 3ms/step - loss:
40312442880.0000 - mae: 126663.9219 - val_loss: 47463727104.0000 -
val_mae: 131832.5781
Epoch 34/100
432/432 [==============================] - 1s 3ms/step - loss:
40219750400.0000 - mae: 126508.1484 - val_loss: 47364272128.0000 -
val_mae: 131534.5938
Epoch 35/100
432/432 [==============================] - 1s 3ms/step - loss:
40127078400.0000 - mae: 126297.2109 - val_loss: 47284809728.0000 -
```

```
val_mae: 131254.0938
Epoch 36/100
432/432 [==============================] - 1s 2ms/step - loss:
40051838976.0000 - mae: 126004.3516 - val_loss: 47137042432.0000 -
val_mae: 131164.7031
Epoch 37/100
432/432 [==============================] - 1s 2ms/step - loss:
39977312256.0000 - mae: 125958.3594 - val_loss: 46983262208.0000 -
val_mae: 131069.2578
Epoch 38/100
432/432 [==============================] - 1s 2ms/step - loss:
39891222528.0000 - mae: 125769.8359 - val_loss: 46876082176.0000 -
val_mae: 130915.9922
Epoch 39/100
432/432 [==============================] - 1s 2ms/step - loss:
39825702912.0000 - mae: 125575.7188 - val_loss: 46789984256.0000 -
val_mae: 130722.4531
Epoch 40/100
432/432 [==============================] - 1s 2ms/step - loss:
39761915904.0000 - mae: 125557.4297 - val_loss: 46681460736.0000 -
val_mae: 130646.9141
Epoch 41/100
432/432 [==============================] - 1s 2ms/step - loss:
39701671936.0000 - mae: 125644.1016 - val_loss: 46724177920.0000 -
val_mae: 130202.6641
Epoch 42/100
432/432 [==============================] - 1s 2ms/step - loss:
39646560256.0000 - mae: 125265.3125 - val_loss: 46551461888.0000 -
val_mae: 130284.2109
Epoch 43/100
432/432 [==============================] - 1s 3ms/step - loss:
39589060608.0000 - mae: 125346.2578 - val_loss: 46518415360.0000 -
val_mae: 130041.8828
Epoch 44/100
432/432 [==============================] - 1s 3ms/step - loss:
39538864128.0000 - mae: 125309.2109 - val_loss: 46565552128.0000 -
val_mae: 129737.8359
Epoch 45/100
432/432 [==============================] - 3s 7ms/step - loss:
39495991296.0000 - mae: 124794.0547 - val_loss: 46312050688.0000 -
val_mae: 129938.3438
Epoch 46/100
432/432 [==============================] - 1s 3ms/step - loss:
39450857472.0000 - mae: 125012.7031 - val_loss: 46198538240.0000 -
val_mae: 129948.5000
Epoch 47/100
432/432 [==============================] - 1s 3ms/step - loss:
39407345664.0000 - mae: 124878.3516 - val_loss: 46189817856.0000 -
val_mae: 129745.1406
```

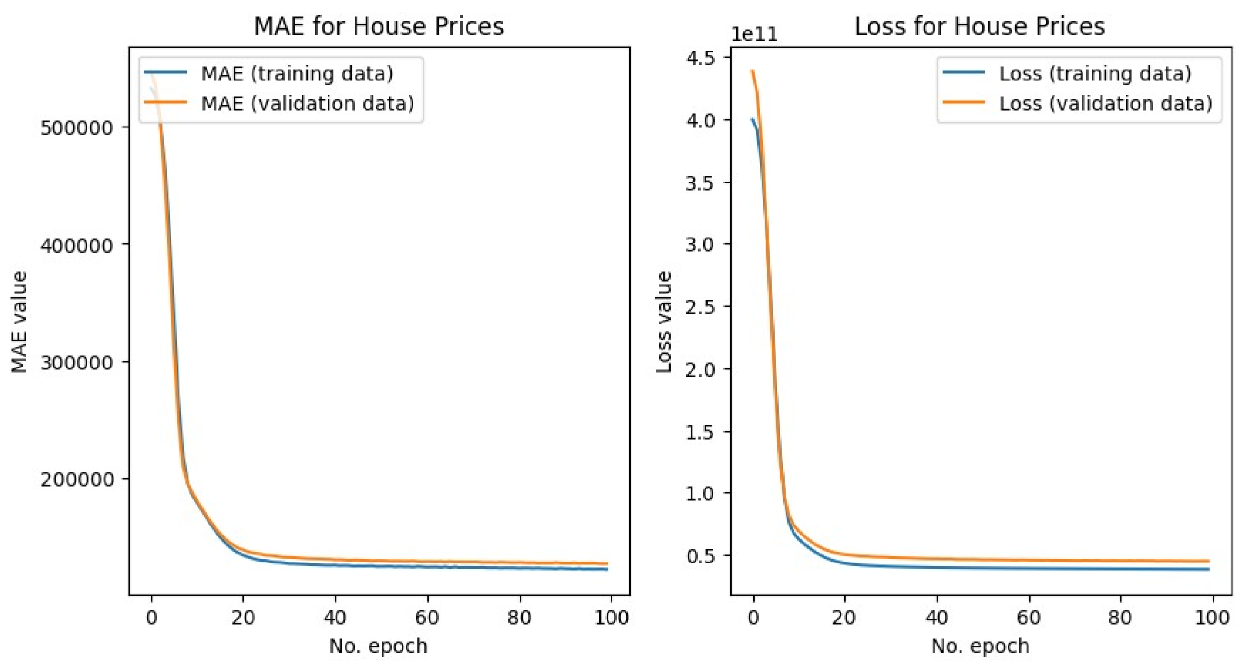
```
Epoch 48/100
432/432 [==============================] - 1s 2ms/step - loss:
39366168576.0000 - mae: 124960.8594 - val_loss: 46135353344.0000 -
val_mae: 129643.5703
Epoch 49/100
432/432 [==============================] - 1s 2ms/step - loss:
39311507456.0000 - mae: 125055.0391 - val_loss: 46207221760.0000 -
val_mae: 129355.8516
Epoch 50/100
432/432 [==============================] - 1s 2ms/step - loss:
39292407808.0000 - mae: 124575.8047 - val_loss: 46053593088.0000 -
val_mae: 129460.9141
Epoch 51/100
432/432 [==============================] - 1s 3ms/step - loss:
39251214336.0000 - mae: 124585.4375 - val_loss: 45939752960.0000 -
val_mae: 129501.3438
Epoch 52/100
432/432 [==============================] - 1s 2ms/step - loss:
39218618368.0000 - mae: 124645.8125 - val_loss: 45919375360.0000 -
val_mae: 129358.9844
Epoch 53/100
432/432 [==============================] - 1s 2ms/step - loss:
39186468864.0000 - mae: 124740.6172 - val_loss: 45938257920.0000 -
val_mae: 129117.3984
Epoch 54/100
432/432 [==============================] - 1s 2ms/step - loss:
39152824320.0000 - mae: 124378.1094 - val_loss: 45859831808.0000 -
val_mae: 129134.8984
Epoch 55/100
432/432 [==============================] - 1s 2ms/step - loss:
39130996736.0000 - mae: 124563.8281 - val_loss: 45815394304.0000 -
val_mae: 129097.2422
Epoch 56/100
432/432 [==============================] - 1s 3ms/step - loss:
39092973568.0000 - mae: 124356.2656 - val_loss: 45799559168.0000 -
val_mae: 129012.3828
Epoch 57/100
432/432 [==============================] - 2s 4ms/step - loss:
39066189824.0000 - mae: 124388.9766 - val_loss: 45745369088.0000 -
val_mae: 128994.5234
Epoch 58/100
432/432 [==============================] - 1s 3ms/step - loss:
39035109376.0000 - mae: 124060.7734 - val_loss: 45580161024.0000 -
val_mae: 129253.1016
Epoch 59/100
432/432 [==============================] - 1s 2ms/step - loss:
39019745280.0000 - mae: 124579.6797 - val_loss: 45697015808.0000 -
val_mae: 128759.2969
Epoch 60/100
```

```
432/432 [==============================] - 1s 2ms/step - loss:
38990049280.0000 - mae: 124295.1875 - val_loss: 45722296320.0000 -
val_mae: 128620.3594
Epoch 61/100
432/432 [==============================] - 1s 2ms/step - loss:
38962257920.0000 - mae: 124011.3984 - val_loss: 45595451392.0000 -
val_mae: 128778.0000
Epoch 62/100
432/432 [==============================] - 1s 3ms/step - loss:
38942380032.0000 - mae: 124102.0000 - val_loss: 45574074368.0000 -
val_mae: 128675.2344
Epoch 63/100
432/432 [==============================] - 1s 2ms/step - loss:
38918696960.0000 - mae: 124162.0469 - val_loss: 45545828352.0000 -
val_mae: 128587.0547
Epoch 64/100
432/432 [==============================] - 1s 2ms/step - loss:
38890418176.0000 - mae: 123792.4453 - val_loss: 45418360832.0000 -
val_mae: 128802.1250
Epoch 65/100
432/432 [==============================] - 1s 2ms/step - loss:
38871130112.0000 - mae: 124292.2188 - val_loss: 45595983872.0000 -
val_mae: 128311.9297
Epoch 66/100
432/432 [==============================] - 1s 2ms/step - loss:
38841868288.0000 - mae: 123502.7969 - val_loss: 45362393088.0000 -
val_mae: 128793.0703
Epoch 67/100
432/432 [==============================] - 1s 2ms/step - loss:
38841741312.0000 - mae: 124183.4766 - val_loss: 45445193728.0000 -
val_mae: 128386.2891
Epoch 68/100
432/432 [==============================] - 1s 3ms/step - loss:
38803378176.0000 - mae: 123676.0156 - val_loss: 45362515968.0000 -
val_mae: 128493.6484
Epoch 69/100
432/432 [==============================] - 1s 3ms/step - loss:
38789992448.0000 - mae: 123753.5078 - val_loss: 45345841152.0000 -
val_mae: 128406.7656
Epoch 70/100
432/432 [==============================] - 2s 4ms/step - loss:
38773575680.0000 - mae: 123750.7891 - val_loss: 45309509632.0000 -
val_mae: 128388.3203
Epoch 71/100
432/432 [==============================] - 1s 3ms/step - loss:
38757003264.0000 - mae: 123690.6875 - val_loss: 45247135744.0000 -
val_mae: 128422.2422
Epoch 72/100
432/432 [==============================] - 1s 3ms/step - loss:
```

```
38736764928.0000 - mae: 123636.1016 - val_loss: 45216485376.0000 -
val_mae: 128384.8672
Epoch 73/100
432/432 [==============================] - 1s 2ms/step - loss:
38717890560.0000 - mae: 123788.8281 - val_loss: 45335879680.0000 -
val_mae: 127999.9141
Epoch 74/100
432/432 [==============================] - 1s 2ms/step - loss:
38694973440.0000 - mae: 123479.1406 - val_loss: 45301985280.0000 -
val_mae: 128021.0391
Epoch 75/100
432/432 [==============================] - 1s 2ms/step - loss:
38679486464.0000 - mae: 123440.3516 - val_loss: 45232709632.0000 -
val_mae: 128052.8984
Epoch 76/100
432/432 [==============================] - 1s 2ms/step - loss:
38669750272.0000 - mae: 123471.3906 - val_loss: 45316538368.0000 -
val_mae: 127830.6250
Epoch 77/100
432/432 [==============================] - 1s 2ms/step - loss:
38650138624.0000 - mae: 123220.2344 - val_loss: 45098250240.0000 -
val_mae: 128215.2109
Epoch 78/100
432/432 [==============================] - 1s 2ms/step - loss:
38638227456.0000 - mae: 123368.1562 - val_loss: 45142401024.0000 -
val_mae: 128013.7734
Epoch 79/100
432/432 [==============================] - 1s 2ms/step - loss:
38610702336.0000 - mae: 123307.2578 - val_loss: 45159141376.0000 -
val_mae: 127874.0625
Epoch 80/100
432/432 [==============================] - 1s 3ms/step - loss:
38594945024.0000 - mae: 123320.5938 - val_loss: 45191061504.0000 -
val_mae: 127764.6953
Epoch 81/100
432/432 [==============================] - 2s 4ms/step - loss:
38578610176.0000 - mae: 123040.7969 - val_loss: 45062971392.0000 -
val_mae: 127913.6797
Epoch 82/100
432/432 [==============================] - 1s 3ms/step - loss:
38568501248.0000 - mae: 123201.5781 - val_loss: 45080846336.0000 -
val_mae: 127797.6406
Epoch 83/100
432/432 [==============================] - 1s 3ms/step - loss:
38545321984.0000 - mae: 123187.2891 - val_loss: 45167841280.0000 -
val_mae: 127561.6172
Epoch 84/100
432/432 [==============================] - 1s 3ms/step - loss:
38535106560.0000 - mae: 122956.3516 - val_loss: 45013409792.0000 -
```

```
val_mae: 127729.1016
Epoch 85/100
432/432 [==============================] - 1s 3ms/step - loss:
38508613632.0000 - mae: 123157.4766 - val_loss: 45169147904.0000 -
val_mae: 127405.1953
Epoch 86/100
432/432 [==============================] - 1s 2ms/step - loss:
38499155968.0000 - mae: 122935.6172 - val_loss: 45102809088.0000 -
val_mae: 127463.2656
Epoch 87/100
432/432 [==============================] - 1s 2ms/step - loss:
38488363008.0000 - mae: 122817.2656 - val_loss: 45041614848.0000 -
val_mae: 127469.6172
Epoch 88/100
432/432 [==============================] - 1s 2ms/step - loss:
38467678208.0000 - mae: 122808.3281 - val_loss: 45045051392.0000 -
val_mae: 127389.3047
Epoch 89/100
432/432 [==============================] - 1s 2ms/step - loss:
38439137280.0000 - mae: 122521.7578 - val_loss: 44828573696.0000 -
val_mae: 127791.5859
Epoch 90/100
432/432 [==============================] - 1s 2ms/step - loss:
38431617024.0000 - mae: 122987.6172 - val_loss: 44943052800.0000 -
val_mae: 127405.2578
Epoch 91/100
432/432 [==============================] - 1s 2ms/step - loss:
38422835200.0000 - mae: 122690.2422 - val_loss: 44964646912.0000 -
val_mae: 127316.1875
Epoch 92/100
432/432 [==============================] - 2s 4ms/step - loss:
38400835584.0000 - mae: 122586.4609 - val_loss: 44929638400.0000 -
val_mae: 127295.4375
Epoch 93/100
432/432 [==============================] - 1s 3ms/step - loss:
38377000960.0000 - mae: 122346.3594 - val_loss: 44776136704.0000 -
val_mae: 127554.6094
Epoch 94/100
432/432 [==============================] - 1s 3ms/step - loss:
38374891520.0000 - mae: 122754.6562 - val_loss: 44891013120.0000 -
val_mae: 127158.4766
Epoch 95/100
432/432 [==============================] - 1s 3ms/step - loss:
38360551424.0000 - mae: 122271.8594 - val_loss: 44777177088.0000 -
val_mae: 127374.7578
Epoch 96/100
432/432 [==============================] - 1s 2ms/step - loss:
38341656576.0000 - mae: 122444.6641 - val_loss: 44768198656.0000 -
val_mae: 127261.7578
```

```
Epoch 97/100
432/432 [==============================] - 1s 3ms/step - loss:
38329466880.0000 - mae: 122279.3203 - val_loss: 44695781376.0000 -
val_mae: 127367.4609
Epoch 98/100
432/432 [==============================] - 1s 3ms/step - loss:
38308651008.0000 - mae: 122373.3438 - val_loss: 44764430336.0000 -
val_mae: 127111.1172
Epoch 99/100
432/432 [==============================] - 1s 2ms/step - loss:
38297440256.0000 - mae: 122376.7031 - val_loss: 44838854656.0000 -
val_mae: 126889.0547
Epoch 100/100
432/432 [==============================] - 1s 2ms/step - loss:
38281707520.0000 - mae: 122163.0547 - val_loss: 44772462592.0000 -
val_mae: 126920.4375
135/135 [==============================] - 0s 2ms/step - loss:
39687159808.0000 - mae: 123552.4844
135/135 [==============================] - 0s 2ms/step
Test Loss: 39687159808.0, Test MAE: 123552.484375
```



Conclusion

For this Neural Network algorithm, from the graph above, we can see that it quickly learns from the dataset and the feeding data to linearize at around 20 epoch for both the MAE and loss. However, we must consider that both values seems to be high with a MAE of 123552.4844 and a loss of 39687159808.0 at epoch 100. This two results are pretty high, precisely the loss. More data and training could be interesting and maybe using a better algorithm for the neural netork could help for this problem.