In Summary;

This project focuses on predicting real estate prices based on various factors such as location, the number of bedrooms and bathrooms, floor count, waterfront views, house grade, and renovation status using Python. After importing the necessary libraries and loading the dataset, I conducted an initial exploratory data analysis (EDA) to understand the structure and distribution of the data. Visualizations were created to explore the relationships between the target variable (price) and key features, such as location, bedroom and bathroom counts, and whether the property had a waterfront view. Additionally, histograms and correlation matrices were used to identify patterns and correlations between numerical variables, providing further insights into how these features impacted property prices.

Once the data was explored, I split the dataset into training and testing sets to train and evaluate predictive models. I visualized the performance of the model to assess the accuracy and success level of the predictions. Based on initial results, I refined the model by adjusting parameters and improving the feature selection process, ultimately achieving better predictive performance. This iterative approach helped optimize the model for more accurate real estate price predictions, demonstrating the power of data visualization, correlation analysis, and model tuning in making informed predictions in the real estate market.

* Import necessary libraries

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from math import sqrt
```

* Import scikit-learn modules

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
```

* Import TensorFlow and Keras modules

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

* Import dataset

```
df =
pd.read_csv('C://Users//Asus//OneDrive//Masaüstü//website//python//PREDICTION//house
_price_prediction//kc_house_data.csv', encoding = 'ISO-8859-1')
```

# Encoding specifies the character encoding of the CSV file.

```
df.tail(5)
df.info()
```

* Column names standardization

```
df.columns = map(str.upper, df.columns)
df.columns = [col.replace(' ', '_') for col in df.columns]
df.tail(5)
```

* Convert datetime

```
df['DATE'] = pd.to_datetime(df['DATE'], format = "%Y%m%dT%H%M%S")
df.tail(5)
```

* Check for null values

```
null_mask = df.isnull()
null_values = null_mask.sum().sum()
```

# Count the total number of null values.

```
print("Number of null values: ", null_values)
```

* Check for dublicate values

```
duplicates_mask = df.duplicated()
num_duplicates = sum(duplicates_mask)
```

# Count the number of duplicate rows.

```
print("Number of duplicate rows: ", num_duplicates)
```

* Understand the dataset

```
df.info()
df.describe()
```

# Exploratory Data Analysis

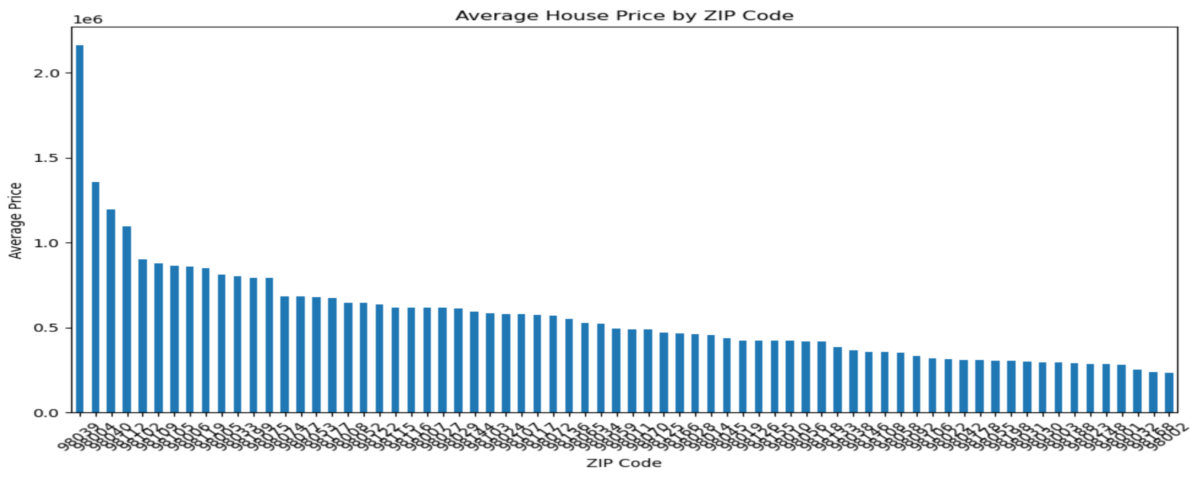
* Descriptive Statistics

```
average_price = df['PRICE'].mean()
cheapest_price = df['PRICE'].min()
average_bathrooms = df['BATHROOMS'].mean()
average_bedrooms = df['BEDROOMS'].mean()
max_bedrooms = df['BEDROOMS'].max()
```

```
        print('Cheapest house price is: ${:,.2f}'.format(cheapest_price))
        print('Average house price is: ${:,.2f}'.format(average_price))
        print('Average number of bathrooms is: ${:,.2f}'.format(average_bathrooms))
        print('Average number of bedrooms is: ${:,.2f}'.format(average_bedrooms))
        print('Maximum number of bedrooms is: ${:,.2f}'.format(max_bedrooms))
```

* Impact of Location on Real Estate Prices
* Grouping data, average price calculation, indetifying key ZIP codes, visualizing insights

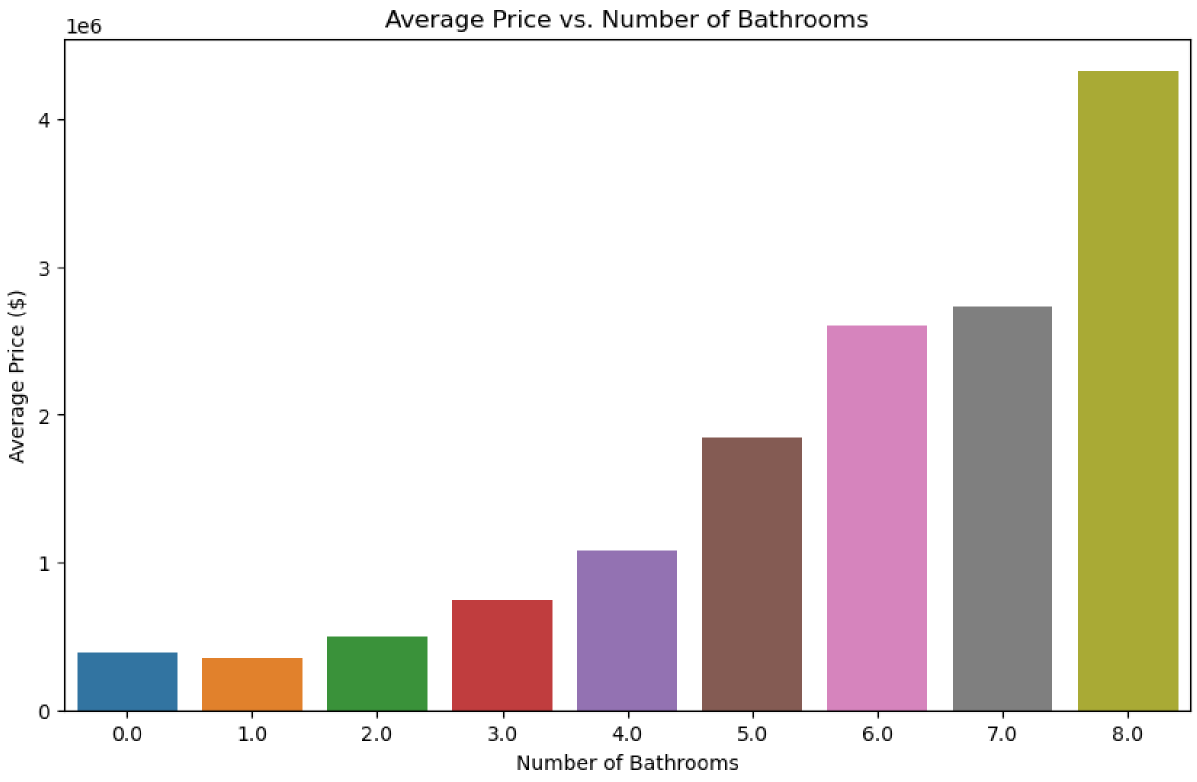
```
        avg_price_by_zip = df.groupby('ZIPCODE')['PRICE'].mean()
        avg_price_by_zip = avg_price_by_zip.sort_values(ascending=False)
        top_10_zip_avg_prices = avg_price_by_zip[0:9]


print('Top 10 ZIP codes with highest average prices: ')
for zip_code, average_price in top_10_zip_avg_prices.items():
  print('ZIP code: {:>5} | Average Price: ${:,.2f}'.format(zip_code, average_price))
        plt.figure(figsize = (10,6))
        avg_price_by_zip.plot(kind='bar')
        plt.title('Average House Price by ZIP Code')
        plt.xlabel('ZIP Code')
        plt.ylabel('Average Price')
        plt.xticks(rotation = 45)
        plt.tight_layout()
        plt.show()
```



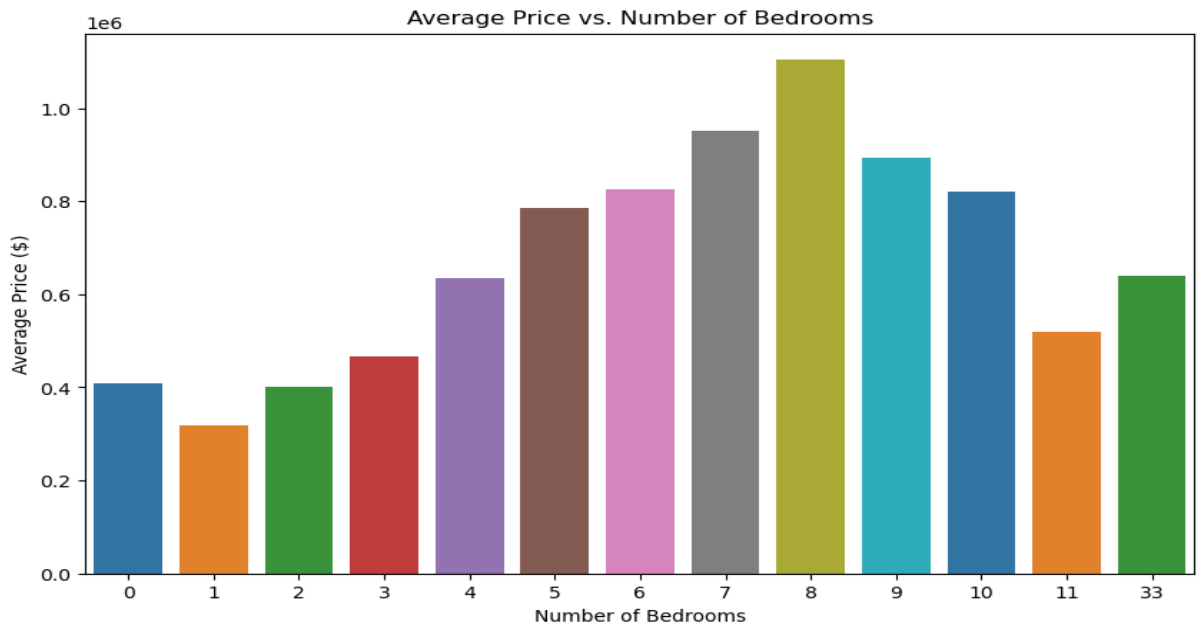
Impact of Housing Features

* Bedrooms and Bathrooms

```
avg_price_per_bedroom = df.groupby('BEDROOMS')['PRICE'].mean()

plt.figure(figsize = (10,6))

sns.barplot(x = avg_price_per_bedroom.index, y=avg_price_per_bedroom.values,
palette=sns.color_palette("tab10", len(avg_price_per_bedroom)))


plt.title('Average Price vs. Number of Bedrooms')

plt.xlabel('Number of Bedrooms')

plt.ylabel('Average Price ($)')

plt.show()
```

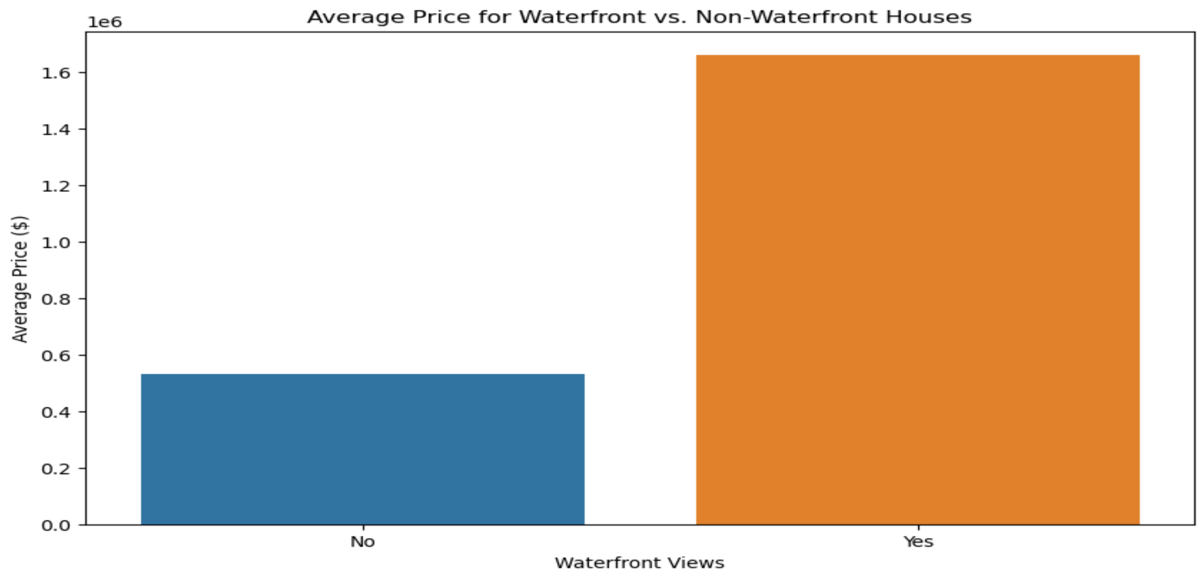


```
df['BATHROOMS'] = df['BATHROOMS'].round()

avg_price_per_bathroom = df.groupby('BATHROOMS')['PRICE'].mean()

plt.figure(figsize = (10,6))

sns.barplot(x = avg_price_per_bathroom.index, y=avg_price_per_bathroom.values,
palette=sns.color_palette("tab10", len(avg_price_per_bathroom)))


plt.title('Average Price vs. Number of Bathrooms')

plt.xlabel('Number of Bathrooms')

plt.ylabel('Average Price ($)')

plt.show()
```



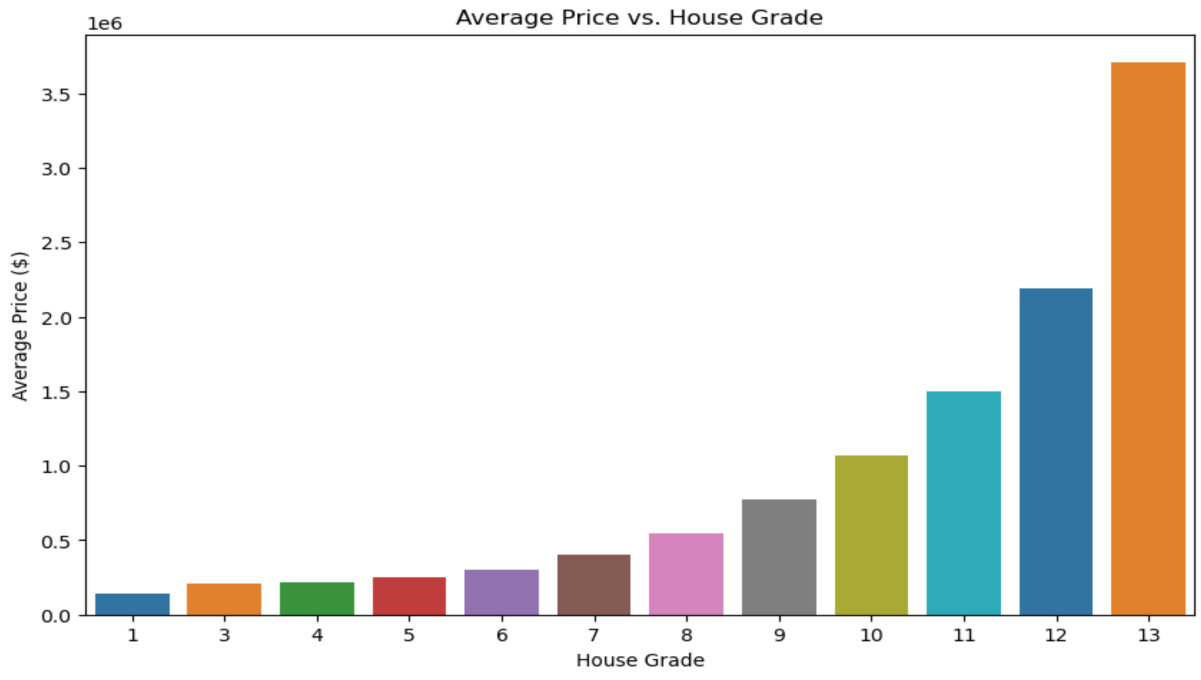
* Waterfront Views

```
avg_price_waterfront = df.groupby('WATERFRONT')['PRICE'].mean()

plt.figure(figsize = (10,6))

sns.barplot(x = avg_price_waterfront.index, y=avg_price_waterfront.values,
palette=sns.color_palette("tab10", len(avg_price_waterfront)))


plt.title('Average Price for Waterfront vs. Non-Waterfront Houses')

plt.xlabel('Waterfront Views')

plt.ylabel('Average Price ($)')

plt.xticks([0,1], ['No', 'Yes'])

plt.show()
```



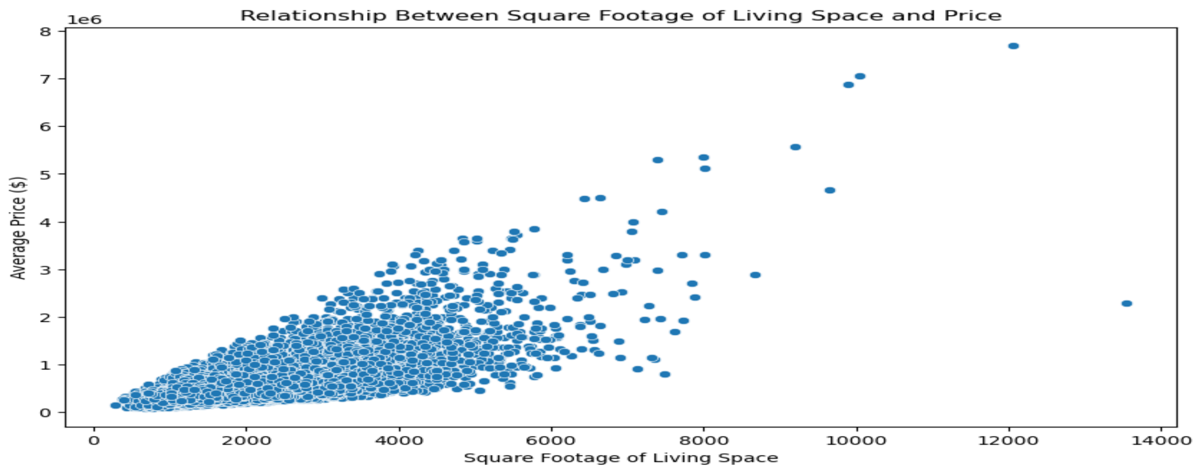
* House Grade Impact

```
avg_price_per_grade = df.groupby('GRADE')['PRICE'].mean()

plt.figure(figsize = (10,6))

sns.barplot(x = avg_price_per_grade.index, y=avg_price_per_grade.values, 
palette=sns.color_palette("tab10", len(avg_price_per_grade)))


plt.title('Average Price vs. House Grade')

plt.xlabel('House Grade')

plt.ylabel('Average Price ($)')

plt.show()
```

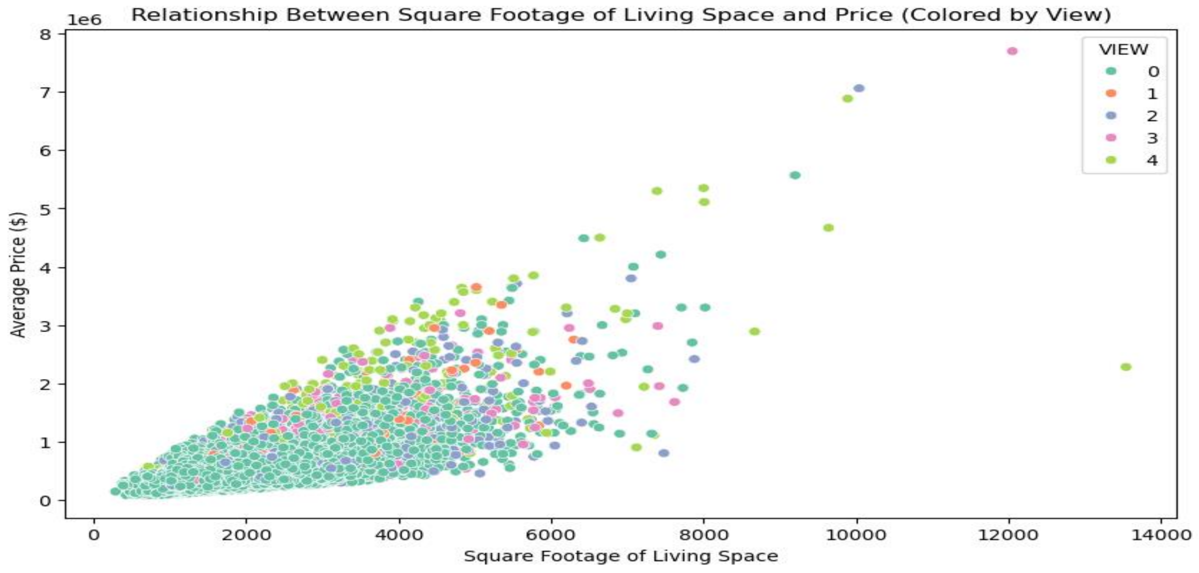


* Living Space and Price

```
plt.figure(figsize = (10,6))
sns.scatterplot(x = 'SQFT_LIVING', y='PRICE', data=df)
plt.title('Relationship Between Square Footage of Living Space and Price')
plt.xlabel('Square Footage of Living Space')
plt.ylabel('Average Price ($)')
plt.show()
```

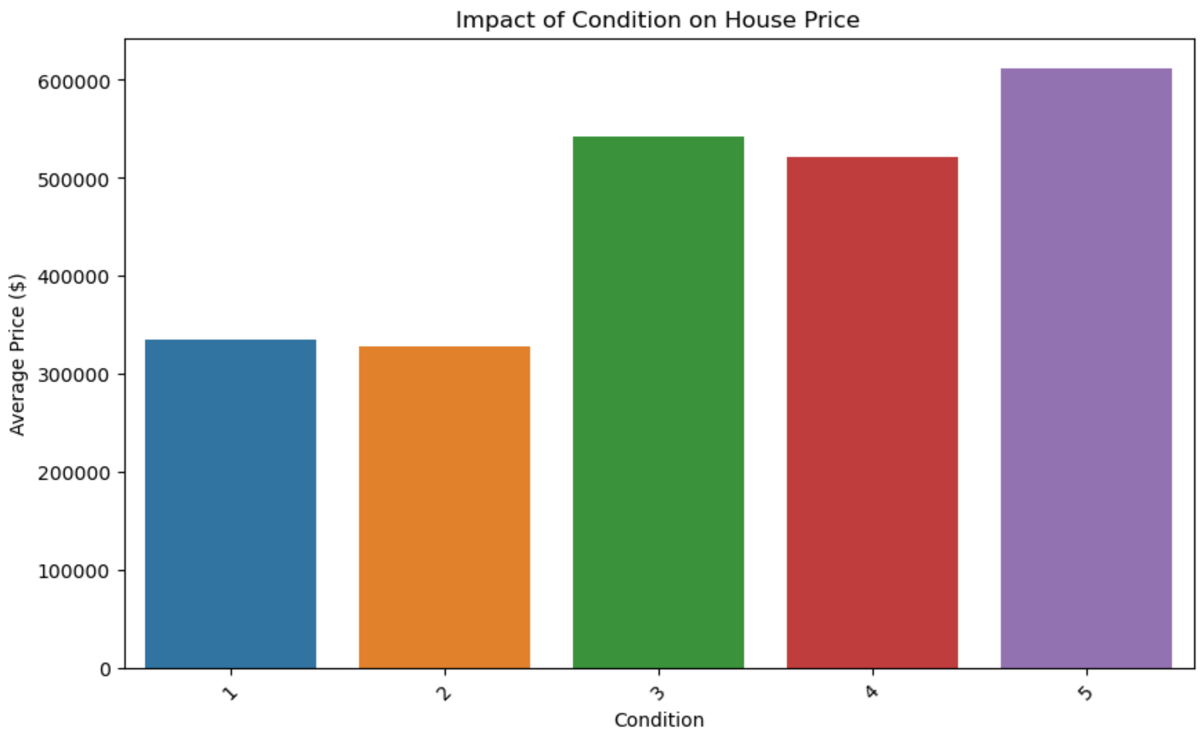


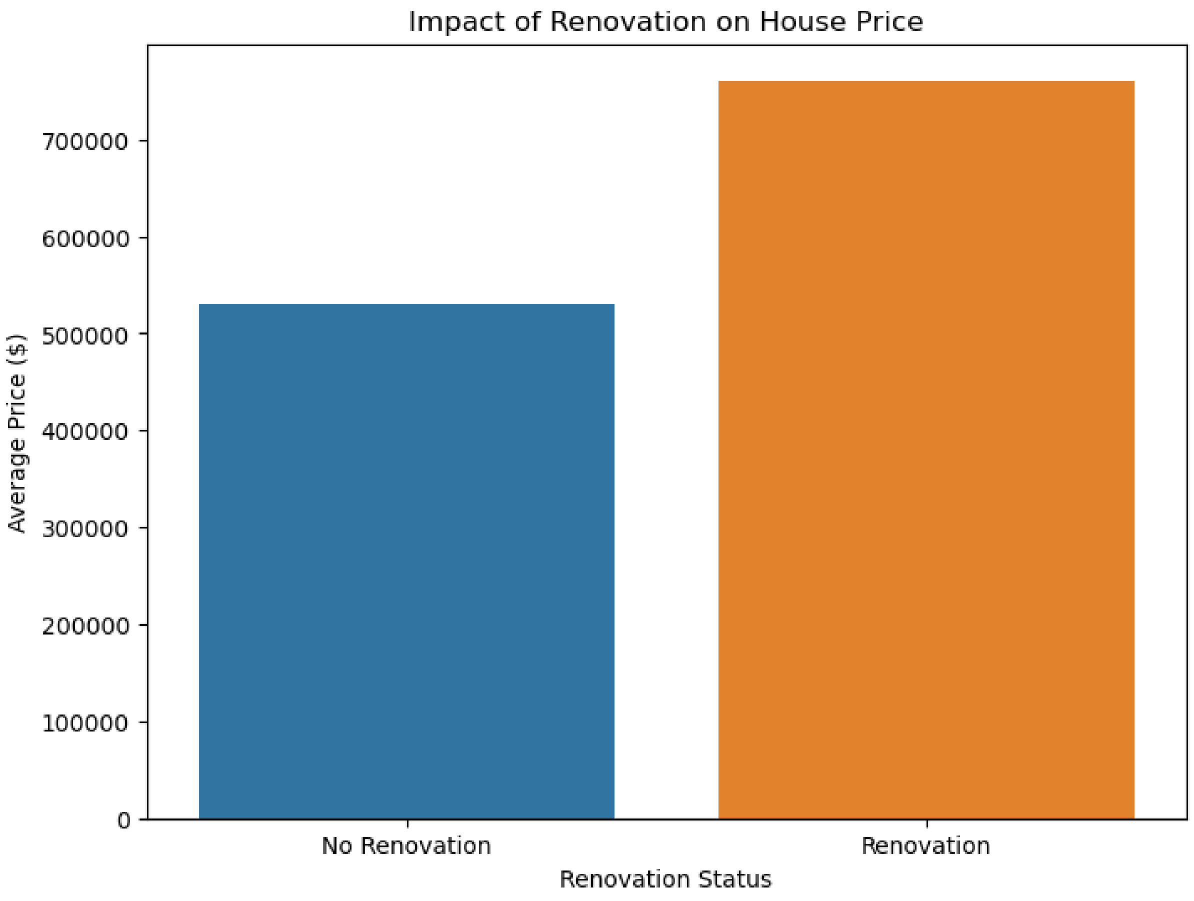
```
plt.figure(figsize = (10,6))
sns.scatterplot(data = df, x = 'SQFT_LIVING', y='PRICE', hue='VIEW', palette = 'Set2')
plt.title('Relationship Between Square Footage of Living Space and Price (Colored by View)')
plt.xlabel('Square Footage of Living Space')
plt.ylabel('Average Price ($)')
plt.show()
```



* Renovation and Condition

```
avg_price_by_condition = df.groupby('CONDITION')['PRICE'].mean()

df['RENOVATION_STATUS'] = df['YR_RENOVATED'].apply(lambda year: 'Renovation' if year >0
else 'No Renovation')

df[['RENOVATION_STATUS', 'YR_RENOVATED']].head(10)

avg_price_by_renovation = df.groupby('RENOVATION_STATUS')['PRICE'].mean()


plt.figure(figsize = (10,6))

sns.barplot(x = avg_price_by_condition.index, y=avg_price_by_condition.values,
palette=sns.color_palette("tab10", len(avg_price_by_condition)))

plt.title('Impact of Condition on House Price')

plt.xlabel('Condition')

plt.ylabel('Average Price ($)')

plt.xticks(rotation=45)

plt.show()


plt.figure(figsize=(8,6))

sns.barplot(x=avg_price_by_renovation.index, y=avg_price_by_renovation.values,
palette=sns.color_palette("tab10", len(avg_price_by_renovation)))

plt.title('Impact of Renovation on House Price')

plt.xlabel('Renovation Status')

plt.ylabel('Average Price ($)')

plt.show()
```


Impact of Condition on House Price
Average Price ($)
600000
500000
400000
300000
200000
100000
0
1
2
3
4
5
Condition

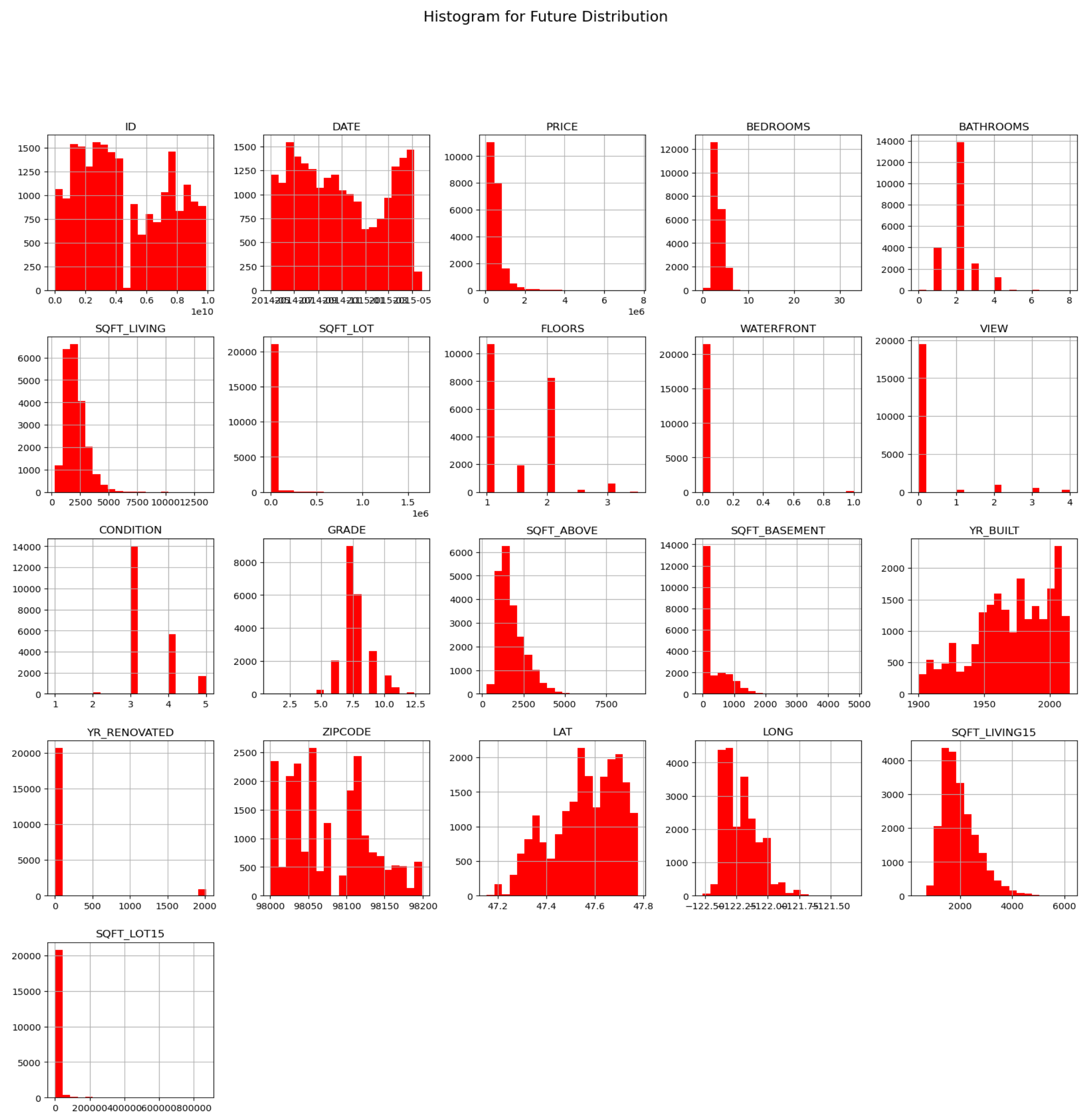

Impact of Renovation on House Price
Average Price ($)
700000
600000
500000
400000
300000
200000
100000
0
No Renovation
Renovation
Renovation Status

# Preliminary Data Exploration for Deep Learning Preparation

* Future Distribution Histogram

# Histogram provide a visual overview of the distribution of each feature.

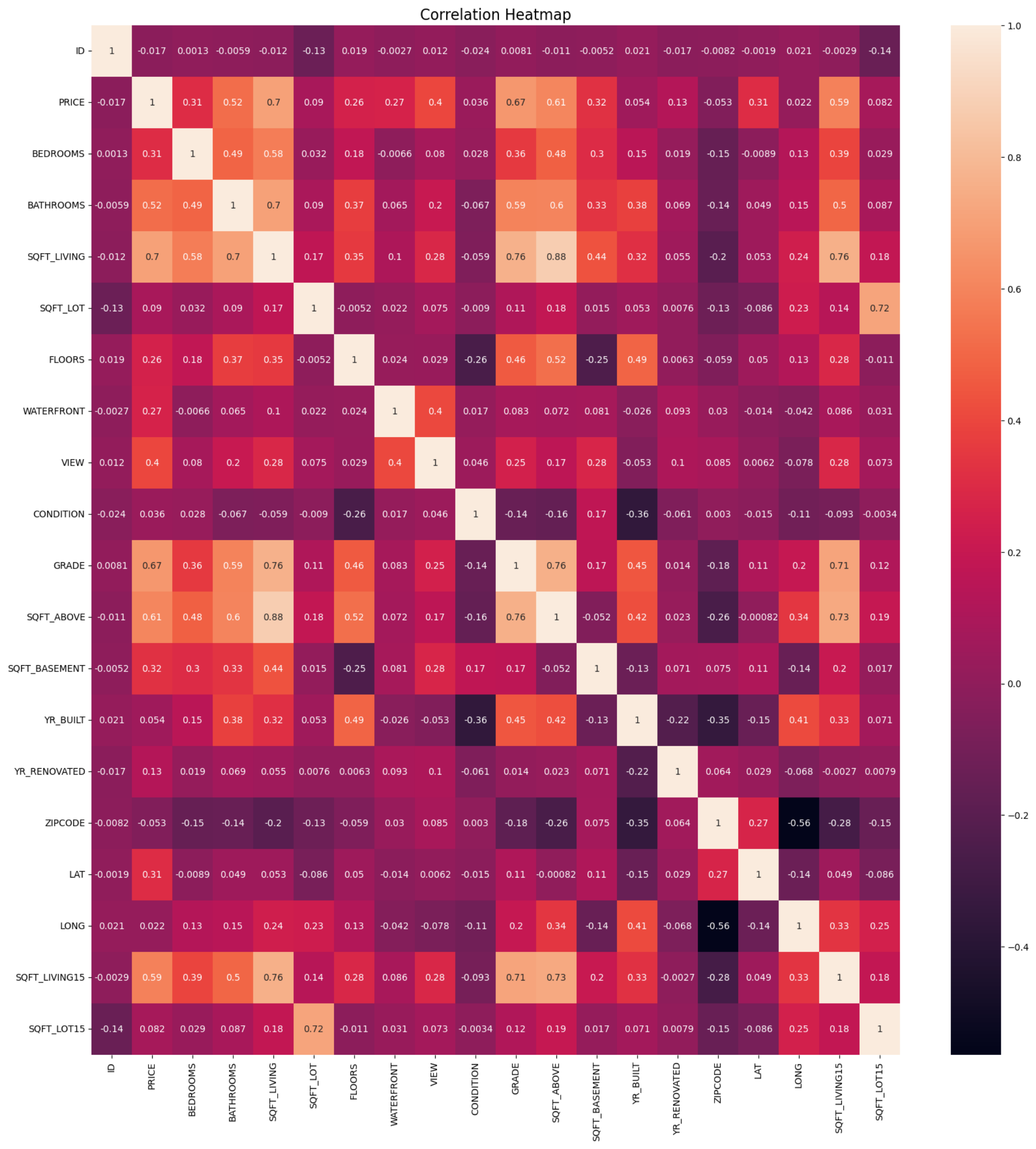
```
        plt.figure(figsize=(20,20))

        df.hist(bins=20, figsize=(20,20), color='r')

        plt.suptitle('Histogram for Future Distribution', fontsize=16)

        plt.show()
```



* Correlation Heatmap

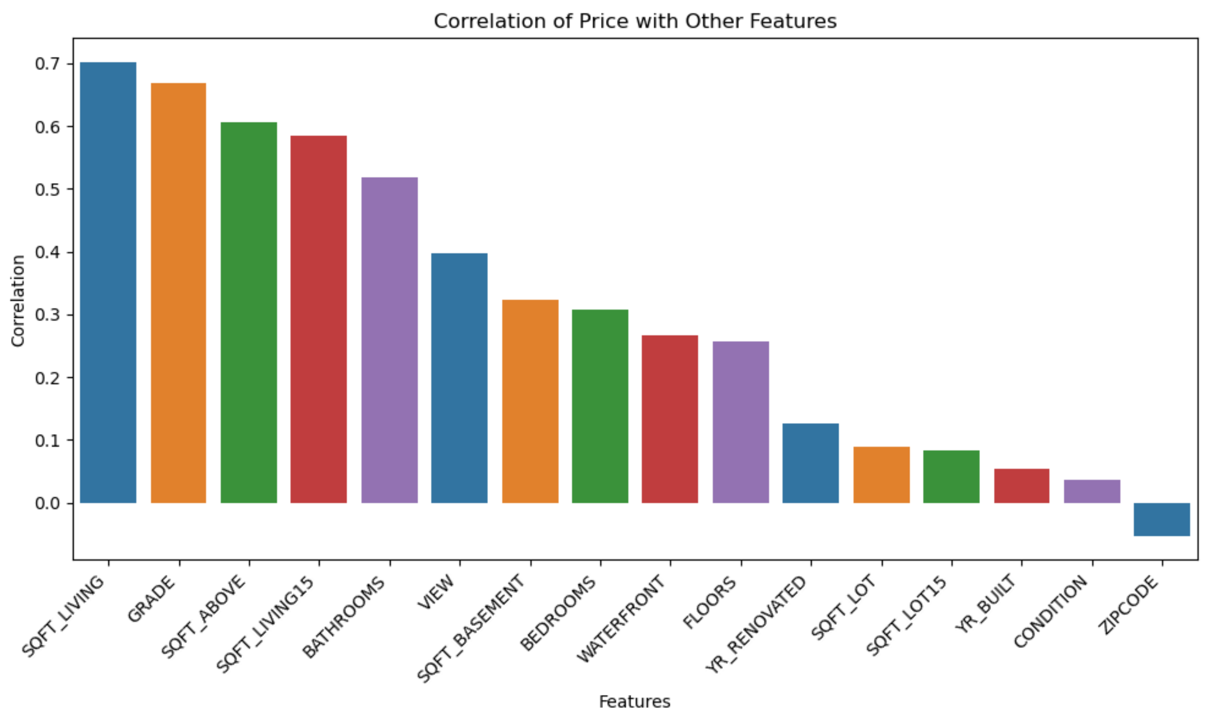
# A correlation heatmap help us a understand the relationship between features.

```
numeric_df = df.select_dtypes(include=['float64', 'int64'])

plt.figure(figsize=(20,20))

sns.heatmap(numeric_df.corr(), annot=True)

plt.title('Correlation Heatmap', fontsize = 16)

plt.show()
```



* Correlation Between Price and Other Features

```
price_correlation = numeric_df.corr()['PRICE'].drop(['PRICE', 'LONG', 'LAT', 'ID'], errors = 'ignore')

sorted_correlations = price_correlation.sort_values(ascending=False)


plt.figure(figsize=(10,6))

sns.barplot(x = sorted_correlations.index, y=sorted_correlations.values, palette=sns.color_palette("tab10", len(avg_price_by_condition)))

plt.title('Correlation of Price with Other Features')

plt.xlabel('Features')

plt.ylabel('Correlation')

plt.xticks(rotation=45, ha='right')

plt.tight_layout()

plt.show()
```



* Deep Learning Model

* Prepare for Data Training

```
selected_features = ['BEDROOMS', 'BATHROOMS', 'SQFT_LIVING', 'SQFT_LOT', 'FLOORS',
'SQFT_ABOVE', 'SQFT_BASEMENT']

x = df[selected_features]

y =df['PRICE']

# x = selected features for input

# y = target variable
```

* Scaling the input features and target variable

```
scaler = MinMaxScaler()

x_scaled = scaler.fit_transform(x)

y = y.values.reshape(-1,1)

y_scaled = scaler.fit_transform(y)

y_scaled
```

* Split the data into training and testing sets

```
x_train, x_test, y_train, y_test = train_test_split(x_scaled, y_scaled, test_size=0.25)

print('Size of training data: ', x_train.shape[0])

print('Size of testing data: ', x_test.shape[0])
```

# Build and Train Deep Learning Model

* Train the model

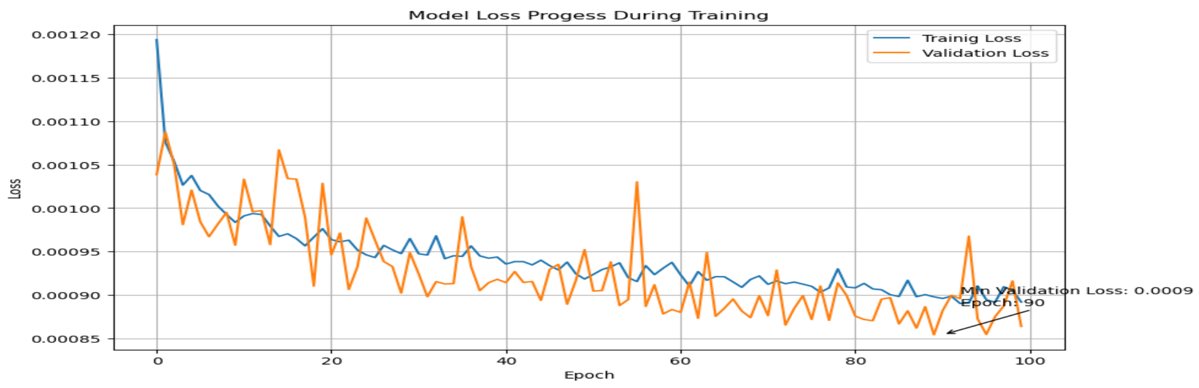
# Assuming x_train and y_train are numpy arrays

```
x_train_tensor = torch.tensor(x_train, dtype=torch.float32)

y_train_tensor = torch.tensor(y_train, dtype=torch.float32)
```

# Create a TensorDataset and DataLoader

```
dataset = TensorDataset(x_train_tensor, y_train_tensor)

train_loader = DataLoader(dataset, batch_size=50, shuffle=True)
```

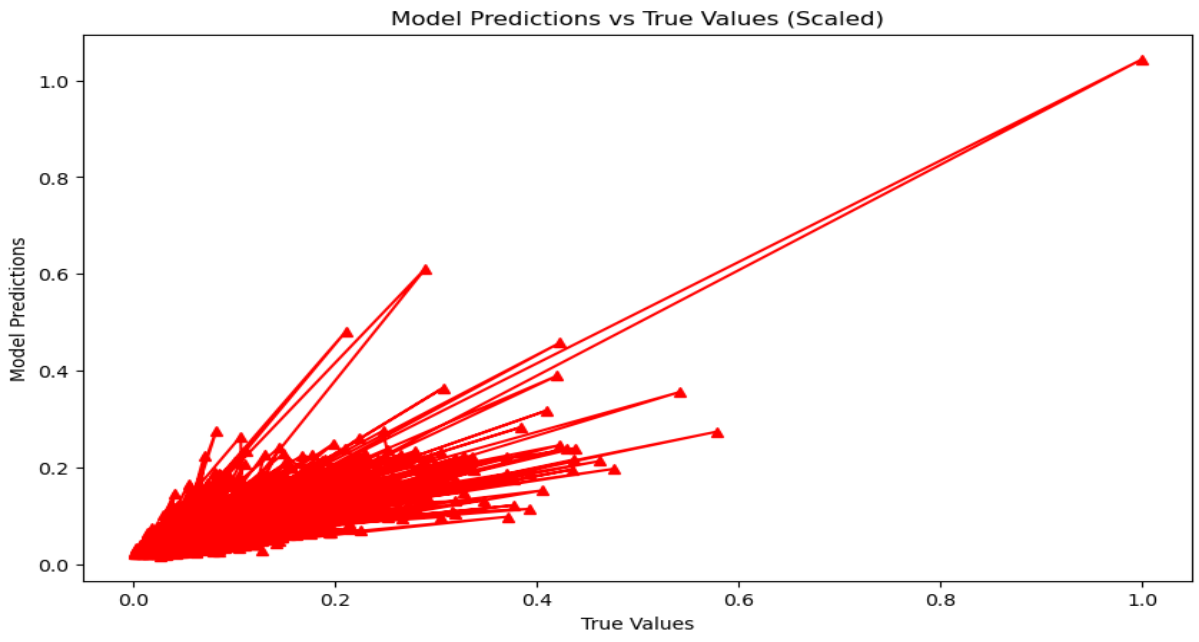
```
# Define the model
    class MyModel(nn.Module):
        def __init__(self):
            super(MyModel, self).__init__()
            self.layer1 = nn.Linear(7, 100)
            self.layer2 = nn.Linear(100, 100)
            self.layer3 = nn.Linear(100, 100)
            self.output_layer = nn.Linear(100, 1)
            self.relu = nn.ReLU()

        def forward(self, x):
            x = self.relu(self.layer1(x))
            x = self.relu(self.layer2(x))
            x = self.relu(self.layer3(x))
            return self.output_layer(x)

    model = MyModel()

# Define loss function and optimizer
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
    epochs = 100
    validation_split = 0.2
    validation_size = int(len(dataset) * validation_split)
    train_size = len(dataset) - validation_size


    train_dataset, val_dataset = torch.utils.data.random_split(dataset, [train_size,
    validation_size])
    val_loader = DataLoader(val_dataset, batch_size=50, shuffle=False)
    train_loss_history = []
    val_loss_history = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0

# Training
        for inputs, targets in train_loader:
            optimizer.zero_grad()  # Zero out gradients
            outputs = model(inputs)  # Forward pass
            loss = criterion(outputs, targets)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)
        train_loss_history.append(train_loss)

# Validation
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for val_inputs, val_targets in val_loader:
                val_outputs = model(val_inputs)
                val_loss += criterion(val_outputs, val_targets).item()

        val_loss /= len(val_loader)
        val_loss_history.append(val_loss)
    print(f"Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}, Validation Loss:
{val_loss:.4f}")

# Results
    print("Training completed.")

* Plot loss progress
    plt.figure(figsize=(10,6))
    plt.plot(train_loss_history, label = 'Trainig Loss')
    plt.plot(val_loss_history, label = 'Validation Loss')
    plt.title('Model Loss Progess During Training')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    min_val_loss = min(val_loss_history)
    min_val_loss_epoch = val_loss_history.index(min_val_loss) +1
    plt.annotate(f'Min Validation Loss: {min_val_loss:.4f}\nEpoch: {min_val_loss_epoch}',
        xy=(min_val_loss_epoch, min_val_loss), xycoords='data',
        xytext=(10,30), textcoords='offset points',
        arrowprops=dict(arrowstyle = '->'))

    plt.grid(True)
    plt.show()
```



* A decrease in both lines indicated the model is learning and adjusting its predictions.

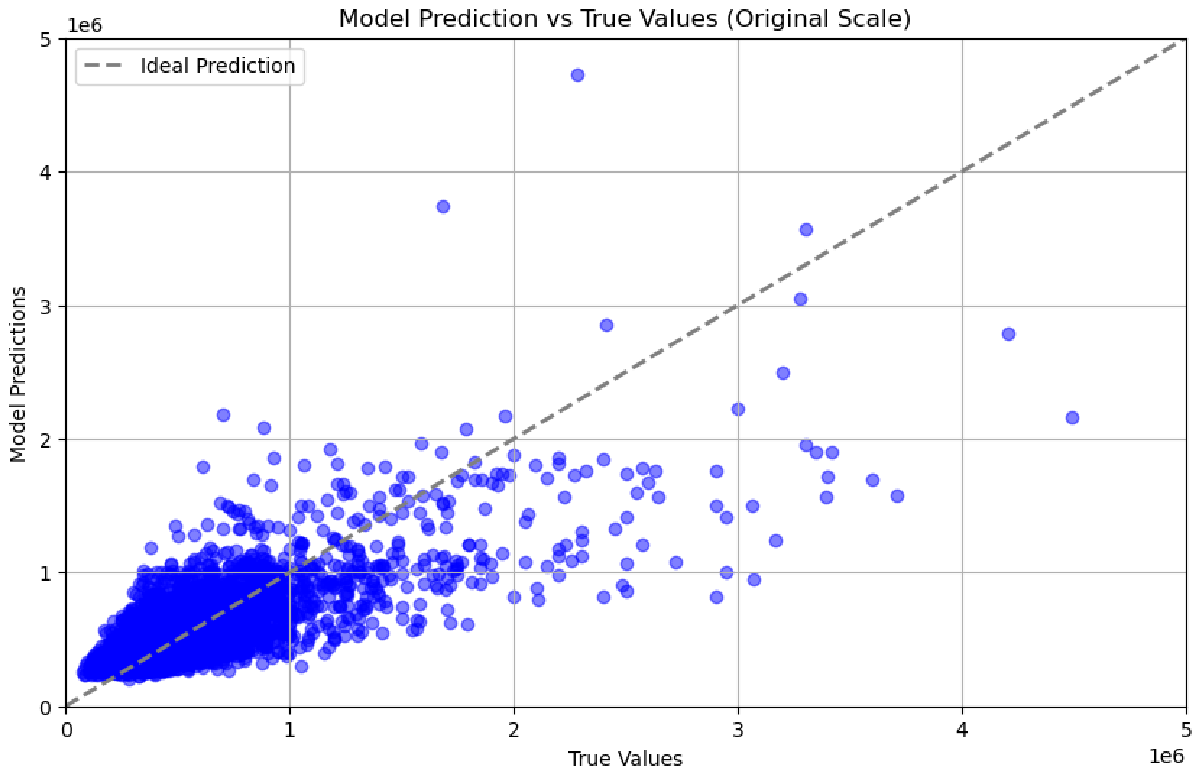
* A sudden drop in training loss coupled with a rise in validation loss may suggest overfitting, where the model becomes too tailored to trainig data.

```
# Model Predictions and Evaluation
        model.eval()


# Convert x_test to a PyTorch tensor and move it to the same device as the model (if needed)
        x_test_tensor = torch.tensor(x_test, dtype=torch.float32)


# Perform inference (model prediction)
        with torch.no_grad():
          y_predict = model(x_test_tensor).numpy()  # Convert prediction to numpy for plotting



        plt.figure(figsize = (10,6))
        plt.plot(y_test, y_predict, marker='^', color='r')
        plt.title('Model Predictions vs True Values (Scaled)')
        plt.xlabel('True Values')
        plt.ylabel('Model Predictions')
        plt.show()
```



* Scaling and Transformation Reminder

```
y_predict_orig = scaler.inverse_transform(y_predict)

y_test_orig = scaler.inverse_transform(y_test)


plt.figure(figsize=(10,6))

plt.scatter(y_test_orig, y_predict_orig, color='blue', alpha=0.5)

plt.plot([0, 5000000], [0, 5000000], linestyle= '--', color='gray', linewidth=2, label = 'Ideal
Prediction')


plt.title('Model Prediction vs True Values (Original Scale)')

plt.xlabel('True Values')

plt.ylabel('Model Predictions')

plt.xlim(0, 5000000)

plt.ylim(0, 5000000)

plt.legend()

plt.grid(True)

plt.show()
```



* Points close to the dashed line indicate accurate predictions.

* Points above the dashed line mean the model underestimated, while points below mean it overestimated.

* Points scattered around the dashed line show different levels of accuracy across price ranges.

```
# Model Performance Evaluation Metrics
```

* Root Mean Squared Error (RMSE), Mean Squared Error (MSE), Mean Absolute Error (MAE), R-Squared (COD), Adjusted R-Squared

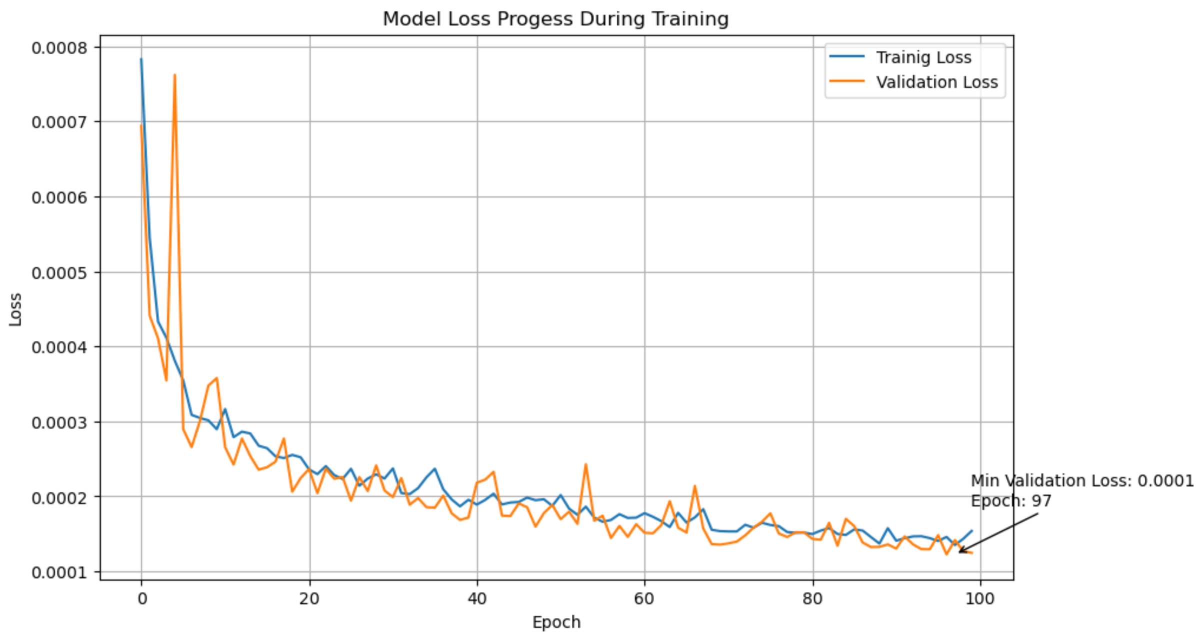
```
    RMSE = np.sqrt(mean_squared_error(y_test_orig, y_predict_orig))
    MSE = mean_squared_error(y_test_orig, y_predict_orig)
    MAE = mean_absolute_error(y_test_orig, y_predict_orig)
    r2 = r2_score(y_test_orig, y_predict_orig)

# Calculate Adjusted R-Squared
    n = len(y_test_orig)
    k = x_test.shape[1]
    adj_r2 = 1-(1-r2)*(n-1)/(n-k-1)

    print('RMSE: ', RMSE)
    print('MSE: ', MSE)
    print('MAE: ', MAE)
    print('R-Squared: ', r2)
    print('Adjusted R-Squared: ', adj_r2)
```

* Model Enhancement and Optimization

```
# Define the list of selected features for model enhancement.
    selected_features = ['BEDROOMS', 'BATHROOMS', 'SQFT_LIVING', 'SQFT_LOT', 'FLOORS',
    'SQFT_ABOVE', 'SQFT_BASEMENT', 'WATERFRONT', 'VIEW', 'CONDITION', 'GRADE',
    'SQFT_LIVING15', 'SQFT_LOT15', 'YR_BUILT', 'YR_RENOVATED', 'ZIPCODE', 'LAT', 'LONG']
    x = df[selected_features]
```

```
# y = target variable
        y = df['PRICE']

# Features scaling using Min Max Scaling
        scaler = MinMaxScaler()

# Scaling the input features (x)
        x_scaled = scaler.fit_transform(x)

# Scaling the target variable (y)
        y = y.values.reshape(-1,1)
        y_scaled = scaler.fit_transform(y)

# Splitting the data into training and testing sets
        x_train, x_test, y_train, y_test = train_test_split(x_scaled, y_scaled, test_size=0.25)

# Display the size of the training and testing data
        print('Size of training data: ', x_train.shape[0])
        print('Size of testing data: ', x_test.shape[0])

# Enhanced Deep Learning Model
# Assuming x_train and y_train are numpy arrays
        x_train_tensor = torch.tensor(x_train, dtype=torch.float32)
        y_train_tensor = torch.tensor(y_train, dtype=torch.float32)

# Create a TensorDataset and DataLoader
        dataset = TensorDataset(x_train_tensor, y_train_tensor)
        train_loader = DataLoader(dataset, batch_size=50, shuffle=True)
```

```
# Define the model
        class MyModel(nn.Module):
          def __init__(self):
            super(MyModel, self).__init__()
            self.layer1 = nn.Linear(18, 100)
            self.layer2 = nn.Linear(100, 100)
            self.layer3 = nn.Linear(100, 100)
            self.output_layer = nn.Linear(100, 1)
            self.relu = nn.ReLU()

          def forward(self, x):
            x = self.relu(self.layer1(x))
            x = self.relu(self.layer2(x))
            x = self.relu(self.layer3(x))
            return self.output_layer(x)

        model = MyModel()

# Define loss function and optimizer
        criterion = nn.MSELoss()
        optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
        epochs = 100
        validation_split = 0.2
        validation_size = int(len(dataset) * validation_split)
        train_size = len(dataset) - validation_size


        train_dataset, val_dataset = torch.utils.data.random_split(dataset, [train_size,
        validation_size])
        val_loader = DataLoader(val_dataset, batch_size=50, shuffle=False)
        train_loss_history = []
        val_loss_history = []

        for epoch in range(epochs):
            model.train()
            running_loss = 0.0

# Training
            for inputs, targets in train_loader:
                optimizer.zero_grad()  # Zero out gradients
                outputs = model(inputs)  # Forward pass
                loss = criterion(outputs, targets)  # Compute loss
                loss.backward()  # Backward pass
                optimizer.step()  # Update weights

                running_loss += loss.item()

            train_loss = running_loss / len(train_loader)
            train_loss_history.append(train_loss)

# Validation
            model.eval()
            val_loss = 0.0
            with torch.no_grad():
                for val_inputs, val_targets in val_loader:
                    val_outputs = model(val_inputs)
                    val_loss += criterion(val_outputs, val_targets).item()

            val_loss /= len(val_loader)
            val_loss_history.append(val_loss)
            print(f"Epoch {epoch+1}/{epochs}, Train Loss: {train_loss:.4f}, Validation Loss: {val_loss:.4f}")


# Results
        print("Training completed.")


* Plot loss progress
# Plot the training and validation loss progress over epochs.
        plt.figure(figsize=(10,6))
        plt.plot(train_loss_history, label = 'Trainig Loss')
        plt.plot(val_loss_history, label = 'Validation Loss')
        plt.title('Model Loss Progess During Training')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()


# Annote the plot with key information.
        min_val_loss = min(val_loss_history)
        min_val_loss_epoch = val_loss_history.index(min_val_loss) +1
        plt.annotate(f'Min Validation Loss: {min_val_loss:.4f}\nEpoch: {min_val_loss_epoch}',
                xy=(min_val_loss_epoch, min_val_loss), xycoords='data',
                xytext=(10,30), textcoords='offset points',
                arrowprops=dict(arrowstyle = '->'))


        plt.grid(True)
        plt.show()
```



* Scaling and Transformation Reminder

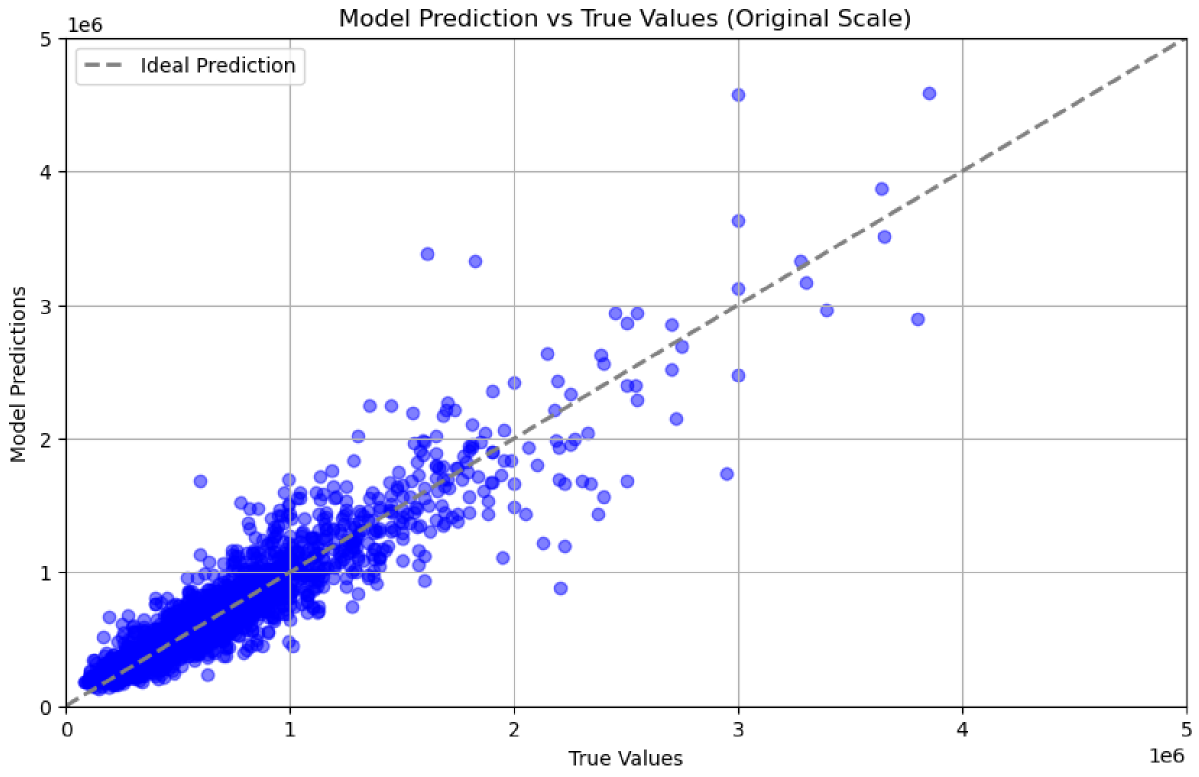
```
# Convert x_test to a PyTorch tensor and move it to the same device as the model (if needed)

        x_test_tensor = torch.tensor(x_test, dtype=torch.float32)


# Perform inference (model prediction)

        with torch.no_grad():

            y_predict = model(x_test_tensor).numpy()  # Convert prediction to numpy for plotting



# Inverse transform predictions and true values to original scale.

        y_predict_orig = scaler.inverse_transform(y_predict)

        y_test_orig = scaler.inverse_transform(y_test)


# Plot model predictions vs True values with echancements

        plt.figure(figsize=(10,6))
```

# Scatter plot of predictions vs true values.

```
plt.scatter(y_test_orig, y_predict_orig, color='blue', alpha=0.5)
plt.plot([0, 5000000], [0, 5000000], linestyle= '--', color='gray', linewidth=2, label = 'Ideal Prediction')
```

# Axis labels and limits

```
plt.title('Model Prediction vs True Values (Original Scale)')
plt.xlabel('True Values')
plt.ylabel('Model Predictions')
plt.xlim(0, 5000000)
plt.ylim(0, 5000000)
plt.legend()
plt.grid(True)
plt.show()
```



```
RMSE = np.sqrt(mean_squared_error(y_test_orig, y_predict_orig))
MSE = mean_squared_error(y_test_orig, y_predict_orig)
MAE = mean_absolute_error(y_test_orig, y_predict_orig)
r2 = r2_score(y_test_orig, y_predict_orig)
```

```
# Calculate Adjusted R-Squared
        n = len(y_test_orig)
        k = x_test.shape[1]
        adj_r2 = 1-(1-r2)*(n-1)/(n-k-1)

        print('RMSE: ', RMSE)
        print('MSE: ', MSE)
        print('MAE: ', MAE)
        print('R-Squared: ', r2)
        print('Adjusted R-Squared: ', adj_r2)
```