King County Housing Analysis

April 16, 2025

1 Predicting King County Housing Prices: A Data-Driven Approach to Feature Correlation Analysis and Machine Learning Model Optimization

1.1 Introduction

Housing prices in King County are influenced by a variety of factors, but which characteristics have the strongest impact, and do these trends differ across neighborhoods? This analysis seeks to answer the question: How do key housing features—such as square footage, renovation status, waterfront location, and proximity to urban centers—affect sale prices, and do these relationships vary by neighborhood? By examining these factors, we can uncover whether buyer preferences prioritize space, modern updates, scenic locations, or urban accessibility—and whether these priorities shift depending on the area.

This question is important because it goes beyond general price trends and explores what truly drives value in different parts of King County. Real estate agents, urban planners, and potential buyers would all benefit from understanding these patterns. For example, if renovated homes command a higher premium in suburban areas but not in downtown Seattle, this could reflect differing buyer expectations. Similarly, if waterfront properties consistently sell for more regardless of neighborhood, it may indicate a broader demand for luxury amenities. These insights could also inform discussions about housing affordability, development priorities, and economic disparities across the region.

To answer this question, we would analyze both quantitative and categorical variables. Key metrics would include sale price (the dependent variable), square footage, year built, and distance to major urban centers like Seattle (all quantitative). Categorical variables would include whether a home is waterfront (yes/no), has been renovated (yes/no), and its neighborhood classification. Geographic coordinates (latitude/longitude) would allow for spatial analysis to detect location-based trends.

Data Science Question How have housing market trends in King County evolved over time? What factors influence property values? Do these trends vary by location or property characteristics? What might these patterns indicate about shifting homeowner preferences or economic conditions?

1.1.1 Libraries

This code imports essential libraries for data analysis, geospatial visualization, and predictive modeling. Pandas and NumPy enable efficient data manipulation and statistical calculations, while Geopandas and Folium facilitate mapping and spatial analysis to explore geographic patterns in property locations. Machine learning tools like Scikit-learn will be used to build regression models, identify key predictors of housing prices (e.g., square footage, waterfront status, or renovation year), and quantify their impact. Together, these tools allow us to visualize price distributions across neighborhoods, analyze historical trends, and uncover the most influential factors driving King County's real estate market dynamics.

[1]:
```python
# ========== Standard Libraries ==========
import os
import json
import datetime
import joblib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.colors
import seaborn as sns
from tqdm import tqdm

# ========== Geospatial Libraries ==========
import folium
from folium import plugins
from folium.plugins import MarkerCluster
from geopy.exc import GeocoderTimedOut, GeocoderServiceError
from geopy.geocoders import Nominatim
from geopy.extra.rate_limiter import RateLimiter

# ========== Plotly for Interactive Visualizations ==========
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# ========== Scikit-Learn Libraries ==========
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split, RandomizedSearchCV,␣
  ↪GridSearchCV
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.metrics import (
    mean_absolute_error,
    mean_squared_error,
    r2_score,
    confusion_matrix,
    ConfusionMatrixDisplay,
    classification_report
)

# ========== Machine Learning Models ==========
# Linear Models
```

```
from sklearn.linear_model import LinearRegression

# Ensemble Models
from sklearn.ensemble import RandomForestRegressor, RandomForestClassifier

# Tree-Based Models
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier

# Distance-Based Models
from sklearn.neighbors import KNeighborsRegressor, KNeighborsClassifier

# Naive Bayes
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler

# Support Vector Machines
from sklearn.svm import SVR, SVC

# Neural Networks
from sklearn.neural_network import MLPRegressor, MLPClassifier
```

1.1.2 Variables

Let’s take a peek at the first few rows of the data frame

[2]:

```
# Read the CSV file
df = pd.read_csv('kc_house_data.csv')

# Display the first few rows of the DataFrame
df.head()
```

[2]:

```
           id             date     price  bedrooms  bathrooms  sqft_living  \
0  7129300520  20141013T000000  221900.0         3       1.00         1180
1  6414100192  20141209T000000  538000.0         3       2.25         2570
2  5631500400  20150225T000000  180000.0         2       1.00          770
3  2487200875  20141209T000000  604000.0         4       3.00         1960
4  1954400510  20150218T000000  510000.0         3       2.00         1680

   sqft_lot  floors  waterfront  view  …  grade  sqft_above  sqft_basement  \
0      5650     1.0           0     0  …      7        1180              0
1      7242     2.0           0     0  …      7        2170            400
2     10000     1.0           0     0  …      6         770              0
3      5000     1.0           0     0  …      7        1050            910
4      8080     1.0           0     0  …      8        1680              0

   yr_built  yr_renovated  zipcode      lat     long  sqft_living15  \
0      1955             0    98178  47.5112 -122.257           1340
```

```
1      1951           1991    98125  47.7210 -122.319            1690
2      1933              0    98028  47.7379 -122.233            2720
3      1965              0    98136  47.5208 -122.393            1360
4      1987              0    98074  47.6168 -122.045            1800

   sqft_lot15
0        5650
1        7639
2        8062
3        5000
4        7503

[5 rows x 21 columns]
```

We notice that there is 21 different features (columns). The id, date, price, bedrooms, bathrooms, sqft_living, sqft_lot, floors, waterfront, view, condition, grade, sqft_above, sqft_basement, yr_built, yr_renovated, zipcode, lat, long, sqft_living15, sqft_lot15.

Columns: - ida: notation for a house - date: The date the house was sold (ranges from May 2014 to May 2015) - price: Price is prediction target - bedrooms: Number of Bedrooms in the house - bathrooms: Number of bathrooms in the house - sqft_living: square footage of the home - sqft_lot: square footage of the lot - floors: Total floors (levels) in the house - waterfront: Determines if the house has a view to the waterfront; in other words is close tp water (Determined by 0 and 1 as 0 being no and 1 being yes) - view: Has been viewed (Determined between 0 and 4) - condition: How good the condition is overall (Determined between 0 and 4) - grade: The overall grade given to the housing unit (based on King County grading system) - sqft_abovesquare: footage of house subtracting the basement - sqft_basement: the square footage of the basement of the house (if the vaklue is 0 then there is no basement in the house) - yr_built: The year the house was built - yr_renovated: The year when the house was renovated - zipcode: zip - lat: Latitude coordinate - long: Longitude coordinate - sqft_living15: square footage of the home in 2015(implies– some renovations) - sqft_lot15: square footage of the lot in 2015(implies– some renovations)

1.1.3 Data Collection

The analysis covers over 20000 addresses spanning from 1900 to 2015 that were sold during May 2014-May 2015 in King County, USA

Dataset was pulled from https://www.kaggle.com/datasets/soylevbeytullah/house-prices-dataset.

[10]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21613 non-null  int64
 1   date           21613 non-null  object
 2   price          21613 non-null  float64
 3   bedrooms       21613 non-null  int64
 4   bathrooms      21613 non-null  float64
 5   sqft_living    21613 non-null  int64
 6   sqft_lot       21613 non-null  int64
 7   floors         21613 non-null  float64
 8   waterfront     21613 non-null  int64
 9   view           21613 non-null  int64
 10  condition      21613 non-null  int64
 11  grade          21613 non-null  int64
 12  sqft_above     21613 non-null  int64
 13  sqft_basement  21613 non-null  int64
 14  yr_built       21613 non-null  int64
 15  yr_renovated   21613 non-null  int64
 16  zipcode        21613 non-null  int64
 17  lat            21613 non-null  float64
 18  long           21613 non-null  float64
 19  sqft_living15  21613 non-null  int64
 20  sqft_lot15     21613 non-null  int64
dtypes: float64(5), int64(15), object(1)
memory usage: 3.5+ MB
```

[11]:

```python
# Get summary statistics
df.describe().round(2)
```

[11]:

```
                 id       price  bedrooms  bathrooms  sqft_living     sqft_lot  \
count  2.161300e+04    21613.00  21613.00   21613.00     21613.00     21613.00
mean   4.580302e+09   540088.14      3.37       2.11      2079.90     15106.97
std    2.876566e+09   367127.20      0.93       0.77       918.44     41420.51
min    1.000102e+06    75000.00      0.00       0.00       290.00       520.00
25%    2.123049e+09   321950.00      3.00       1.75      1427.00      5040.00
50%    3.904930e+09   450000.00      3.00       2.25      1910.00      7618.00
75%    7.308900e+09   645000.00      4.00       2.50      2550.00     10688.00
max    9.900000e+09  7700000.00     33.00       8.00     13540.00   1651359.00

         floors  waterfront      view  condition     grade  sqft_above  \
count  21613.00    21613.00  21613.00   21613.00  21613.00    21613.00
mean       1.49        0.01      0.23       3.41      7.66     1788.39
std        0.54        0.09      0.77       0.65      1.18      828.09
min        1.00        0.00      0.00       1.00      1.00      290.00
25%        1.00        0.00      0.00       3.00      7.00     1190.00
50%        1.50        0.00      0.00       3.00      7.00     1560.00
75%        2.00        0.00      0.00       4.00      8.00     2210.00
max        3.50        1.00      4.00       5.00     13.00     9410.00

       sqft_basement  yr_built  yr_renovated   zipcode       lat      long  \
count       21613.00  21613.00      21613.00  21613.00  21613.00  21613.00
mean          291.51   1971.01         84.40  98077.94     47.56   -122.21
std           442.58     29.37        401.68     53.51      0.14      0.14
```

```
min          0.00   1900.00          0.00  98001.00     47.16   -122.52
25%          0.00   1951.00          0.00  98033.00     47.47   -122.33
50%          0.00   1975.00          0.00  98065.00     47.57   -122.23
75%        560.00   1997.00          0.00  98118.00     47.68   -122.12
max       4820.00   2015.00       2015.00  98199.00     47.78   -121.32

       sqft_living15  sqft_lot15
count       21613.00    21613.00
mean         1986.55    12768.46
std           685.39    27304.18
min           399.00      651.00
25%          1490.00     5100.00
50%          1840.00     7620.00
75%          2360.00    10083.00
max          6210.00   871200.00
```

[12]:

```
df.columns
```

[12]:

```
Index(['id', 'date', 'price', 'bedrooms', 'bathrooms', 'sqft_living',
       'sqft_lot', 'floors', 'waterfront', 'view', 'condition', 'grade',
       'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode',
       'lat', 'long', 'sqft_living15', 'sqft_lot15'],
      dtype='object')
```

[13]:

```
# How many columns and rows in the dataset
df.shape
```

[13]:

```
(21613, 21)
```

[14]:

```
# Total number of cells in the dataset
df.size
```

[14]:

```
453873
```

[15]:

```
# Checking to see if there any duplicated values
df.duplicated().sum()
```

[15]:

```
0
```

[16]:

```
# Checking to see if any data is null/empty
df.isnull().sum()
```

[16]:

```
id                 0
date               0
price              0
bedrooms           0
bathrooms          0
sqft_living        0
```

```
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

1.2 Data Cleaning

1.2.1 Transformation

Once I had gotten the basic information of the dataset, there were columns I needed to add and fix.

Firstly, I will add a description column that essentially groups all information of the property into one paragraph. Variable will be called home_description

[17]:

```
# Function to create a house description
def create_property_description(row):
    description = f"This {row['bedrooms']} bedroom, {row['bathrooms']} bathroom␣
 ↪home features "
    description += f"{row['sqft_living']:,} sqft of living space on a␣
 ↪{row['sqft_lot']:,} sqft lot. "
    description += f"The property has {row['floors']} floor(s) and includes "

    # Waterfront information
    if row['waterfront'] == 1:
        description += "a waterfront view, "

    # View quality
    view_descriptions = {
        0: "no special views",
        1: "a limited view",
        2: "a decent view",
        3: "a good view",
        4: "an excellent view"
    }
    description += f"{view_descriptions.get(row['view'], 'a view')} and is in "
    # Condition description
    condition_descriptions = {
        1: "poor",
        2: "fair",
        3: "average",
        4: "good",
        5: "excellent"
    }
    description += f"{condition_descriptions.get(row['condition'], 'unknown')}␣
 ↪condition. "

    # Square footage details
    description += f"It has {row['sqft_above']:,} sqft above ground. "
    if row['sqft_basement'] > 0:
        description += f"and {row['sqft_basement']:,} sqft of basement space. "

    # Year and renovation information
    description += f"Built in {row['yr_built']}"
    if row['yr_renovated'] > 0:
        description += f", renovated in {row['yr_renovated']}. "
        description += (f"In 2015, the home had a living area of␣
 ↪{row['sqft_living15']:,} sqft and "
                        f"a lot size of {row['sqft_lot15']:,} sqft, compared to␣
 ↪the current "
                        f"{row['sqft_living']:,} sqft living space and␣
 ↪{row['sqft_lot']:,} sqft lot. ")
    else:
        description += ". "

    # Overall grade information
    description += f"It has an overall grade of {row['grade']}, according to␣
 ↪the King County grading system."

    return description

# Assuming your DataFrame is named 'df'
df['home_description'] = df.apply(create_property_description, axis=1)
```

Example property description

[18]:
```
# Display a sample description from the first
df['home_description'].iloc[0]
```

[18]:
```
'This 3 bedroom, 1.0 bathroom home features 1,180 sqft of living space on a
5,650 sqft lot. The property has 1.0 floor(s) and includes no special views and
is in average condition. It has 1,180 sqft above ground. Built in 1955. It has
```

```
an overall grade of 7, according to the King County grading system.'
```

Then I fixed the date column as the date looked in this format “20141013T000000”, the T000000 at the end of the date is not needed so I removed it. Also I formatted the date as well.

```
[20]: # Assuming your DataFrame is named 'df' and the column with the date is named␣
        ↪'date'
      df['date'] = df['date'].str.replace('T000000', '')  # Remove 'T000000'
      df['date'] = pd.to_datetime(df['date'], format='%Y%m%d')  # Convert to datetime␣
        ↪format
      df['date'] = df['date'].dt.strftime('%Y/%m/%d')  # Format as 'yyyy/mm/dd'

      df.head()
```

```
[20]:            id        date     price  bedrooms  bathrooms  sqft_living  \
      0  7129300520  2014/10/13  221900.0         3       1.00         1180
      1  6414100192  2014/12/09  538000.0         3       2.25         2570
      2  5631500400  2015/02/25  180000.0         2       1.00          770
      3  2487200875  2014/12/09  604000.0         4       3.00         1960
      4  1954400510  2015/02/18  510000.0         3       2.00         1680

         sqft_lot  floors  waterfront  view  …  sqft_above  sqft_basement  \
      0      5650     1.0           0     0  …        1180              0
      1      7242     2.0           0     0  …        2170            400
      2     10000     1.0           0     0  …         770              0
      3      5000     1.0           0     0  …        1050            910
      4      8080     1.0           0     0  …        1680              0

         yr_built  yr_renovated  zipcode      lat     long  sqft_living15  \
      0      1955             0    98178  47.5112 -122.257           1340
      1      1951          1991    98125  47.7210 -122.319           1690
      2      1933             0    98028  47.7379 -122.233           2720
      3      1965             0    98136  47.5208 -122.393           1360
      4      1987             0    98074  47.6168 -122.045           1800

         sqft_lot15                                   home_description
      0        5650  This 3 bedroom, 1.0 bathroom home features 1,1…
      1        7639  This 3 bedroom, 2.25 bathroom home features 2,…
      2        8062  This 2 bedroom, 1.0 bathroom home features 770…
      3        5000  This 4 bedroom, 3.0 bathroom home features 1,9…
      4        7503  This 3 bedroom, 2.0 bathroom home features 1,6…

      [5 rows x 22 columns]
```

1.3 Visualizations

This code creates the color map based on the price of the propterty

[23]:

```
# Define the create_color_map function
def create_color_map(color_coords, color_bounds):
    def to_cmap_coord(x, level=0.0):
        return (level, np.interp(x, xp=[0, 255], fp=[0, 1]), np.interp(x,␣
 ↪xp=[0, 255], fp=[0, 1]))

    cmap_price_bounds = [np.interp(p, xp=[min(color_bounds),␣
 ↪max(color_bounds)], fp=[0, 1]) for p in color_bounds]

    c_dict = {
        'red': tuple(to_cmap_coord(color_coords[i][0], cmap_price_bounds[i])␣
 ↪for i in range(len(color_coords))),
        'green': tuple(to_cmap_coord(color_coords[i][1], cmap_price_bounds[i])␣
 ↪for i in range(len(color_coords))),
        'blue': tuple(to_cmap_coord(color_coords[i][2], cmap_price_bounds[i])␣
 ↪for i in range(len(color_coords))),
    }

    return matplotlib.colors.LinearSegmentedColormap('cmap', segmentdata=c_dict)
```

[24]:

```
# Color Boundaries (in $s)
cmap_price_bounds = [0, 800000, 1500000, 10000000]
# Color Definitions (in RGB)
color_coords = [
    (47, 216, 58),     # Green ($0 - $800,000)
    (215, 237, 23),    # Yellow ($800,000 - $1,500,000)
    (239, 32, 21),     # Red ($1,500,000 - $10,000,000)
    (117, 11, 5)       # Darker Red (> $10,000,000)
]
```

Finally, I used lat and long to find the addresses which was the new column as well to add. Variable will be called address.

This is where I find the addresses of every location based on the lat and long. Then the address found will be placed in the table.

Originally I had it that this ran everytime I clicked it, issue was due to the large size of the dataset, it took 8 hours to get all addresses, with improvements I instead added the addresses found in a cache file so its faster to get and retrieve them without doing the whole process again.

[25]:

```
# Initialize Nominatim API
geolocator = Nominatim(user_agent="real_estate_map")

# Add rate limiting to avoid hitting API limits
reverse = RateLimiter(geolocator.reverse, min_delay_seconds=1)

# Define a function to get the address from latitude and longitude
def get_address(lat, lon):
    location = reverse((lat, lon), exactly_one=True)
    return location.address if location else "Address not found"

# Cache file path
cache_file = "address_cache.json"

# Load existing cache if it exists
if os.path.exists(cache_file):
    with open(cache_file, "r") as f:
        address_cache = json.load(f)
else:
    address_cache = {}

# Function to fetch address with caching
def get_cached_address(lat, lon):
    # Create a unique key for the coordinates
    key = f"{lat},{lon}"

    # Check if the address is already in the cache
    if key in address_cache:
        return address_cache[key]

    # If not in cache, fetch the address and save it
    address = get_address(lat, lon)
    address_cache[key] = address

    # Save the updated cache to the file
    with open(cache_file, "w") as f:
        json.dump(address_cache, f)

    return address

# Use tqdm to show progress while fetching addresses
tqdm.pandas(desc="Fetching addresses")  # Initialize tqdm for pandas

# Add a new column 'address' to the original DataFrame df
df['address'] = df.progress_apply(lambda row: get_cached_address(row['lat'],␣
 ↪row['long']), axis=1)

# Display the first few rows of the updated DataFrame
df.head()
```

```
Fetching addresses: 100%|                             |
21613/21613 [00:00<00:00, 25805.50it/s]
```

[25]:

```
           id        date     price  bedrooms  bathrooms  sqft_living  \
0  7129300520  2014/10/13  221900.0         3       1.00         1180
```

```
1  6414100192  2014/12/09  538000.0         3       2.25         2570
2  5631500400  2015/02/25  180000.0         2       1.00          770
3  2487200875  2014/12/09  604000.0         4       3.00         1960
4  1954400510  2015/02/18  510000.0         3       2.00         1680

   sqft_lot  floors  waterfront  view  …  sqft_basement  yr_built  \
0      5650     1.0           0     0  …              0      1955
1      7242     2.0           0     0  …            400      1951
2     10000     1.0           0     0  …              0      1933
3      5000     1.0           0     0  …            910      1965
4      8080     1.0           0     0  …              0      1987

   yr_renovated  zipcode      lat     long  sqft_living15  sqft_lot15  \
0             0    98178  47.5112 -122.257           1340        5650
1          1991    98125  47.7210 -122.319           1690        7639
2             0    98028  47.7379 -122.233           2720        8062
3             0    98136  47.5208 -122.393           1360        5000
4             0    98074  47.6168 -122.045           1800        7503

                                    home_description  \
0  This 3 bedroom, 1.0 bathroom home features 1,1…
1  This 3 bedroom, 2.25 bathroom home features 2,…
2  This 2 bedroom, 1.0 bathroom home features 770…
3  This 4 bedroom, 3.0 bathroom home features 1,9…
4  This 3 bedroom, 2.0 bathroom home features 1,6…

                                             address
0  10012, 61st Avenue South, Rainier Beach, Seatt…
1  837, Northeast 127th Street, Northgate, Seattl…
2  15109, 81st Avenue Northeast, Moorlands, Kenmo…
3  9236, Fauntleroy Way Southwest, Fauntleroy, Se…
4  901, 221st Avenue Northeast, Sammamish, King C…

[5 rows x 23 columns]
```

[26]:

```
# Create the colormap
cmap = create_color_map(color_coords, cmap_price_bounds)

# Create the folium map instance
real_estate_map = folium.Map(location=[47.61038000, -122.20068000],␣
 ↪zoom_start=12)

# Define the cmap_func function
def cmap_func(row, cmap):
    r_interp = np.interp(row['price'], xp=[df['price'].min(), df['price'].
 ↪max()], fp=[5, 20])
```

```
    c_interp = np.interp(row['price'], xp=[df['price'].min(), df['price'].
 ↪max()], fp=[0, 1])
    o_interp = np.interp(row['price'], xp=[df['price'].min(), df['price'].
 ↪max()], fp=[0.2, 1.0])
    inner_color = matplotlib.colors.to_hex(list(cmap(c_interp))[:3] +␣
 ↪[o_interp])

    popup_text = folium.Popup(
    "Price: " + '${:,d}'.format(int(row['price'])) +
    "<br>Address: " + row['address'] +
    "<br>Description: " + row['home_description'],
    max_width=450
)


    folium.CircleMarker(location=[row['lat'], row['long']],
                        radius=r_interp, weight=0.9, color='black',␣
 ↪fill_color=inner_color,
                        popup=popup_text).add_to(real_estate_map)

# Apply the cmap_func to each row in the DataFrame
df.apply(lambda row: cmap_func(row, cmap), axis=1)

# Display the map
real_estate_map
```

[26]:

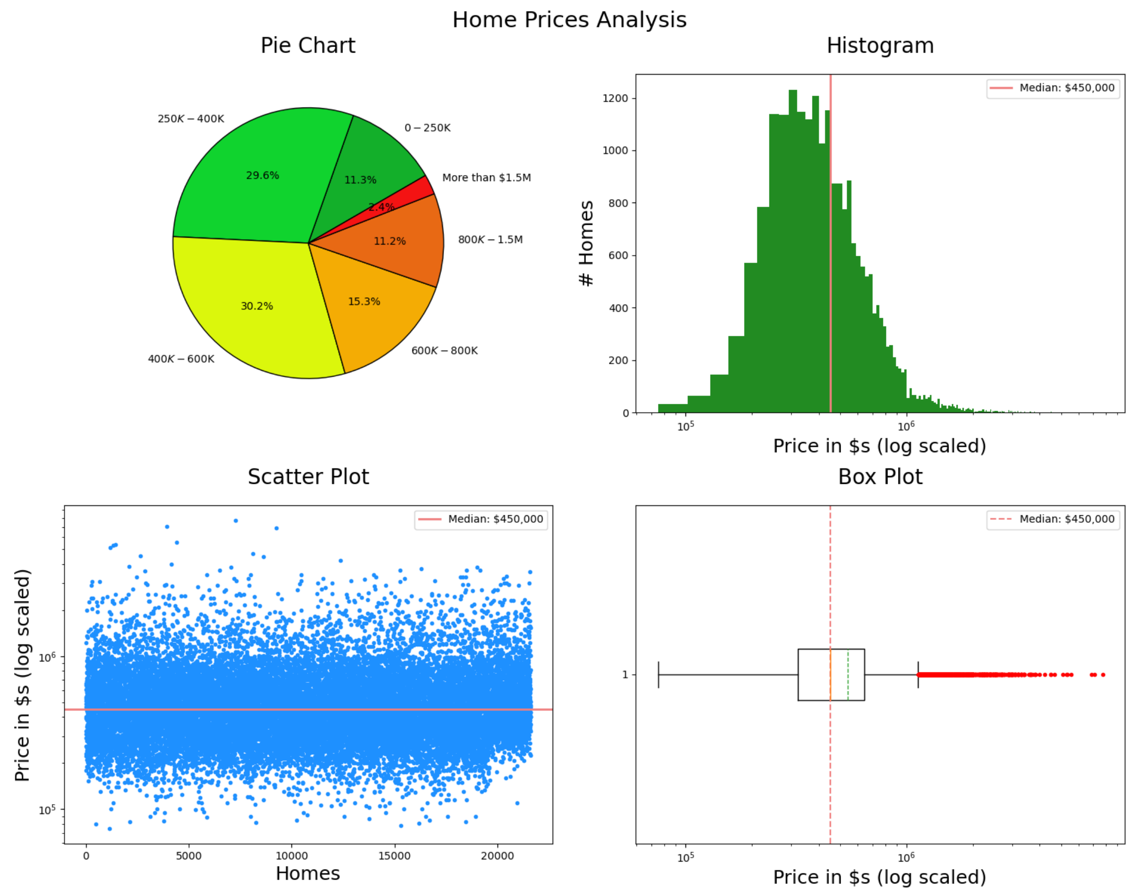
```
<folium.folium.Map at 0x247e7f43860>
```

[3]:

```
# Define price intervals and labels for the pie chart
labels = ['$0 - $250K', '$250K - $400K', '$400K - $600K', '$600K - $800K',␣
 ↪'$800K - $1.5M', 'More than $1.5M']
colors = ['#13af2a', '#10d32e', '#dbf70c', '#f4ac04', '#e86914', '#f41313']
intervals = [0, 250000, 400000, 600000, 800000, 1500000, df['price'].max()]  #␣
 ↪Price intervals in $s

# Calculate the size of each pie chart slice
chart_slice_sizes = df.groupby(pd.cut(df['price'], intervals)).size().values

# Calculate median price for annotation
median_price = df['price'].median()

# Create the parent figure
fig, ax = plt.subplots(2, 2)
fig.set_size_inches(15, 12)
fig.suptitle('Home Prices Analysis', size=21)

# Create pie chart
```

```
ax[0][0].set_title('Pie Chart', size=20, pad=20)
ax[0][0].pie(chart_slice_sizes, labels=labels, colors=colors, startangle=30,␣
 ↪autopct='%1.1f%%',
             wedgeprops={'edgecolor': 'black'})

# Create histogram
ax[0][1].set_title('Histogram', size=20, pad=20)
ax[0][1].hist(df['price'], bins=280, color='forestgreen')
ax[0][1].set_ylabel('# Homes', size=18)
ax[0][1].set_xlabel('Price in $s (log scaled)', size=18)
ax[0][1].set_xscale('log')
median_line = ax[0][1].axvline(median_price, color='lightcoral', label=f'Median:
 ↪ ${median_price:,.0f}', linewidth=2)
ax[0][1].legend()

# Create Scatterplot
ax[1][0].set_title('Scatter Plot', size=20, pad=20)
ax[1][0].scatter(range(len(df)), df['price'], marker='.', color='dodgerblue')
ax[1][0].set_ylabel('Price in $s (log scaled)', size=18)
ax[1][0].set_xlabel('Homes', size=18)
ax[1][0].set_yscale('log')
median_line = ax[1][0].axhline(median_price, color='lightcoral', label=f'Median:
 ↪ ${median_price:,.0f}', linewidth=2)
ax[1][0].legend()

# Create Boxplot
ax[1][1].set_title('Box Plot', size=20, pad=20)
ax[1][1].boxplot(df['price'], sym='r.', vert=False, showmeans=True,␣
 ↪meanline=True)
ax[1][1].set_xlabel('Price in $s (log scaled)', size=18)
ax[1][1].set_xscale('log')

# Add median price annotation to boxplot
ax[1][1].axvline(median_price, color='lightcoral', linestyle='--',␣
 ↪label=f'Median: ${median_price:,.0f}')
ax[1][1].legend()

# Adjust layout and display the plot
fig.tight_layout()
plt.show()
```

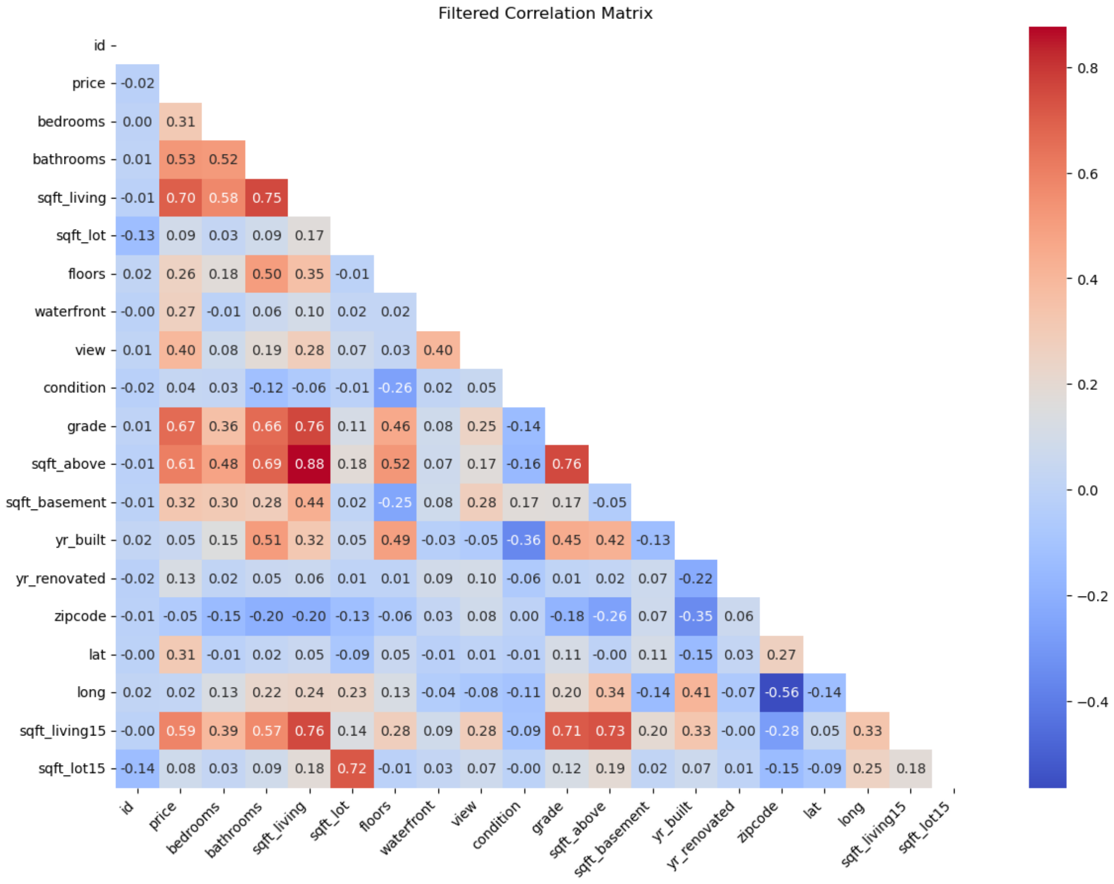
```
C:\Users\emman\AppData\Local\Temp\ipykernel_26808\1634433732.py:7:
FutureWarning: The default of observed=False is deprecated and will be changed
to True in a future version of pandas. Pass observed=False to retain current
behavior or observed=True to adopt the future default and silence this warning.
  chart_slice_sizes = df.groupby(pd.cut(df['price'], intervals)).size().values
```



[9]:

```
# Assuming 'data' is the DataFrame containing your dataset
def filter_correlations(corr_matrix, threshold=0.5):
    mask = np.abs(corr_matrix) >= threshold
    filtered_corr = corr_matrix.where(mask).fillna(0)
    return filtered_corr

def plot_heatmap(matrix, title, size=(14, 10)):
    mask = np.triu(np.ones_like(matrix, dtype=bool))  # Create a mask for the␣
 ↪upper triangle
    plt.figure(figsize=size)
    sns.heatmap(matrix, mask=mask, annot=True, cmap='coolwarm', fmt=".2f",␣
 ↪annot_kws={"size": 10})
    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=0)
    plt.title(title)
    plt.show()

# Select only numeric columns from the DataFrame
numeric_cols = df.select_dtypes(include=['number'])
```

```
# Compute the correlation matrix
corr_matrix = numeric_cols.corr()

# Filter the correlation matrix to only show high correlations
filtered_corr_matrix = filter_correlations(corr_matrix, threshold=0)

# Plot heatmap with the upper triangle masked
plot_heatmap(filtered_corr_matrix, 'Filtered Correlation Matrix')
```

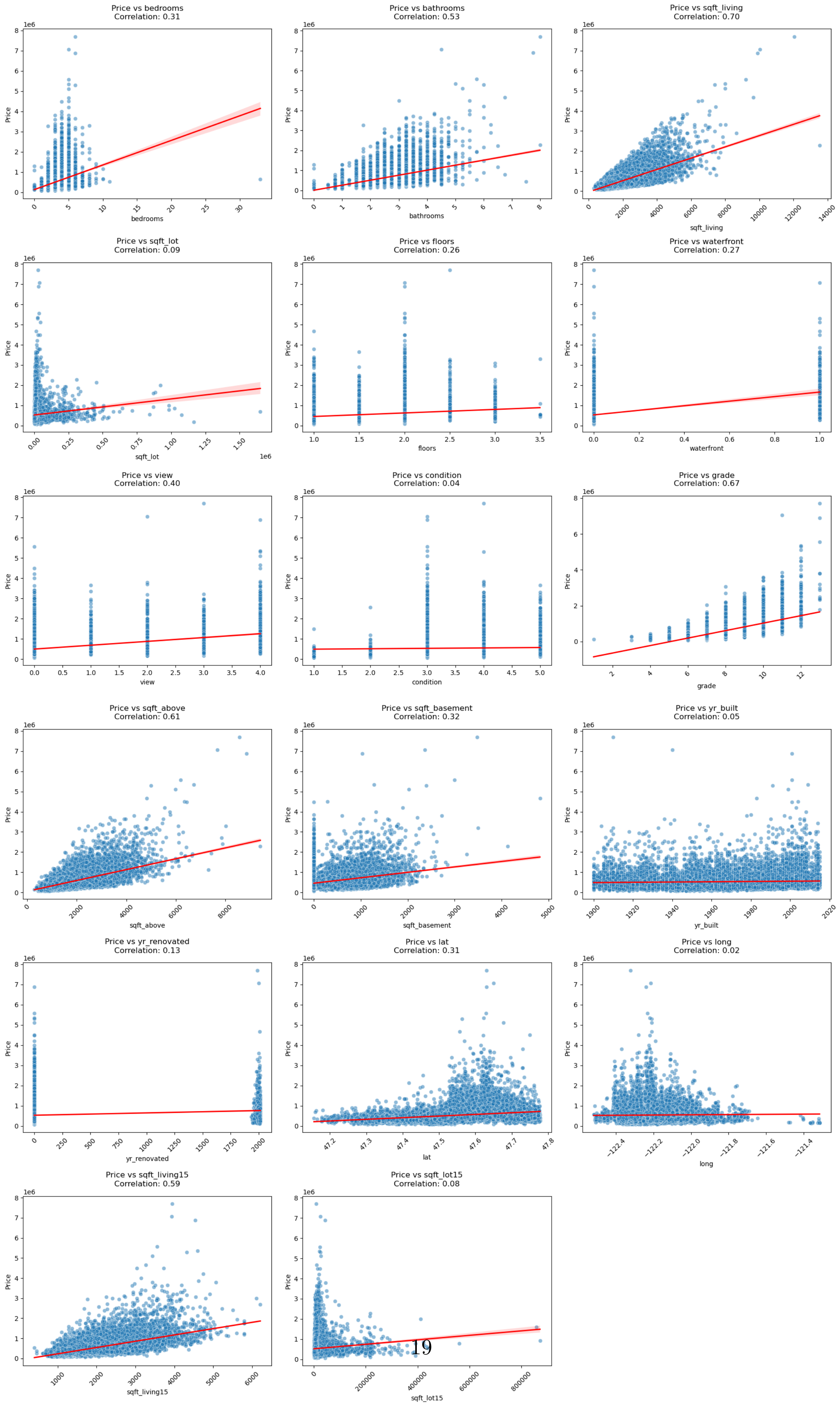


[10]:
```
# Extract correlations for "price"
price_corr = corr_matrix['price']

# Sort the correlations by absolute value (distance from 0) in descending order
# While keeping the original sign of the correlation
sorted_price_corr = price_corr.iloc[(-price_corr.abs()).argsort()]

# Format the output for better readability
print("Features sorted by strength of correlation with price (absolute value):")
print("------------------------------------------------------")
print(f"{'Feature':<25} {'Correlation':>10}")
print("-" * 36)
for feature, corr in sorted_price_corr.items():
    print(f"{feature:<25} {corr:>10.3f}")
```

```
Features sorted by strength of correlation with price (absolute value):
------------------------------------------------------
Feature                   Correlation
------------------------------------
price                          1.000
sqft_living                    0.702
grade                          0.667
sqft_above                     0.606
sqft_living15                  0.585
bathrooms                      0.525
view                           0.397
sqft_basement                  0.324
bedrooms                       0.308
lat                            0.307
waterfront                     0.266
floors                         0.257
yr_renovated                   0.126
sqft_lot                       0.090
sqft_lot15                     0.082
yr_built                       0.054
zipcode                       -0.053
condition                      0.036
long                           0.022
id                            -0.017
```

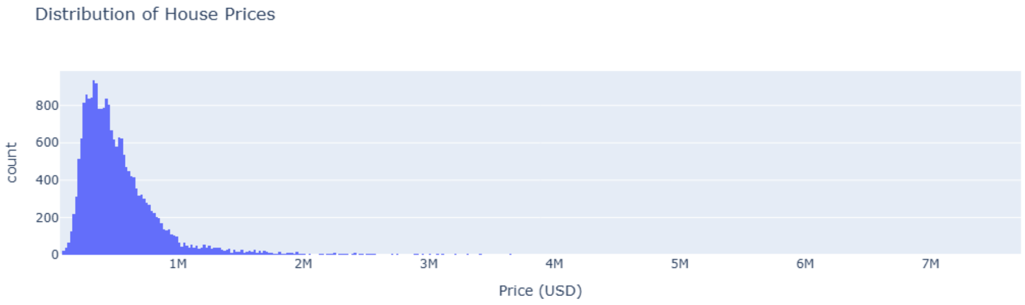
[4]:

```
def plot_scatter_subplots(df):
    numerical_features = ['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot',␣
 ↪'floors',
                          'waterfront', 'view', 'condition', 'grade',␣
 ↪'sqft_above',
                          'sqft_basement', 'yr_built', 'yr_renovated', 'lat',␣
 ↪'long',
                          'sqft_living15', 'sqft_lot15']

    num_plots = len(numerical_features)
    rows = (num_plots // 3) + 1
    fig, axes = plt.subplots(rows, 3, figsize=(18, 5 * rows))  # Slightly wider␣
 ↪figure
    axes = axes.flatten()

    for i, feature in enumerate(numerical_features):
        # Create scatter plot
        sns.scatterplot(x=df[feature], y=df['price'], alpha=0.5, ax=axes[i])

        # Add regression line
        sns.regplot(x=df[feature], y=df['price'], scatter=False,
                    color='red', line_kws={'linewidth': 2}, ax=axes[i])

        # Calculate correlation coefficient
        corr = df[feature].corr(df['price'])

        # Format title with correlation
        axes[i].set_title(f'Price vs {feature}\nCorrelation: {corr:.2f}',␣
 ↪pad=15)
        axes[i].set_xlabel(feature)
        axes[i].set_ylabel('Price')

        # Rotate x-axis labels if needed for better readability
        if len(df[feature].unique()) > 10:  # For features with many unique␣
 ↪values
            for tick in axes[i].get_xticklabels():
                tick.set_rotation(45)

    # Remove empty subplots
    for j in range(i + 1, len(axes)):
        fig.delaxes(axes[j])

    plt.tight_layout()
    plt.show()

# Call the function with the DataFrame
plot_scatter_subplots(df)
```

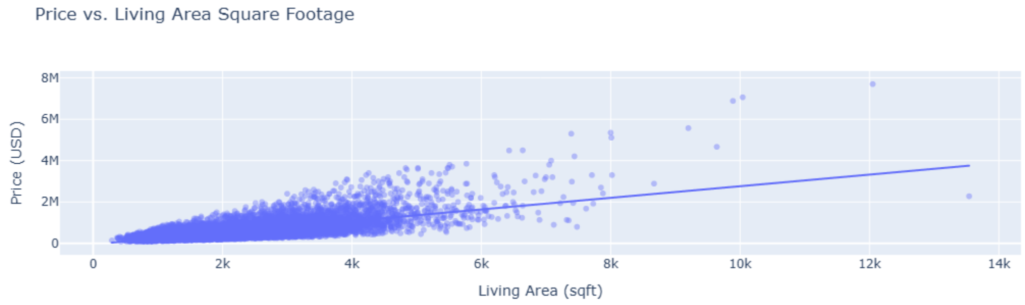

Price vs bedrooms
Correlation: 0.31
1e6
Price
bedrooms
0
5
10
15
20
25
30
Price vs bathrooms
Correlation: 0.53
1e6
Price
bathrooms
Price vs sqft_living
Correlation: 0.70
1e6
Price
sqft_living
0
2000
4000
6000
8000
10000
12000
14000
Price vs sqft_lot
Correlation: 0.09
1e6
Price
sqft_lot
1e6
0.00
0.25
0.50
0.75
1.00
1.25
1.50
Price vs floors
Correlation: 0.26
1e6
Price
floors
1.0
1.5
2.0
2.5
3.0
3.5
Price vs waterfront
Correlation: 0.27
1e6
Price
waterfront
0.0
0.2
0.4
0.6
0.8
1.0
Price vs view
Correlation: 0.40
1e6
Price
view
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Price vs condition
Correlation: 0.04
1e6
Price
condition
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Price vs grade
Correlation: 0.67
1e6
Price
grade
2
4
6
8
10
12
Price vs sqft_above
Correlation: 0.61
1e6
Price
sqft_above
0
2000
4000
6000
8000
Price vs sqft_basement
Correlation: 0.32
1e6
Price
sqft_basement
0
1000
2000
3000
4000
5000
Price vs yr_built
Correlation: 0.05
1e6
Price
yr_built
1900
1920
1940
1960
1980
2000
2020
Price vs yr_renovated
Correlation: 0.13
1e6
Price
yr_renovated
0
250
500
750
1000
1250
1500
1750
2000
Price vs lat
Correlation: 0.31
1e6
Price
lat
47.2
47.3
47.4
47.5
47.6
47.7
47.8
Price vs long
Correlation: 0.02
1e6
Price
long
−122.4
−122.2
−122.0
−121.8
−121.6
−121.4
Price vs sqft_living15
Correlation: 0.59
1e6
Price
sqft_living15
1000
2000
3000
4000
5000
6000
Price vs sqft_lot15
Correlation: 0.08
1e6
Price
sqft_lot15
0
200000
400000
600000
800000

1.3.1 Basic Visualizations

```
[11]: fig = px.histogram(df, x='price',
                         title='Distribution of House Prices',
                         labels={'price': 'Price (USD)'},
                         log_y=False)

      fig.show()
```

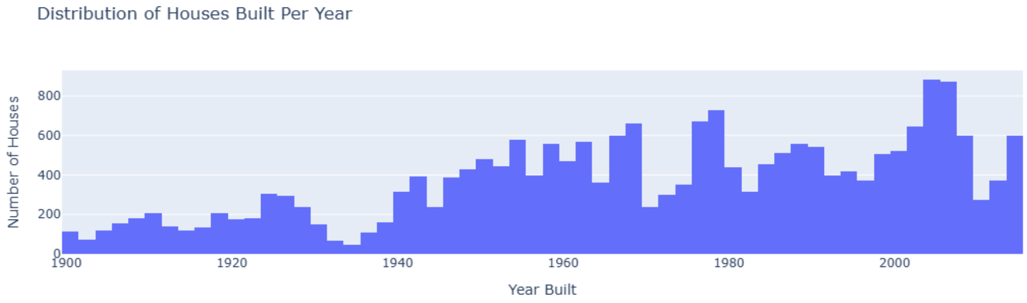


```
[15]: fig = px.scatter(df, x='sqft_living', y='price',
                       title='Price vs. Living Area Square Footage',
                       labels={'sqft_living': 'Living Area (sqft)', 'price': 'Price␣
        ↪(USD)'},
                       trendline='ols',
                       opacity=0.4)
      fig.show()
```



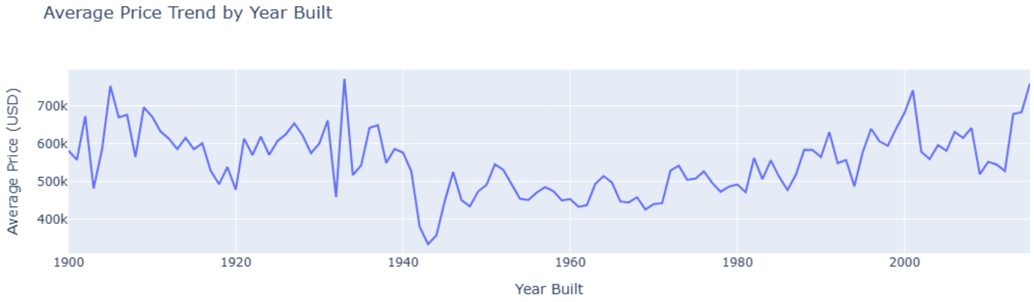
[22]:

```python
# Create the histogram
fig = px.histogram(df, x='yr_built',
                   title='Distribution of Houses Built Per Year',
                   labels={'yr_built': 'Year Built'})

# Customize axis labels
fig.update_layout(
    xaxis_title='Year Built',
    yaxis_title='Number of Houses'
)

# Show the figure
fig.show()
```

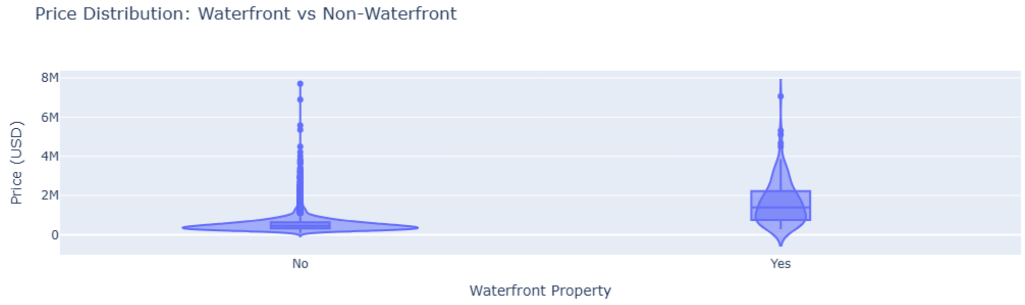


[21]:

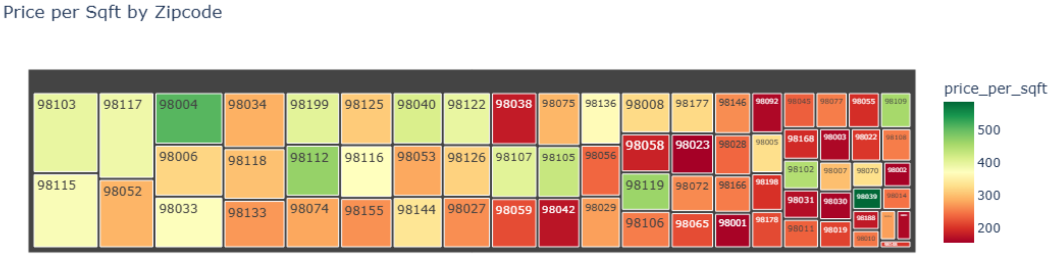
```python
year_price = df.groupby('yr_built')['price'].mean().reset_index()
fig = px.line(year_price, x='yr_built', y='price',
              title='Average Price Trend by Year Built',
              labels={'yr_built': 'Year Built', 'price': 'Average Price (USD)'})
fig.show()
```



[26]:
```
fig = px.violin(df, x='waterfront', y='price',
                title='Price Distribution: Waterfront vs Non-Waterfront',
                labels={'waterfront': 'Waterfront Property', 'price': 'Price␣
 ↪(USD)'},
                box=True)
fig.update_layout(xaxis=dict(tickvals=[0, 1], ticktext=['No', 'Yes']))
fig.show()
```

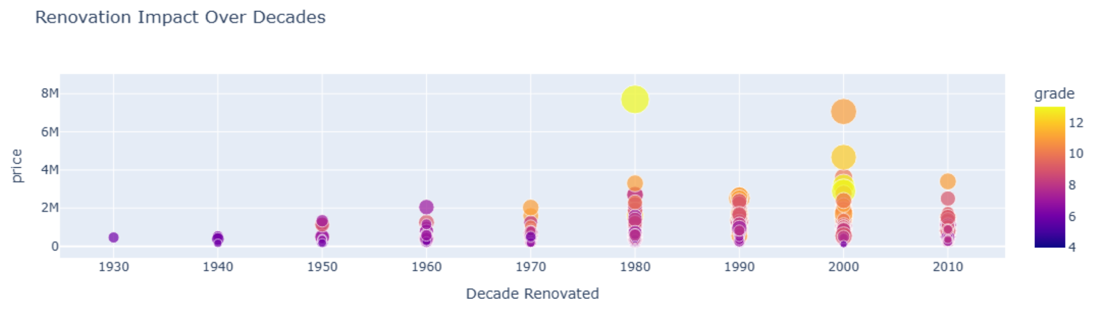


[27]:
```
df['price_per_sqft'] = df['price'] / df['sqft_living']
fig = px.treemap(df, path=['zipcode'], values='price_per_sqft',
                 color='price_per_sqft', color_continuous_scale='RdYlGn',
                 title='Price per Sqft by Zipcode')
fig.show()
```



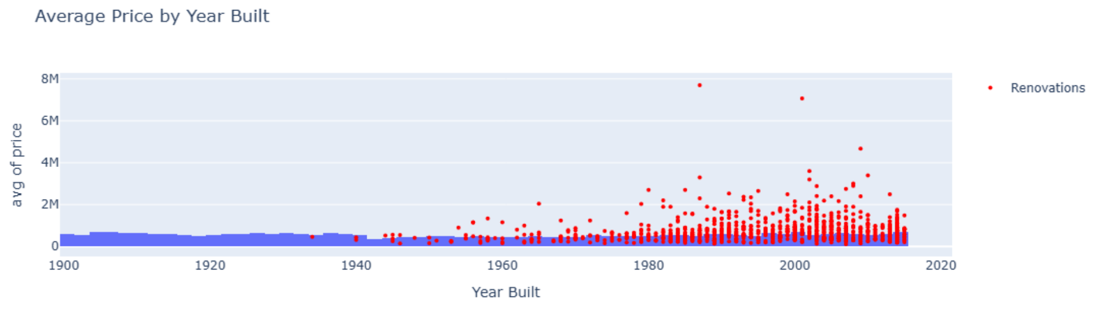
[28]:
```
renovated = df[df['yr_renovated'] > 0].copy()
renovated['decade_renovated'] = (renovated['yr_renovated'] // 10) * 10
fig = px.scatter(renovated, x='decade_renovated', y='price',
                 size='sqft_living', color='grade',
                 title='Renovation Impact Over Decades',
                 labels={'decade_renovated': 'Decade Renovated'})
```

```
fig.show()
```



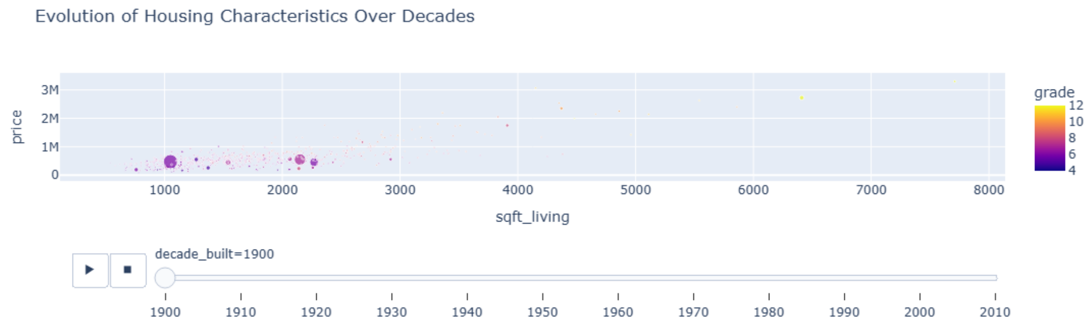
[30]:

```
fig = px.histogram(df, x='yr_built', y='price',
                   histfunc='avg',
                   title='Average Price by Year Built',
                   labels={'yr_built': 'Year Built'})
fig.add_scatter(x=renovated['yr_renovated'], y=renovated['price'],
                 mode='markers', name='Renovations',
                 marker=dict(color='red', size=4))
fig.show()
```



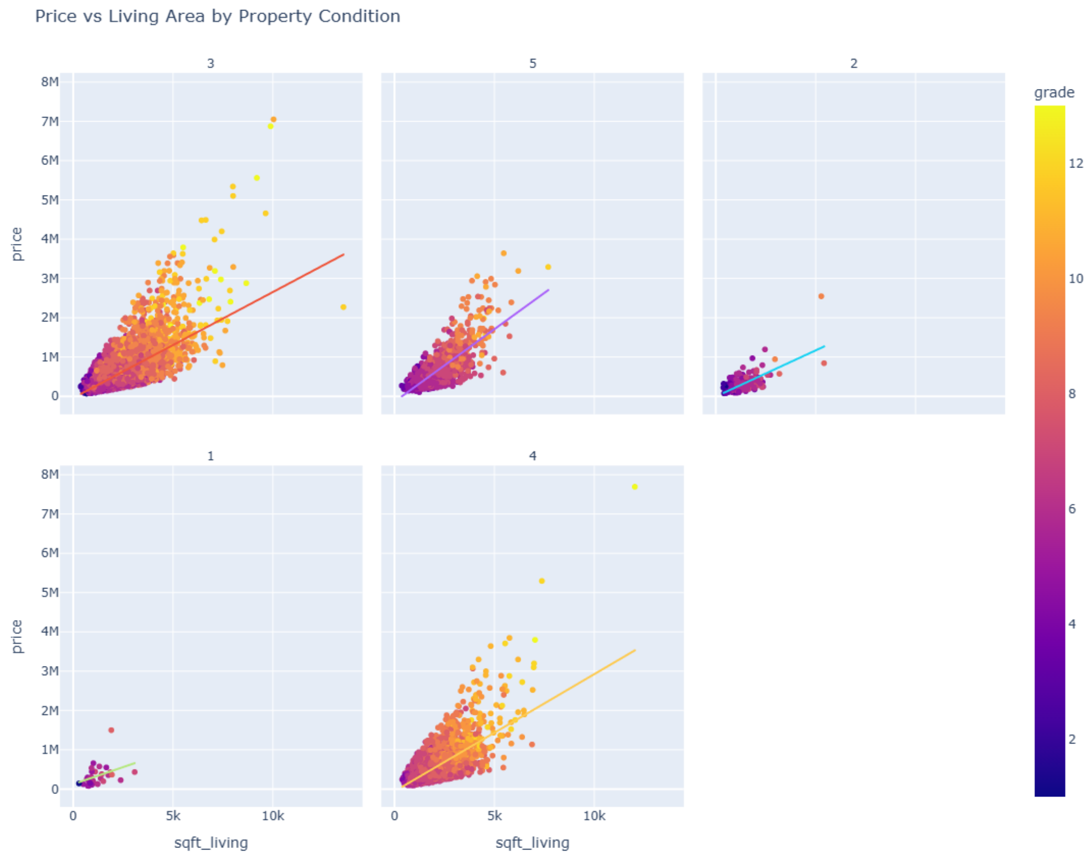
[39]:

```
# Sort the DataFrame by 'decade_built' to ensure proper order
df['decade_built'] = (df['yr_built'] // 10) * 10
df = df.sort_values(by='decade_built')

# Create the scatter plot
fig = px.scatter(df, x='sqft_living', y='price',
                  animation_frame='decade_built',
                  size='sqft_lot', color='grade',
                  range_x=[0, 10000], range_y=[0, 4e6],
                  title='Evolution of Housing Characteristics Over Decades')

# Adjust animation speed
fig.layout.updatemenus[0].buttons[0].args[1]['frame']['duration'] = 1000

# Show the figure
fig.show()
```



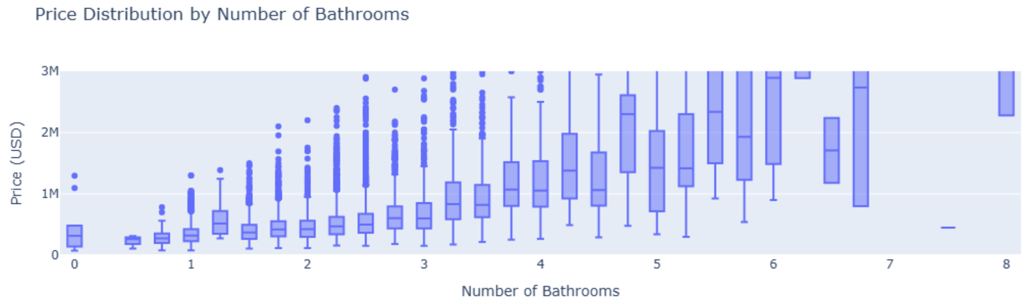
[42]:

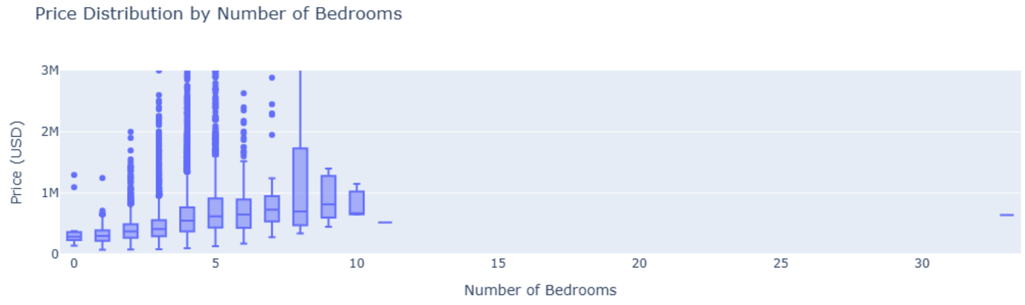
```
fig = px.scatter(df, x='sqft_living', y='price',
                  facet_col='condition', facet_col_wrap=3,
                  color='grade', trendline='ols',
                  title='Price vs Living Area by Property Condition',
                  height=900)
fig.for_each_annotation(lambda a: a.update(text=a.text.split("=")[1]))
fig.show()
```



[18]:

```
# Chart 1: Bathrooms vs. Price
fig = px.box(df, x='bathrooms', y='price',
              title='Price Distribution by Number of Bathrooms',
              labels={'bathrooms': 'Number of Bathrooms', 'price': 'Price␣
 ↪(USD)'})
fig.update_layout(yaxis_range=[0, 3e6])
fig.show()

# Chart 2: Bedrooms vs. Price
fig = px.box(df, x='bedrooms', y='price',
              title='Price Distribution by Number of Bedrooms',
              labels={'bedrooms': 'Number of Bedrooms', 'price': 'Price (USD)'})
fig.update_layout(yaxis_range=[0, 3e6])
fig.show()
```





1.3.2 Machine Learning

We will use the variables to predict the prices of properties and see the accuracy of different ML models (learned from Intro to Data Science).

We will use four different models: - Linear Regression - Random Forest Regressor - K-Nearest Neighbors Regressor - Naive Bayes Regressor

[6]:
```
# Assuming df is your DataFrame
X = df[['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',␣
 ↪'waterfront',
        'view', 'condition', 'grade', 'sqft_above', 'sqft_basement', 'yr_built',
        'lat', 'long', 'sqft_living15', 'sqft_lot15']]
y = df['price']

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

# Initialize selected models
models = {
    'Linear Regression': make_pipeline(StandardScaler(), LinearRegression()),
    'Random Forest Regressor': RandomForestRegressor(n_estimators=100,␣
 ↪random_state=42),
    'K-Nearest Neighbors Regressor': KNeighborsRegressor(n_neighbors=5),
    'Naive Bayes Regressor': GaussianNB()  # Note: GaussianNB is actually a␣
 ↪classifier, not a regressor
}

# Dictionary to store results
results = {}

# Train each model, make predictions, and evaluate performance
for model_name, model in models.items():
    # Train the model
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate the model
    mae = mean_absolute_error(y_test, y_pred)
    mse = mean_squared_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    # Store results
    results[model_name] = {'MAE': mae, 'MSE': mse, 'R2': r2}

# Convert results into a DataFrame
results_df = pd.DataFrame(results).T

# Display the comparison of models
results_df
```

[6]:

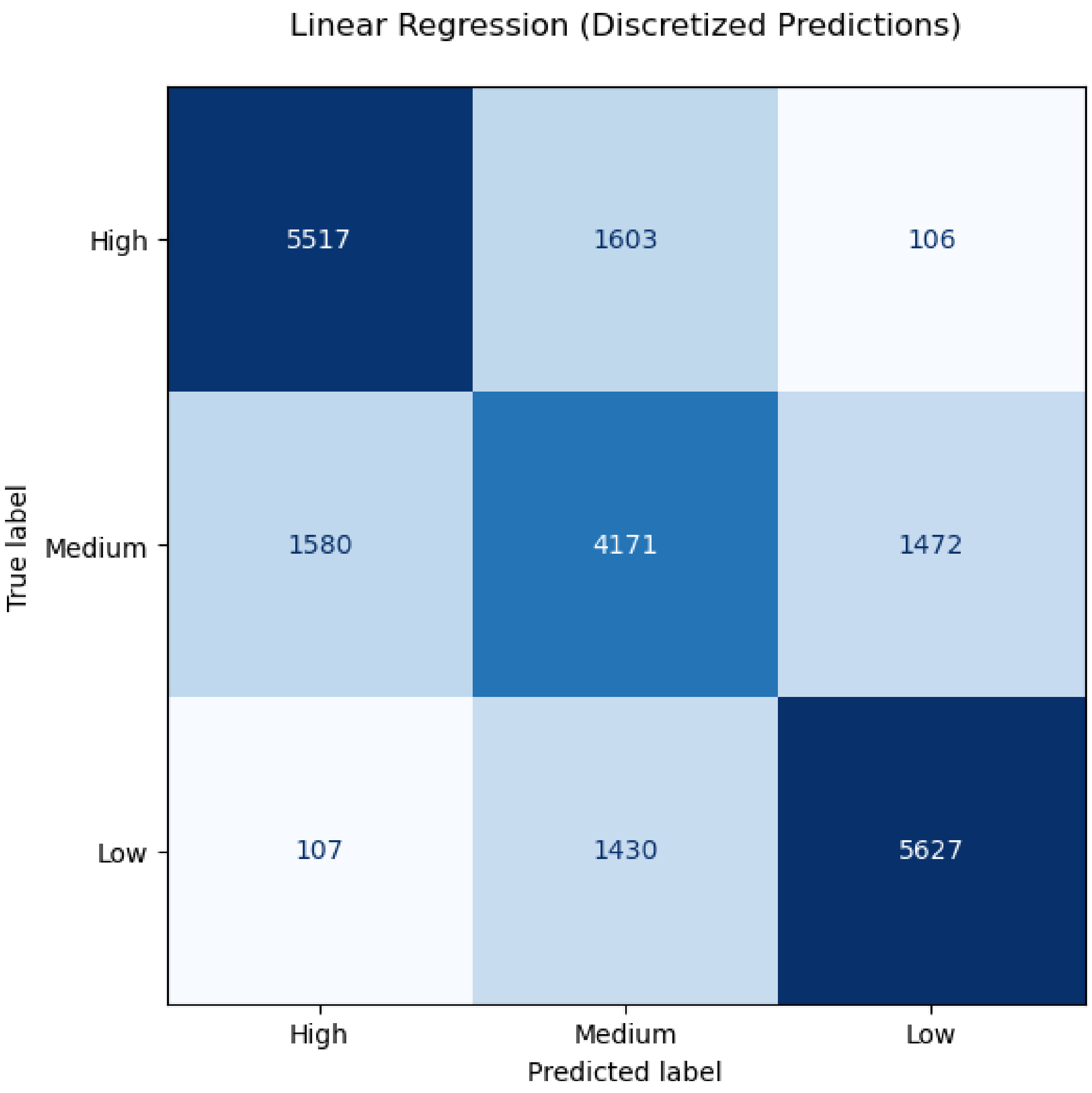
```
                                         MAE           MSE        R2
Linear Regression              128287.670990  4.605589e+10  0.695351
Random Forest Regressor         73451.395158  2.256044e+10  0.850768
K-Nearest Neighbors Regressor  166014.793569  7.466949e+10  0.506078
Naive Bayes Regressor          183768.454777  8.297760e+10  0.451122
```

- Mean Absolute Error (MAE): Measures the average absolute difference between predicted and actual values. It gives an idea of how far off predictions are, without considering direction.
- Mean Squared Error (MSE): Similar to MAE but squares the errors before averaging them. This penalizes larger errors more, making it useful when you want to emphasize big mistakes.
- R-Squared (R^2): Also known as the coefficient of determination, it indicates how well the model explains the variance in the data. A value closer to 1 means better predictive power.

[11]:

```
# Train Linear Regression model
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)  # y_train is original continuous values

# Get continuous predictions on full dataset
y_full_cont_pred = lin_reg.predict(X)

# Discretize the predictions using the same quantile bins as the original
y_full_pred_cat = pd.qcut(y_full_cont_pred, q=3, labels=class_labels)

# Generate confusion matrix
cm = confusion_matrix(y_full_cat, y_full_pred_cat, labels=class_labels)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=class_labels)

# Plot
fig, ax = plt.subplots(figsize=(6, 6))
disp.plot(ax=ax, cmap='Blues', colorbar=False)
ax.set_title('Linear Regression (Discretized Predictions)', pad=20)

# Print classification report
print("\nClassification Report for Linear Regression (Full Dataset):")
print(classification_report(y_full_cat, y_full_pred_cat,␣
 ↪target_names=class_labels))

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

```
Classification Report for Linear Regression (Full Dataset):
              precision    recall  f1-score   support

        High       0.77      0.76      0.76      7226
      Medium       0.78      0.79      0.78      7164
         Low       0.58      0.58      0.58      7223

    accuracy                           0.71     21613
   macro avg       0.71      0.71      0.71     21613
weighted avg       0.71      0.71      0.71     21613
```



Let's take a deep dive into the classification report

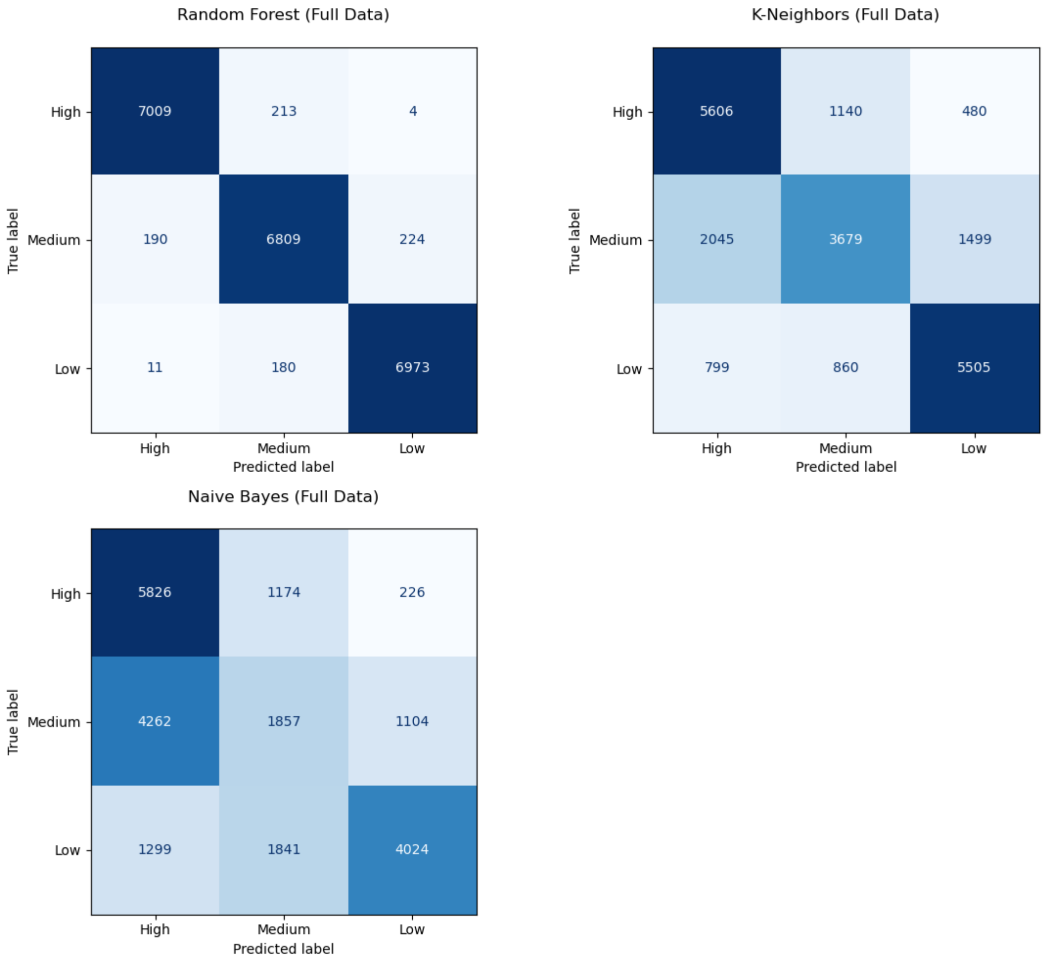
- Precision: The proportion of true positive predictions relative to all positive predictions made.
- Recall: The proportion of actual positive instances that were correctly identified.
- F1-Score: A harmonic mean of precision and recall, providing a single metric to evaluate the balance between them.
- Support: The count of actual instances in each class

[30]:
```
# Define class order (High, Medium, Low)
class_labels = ["High", "Medium", "Low"]

# Convert ALL prices to categories (not just train/test)
y_full_cat = pd.qcut(y, q=3, labels=class_labels)  # Apply to entire y
```

```
# Initialize models
class_models = {
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
    'K-Neighbors': KNeighborsClassifier(n_neighbors=5),
    'Naive Bayes': GaussianNB()
}

# Create 2x2 grid
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
axes = axes.ravel()

# Train on training set, predict on FULL dataset
for idx, (model_name, model) in enumerate(class_models.items()):
    model.fit(X_train, y_train_cat)  # Train on training data
    y_full_pred = model.predict(X)     # Predict on ALL data (X instead of␣
 ↪X_test)

    # Generate confusion matrix (full dataset)
    cm = confusion_matrix(y_full_cat, y_full_pred, labels=class_labels)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,␣
 ↪display_labels=class_labels)

    # Plot
    disp.plot(ax=axes[idx], cmap='Blues', colorbar=False)
    axes[idx].set_title(f'{model_name} (Full Data)', pad=20)

    # Print classification report (full dataset)
    print(f"\nClassification Report for {model_name} (Full Dataset):")
    print(classification_report(y_full_cat, y_full_pred,␣
 ↪target_names=class_labels))
    print("=" * 60)

# Hide empty subplot
axes[3].axis('off')

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

```
Classification Report for Random Forest (Full Dataset):
              precision    recall  f1-score   support

        High       0.97      0.97      0.97      7226
      Medium       0.97      0.97      0.97      7164
         Low       0.95      0.94      0.94      7223
```

```
    accuracy                           0.96     21613
   macro avg       0.96      0.96      0.96     21613
weighted avg       0.96      0.96      0.96     21613

============================================================

Classification Report for K-Neighbors (Full Dataset):
              precision    recall  f1-score   support

        High       0.66      0.78      0.72      7226
      Medium       0.74      0.77      0.75      7164
         Low       0.65      0.51      0.57      7223

    accuracy                           0.68     21613
   macro avg       0.68      0.68      0.68     21613
weighted avg       0.68      0.68      0.68     21613

============================================================

Classification Report for Naive Bayes (Full Dataset):
              precision    recall  f1-score   support

        High       0.51      0.81      0.63      7226
      Medium       0.75      0.56      0.64      7164
         Low       0.38      0.26      0.31      7223

    accuracy                           0.54     21613
   macro avg       0.55      0.54      0.53     21613
weighted avg       0.55      0.54      0.53     21613

============================================================
```



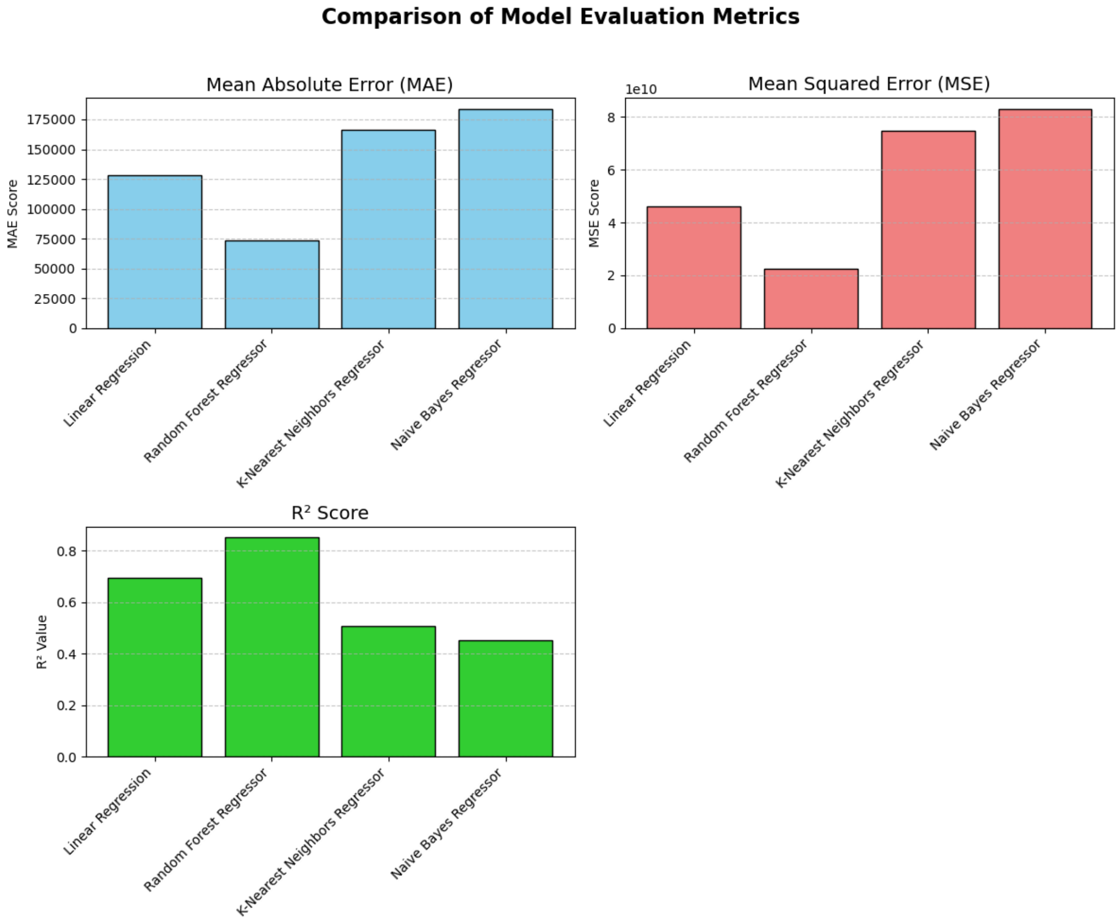
[31]:

```
# Create a 2x2 grid
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
fig.suptitle('Comparison of Model Evaluation Metrics', fontsize=16,␣
 ↪fontweight='bold')

# Identify the best and worst models for each metric (kept for reference but␣
 ↪not used in coloring)
best_mae_model = results_df['MAE'].idxmin()
worst_mae_model = results_df['MAE'].idxmax()

best_mse_model = results_df['MSE'].idxmin()
worst_mse_model = results_df['MSE'].idxmax()

best_r2_model = results_df['R2'].idxmax()
worst_r2_model = results_df['R2'].idxmin()

# Print the best and worst models with their respective values
```

```python
print(f"Best MAE Model: {best_mae_model} (MAE: {results_df.loc[best_mae_model, 'MAE']:.4f})")
print(f"Worst MAE Model: {worst_mae_model} (MAE: {results_df.loc[worst_mae_model, 'MAE']:.4f})\n")

print(f"Best MSE Model: {best_mse_model} (MSE: {results_df.loc[best_mse_model, 'MSE']:.4f})")
print(f"Worst MSE Model: {worst_mse_model} (MSE: {results_df.loc[worst_mse_model, 'MSE']:.4f})\n")

print(f"Best R² Model: {best_r2_model} (R²: {results_df.loc[best_r2_model, 'R2']:.4f})")
print(f"Worst R² Model: {worst_r2_model} (R²: {results_df.loc[worst_r2_model, 'R2']:.4f})\n")

# Plot MAE with uniform color
axes[0, 0].bar(results_df.index, results_df['MAE'], color='skyblue', edgecolor='black')
axes[0, 0].set_title('Mean Absolute Error (MAE)', fontsize=14)
axes[0, 0].set_ylabel('MAE Score')
axes[0, 0].set_xticks(range(len(results_df.index)))
axes[0, 0].set_xticklabels(results_df.index, rotation=45, ha='right', fontsize=10)
axes[0, 0].grid(axis='y', linestyle='--', alpha=0.7)

# Plot MSE with uniform color
axes[0, 1].bar(results_df.index, results_df['MSE'], color='lightcoral', edgecolor='black')
axes[0, 1].set_title('Mean Squared Error (MSE)', fontsize=14)
axes[0, 1].set_ylabel('MSE Score')
axes[0, 1].set_xticks(range(len(results_df.index)))
axes[0, 1].set_xticklabels(results_df.index, rotation=45, ha='right', fontsize=10)
axes[0, 1].grid(axis='y', linestyle='--', alpha=0.7)

# Plot R² with uniform color
axes[1, 0].bar(results_df.index, results_df['R2'], color='limegreen', edgecolor='black')
axes[1, 0].set_title('R² Score', fontsize=14)
axes[1, 0].set_ylabel('R² Value')
axes[1, 0].set_xticks(range(len(results_df.index)))
axes[1, 0].set_xticklabels(results_df.index, rotation=45, ha='right', fontsize=10)
axes[1, 0].grid(axis='y', linestyle='--', alpha=0.7)

# Leave the bottom-right slot blank
```

```
axes[1, 1].axis('off')

# Adjust layout
plt.tight_layout(rect=[0, 0, 1, 0.96])

# Show the grid
plt.show()
```

```
Best MAE Model: Random Forest Regressor (MAE: 73451.3952)
Worst MAE Model: Naive Bayes Regressor (MAE: 183768.4548)

Best MSE Model: Random Forest Regressor (MSE: 22560437679.1711)
Worst MSE Model: Naive Bayes Regressor (MSE: 82977602741.9500)

Best R² Model: Random Forest Regressor (R²: 0.8508)
Worst R² Model: Naive Bayes Regressor (R²: 0.4511)
```



Linear regression and random forest regressors tend to perform better than K-Nearest Neighbors

(KNN) and Naive Bayes for housing price prediction datasets due to their inherent strengths in handling structured, numerical data with complex relationships. Housing datasets typically contain a mix of numerical features like square footage and number of bedrooms, along with categorical variables like neighborhood or property type. Linear regression excels in this scenario because it effectively models the linear relationships between these features and the target price variable. It provides interpretable coefficients that show how each feature contributes to the price, such as quantifying how much an additional bedroom increases a home's value. The algorithm also handles continuous numerical data naturally without requiring extensive preprocessing, making it well-suited for housing market analysis where precise price predictions are crucial.

Random forest regressors outperform other models because they automatically capture both linear and non-linear relationships in the data through their ensemble decision tree approach. Housing prices are influenced by complex interactions between features—for example, a large backyard may significantly increase value in suburban areas but matter less in urban settings. Random forests naturally model these interactions without explicit feature engineering. They also handle mixed data types well, are robust to outliers (like unusually large or small properties), and provide feature importance scores that reveal which factors most impact pricing. Additionally, their ability to process high-dimensional data makes them ideal for datasets with numerous features like those found in real estate.

KNN struggles with housing data primarily due to the "curse of dimensionality." As housing datasets often contain 10-20 features (square footage, bedrooms, location coordinates, etc.), the concept of distance becomes less meaningful in such high-dimensional spaces. This makes it difficult for KNN to find truly comparable neighboring properties. The algorithm also requires all features to be carefully scaled, as it weighs bedroom count equally with square footage by default, which doesn't reflect real-world pricing dynamics. Moreover, KNN's computational inefficiency becomes problematic with large housing datasets, as it must compare each property against all others during prediction. While KNN might work decently for location-based pricing using just latitude and longitude, it generally underperforms for comprehensive housing price prediction.

Naive Bayes is poorly suited for housing price prediction because its fundamental assumption of feature independence rarely holds in real estate data. In practice, housing features are highly interdependent—the number of bedrooms correlates with square footage, which in turn relates to neighborhood demographics. The algorithm also struggles with continuous numerical variables, requiring artificial discretization that loses valuable pricing information. Unlike regression models that can output precise dollar amounts, Naive Bayes predicts probability distributions, making it ill-equipped for exact price estimation. While it could potentially work for categorical predictions like "above/below median price," even this approach would be outperformed by random forests or logistic regression due to Naive Bayes' inability to model feature interactions properly. For these reasons, it typically ranks as the weakest choice among the four algorithms for housing price prediction tasks.

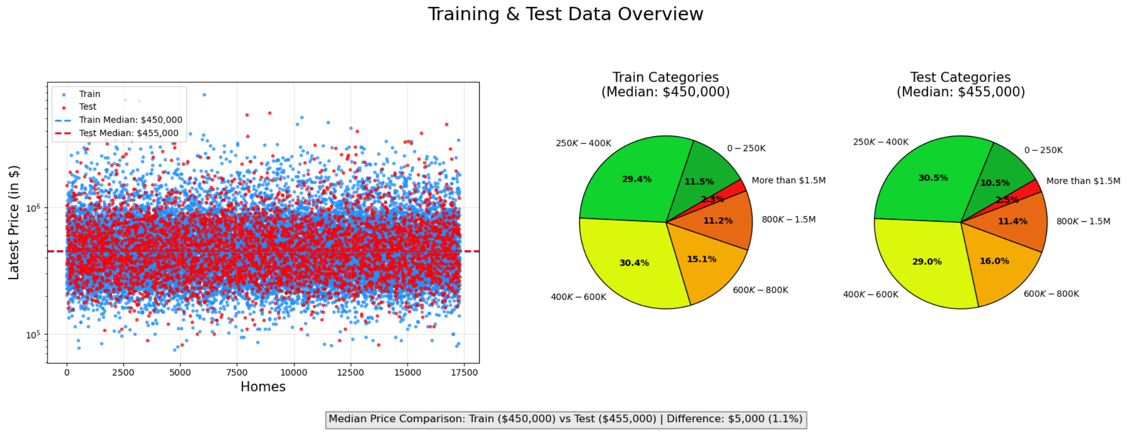
[7]:

```python
# Create a figure with subplots
fig, ax = plt.subplots(1, 3, gridspec_kw={'width_ratios': [6, 3, 3]})
fig.set_size_inches(18, 6)  # Increased figure size for better visibility
fig.suptitle('Training & Test Data Overview', size=21, y=1.05)

# Calculate medians
train_median = np.median(y_train)
```

```
test_median = np.median(y_test)

# Scatter plot for training and test data
s1 = ax[0].scatter(np.arange(len(y_train)), y_train, marker='.',␣
 ↪color='dodgerblue', alpha=0.7, label='Train')
s2 = ax[0].scatter(np.linspace(0, len(y_train), len(y_test)), y_test, marker='.
 ↪', color='red', alpha=0.7, label='Test')

# Add median lines with labels
ax[0].axhline(train_median, color='dodgerblue', linestyle='--', linewidth=2,
              label=f'Train Median: ${train_median:,.0f}')
ax[0].axhline(test_median, color='red', linestyle='--', linewidth=2,
              label=f'Test Median: ${test_median:,.0f}')

ax[0].legend(loc='upper left', fontsize=10)
ax[0].set_ylabel('Latest Price (in $)', size=15)
ax[0].set_xlabel('Homes', size=15)
ax[0].set_yscale('log')
ax[0].grid(True, alpha=0.3)

# Define intervals for price ranges
labels = ['$0 - $250K', '$250K - $400K', '$400K - $600K', '$600K - $800K',␣
 ↪'$800K - $1.5M', 'More than $1.5M']
colors = ['#13af2a', '#10d32e', '#dbf70c', '#f4ac04', '#e86914', '#f41313']
intervals = [0, 250000, 400000, 600000, 800000, 1500000, max(y_train.max(),␣
 ↪y_test.max())]

# Pie chart (Train set)
train_data = pd.DataFrame({'price': y_train})
chart_slice_sizes = train_data.groupby(pd.cut(train_data['price'], intervals)).
 ↪size().values
ax[1].set_title(f'Train Categories\n(Median: ${train_median:,.0f})', size=15,␣
 ↪pad=20)
wedges, texts, autotexts = ax[1].pie(chart_slice_sizes, labels=labels,␣
 ↪colors=colors,
                                     startangle=30, autopct='%1.1f%%',␣
 ↪wedgeprops={'edgecolor': 'black'})
plt.setp(autotexts, size=10, weight="bold")  # Make percentages more readable

# Pie chart (Test set)
test_data = pd.DataFrame({'price': y_test})
chart_slice_sizes = test_data.groupby(pd.cut(test_data['price'], intervals)).
 ↪size().values
ax[2].set_title(f'Test Categories\n(Median: ${test_median:,.0f})', size=15,␣
 ↪pad=20)
```

```
wedges, texts, autotexts = ax[2].pie(chart_slice_sizes, labels=labels,␣
 ↪colors=colors,
                                        startangle=30, autopct='%1.1f%%',␣
 ↪wedgeprops={'edgecolor': 'black'})
plt.setp(autotexts, size=10, weight="bold")  # Make percentages more readable

# Add median comparison annotation
fig.text(0.5, -0.05,
          f"Median Price Comparison: Train (${train_median:,.0f}) vs Test␣
 ↪(${test_median:,.0f}) | Difference: ${abs(train_median-test_median):,.0f}␣
 ↪({abs(train_median-test_median)/train_median*100:.1f}%)",
          ha='center', fontsize=12, bbox=dict(facecolor='lightgray', alpha=0.5))

fig.tight_layout()
plt.subplots_adjust(top=0.85)  # Adjust top spacing for main title
plt.show()
```

```
C:\Users\emman\AppData\Local\Temp\ipykernel_26808\3031443393.py:33:
FutureWarning: The default of observed=False is deprecated and will be changed
to True in a future version of pandas. Pass observed=False to retain current
behavior or observed=True to adopt the future default and silence this warning.
  chart_slice_sizes = train_data.groupby(pd.cut(train_data['price'],
intervals)).size().values
C:\Users\emman\AppData\Local\Temp\ipykernel_26808\3031443393.py:41:
FutureWarning: The default of observed=False is deprecated and will be changed
to True in a future version of pandas. Pass observed=False to retain current
behavior or observed=True to adopt the future default and silence this warning.
  chart_slice_sizes = test_data.groupby(pd.cut(test_data['price'],
intervals)).size().values
```



1.4 Analysis

The King County housing dataset reveals significant trends in property values, driven by location, size, and amenities. Prices range widely from $75,000$ to 7.7 million, with a median of $450,000, reflecting a diverse market. Waterfront properties and renovated homes command substantial premiums, particularly in affluent neighborhoods like Mercer Island and Queen Anne. Geographic coordinates (lat and long) highlight clustering of high-value properties near Seattle's urban core and waterfronts, while suburban areas such as Auburn and Kent feature lower prices but larger lot sizes. Square footage emerges as a critical factor, with larger homes (averaging 2,080 sqft of living space) correlating strongly with higher prices. Most properties have 3 bedrooms and 2 bathrooms, though outliers like homes with 33 bedrooms suggest potential data anomalies. Renovations, while rare (only 3.9% of homes), boost value, particularly when paired with modern amenities.

1.5 Conclusion

Location, size, and amenities are the primary drivers of housing prices in King County. Proximity to urban centers like Seattle and Bellevue, as well as waterfront access, significantly elevates property values. Square footage, particularly living space and above-ground area, is the strongest predictor of price, underscoring buyer preference for spacious homes. Waterfront status and high construction grades further amplify premiums, reflecting demand for luxury and quality. However, affordability challenges persist in high-demand urban zones, pushing buyers toward suburban areas where larger lots are more accessible. The dataset also hints at broader societal trends, such as growing emphasis on sustainability through renovations and energy-efficient designs.