US House Market Prediction

In [3]:
```
import copy
import warnings

import joblib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import torch
import torch.nn as nn
import torchinfo
import tqdm
from geopy.geocoders import Nominatim
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import RobustScaler, StandardScaler
from torch.optim.lr_scheduler import ReduceLROnPlateau, StepLR
from torch.utils.data import DataLoader, TensorDataset

warnings.filterwarnings('ignore')
```

Data Loading

In [2]:
```
df = pd.read_csv('datasets/realtordataUS.csv', encoding='UTF-8')
df
```

Out[2]:

	brokered_by	status	price	bed	bath	acre_lot	street	city	state	zip_code	house_size	prev_sol
0	103378.0	for_sale	105000.0	3.0	2.0	0.12	1962661.0	Adjuntas	Puerto Rico	601.0	920.0	
1	52707.0	for_sale	80000.0	4.0	2.0	0.08	1902874.0	Adjuntas	Puerto Rico	601.0	1527.0	
2	103379.0	for_sale	67000.0	2.0	1.0	0.15	1404990.0	Juana Diaz	Puerto Rico	795.0	748.0	
3	31239.0	for_sale	145000.0	4.0	2.0	0.10	1947675.0	Ponce	Puerto Rico	731.0	1800.0	
4	34632.0	for_sale	65000.0	6.0	2.0	0.05	331151.0	Mayaguez	Puerto Rico	680.0	NaN	
...	...	...	...	...	...	...	...	...	...	...	...	
2226377	23009.0	sold	359900.0	4.0	2.0	0.33	353094.0	Richland	Washington	99354.0	3600.0	2022
2226378	18208.0	sold	350000.0	3.0	2.0	0.10	1062149.0	Richland	Washington	99354.0	1616.0	2022
2226379	76856.0	sold	440000.0	6.0	3.0	0.50	405677.0	Richland	Washington	99354.0	3200.0	2022
2226380	53618.0	sold	179900.0	2.0	1.0	0.09	761379.0	Richland	Washington	99354.0	933.0	2022
2226381	108243.0	sold	580000.0	5.0	3.0	0.31	307704.0	Richland	Washington	99354.0	3615.0	2022

2226382 rows × 12 columns

Null/Missing Values Check

In [3]:
```
df.isnull().sum()
```

Out[3]:
```
brokered_by         4533
status                 0
price               1541
bed               481317
bath              511771
acre_lot          325589
street             10866
city                1407
state                  8
zip_code             299
house_size        568484
prev_sold_date    734297
dtype: int64
```

Statistics of Dataset

In [4]:
```
df.describe()
```

Out[4]:

	brokered_by	price	bed	bath	acre_lot	street	zip_code	house_size
count	2.221849e+06	2.224841e+06	1.745065e+06	1.714611e+06	1.900793e+06	2.215516e+06	2.226083e+06	1.657898e+06
mean	5.293989e+04	5.241955e+05	3.275841e+00	2.496440e+00	1.522303e+01	1.012325e+06	5.218668e+04	2.714471e+03
std	3.064275e+04	2.138893e+06	1.567274e+00	1.652573e+00	7.628238e+02	5.837635e+05	2.895408e+04	8.081635e+05
min	0.000000e+00	0.000000e+00	1.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	4.000000e+00
25%	2.386100e+04	1.650000e+05	3.000000e+00	2.000000e+00	1.500000e-01	5.063128e+05	2.961700e+04	1.300000e+03
50%	5.288400e+04	3.250000e+05	3.000000e+00	2.000000e+00	2.600000e-01	1.012766e+06	4.838200e+04	1.760000e+03
75%	7.918300e+04	5.500000e+05	4.000000e+00	3.000000e+00	9.800000e-01	1.521173e+06	7.807000e+04	2.413000e+03
max	1.101420e+05	2.147484e+09	4.730000e+02	8.300000e+02	1.000000e+05	2.001357e+06	9.999900e+04	1.040400e+09

Data Cleaning

First we will drop all unncessary columns in the dataset

In [5]:
```
df = df.drop(columns=["brokered_by", "status", "street", "prev_sold_date"])
df
```

Out[5]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size
0	105000.0	3.0	2.0	0.12	Adjuntas	Puerto Rico	601.0	920.0
1	80000.0	4.0	2.0	0.08	Adjuntas	Puerto Rico	601.0	1527.0
2	67000.0	2.0	1.0	0.15	Juana Diaz	Puerto Rico	795.0	748.0
3	145000.0	4.0	2.0	0.10	Ponce	Puerto Rico	731.0	1800.0
4	65000.0	6.0	2.0	0.05	Mayaguez	Puerto Rico	680.0	NaN
...	...	...	...	...	...	...	...	...
2226377	359900.0	4.0	2.0	0.33	Richland	Washington	99354.0	3600.0
2226378	350000.0	3.0	2.0	0.10	Richland	Washington	99354.0	1616.0
2226379	440000.0	6.0	3.0	0.50	Richland	Washington	99354.0	3200.0
2226380	179900.0	2.0	1.0	0.09	Richland	Washington	99354.0	933.0
2226381	580000.0	5.0	3.0	0.31	Richland	Washington	99354.0	3615.0

2226382 rows × 8 columns

We will also drop all null values present in the dataset since there are tons from the Null Value Check

In [6]:
```
df = df.dropna()
df
```

Out[6]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size
0	105000.0	3.0	2.0	0.12	Adjuntas	Puerto Rico	601.0	920.0
1	80000.0	4.0	2.0	0.08	Adjuntas	Puerto Rico	601.0	1527.0
2	67000.0	2.0	1.0	0.15	Juana Diaz	Puerto Rico	795.0	748.0
3	145000.0	4.0	2.0	0.10	Ponce	Puerto Rico	731.0	1800.0
5	179000.0	4.0	3.0	0.46	San Sebastian	Puerto Rico	612.0	2520.0
...	...	...	...	...	...	...	...	...
2226377	359900.0	4.0	2.0	0.33	Richland	Washington	99354.0	3600.0
2226378	350000.0	3.0	2.0	0.10	Richland	Washington	99354.0	1616.0
2226379	440000.0	6.0	3.0	0.50	Richland	Washington	99354.0	3200.0
2226380	179900.0	2.0	1.0	0.09	Richland	Washington	99354.0	933.0
2226381	580000.0	5.0	3.0	0.31	Richland	Washington	99354.0	3615.0

1360347 rows × 8 columns

We also want to drop all duplicate rows if there are any.

In [7]:
```
df = df.drop_duplicates()
df
```

Out[7]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size
0	105000.0	3.0	2.0	0.12	Adjuntas	Puerto Rico	601.0	920.0
1	80000.0	4.0	2.0	0.08	Adjuntas	Puerto Rico	601.0	1527.0
2	67000.0	2.0	1.0	0.15	Juana Diaz	Puerto Rico	795.0	748.0
3	145000.0	4.0	2.0	0.10	Ponce	Puerto Rico	731.0	1800.0
5	179000.0	4.0	3.0	0.46	San Sebastian	Puerto Rico	612.0	2520.0
...	...	...	...	...	...	...	...	...
2226377	359900.0	4.0	2.0	0.33	Richland	Washington	99354.0	3600.0
2226378	350000.0	3.0	2.0	0.10	Richland	Washington	99354.0	1616.0
2226379	440000.0	6.0	3.0	0.50	Richland	Washington	99354.0	3200.0
2226380	179900.0	2.0	1.0	0.09	Richland	Washington	99354.0	933.0
2226381	580000.0	5.0	3.0	0.31	Richland	Washington	99354.0	3615.0

1259352 rows × 8 columns

We also want to remove outliers such as houses with 10 or more bedrooms/bathrooms (these could be buildings, multi-family residentals, extreme luxary mansions, or data entry errors)

In [8]:
```
df = df[(df['bed'] < 10) & (df['bath'] < 10) & (df['price'] > 50000) & (df['price'] < 50000000)]
df
```

Out[8]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size
0	105000.0	3.0	2.0	0.12	Adjuntas	Puerto Rico	601.0	920.0
1	80000.0	4.0	2.0	0.08	Adjuntas	Puerto Rico	601.0	1527.0
2	67000.0	2.0	1.0	0.15	Juana Diaz	Puerto Rico	795.0	748.0
3	145000.0	4.0	2.0	0.10	Ponce	Puerto Rico	731.0	1800.0
5	179000.0	4.0	3.0	0.46	San Sebastian	Puerto Rico	612.0	2520.0
...	...	...	...	...	...	...	...	...
2226377	359900.0	4.0	2.0	0.33	Richland	Washington	99354.0	3600.0
2226378	350000.0	3.0	2.0	0.10	Richland	Washington	99354.0	1616.0
2226379	440000.0	6.0	3.0	0.50	Richland	Washington	99354.0	3200.0
2226380	179900.0	2.0	1.0	0.09	Richland	Washington	99354.0	933.0
2226381	580000.0	5.0	3.0	0.31	Richland	Washington	99354.0	3615.0

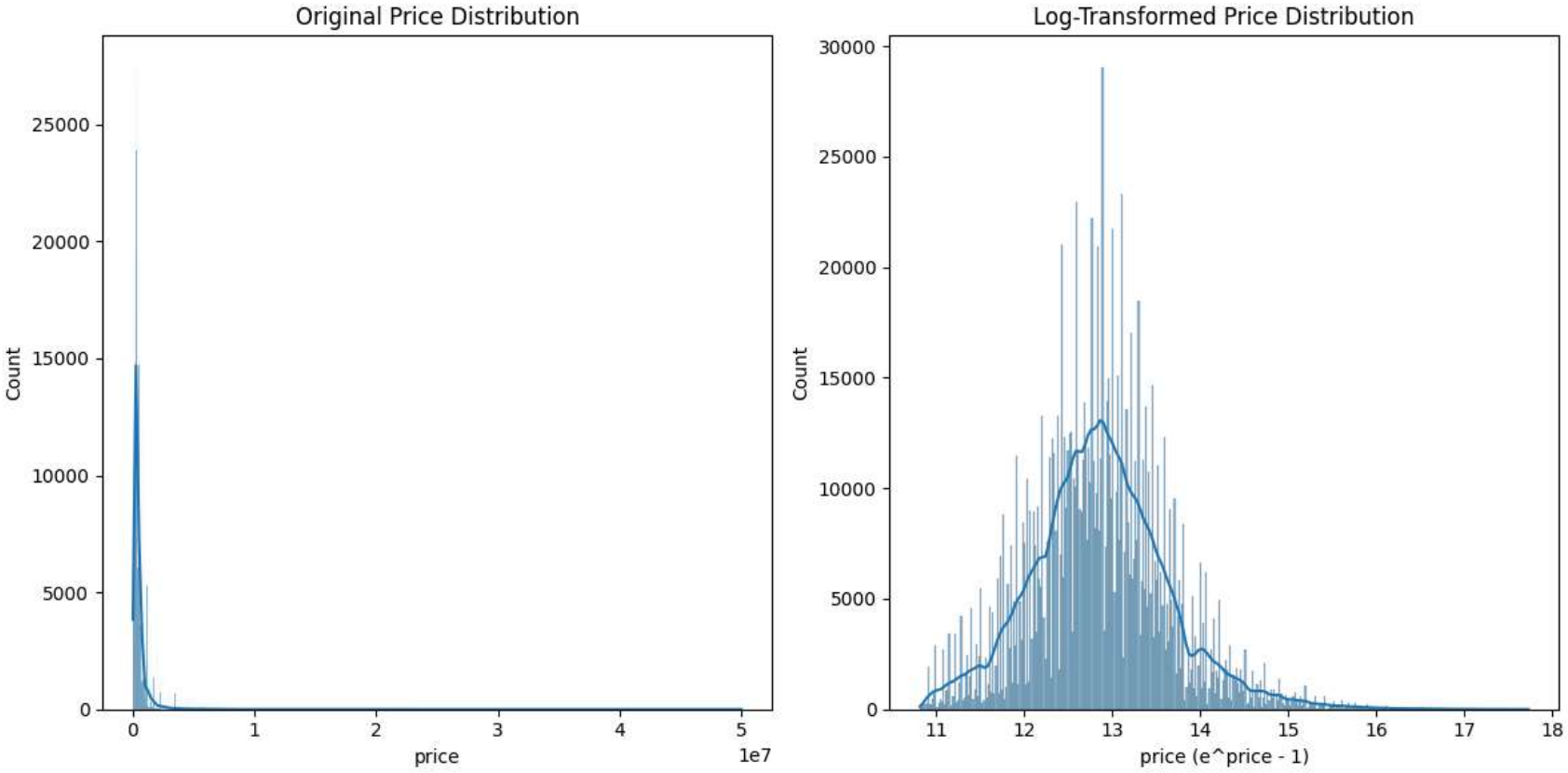
1239185 rows × 8 columns

Visualization & Exploratory Data Analysis

Lets first see the count of all priced houses. If we do it regularly its incredibly difficult to see due to the outliers and they dominate the scale of our plot. We can then use a logarithmic scale to further see the relationships.

In [9]:
```
# Apply log transformation to Price
prices = np.log1p(df['price'])  # log1p handles log(1 + x), avoiding log(0)

# Visualize the original and transformed distributions
plt.figure(figsize=(12, 6))

# Original Price Distribution
plt.subplot(1, 2, 1)
sns.histplot(df['price'], kde=True)
plt.title('Original Price Distribution')
```

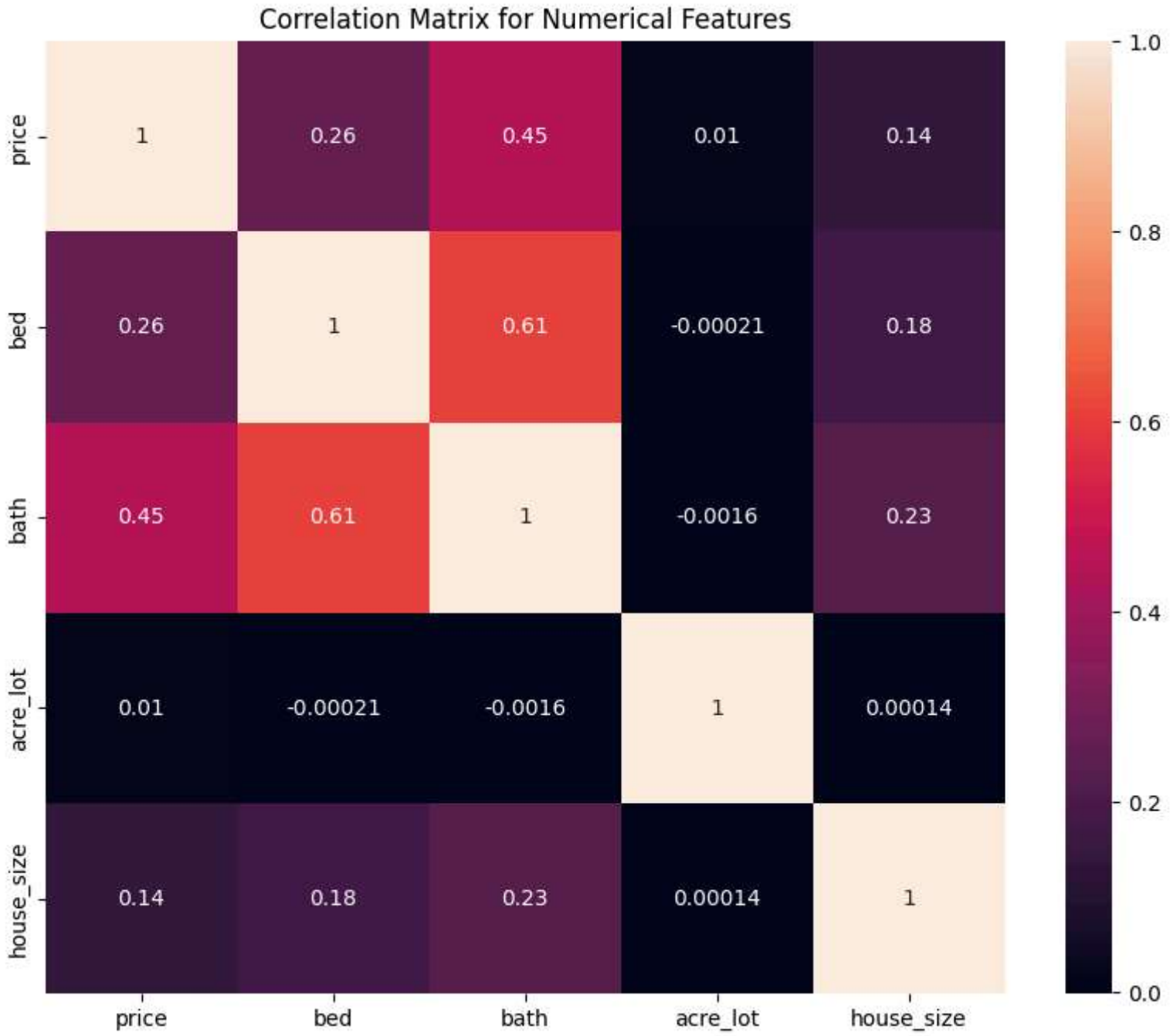
```
# Log-Transformed Price Distribution
plt.subplot(1, 2, 2)
sns.histplot(prices, kde=True)
plt.title('Log-Transformed Price Distribution')
plt.xlabel("price (e^price - 1)")

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



We can also plot this dataset using a heat map to show the correlation matrix of all numerical features.

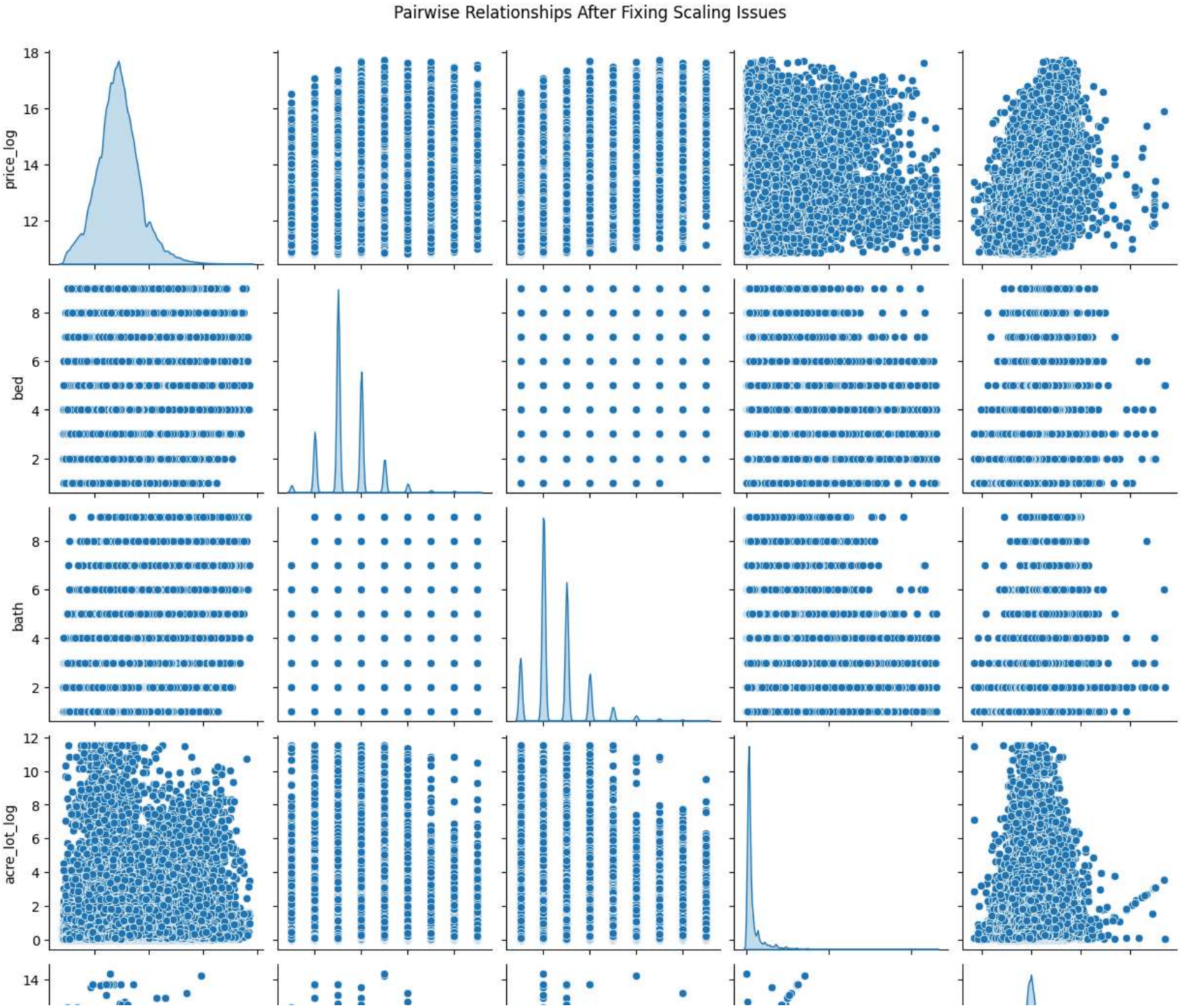
In [10]:
```
# Compute the correlation matrix
corr_columns = ['price', 'bed', 'bath', 'acre_lot', 'house_size']
corr_matrix = df[corr_columns].corr()
```

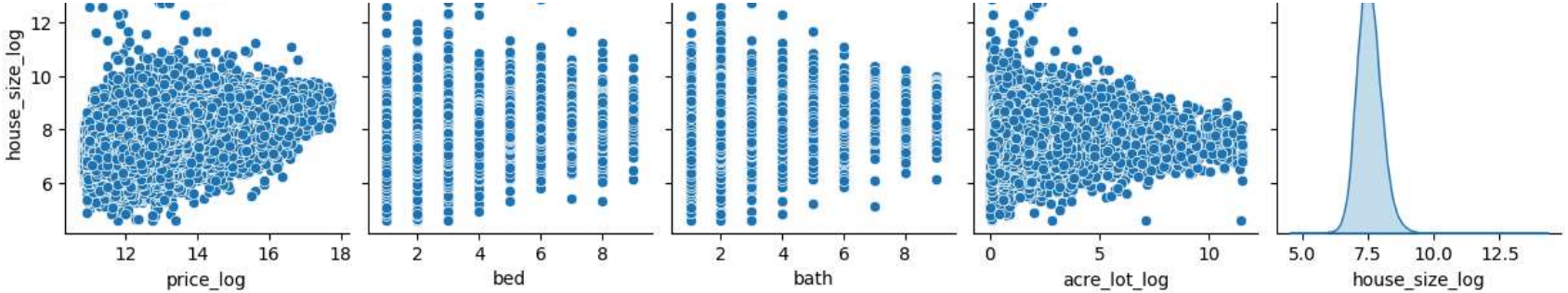
```
# Plot the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True)
plt.title('Correlation Matrix for Numerical Features')
plt.show()
```


Correlation Matrix for Numerical Features
price
bed
bath
acre_lot
house_size
1
0.26
0.45
0.01
0.14
0.26
1
0.61
-0.00021
0.18
0.45
0.61
1
-0.0016
0.23
0.01
-0.00021
-0.0016
1
0.00014
0.14
0.18
0.23
0.00014
1
price
bed
bath
acre_lot
house_size
1.0
0.8
0.6
0.4
0.2
0.0

Here is the correlation graphings and the pairplots for all features as well. These are also log-scaled due to the extreme outlier situation we faced earlier.

In [11]:
```
# Apply Log to handle scaling issues
df_plot = df[:]
df_plot['price_log'] = np.log1p(df['price'])
df_plot['house_size_log'] = np.log1p(df['house_size'])
df_plot['acre_lot_log'] = np.log1p(df['acre_lot'])

sns.pairplot(
    df_plot,
    vars=['price_log', 'bed', 'bath', 'acre_lot_log', 'house_size_log'],
    diag_kind='kde'
)
plt.suptitle('Pairwise Relationships After Fixing Scaling Issues', y=1.02)
plt.show()
```


Pairwise Relationships After Fixing Scaling Issues
price_log
bed
bath
acre_lot_log



In [12]:
```
# Save the cleaned dataset
df.to_csv('datasets/realtordataUS_cleaned.csv', index=False)
```

Data Preproccessing

In [13]:
```
df_cleaned = pd.read_csv('datasets/realtordataUS_cleaned.csv', encoding='UTF-8')
df_cleaned
```

Out[13]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size
0	105000.0	3.0	2.0	0.12	Adjuntas	Puerto Rico	601.0	920.0
1	80000.0	4.0	2.0	0.08	Adjuntas	Puerto Rico	601.0	1527.0
2	67000.0	2.0	1.0	0.15	Juana Diaz	Puerto Rico	795.0	748.0
3	145000.0	4.0	2.0	0.10	Ponce	Puerto Rico	731.0	1800.0
4	179000.0	4.0	3.0	0.46	San Sebastian	Puerto Rico	612.0	2520.0
...	...	...	...	...	...	...	...	...
1239180	359900.0	4.0	2.0	0.33	Richland	Washington	99354.0	3600.0
1239181	350000.0	3.0	2.0	0.10	Richland	Washington	99354.0	1616.0
1239182	440000.0	6.0	3.0	0.50	Richland	Washington	99354.0	3200.0
1239183	179900.0	2.0	1.0	0.09	Richland	Washington	99354.0	933.0
1239184	580000.0	5.0	3.0	0.31	Richland	Washington	99354.0	3615.0

1239185 rows × 8 columns

Due to our computation limit, we will only use 43000 entries from state California

In [ ]:
```
df_california = df_cleaned[df_cleaned['state'] == 'California']
df_sample = df_california.sample(n=43000)
df_sample.to_csv('datasets/realtordataUS_cleaned_california_sample.csv', index=False)
df_remaining = df_california.drop(df_california.index)
df_remaining.to_csv('datasets/realtordataUS_cleaned_california_remaining.csv', index=False)
df_sample
```

Out[ ]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size
679788	305000.0	3.0	2.0	0.35	Pope Valley	California	94567.0	1698.0
1202601	470740.0	3.0	2.0	0.31	Plumas Lake	California	95961.0	1818.0
1141229	581990.0	3.0	3.0	0.07	Murrieta	California	92563.0	1864.0
1190201	325000.0	3.0	2.0	0.37	Pioneer	California	95666.0	1724.0
1152366	190000.0	1.0	1.0	0.01	Palm Springs	California	92262.0	589.0
...	...	...	...	...	...	...	...	...
1152496	369000.0	4.0	2.0	0.11	La Quinta	California	92253.0	1470.0
1171830	2400000.0	4.0	3.0	0.15	Piedmont	California	94610.0	3237.0
1181247	769950.0	4.0	3.0	0.12	Tracy	California	95377.0	2498.0
1165907	550000.0	4.0	3.0	0.32	Bakersfield	California	93312.0	2592.0
1120570	725000.0	3.0	2.0	0.23	Los Angeles	California	90006.0	1256.0

43000 rows × 8 columns

First, we convert the categorical location data (city, state, zip code) into numerical data. There is also an option to use Embedding layers, however that will limit our prediction inputs (since the categorical data of this dataset is not fixed)

In [82]:
```
# Merge city, state, and zip_code into a location string for GeoPy API to convert to longitude and latitude coordinat
df_sample['zip_code'] = df_sample['zip_code'].astype(int)
df_sample['location'] = df_sample['city'] + ', ' + df_sample['state'] + ', ' + df_sample['zip_code'].astype(str)
df_sample
```

Out[82]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size	location
679788	305000.0	-0.260594	-0.475677	-0.017618	Pope Valley	California	94567	-0.186913	Pope Valley, California, 94567
1202601	470740.0	-0.260594	-0.475677	-0.017655	Plumas Lake	California	95961	-0.082726	Plumas Lake, California, 95961
1141229	581990.0	-0.260594	0.440777	-0.017876	Murrieta	California	92563	-0.042788	Murrieta, California, 92563
1190201	325000.0	-0.260594	-0.475677	-0.017599	Pioneer	California	95666	-0.164339	Pioneer, California, 95666
1152366	190000.0	-2.100705	-1.392130	-0.017931	Palm Springs	California	92262	-1.149775	Palm Springs, California, 92262
...	...	...	...	...	...	...	...	...	...
1152496	369000.0	0.659461	-0.475677	-0.017839	La Quinta	California	92253	-0.384869	La Quinta, California, 92253
1171830	2400000.0	0.659461	0.440777	-0.017802	Piedmont	California	94610	1.149286	Piedmont, California, 94610
1181247	769950.0	0.659461	0.440777	-0.017830	Tracy	California	95377	0.507667	Tracy, California, 95377
1165907	550000.0	0.659461	0.440777	-0.017645	Bakersfield	California	93312	0.589280	Bakersfield, California, 93312
1120570	725000.0	-0.260594	-0.475677	-0.017728	Los Angeles	California	90006	-0.570669	Los Angeles, California, 90006

43000 rows × 9 columns

Convert location string to longitude and latitude coordinates

In [ ]:

```
geolocator = Nominatim(user_agent='ontariotechu')

def get_long_lat_coordinates(location):
    try:
        loc = geolocator.geocode(location.split(',')[-1].strip()) # try with only the zip code
        if loc:
            return loc.latitude, loc.longitude
        else:
            loc = geolocator.geocode(location)
```

```
            if loc:
                return loc.latitude, loc.longitude
            else:
                return None, None
    except Exception as e:
        return None, None

df_sample_longlat = df_sample[:]
df_sample_longlat[['loc_lat', 'loc_long']] = df_sample_longlat.apply(lambda row: pd.Series(get_long_lat_coordinates(r
df_sample_longlat.to_csv('datasets/realtordataUS_cleaned_california_sample_longlat.csv', index=False)
```

In [4]:
```
df_sample_longlat = pd.read_csv('datasets/realtordataUS_cleaned_california_sample_longlat.csv', encoding='UTF-8')
df_sample_longlat
```

Out[4]:

	price	bed	bath	acre_lot	city	state	zip_code	house_size	location	loc_lat	loc_long
0	305000	3	2	0.35	Pope Valley	California	94567	1698	Pope Valley, California, 94567	38.615184	-122.427757
1	470740	3	2	0.31	Plumas Lake	California	95961	1818	Plumas Lake, California, 95961	39.020402	-121.551226
2	581990	3	3	0.07	Murrieta	California	92563	1864	Murrieta, California, 92563	33.560832	-117.210656
3	325000	3	2	0.37	Pioneer	California	95666	1724	Pioneer, California, 95666	38.431855	-120.571872
4	190000	1	1	0.01	Palm Springs	California	92262	589	Palm Springs, California, 92262	33.824627	-116.540303
...	...	...	...	...	...	...	...	...	...	...	...
42995	369000	4	2	0.11	La Quinta	California	92253	1470	La Quinta, California, 92253	33.677474	-116.295879
42996	2400000	4	3	0.15	Piedmont	California	94610	3237	Piedmont, California, 94610	37.824371	-122.231635
42997	769950	4	3	0.12	Tracy	California	95377	2498	Tracy, California, 95377	37.738551	-121.420139
42998	550000	4	3	0.32	Bakersfield	California	93312	2592	Bakersfield, California, 93312	35.373871	-119.019463
42999	725000	3	2	0.23	Los Angeles	California	90006	1256	Los Angeles, California, 90006	34.053691	-118.242766

43000 rows × 11 columns

Then we have to convert longitude and latitude coordinates to the Earth's Cartesian coordinates, so that our model can do spatial calculation

In [5]:
```
def get_cartesian_coordinates(df):
    df_cartesian = df[:]
```

```
    # Convert latitude and longitude to radians
    df_cartesian['loc_lat_rad'] = np.radians(df_cartesian['loc_lat'])
    df_cartesian['loc_long_rad'] = np.radians(df_cartesian['loc_long'])

    # Calculate Cartesian coordinates
    df_cartesian['loc_x'] = np.cos(df_cartesian['loc_lat_rad']) * np.cos(df_cartesian['loc_long_rad'])
    df_cartesian['loc_y'] = np.cos(df_cartesian['loc_lat_rad']) * np.sin(df_cartesian['loc_long_rad'])
    df_cartesian['loc_z'] = np.sin(df_cartesian['loc_lat_rad'])

    # Now we can remove all categorical columns and intermediate math columns
    df_out = df_cartesian.drop(columns=['city', 'state', 'zip_code', 'location', 'loc_lat', 'loc_long', 'loc_lat_rad
    return df_out # The Cartesian data is already normalized

df_sample_cartesian = get_cartesian_coordinates(df_sample_longlat)[:]
df_sample_cartesian
```

Out[5]:

	price	bed	bath	acre_lot	house_size	loc_x	loc_y	loc_z
0	305000	3	2	0.35	1698	-0.418991	-0.659517	0.624087
1	470740	3	2	0.31	1818	-0.406533	-0.662072	0.629597
2	581990	3	3	0.07	1864	-0.381037	-0.741079	0.552822
3	325000	3	2	0.37	1724	-0.398426	-0.674456	0.621583
4	190000	1	1	0.01	589	-0.371200	-0.743202	0.556653
...	...	...	...	...	...	...	...	...
42995	369000	4	2	0.11	1470	-0.368658	-0.746058	0.554517
42996	2400000	4	3	0.15	3237	-0.421285	-0.668171	0.613243
42997	769950	4	3	0.12	2498	-0.412258	-0.674853	0.612059
42998	550000	4	3	0.32	2592	-0.395552	-0.713023	0.578909
42999	725000	3	2	0.23	1256	-0.392059	-0.729879	0.559970

43000 rows × 8 columns

Finally, we should normalize our numeric features because the values range vastly across the dataset. This will ensure our data is on similar scales and this could improve our models performance down the line.

In [6]:
```
# Initialize numerical scaler
numerical_features_to_scale = ['bed', 'bath', 'acre_lot', 'house_size']
scaler = StandardScaler()
scaler.fit(df_sample_cartesian[numerical_features_to_scale])
joblib.dump(scaler, "../app/feature_scaler.joblib")
price_scaler = StandardScaler()
price_scaler.fit(df_sample_cartesian[['price']])
joblib.dump(price_scaler, "../app/price_scaler.joblib")

# Transform sample data
df_inputs_norm = df_sample_cartesian[:]
df_inputs_norm[numerical_features_to_scale] = scaler.transform(df_inputs_norm[numerical_features_to_scale])
df_inputs_norm['price'] = price_scaler.transform(df_inputs_norm[['price']])
df_inputs_norm
```

Out[6]:

	price	bed	bath	acre_lot	house_size	loc_x	loc_y	loc_z
0	-0.493320	-0.263561	-0.478407	-0.019058	-0.180694	-0.418991	-0.659517	0.624087
1	-0.384415	-0.263561	-0.478407	-0.019089	-0.083815	-0.406533	-0.662072	0.629597
2	-0.311315	-0.263561	0.436492	-0.019278	-0.046678	-0.381037	-0.741079	0.552822
3	-0.480178	-0.263561	-0.478407	-0.019042	-0.159703	-0.398426	-0.674456	0.621583
4	-0.568884	-2.109199	-1.393307	-0.019325	-1.076013	-0.371200	-0.743202	0.556653
...	...	...	...	...	...	...	...	...
42995	-0.451267	0.659258	-0.478407	-0.019247	-0.364763	-0.368658	-0.746058	0.554517
42996	0.883267	0.659258	0.436492	-0.019215	1.061774	-0.421285	-0.668171	0.613243
42997	-0.187810	0.659258	0.436492	-0.019239	0.465164	-0.412258	-0.674853	0.612059
42998	-0.332335	0.659258	0.436492	-0.019082	0.541052	-0.395552	-0.713023	0.578909
42999	-0.217345	-0.263561	-0.478407	-0.019152	-0.537530	-0.392059	-0.729879	0.559970

43000 rows × 8 columns

Now we can start training our model

First, we initialize our models

In [7]:
```
class LinearRegressionModel(nn.Module):
    def __init__(self, in_dims):
        super().__init__()
        self.nn = nn.Linear(in_dims, 1) # out dimension is 1, predicting the price

    def forward(self, x):
        return self.nn(x)
```

In [49]:
```
class MLPRegressionModel(nn.Module):
    def __init__(self, in_dims):
        super().__init__()
        # Define the MLP architecture
```

```
        self.nn = nn.Sequential(
            nn.Linear(in_dims, 512),
            nn.ELU(), # Neurons with negative inputs won't become inactive
            nn.Dropout(0.15), # Add dropout with a 10% probability to reduce overfitting
            nn.Linear(512, 256),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(256, 128),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(128, 64),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(64, 32),
            nn.ELU(),
            nn.Dropout(0.15),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.nn(x)
```

Next, we initialize training and testing DataLoader

In [58]:
```
# Initialize training and testing DataLoader
X = df_inputs_norm.drop(columns=['price']).to_numpy()
y = df_inputs_norm['price'].to_numpy()

# Train-test split for model evaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, shuffle=True)

# Convert to 2D PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)
in_dims = X_train.shape[1]

# Create TensorDataset and DataLoader for batching
train_dataset = TensorDataset(X_train, y_train)
test_dataset = TensorDataset(X_test, y_test)
```

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Train models and evaluate

In [54]:

```
def train(model_class: nn.Module, n_epochs: int, weight_decay=0.0):
    model = model_class(in_dims)
    print(torchinfo.summary(model))

    # Loss function
    # loss_fn = nn.MSELoss() # Mean Squared Error
    loss_fn = nn.HuberLoss(delta=0.8) # less sensitive to outliers

    # Optimizer
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=weight_decay)

    # Learning rate scheduler: Reduce learning rate when validation loss plateaus
    scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.2, patience=5, verbose=True)

    # Statistics
    best_train_mse = np.inf # Initialize to infinity
    best_mse = np.inf # Initialize to infinity
    best_weights = None
    history_mse = []
    history_r2 = []
    history_train_mse = []  # To store training MSE
    history_train_r2 = []  # To store training R2

    # Early stopping parameters
    patience = 20 # Number of epochs to wait for improvement
    epochs_without_improvement = 0 # Track epochs without improvement

    for epoch in range(n_epochs):
        model.train()

        # Variables to accumulate training loss and predictions
        total_train_loss = 0
        y_train_true_all = []
        y_train_pred_all = []
```

```
        with tqdm.tqdm(train_loader, unit="batch", mininterval=0, disable=True) as bar:
            bar.set_description(f"Epoch {epoch}")
            for X_batch, y_batch in bar:
                # Forward pass
                y_pred = model(X_batch)
                loss = loss_fn(y_pred, y_batch)
                # Backward pass
                optimizer.zero_grad()
                loss.backward()
                # Update weights
                optimizer.step()
                # Print progress
                bar.set_postfix(mse=float(loss))

                total_train_loss += loss.item()
                y_train_true_all.append(y_batch.cpu().numpy())
                y_train_pred_all.append(y_pred.cpu().detach().numpy())

        # Calculate training MSE
        train_mse = total_train_loss / len(train_loader)
        if train_mse < best_train_mse: best_train_mse = train_mse
        history_train_mse.append(train_mse)
        # Flatten training true and predicted values for R2 calculation
        y_train_true_all = np.concatenate(y_train_true_all, axis=0)
        y_train_pred_all = np.concatenate(y_train_pred_all, axis=0)
        # Calculate training R2
        train_r2 = r2_score(y_train_true_all, y_train_pred_all)
        history_train_r2.append(train_r2)

        # Evaluate accuracy at the end of each epoch
        model.eval()
        total_loss = 0
        y_true_all = []
        y_pred_all = []

        with torch.no_grad():
            for X_batch, y_batch in test_loader:
                y_pred = model(X_batch)
                loss = loss_fn(y_pred, y_batch)
                total_loss += loss.item()

                # Store true and predicted values for R2 calculation
```

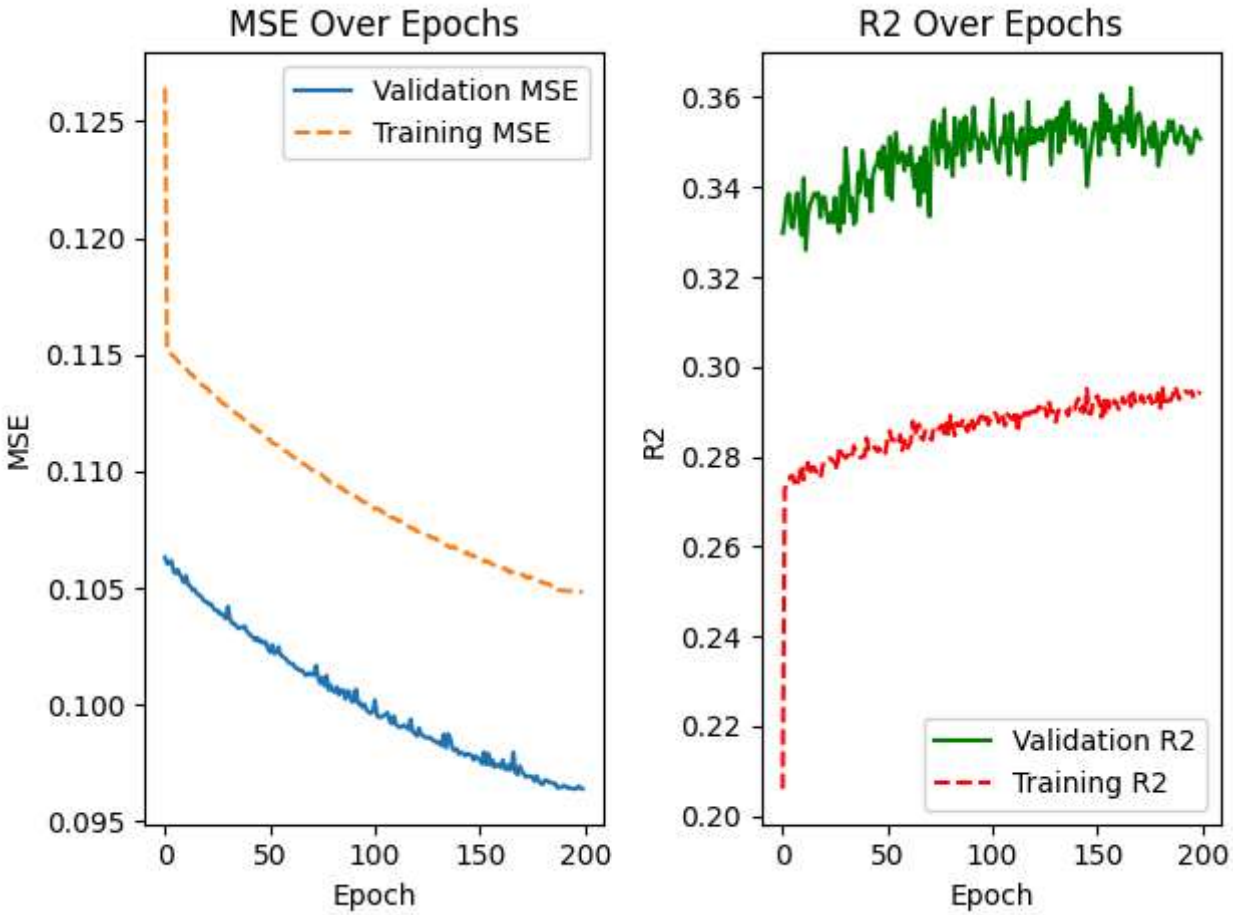
```
            y_true_all.append(y_batch.cpu().numpy())
            y_pred_all.append(y_pred.cpu().numpy())

    mse = total_loss / len(test_loader)
    mse = float(mse)
    history_mse.append(mse)

    # Flatten true and predicted values for R2 calculation
    y_true_all = np.concatenate(y_true_all, axis=0)
    y_pred_all = np.concatenate(y_pred_all, axis=0)
    # Calculate R2
    r2 = r2_score(y_true_all, y_pred_all)
    history_r2.append(r2)

    if mse < best_mse:
        best_mse = mse
        best_weights = copy.deepcopy(model.state_dict())
        epochs_without_improvement = 0
    else:
        epochs_without_improvement += 1
        # If no improvement for 'patience' epochs, stop training
        if epochs_without_improvement >= patience:
            print(f"Early stopping at epoch {epoch+1}")
            break

    # Step the scheduler to adjust the learning rate based on validation loss
    scheduler.step(mse)

# Restore model and return best accuracy
model.load_state_dict(best_weights)
print("Train MSE: %.2f" % best_train_mse)
print("Train RMSE: %.2f" % np.sqrt(best_train_mse))
print("Train R2: %.2f" % max(history_train_r2))
print("MSE: %.2f" % best_mse)
print("RMSE: %.2f" % np.sqrt(best_mse))
print("R2: %.2f" % max(history_r2))

# Plot MSE
plt.subplot(1, 2, 1)
plt.title('MSE Over Epochs')
plt.plot(history_mse, label='Validation MSE')
plt.plot(history_train_mse, label='Training MSE', linestyle='--')
```

```
    plt.xlabel('Epoch')
    plt.ylabel('MSE')
    plt.legend()

    # Plot R2
    plt.subplot(1, 2, 2)
    plt.title('R2 Over Epochs')
    plt.plot(history_r2, label='Validation R2', color='green')
    plt.plot(history_train_r2, label='Training R2', linestyle='--', color='red')
    plt.xlabel('Epoch')
    plt.ylabel('R2')
    plt.legend()
    plt.tight_layout()
    plt.show()

    return model
```

In [187…

```
lr_model = train(LinearRegressionModel, 200)
```

```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
LinearRegressionModel                    --
├─Linear: 1-1                            8
==========================================================================
Total params: 8
Trainable params: 8
Non-trainable params: 0
==========================================================================
Epoch 00187: reducing learning rate of group 0 to 2.0000e-04.
Train MSE: 0.10
Train RMSE: 0.32
Train R2: 0.30
MSE: 0.10
RMSE: 0.31
R2: 0.36
```

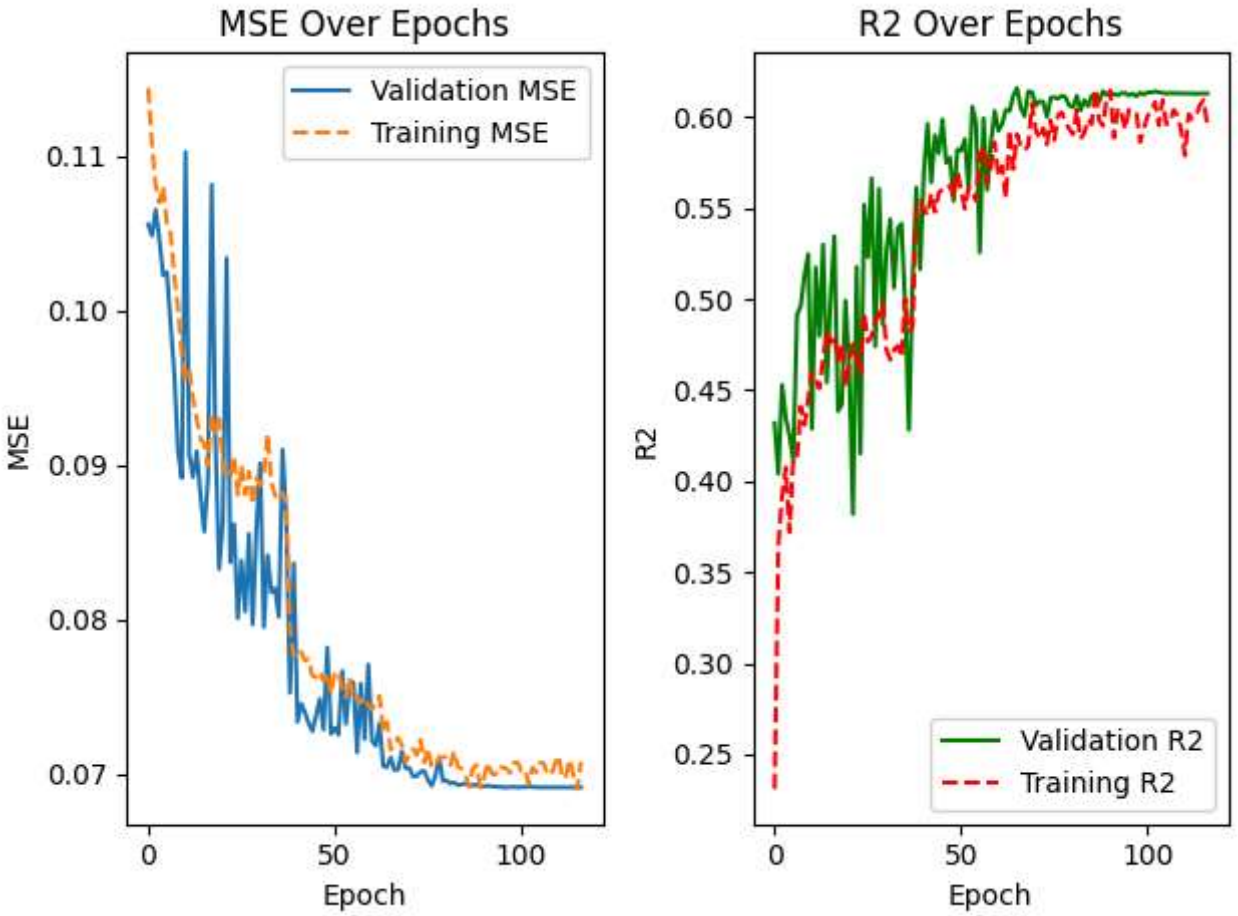


We can see that the simple linear regression model is too simple and underfitting for this dataset

However, using a MLP model with more layers will allow fine-tuning and improve performance

In [59]:
```
mlp_model = train(MLPRegressionModel, 200)
```

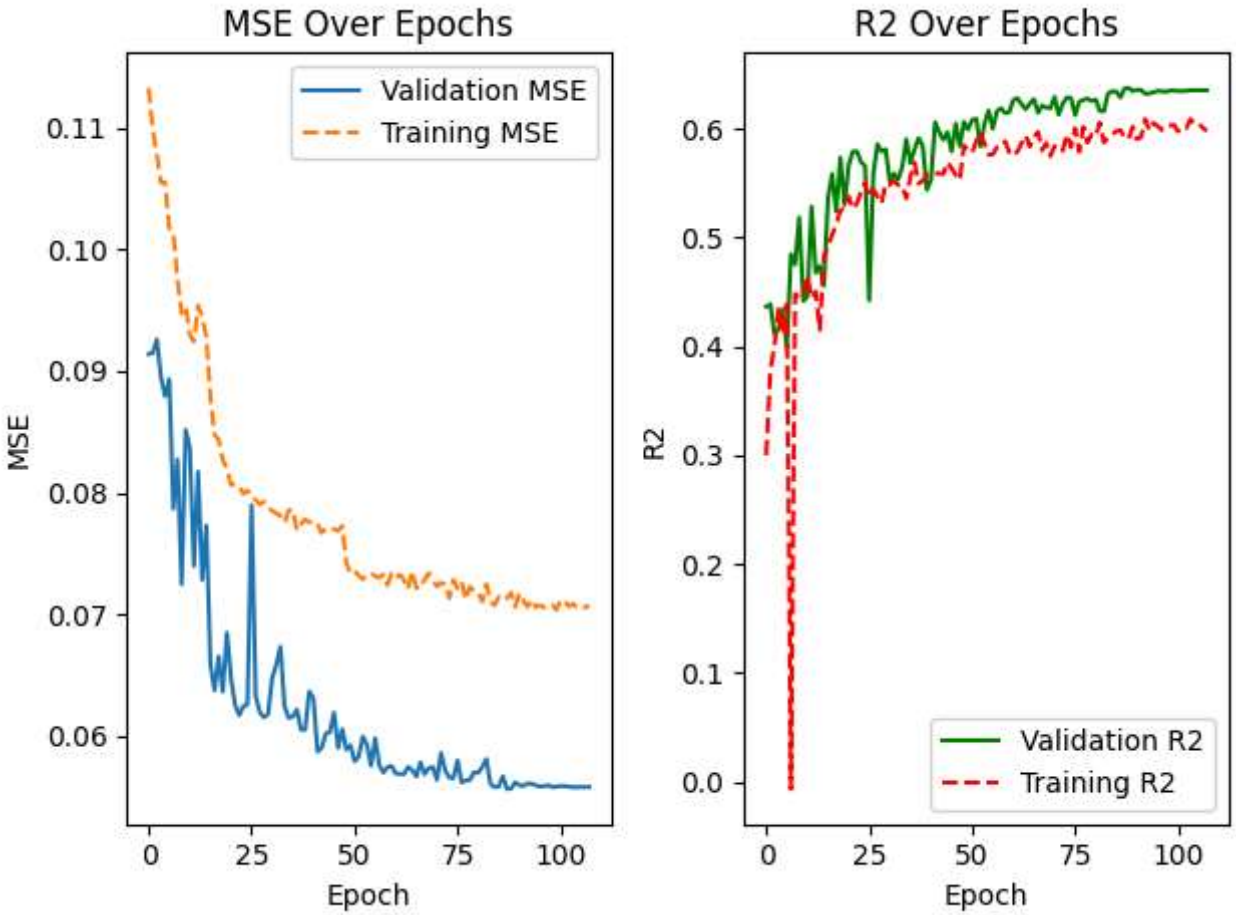
```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
MLPRegressionModel                       --
├─Sequential: 1-1                        --
│    └─Linear: 2-1                       4,096
│    └─ELU: 2-2                          --
│    └─Dropout: 2-3                      --
│    └─Linear: 2-4                       131,328
│    └─ELU: 2-5                          --
│    └─Dropout: 2-6                      --
│    └─Linear: 2-7                       32,896
│    └─ELU: 2-8                          --
│    └─Dropout: 2-9                      --
│    └─Linear: 2-10                      8,256
│    └─ELU: 2-11                         --
│    └─Dropout: 2-12                     --
│    └─Linear: 2-13                      2,080
│    └─ELU: 2-14                         --
│    └─Dropout: 2-15                     --
│    └─Linear: 2-16                      33
==========================================================================
Total params: 178,689
Trainable params: 178,689
Non-trainable params: 0
==========================================================================
Epoch 00038: reducing learning rate of group 0 to 2.0000e-04.
Epoch 00063: reducing learning rate of group 0 to 4.0000e-05.
Epoch 00083: reducing learning rate of group 0 to 8.0000e-06.
Epoch 00089: reducing learning rate of group 0 to 1.6000e-06.
Epoch 00103: reducing learning rate of group 0 to 3.2000e-07.
Epoch 00109: reducing learning rate of group 0 to 6.4000e-08.
Epoch 00115: reducing learning rate of group 0 to 1.2800e-08.
Early stopping at epoch 117
Train MSE: 0.07
Train RMSE: 0.26
Train R2: 0.62
MSE: 0.07
RMSE: 0.26
R2: 0.62
```



We will use the MLP model for deployment

In [56]:
```
X_train = torch.tensor(X, dtype=torch.float32)
y_train = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)
in_dims = X_train.shape[1]

# Create TensorDataset and DataLoader for batching (using 100% data for training)
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

# Create and train model
mlp_model = train(MLPRegressionModel, 200)
```

```
==========================================================================
Layer (type:depth-idx)                   Param #
==========================================================================
MLPRegressionModel                       --
├─Sequential: 1-1                        --
│    └─Linear: 2-1                       4,096
│    └─ELU: 2-2                          --
│    └─Dropout: 2-3                      --
│    └─Linear: 2-4                       131,328
│    └─ELU: 2-5                          --
│    └─Dropout: 2-6                      --
│    └─Linear: 2-7                       32,896
│    └─ELU: 2-8                          --
│    └─Dropout: 2-9                      --
│    └─Linear: 2-10                      8,256
│    └─ELU: 2-11                         --
│    └─Dropout: 2-12                     --
│    └─Linear: 2-13                      2,080
│    └─ELU: 2-14                         --
│    └─Dropout: 2-15                     --
│    └─Linear: 2-16                      33
==========================================================================
Total params: 178,689
Trainable params: 178,689
Non-trainable params: 0
==========================================================================
Epoch 00015: reducing learning rate of group 0 to 2.0000e-04.
Epoch 00048: reducing learning rate of group 0 to 4.0000e-05.
Epoch 00083: reducing learning rate of group 0 to 8.0000e-06.
Epoch 00094: reducing learning rate of group 0 to 1.6000e-06.
Epoch 00100: reducing learning rate of group 0 to 3.2000e-07.
Epoch 00106: reducing learning rate of group 0 to 6.4000e-08.
Early stopping at epoch 108
Train MSE: 0.07
Train RMSE: 0.27
Train R2: 0.61
MSE: 0.06
RMSE: 0.24
R2: 0.64
```


MSE Over Epochs
Validation MSE
Training MSE
MSE
0.11
0.10
0.09
0.08
0.07
0.06
0
25
50
75
100
Epoch
R2 Over Epochs
R2
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Validation R2
Training R2
0
25
50
75
100
Epoch

In [57]:

```
import onnx

mlp_model.eval()
torch.onnx.export(
    mlp_model,
    torch.randn(1, 7),
    "../app/my_model.onnx",
    export_params=True,
    opset_version=11,
    input_names=["input"],
    output_names=["output"],
    dynamic_axes={"input": {0: "batch_size"}, "output": {0: "batch_size"}}
)
```