Real Estate Investment Analysis for Zillow Data

#Project Overview

In this project, we aim to assist Zillow real-estate investment firm in identifying the top 5 states for investment based on historical housing data from Zillow. The firm seeks to maximize returns by strategically investing in areas with high potential for property value appreciation over the next few years. Our analysis will leverage time series forecasting techniques to predict future property prices and recommend the most promising states for investment.

#Business Problem

Zillow investment firm faces the challenge of selecting the most lucrative states for investment amidst fluctuating real estate markets. By utilizing historical data and advanced analytics, our goal is to provide actionable insights taking into account:

Profit potential: Expected price appreciation and potential rental income.

Risk: Volatility and stability of the price forecasts.

Time horizon: Appropriate time frame for the investment (e.g., short-term, long-term).

#Stakeholder

The primary stakeholder for this project is the investment committee of the Zillow real-estate investment firm. They are interested in understanding the historical trends, seasonality, and predictive patterns in housing market data to identify areas that offer the highest potential for property value growth and rental income.

#Objectives

Data Collection: Gather and preprocess historical housing data from Zillow, focusing on key metrics such as median home prices, rental yields, and historical price trends.

Exploratory Data Analysis (EDA): Explore the dataset to identify trends, seasonality, and correlations between variables. Visualize key metrics to gain insights into historical market behavior.

Time Series Modeling: Apply time series forecasting techniques (e.g., ARIMA) to model and predict future property prices for selected states.

Ranking and Recommendation: Develop a ranking system based on forecasted price appreciation, rental yields, and risk metrics to recommend the top 5 states for investment.

#Methodology

###Data Loading and Filtering:

Load the Zillow dataset and filter it for the states we are interested in.

###Data Preprocessing:

Convert the column names representing dates into datetime objects.

###Exploratory Data Analysis (EDA) and Visualization:

Visualize the historical price trends for the selected states. Reshape Data from Wide to Long Format:

Convert the data into a format suitable for time series modeling.

###ARIMA Modeling:

Fit ARIMA models to forecast future housing prices for each selected zip code.

###Interpreting Results:

Evaluate the forecasts and provide a detailed recommendation.

Profit Potential: Consider both the forecasted increase in housing prices and the confidence intervals.

Risk Assessment: Evaluate the stability and trends in the historical data.

Time Horizon: Assess the appropriate investment duration based on the forecasts and market trends.

Conclusion Summarize our findings and provide final thoughts on the investment opportunities in the selected states.

Data Understanding

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import scipy.stats as stats
import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from sklearn.metrics import mean_squared_error

excel_file_path = "/content/zillow_data.csv"

df = pd.read_csv(excel_file_path)


df.head()
```

{"type":"dataframe","variable_name":"df"}

Data Preprocessing

```
# Step 1: Handle missing values
# Check for missing values
missing_values = df.isnull().sum()

# Display columns with missing values
missing_values[missing_values > 0]
```

```
Metro      1043
1996-04    1039
1996-05    1039
1996-06    1039
1996-07    1039
           ...
2014-02      56
2014-03      56
2014-04      56
2014-05      56
2014-06      56
Length: 220, dtype: int64
```

```
# Define a threshold for dropping columns
threshold = 0.2 * len(df)

# Drop columns with more than 20% missing values
df_dropped = df.dropna(thresh=threshold, axis=1)

# Forward fill the remaining missing values
df_filled = df_dropped.fillna(method='ffill').fillna(method='bfill')

# Check if there are any remaining missing values
remaining_missing_values = df_filled.isnull().sum()
remaining_missing_values[remaining_missing_values > 0]
remaining_missing_values
```

```
RegionID      0
RegionName    0
City          0
State         0
Metro         0
             ..
2017-12       0
2018-01       0
2018-02       0
2018-03       0
2018-04       0
Length: 272, dtype: int64
```

```
df_filled.head()
```

{"type":"dataframe","variable_name":"df_filled"}

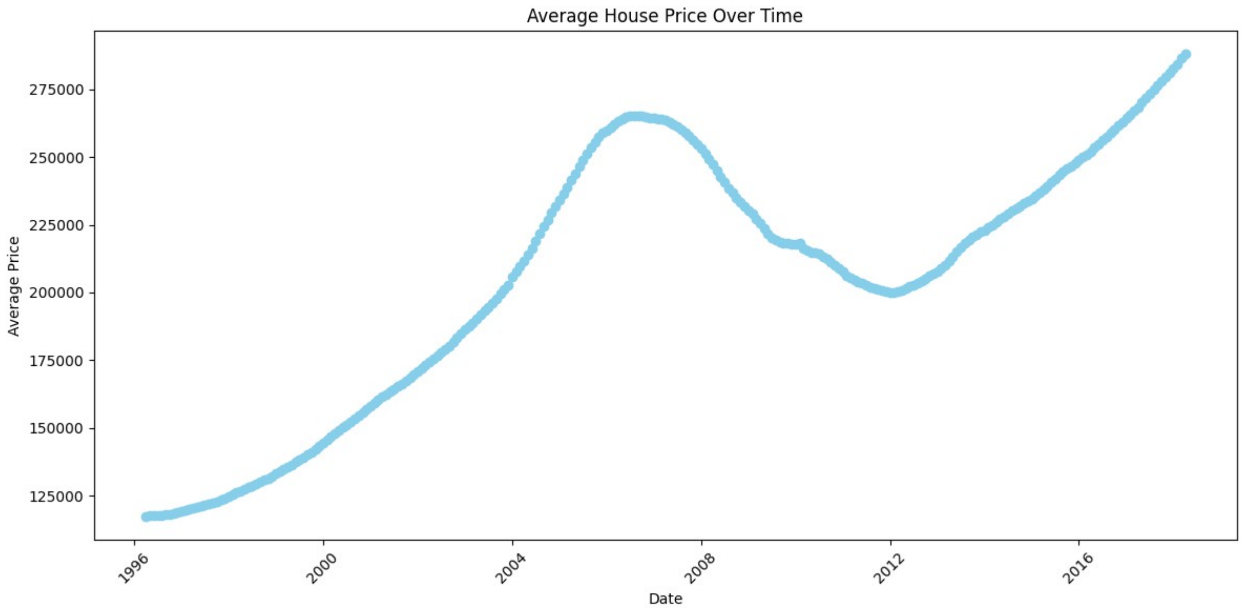
EDA and Visualization

Univariate analysis

```
# Melt the DataFrame to long format using date columns
df_melted = pd.melt(df_filled, value_vars=df_filled.columns,
var_name='Date', value_name='Price')
# Drop rows where 'Price' is NaN
df_melted = df_melted.dropna(subset=['Price'])

# Convert 'Price' to numeric if needed
df_melted['Price'] = pd.to_numeric(df_melted['Price'],
errors='coerce')

# Convert 'Date' to datetime format
df_melted['Date'] = pd.to_datetime(df_melted['Date'], format='%Y-%m',
errors='coerce')

# Remove rows where 'Date' is NaT (not a valid date)
df_melted = df_melted.dropna(subset=['Date'])

# Calculate average price for each date
avg_prices = df_melted.groupby('Date')['Price'].mean()

df_melted.head()
```

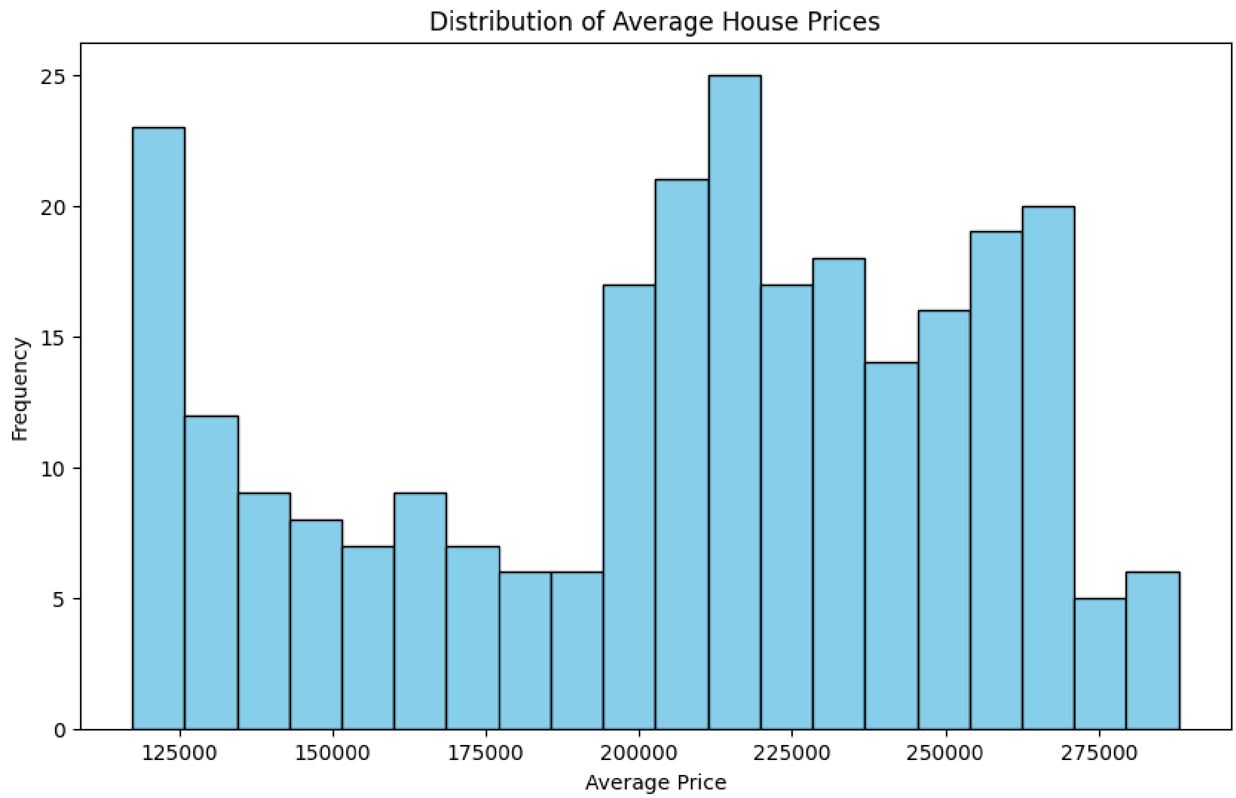
{"type":"dataframe","variable_name":"df_melted"}

```
# Plot line graph
plt.figure(figsize=(12, 6))
plt.plot(avg_prices.index, avg_prices.values, marker='o',
linestyle='-', color='skyblue')
plt.title('Average House Price Over Time')
plt.xlabel('Date')
plt.ylabel('Average Price')
plt.xticks(rotation=45)
plt.grid(False)
plt.tight_layout()
plt.show()
```

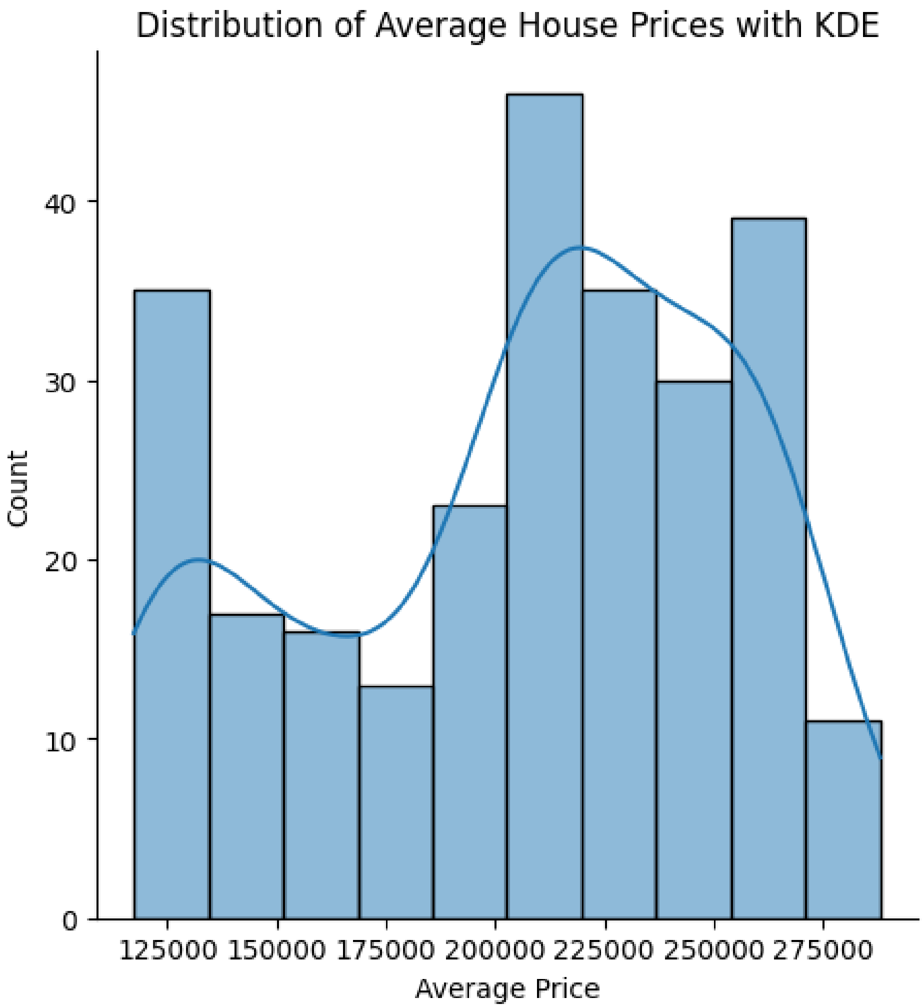


Shows a steady upward trend with a decline between 2008 and 2012 as a result of the financial crisis caused by the bursting of the housing market bubble.

```
# Plot histogram
plt.figure(figsize=(10, 6))
plt.hist(avg_prices.values, bins=20, color='skyblue',
edgecolor='black')
plt.title('Distribution of Average House Prices')
plt.xlabel('Average Price')
plt.ylabel('Frequency')
plt.grid(False)
plt.show()

# kernel density plot, use Seaborn
sns.displot(avg_prices.values, kde=True)
plt.title('Distribution of Average House Prices with KDE')
plt.xlabel('Average Price')
plt.show()
```



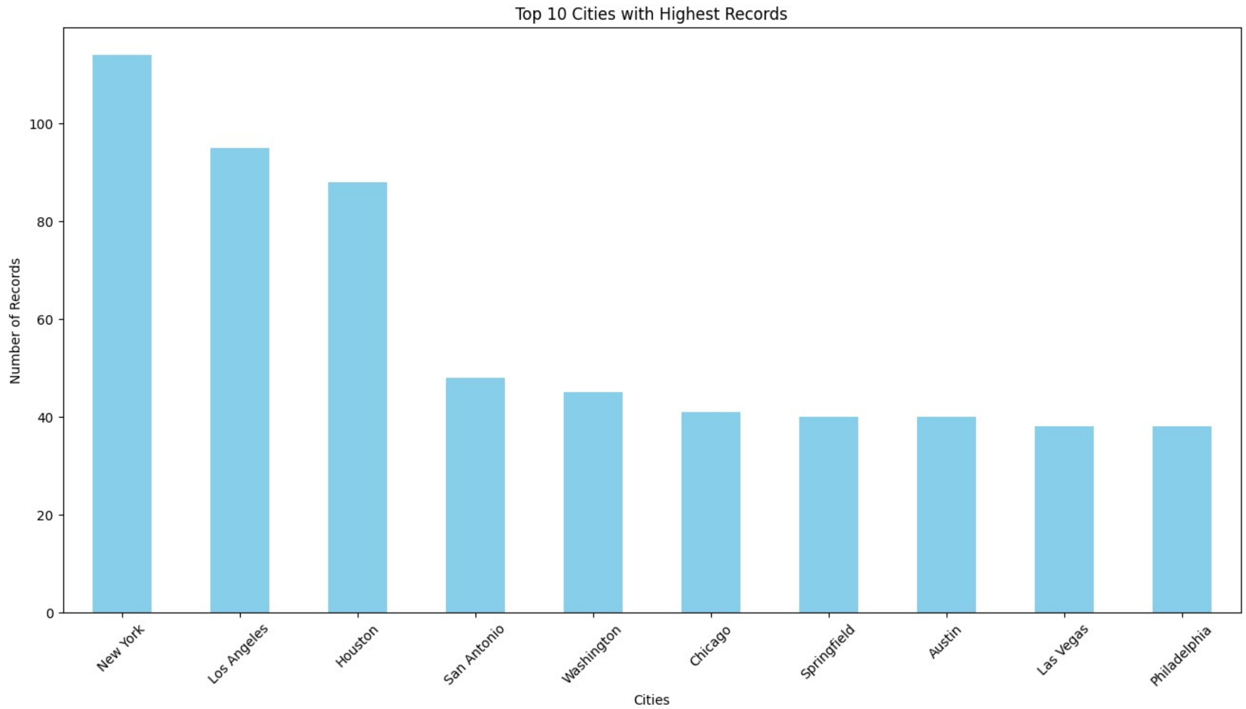
Distribution of Average House Prices
Frequency
Average Price
0
5
10
15
20
25
125000
150000
175000
200000
225000
250000
275000

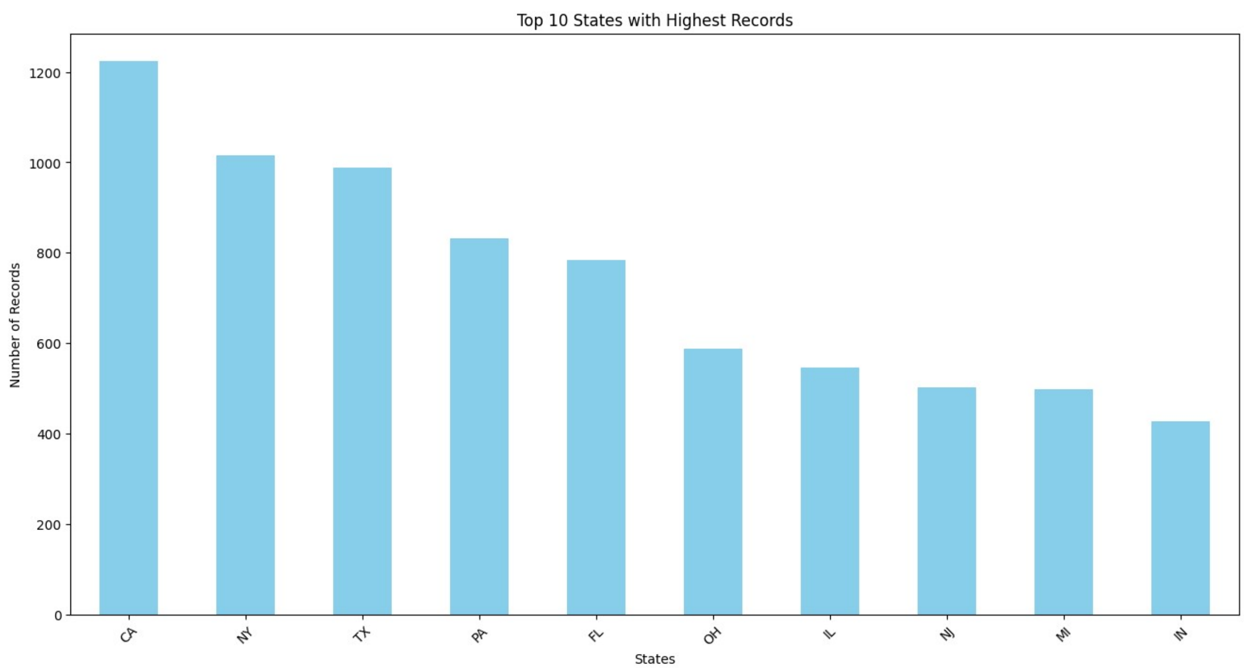


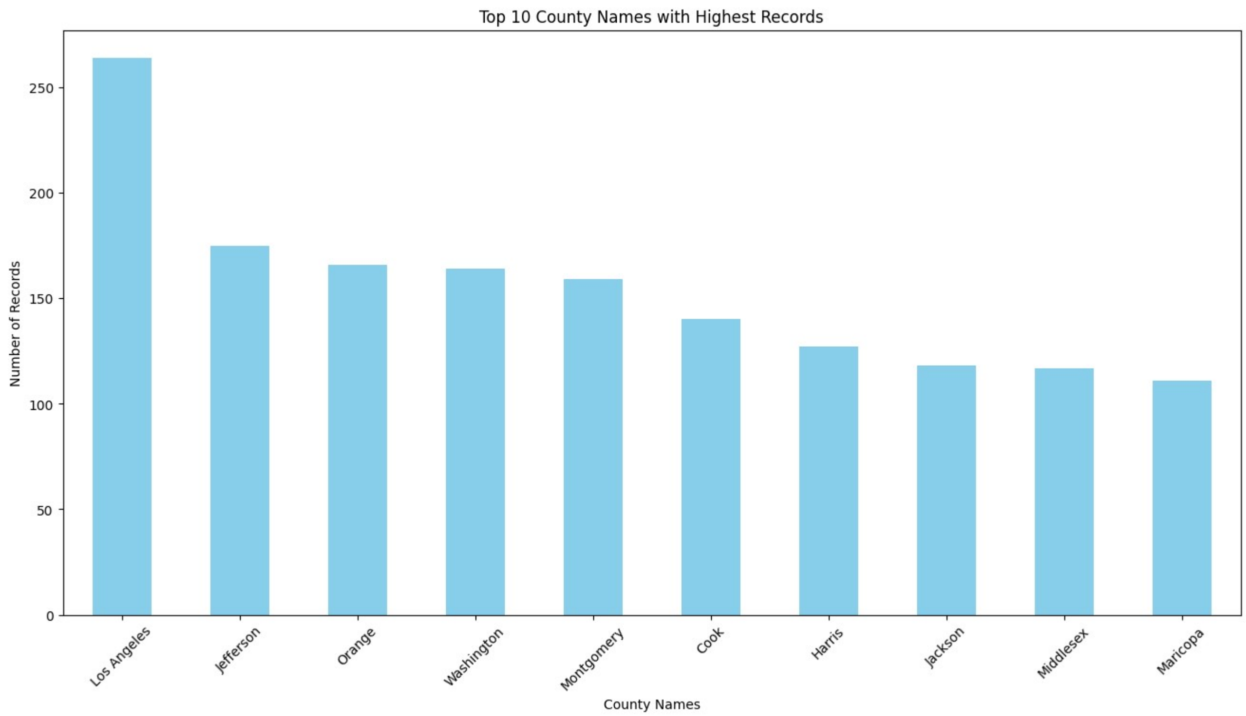
The average house prices are not evenly distributed. There are more houses in the middle price range, and fewer houses that are very expensive or very inexpensive.

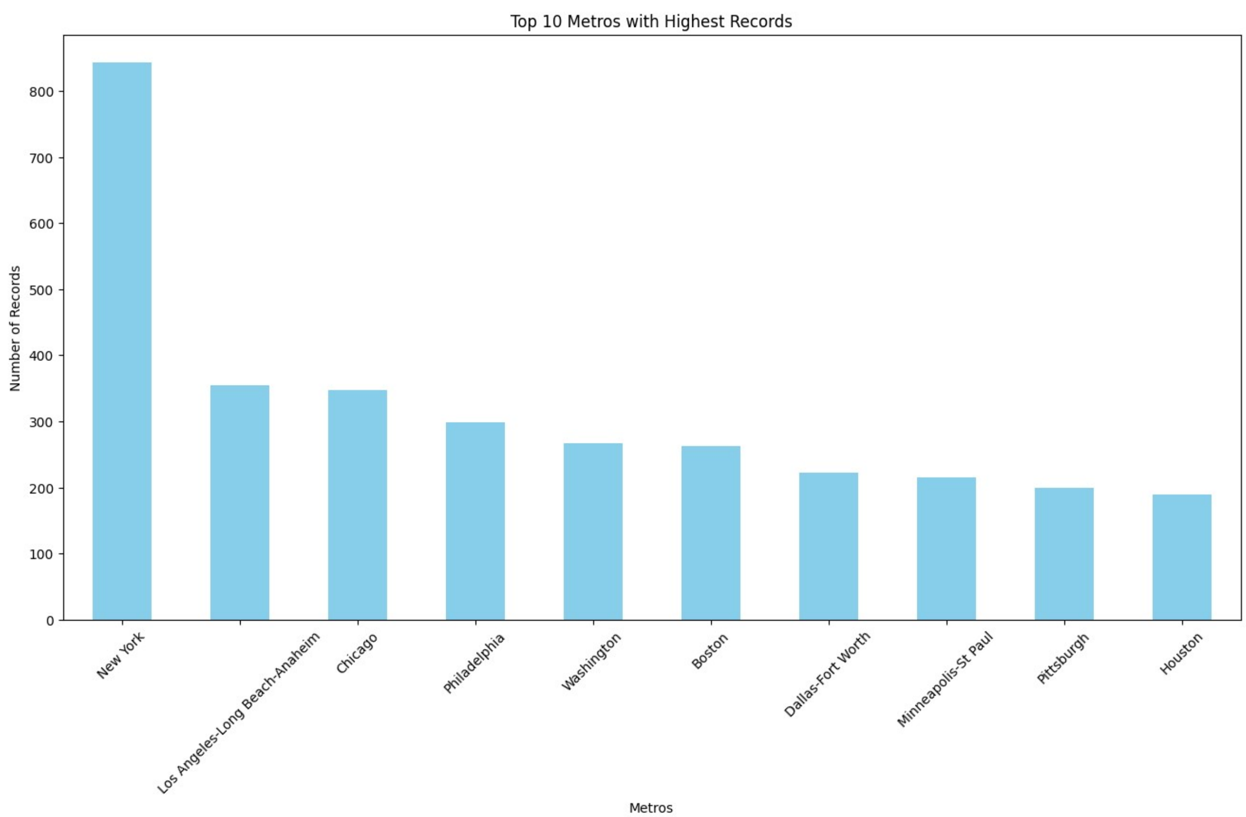
```
def plot_top_10(df, column_name, title):
    counts = df[column_name].value_counts().head(10)
    plt.figure(figsize=(16, 8))
    counts.plot(kind='bar', color='skyblue')
    plt.title(f'Top 10 {title} with Highest Records')
    plt.xlabel(title)
    plt.ylabel('Number of Records')
    plt.xticks(rotation=45)
    plt.show()

# Plot for City
plot_top_10(df_filled, 'City', 'Cities')

# Plot for State
plot_top_10(df_filled, 'State', 'States')
```

```python
# Plot for CountyName
plot_top_10(df_filled, 'CountyName', 'County Names')

# Plot for Metro
plot_top_10(df_filled, 'Metro', 'Metros')
```









New York, California and the cities/counties within them generally have a high region count. Possibly indicating a high population.

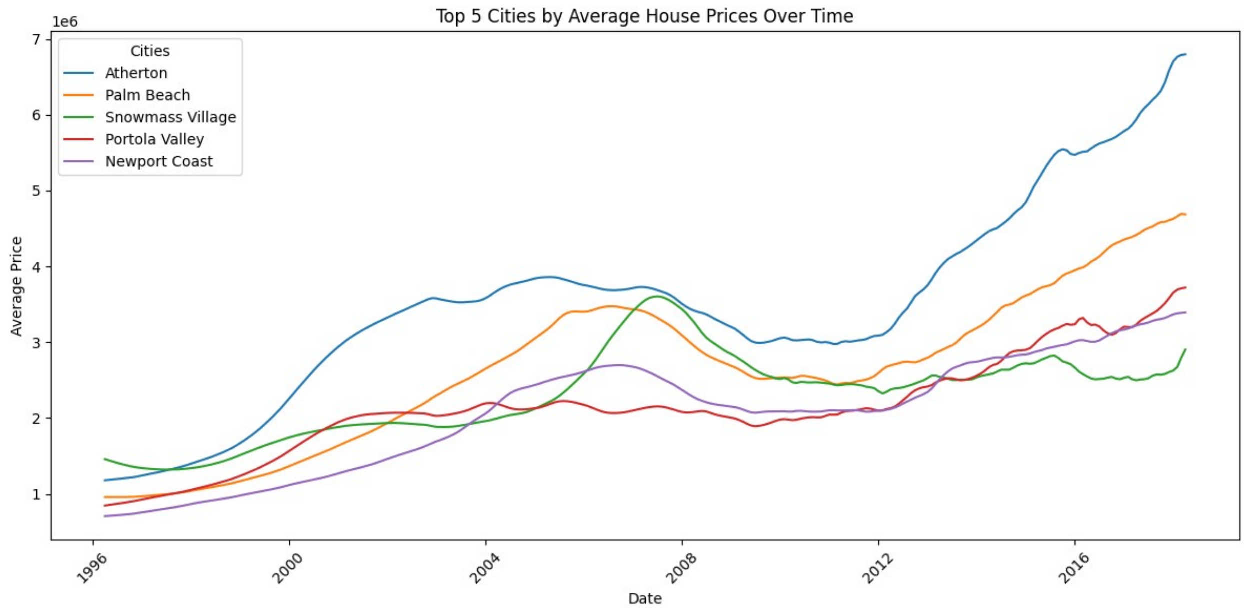
Bivariate analysis

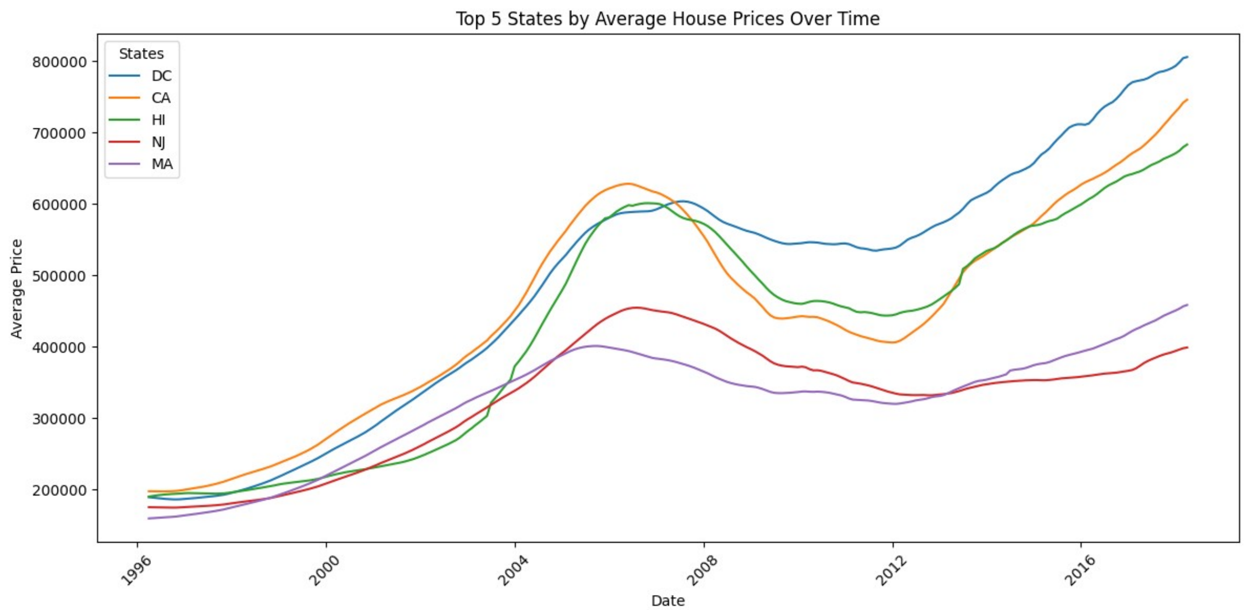
```
# Identify date columns
date_columns = [col for col in df_filled.columns if '-' in col and
col[:4].isdigit()]

# Melt the DataFrame to long format using date columns
df_melted = pd.melt(df_filled, id_vars=['City', 'State', 'CountyName',
'Metro'],
                    value_vars=date_columns, var_name='Date',
value_name='Price')

# Drop rows where 'Price' is NaN
df_melted = df_melted.dropna(subset=['Price'])

# Convert 'Price' to numeric if needed
df_melted['Price'] = pd.to_numeric(df_melted['Price'],
errors='coerce')

# Convert 'Date' to datetime format
df_melted['Date'] = pd.to_datetime(df_melted['Date'], format='%Y-%m')

df_melted.head()
```

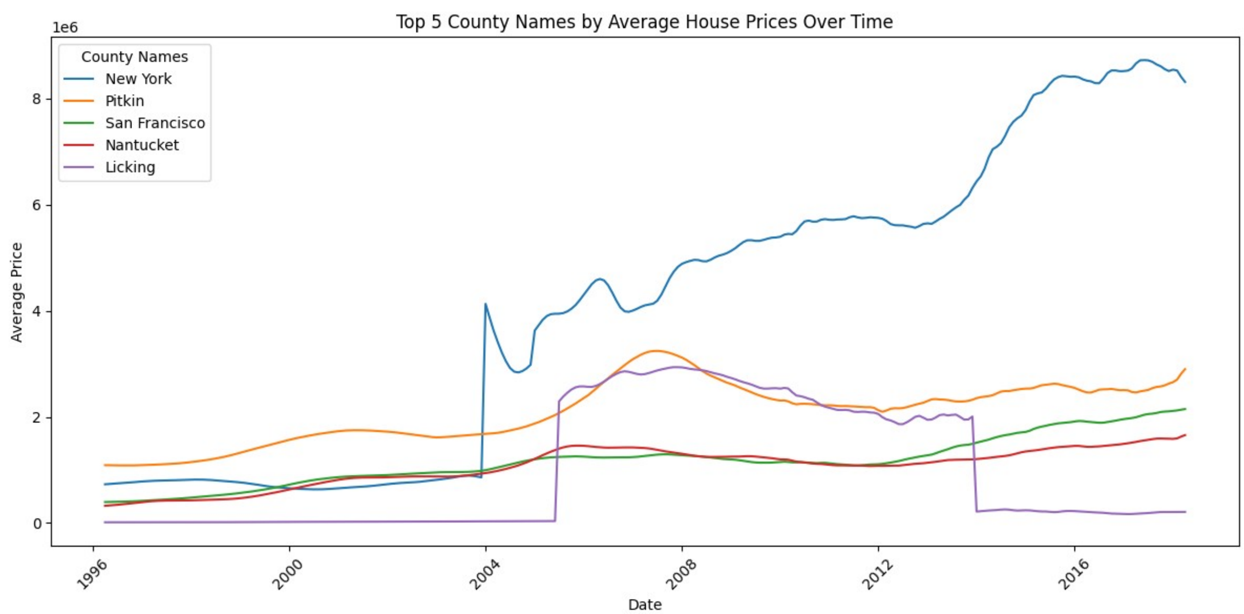
{"type":"dataframe","variable_name":"df_melted"}

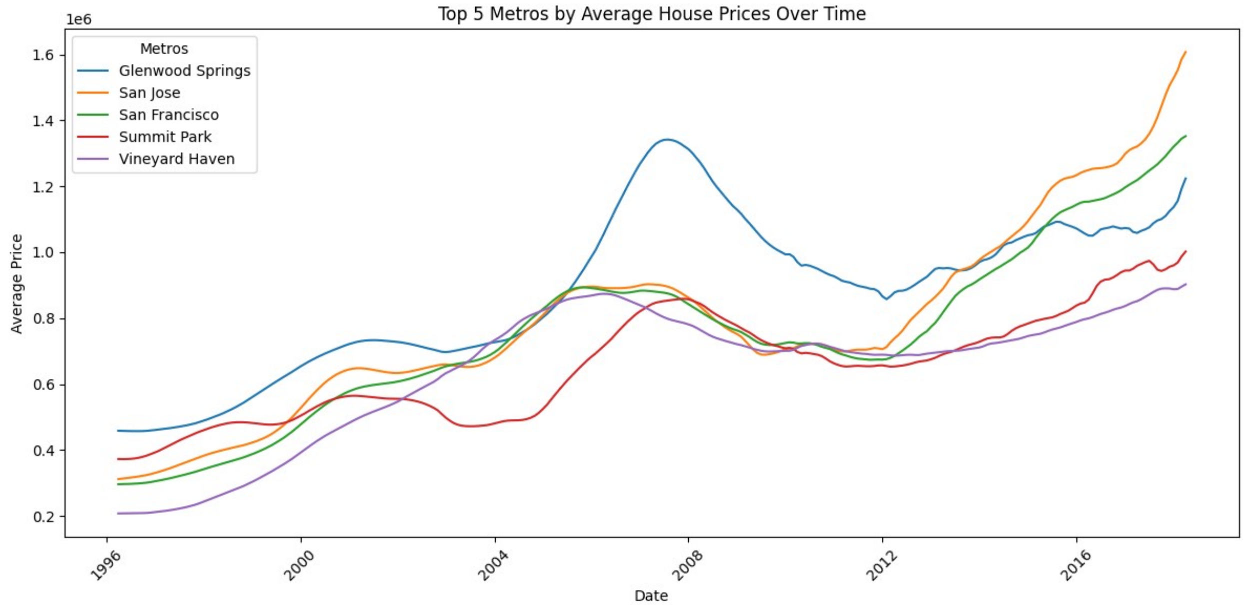
```
# Create a function to plot line graphs for average prices in
different columns
def plot_top_5_prices(df, group_by_column, title):
    # Calculate average prices for each group and date
    avg_prices = df.groupby([group_by_column, 'Date'])
['Price'].mean().reset_index()

    # Find top 5 groups by average price over all dates
    top_5_groups = avg_prices.groupby(group_by_column)
['Price'].mean().sort_values(ascending=False).head(5).index

    # Filter to only include top 5 groups
    top_5_avg_prices =
avg_prices[avg_prices[group_by_column].isin(top_5_groups)]

    # Plot line graph
    plt.figure(figsize=(12, 6))  # Adjust figure size as needed
    for group in top_5_groups:
        group_data =
top_5_avg_prices[top_5_avg_prices[group_by_column] == group]
        plt.plot(group_data['Date'], group_data['Price'], label=group)

    plt.title(f'Top 5 {title} by Average House Prices Over Time')
    plt.xlabel('Date')
    plt.ylabel('Average Price')
```

```
    plt.legend(title=title)
    plt.xticks(rotation=45)
    plt.grid(False)
    plt.tight_layout()  # Ensures tight layout to prevent clipping
    plt.show()


# Plot for City
plot_top_5_prices(df_melted, 'City', 'Cities')

# Plot for State
plot_top_5_prices(df_melted, 'State', 'States')

# Plot for CountyName
plot_top_5_prices(df_melted, 'CountyName', 'County Names')

# Plot for Metro
plot_top_5_prices(df_melted, 'Metro', 'Metros')
```



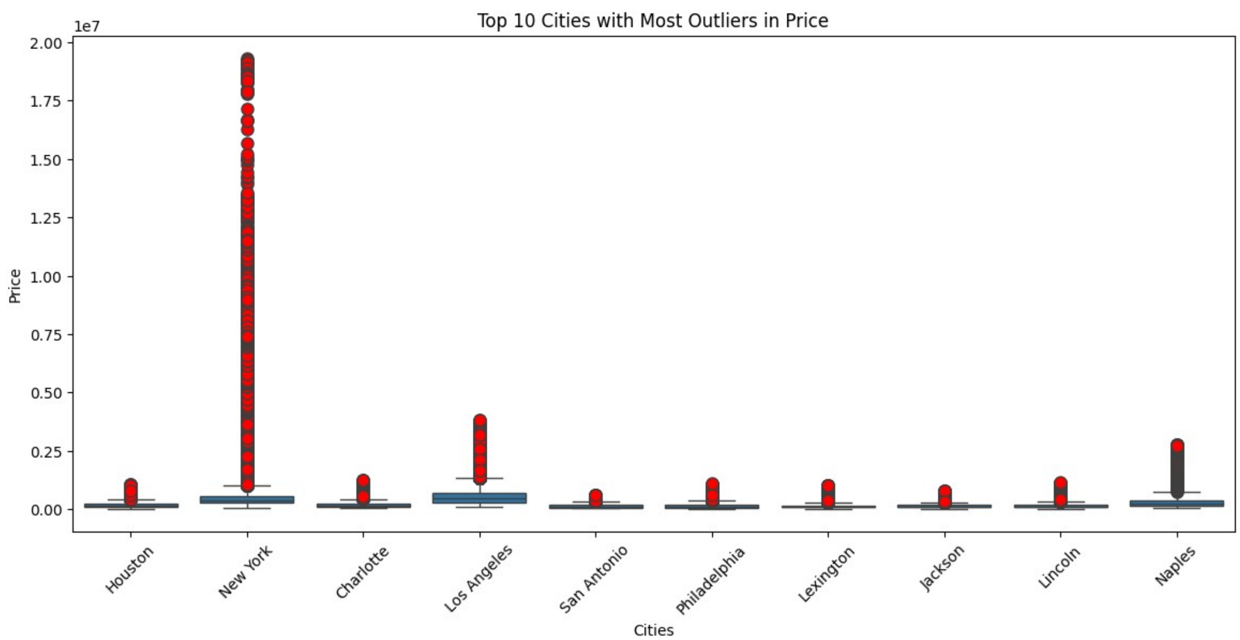

Top 5 States by Average House Prices Over Time
States
DC
CA
HI
NJ
MA
800000
700000
600000
500000
400000
300000
200000
Average Price
1996
2000
2004
2008
2012
2016
Date

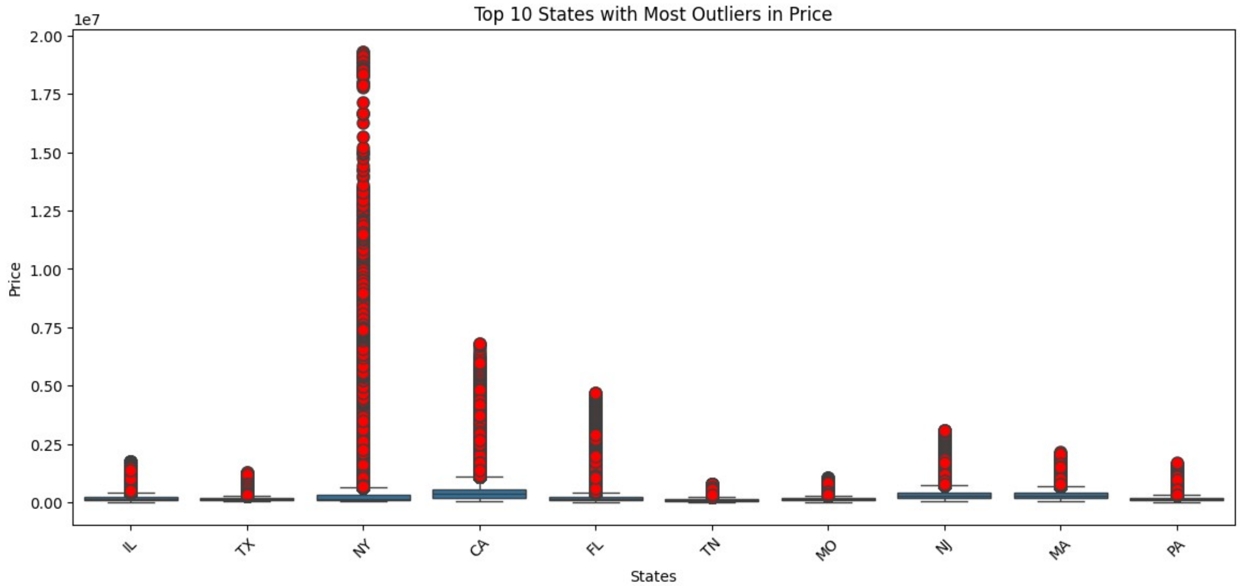

Top 5 County Names by Average House Prices Over Time
1e6
County Names
New York
Pitkin
San Francisco
Nantucket
Licking
8
6
4
2
0
Average Price
1996
2000
2004
2008
2012
2016
Date

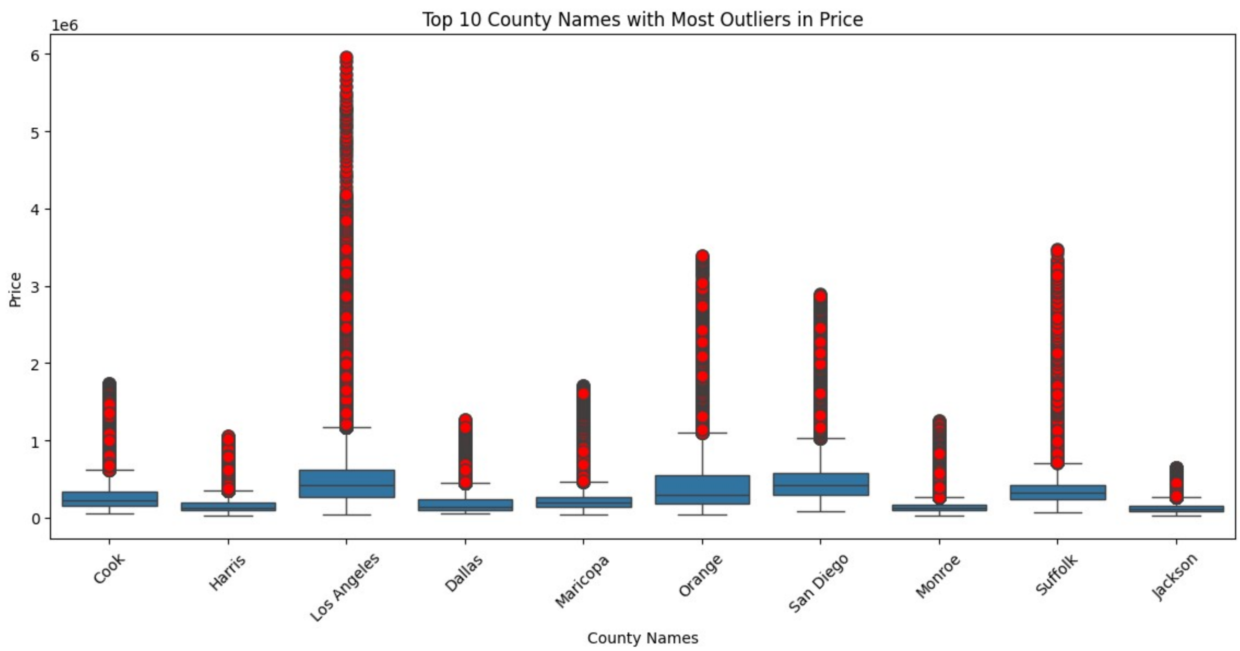


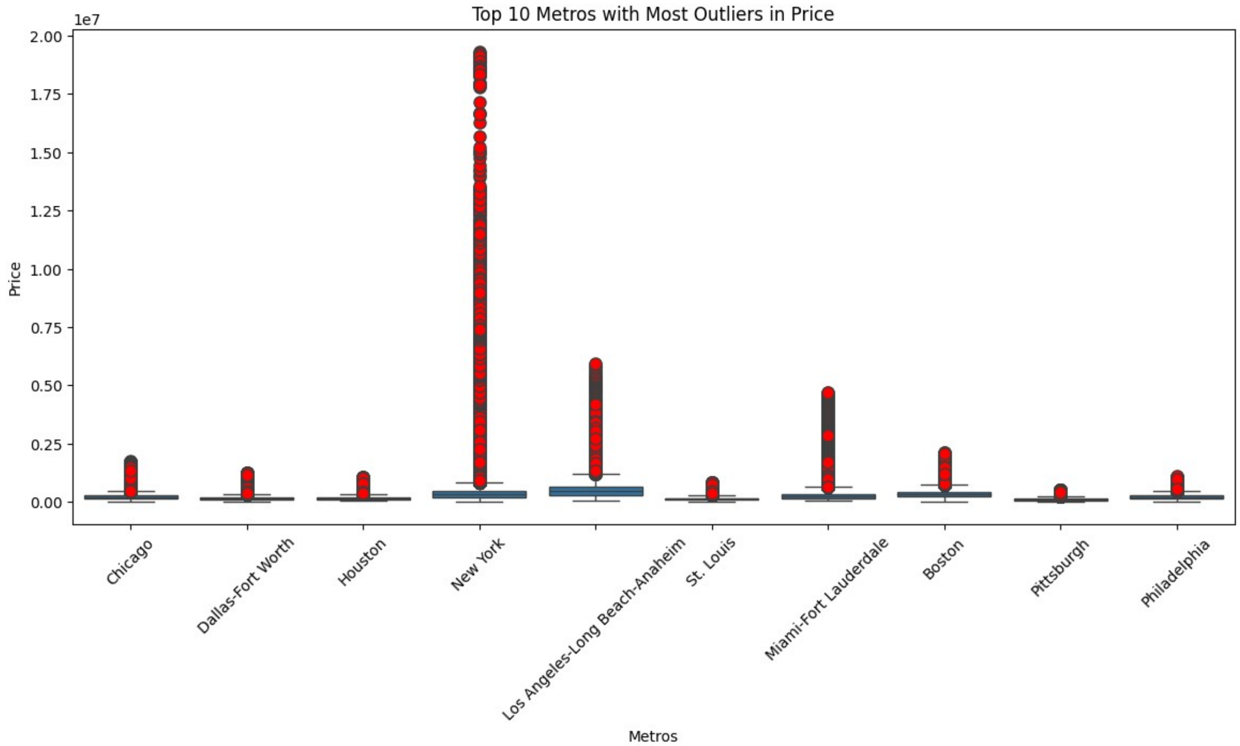

```
def plot_top_10_outliers(df, group_by_column, value_column, title):
    """
    Plots boxplots for the top 10 groups with the most outliers in the
given value column.

    Args:
        df (pd.DataFrame): The DataFrame containing the data.
        group_by_column (str): The column to group the data by.
        value_column (str): The column containing the values to plot.
        title (str): The title of the plot.
    """

    # Calculate the number of outliers for each group
    outlier_counts = df.groupby(group_by_column)
[value_column].apply(lambda x: (np.abs(x - x.mean()) > 4 *
x.std()).sum())

    # Get the top 10 groups with the most outliers
    top_10_groups =
outlier_counts.sort_values(ascending=False).head(10).index

    # Filter the DataFrame to include only the top 10 groups
    df_filtered = df[df[group_by_column].isin(top_10_groups)]

    # Plot the boxplots
    plt.figure(figsize=(14, 6))
    # Removed the unsupported 'show_fliers' argument and used
'flierprops' instead.
    # 'flierprops' allows for customization of outlier marker
appearance.
    sns.boxplot(x=group_by_column, y=value_column, data=df_filtered,
```

```
flierprops={'marker': 'o', 'markerfacecolor': 'red', 'markersize': 8})
    plt.title(f'Top 10 {title} with Most Outliers in {value_column}')
    plt.xlabel(title)
    plt.ylabel(value_column)
    plt.xticks(rotation=45)
    plt.show()

# Example usage:
plot_top_10_outliers(df_melted, 'City', 'Price', 'Cities')
plot_top_10_outliers(df_melted, 'State', 'Price', 'States')
plot_top_10_outliers(df_melted, 'CountyName', 'Price', 'County Names')
plot_top_10_outliers(df_melted, 'Metro', 'Price', 'Metros')
```




Top 10 States with Most Outliers in Price
1e7
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Price
IL
TX
NY
CA
FL
TN
MO
NJ
MA
PA
States


Top 10 County Names with Most Outliers in Price
1e6
6
5
4
3
2
1
0
Price
Cook
Harris
Los Angeles
Dallas
Maricopa
Orange
San Diego
Monroe
Suffolk
Jackson
County Names



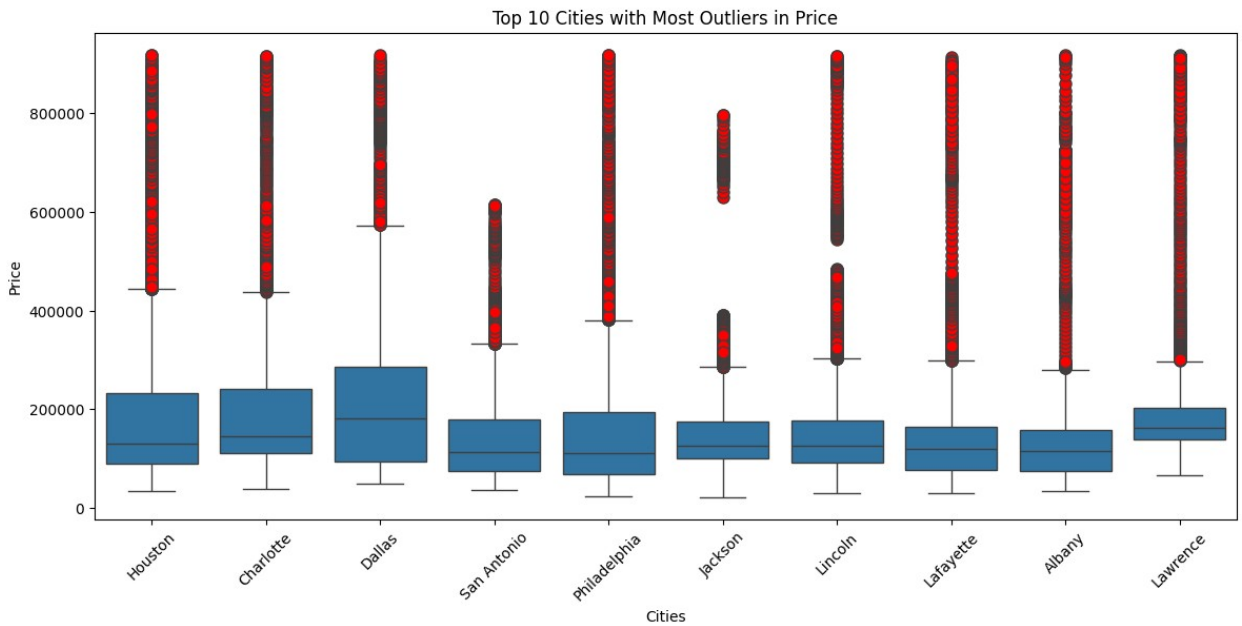
New York and California/Los Angeles feature again with a high number of outliers indicating high end properties in the areas and a rich population.

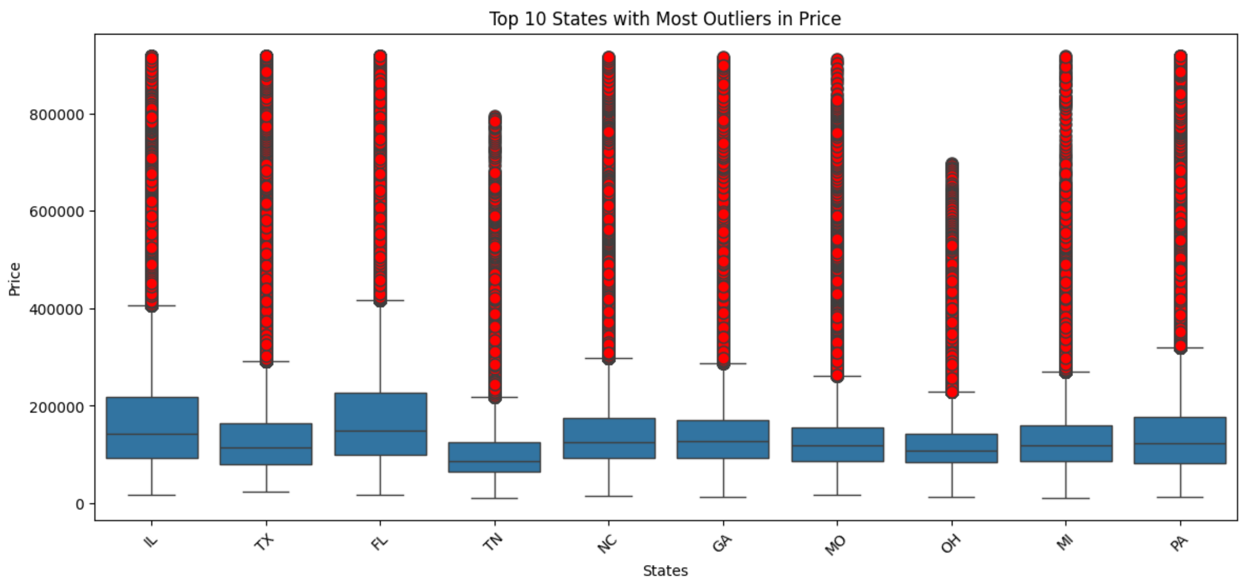
```
df_melted.columns

Index(['City', 'State', 'CountyName', 'Metro', 'Date', 'Price'],
dtype='object')

def drop_numerical_outliers(df, z_thresh=3):
    numeric_cols = df.select_dtypes(include=[np.number]).columns

    # Apply Z-score filter to the selected columns
    constrains = df[numeric_cols].apply(lambda x:
np.abs(stats.zscore(x)) < z_thresh).all(axis=1)

    # Drop the rows that do not satisfy the constraints
    df.drop(df.index[~constrains], inplace=True)
    return df

 df_no_outliers = drop_numerical_outliers(df_melted.copy())

df_no_outliers.head()

{"type":"dataframe","variable_name":"df_no_outliers"}

print("Before dropping numerical outliers, length of the dataframe is:
", len(df_melted))
```

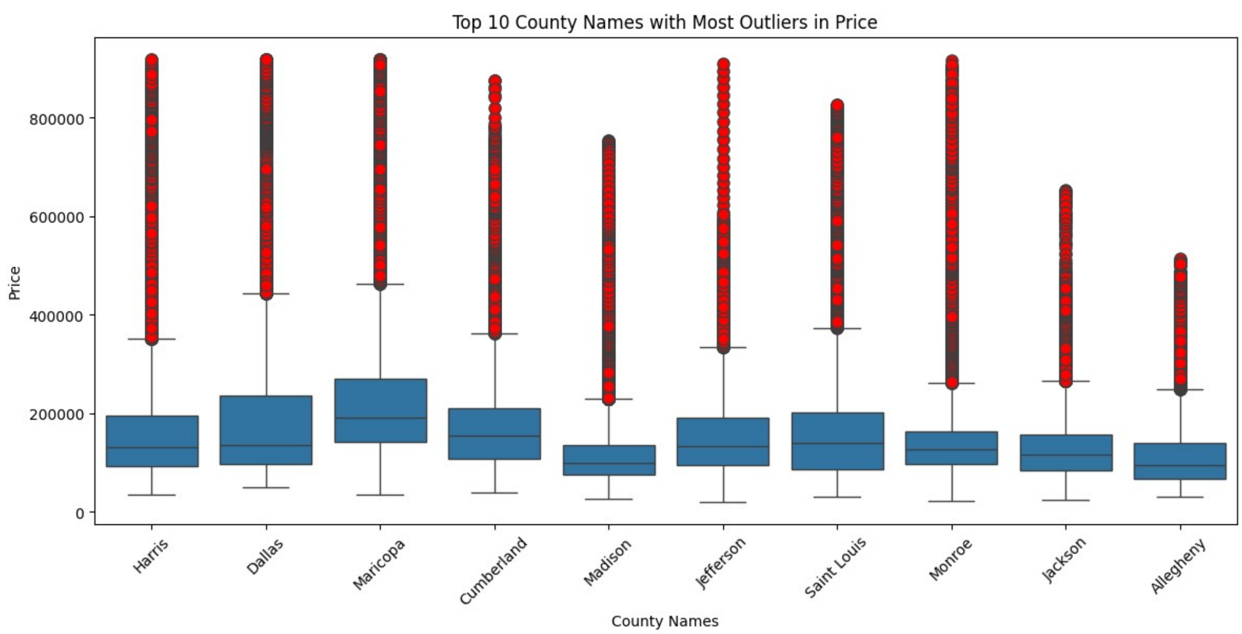
```
print("After dropping numerical outliers, length of the dataframe is:
", len(df_no_outliers))
```

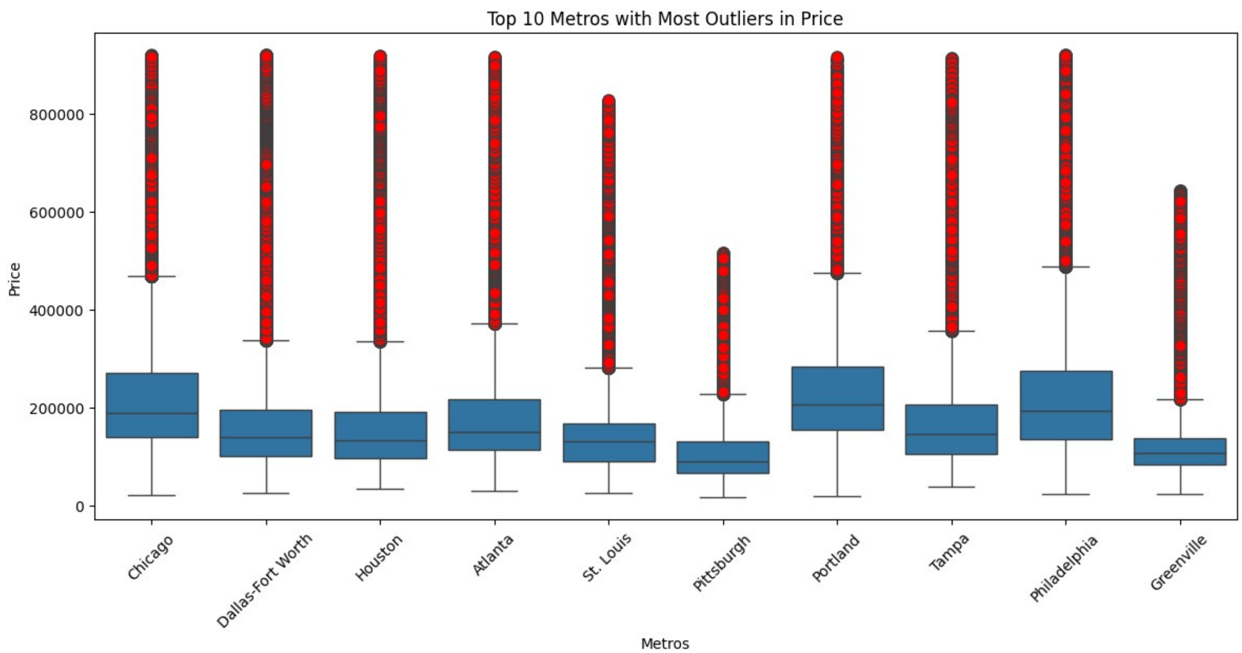
```
Before dropping numerical outliers, length of the dataframe is:
3901595
After dropping numerical outliers, length of the dataframe is:
3854307
```

```
plot_top_10_outliers(df_no_outliers, 'City', 'Price', 'Cities')
plot_top_10_outliers(df_no_outliers, 'State', 'Price', 'States')
plot_top_10_outliers(df_no_outliers, 'CountyName', 'Price', 'County
Names')
plot_top_10_outliers(df_no_outliers, 'Metro', 'Price', 'Metros')
```




Top 10 States with Most Outliers in Price
Price
800000
600000
400000
200000
0
IL
TX
FL
TN
NC
GA
MO
OH
MI
PA
States

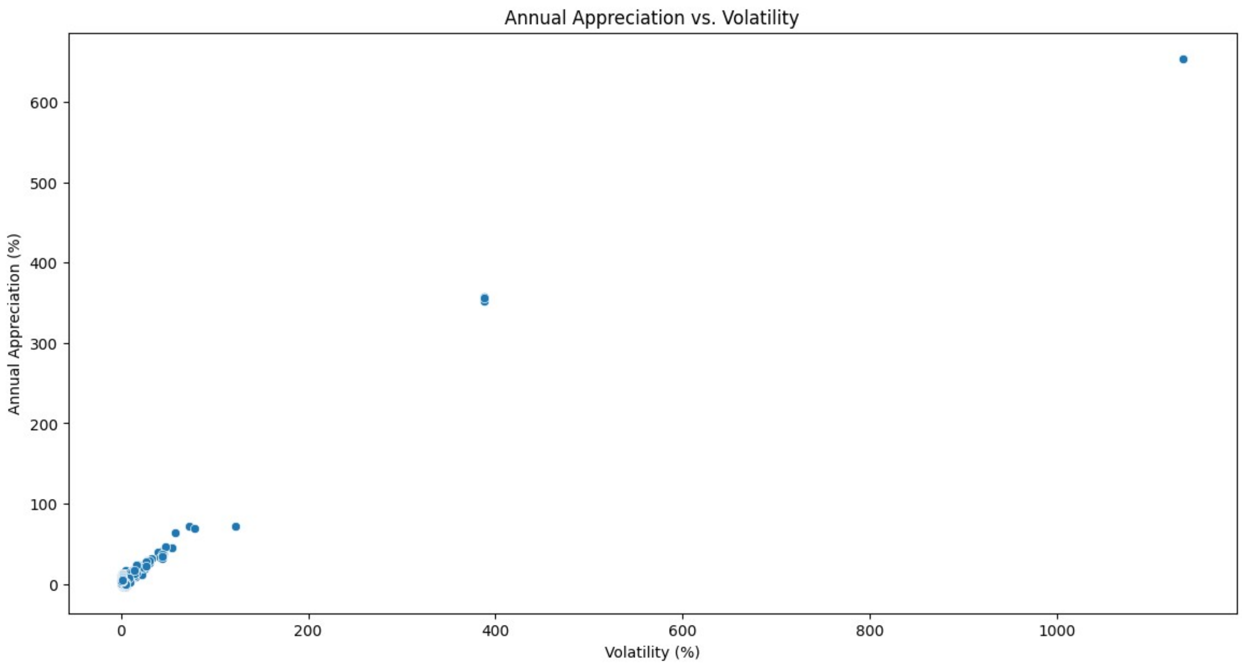

Top 10 County Names with Most Outliers in Price
Price
800000
600000
400000
200000
0
Harris
Dallas
Maricopa
Cumberland
Madison
Jefferson
Saint Louis
Monroe
Jackson
Allegheny
County Names



The dataset contains fat tails showing a high number of extreme high prices and low prices that are hard to eliminate with a z-score threshold.

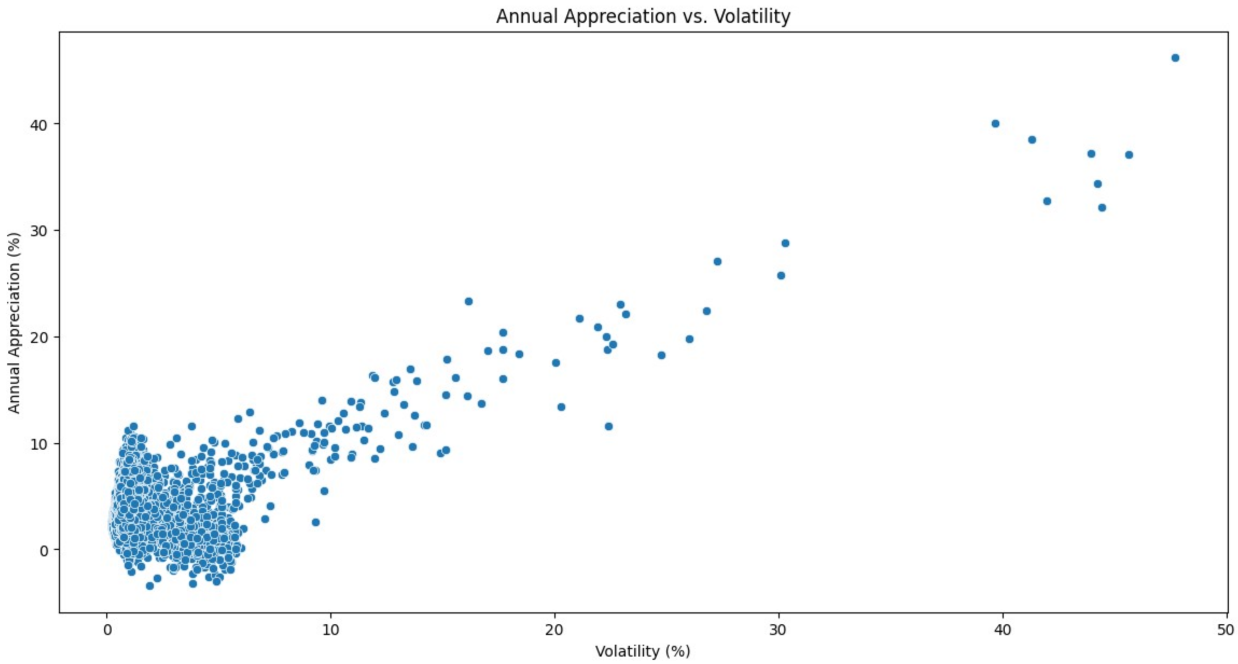
```
# Redefine price_data
price_data = df_filled.set_index(['RegionID', 'RegionName', 'City',
'State', 'Metro', 'CountyName', 'SizeRank']).T.iloc[7:]

# Calculate annual appreciation rates
annual_appreciation_rates = price_data.pct_change(periods=12).mean() *
100

# Calculate the standard deviation of monthly price changes
(volatility)
volatility = price_data.pct_change().std() * 100

# Combine into a single dataframe
metrics_df = pd.DataFrame({
    'RegionName':
annual_appreciation_rates.index.get_level_values('RegionName'),
    'AnnualAppreciation': annual_appreciation_rates.values,
    'Volatility': volatility.values
})

# Bivariate plot: Annual appreciation vs. volatility
plt.figure(figsize=(14, 7))
sns.scatterplot(x='Volatility', y='AnnualAppreciation',
data=metrics_df)
plt.xlabel('Volatility (%)')
plt.ylabel('Annual Appreciation (%)')
```

```
plt.title('Annual Appreciation vs. Volatility')
plt.show()
```




```
# Df without outliers
df_filled_no_outliers = drop_numerical_outliers(df_filled.copy())

# Redefine price_data
price_data = df_filled_no_outliers.set_index(['RegionID',
'RegionName', 'City', 'State', 'Metro', 'CountyName',
'SizeRank']).T.iloc[7:]

# Calculate annual appreciation rates
annual_appreciation_rates = price_data.pct_change(periods=12).mean() *
100

# Calculate the standard deviation of monthly price changes
(volatility)
volatility = price_data.pct_change().std() * 100

# Combine into a single dataframe
metrics_df = pd.DataFrame({
    'RegionName':
annual_appreciation_rates.index.get_level_values('RegionName'),
    'AnnualAppreciation': annual_appreciation_rates.values,
    'Volatility': volatility.values
})

# Bivariate plot: Annual appreciation vs. volatility
plt.figure(figsize=(14, 7))
```

```
sns.scatterplot(x='Volatility', y='AnnualAppreciation',
data=metrics_df)
plt.xlabel('Volatility (%)')
plt.ylabel('Annual Appreciation (%)')
plt.title('Annual Appreciation vs. Volatility')
plt.show()
```



Price vs. Time:

There is a strong positive correlation between time and median sales prices across most zip codes. Some zip codes exhibit stronger growth trends compared to others.

Price vs. Region:

Certain regions, especially major metropolitan areas, consistently show higher median prices. Suburban and rural areas tend to have lower median prices but may show significant growth potential.

Price vs. SizeRank:

There is a noticeable correlation between SizeRank and median prices, with higher-ranked areas often having higher prices. This could indicate that more populous areas, which tend to have higher SizeRanks, are more valuable.

Price vs. Volatility:

Zip codes with higher median prices tend to show more volatility. This could indicate that high-value areas are subject to more significant fluctuations in market conditions.

Reshape from Wide to Long Format

```
def melt_data(df_filled):
    """
    Takes the zillow_data dataset in wide form or a subset of the
zillow_dataset.
    Returns a long-form datetime dataframe
    with the datetime column names as the index and the values as the
'values' column.

    If more than one row is passes in the wide-form dataset, the
values column
    will be the mean of the values from the datetime columns in all of
the rows.
    """

    melted = pd.melt(df_filled, id_vars=['RegionName', 'RegionID',
'SizeRank', 'City', 'State', 'Metro', 'CountyName'], var_name='time')
    melted['time'] = pd.to_datetime(melted['time'])
    melted = melted.dropna(subset=['value'])
    return melted.groupby('time').aggregate({'value':'mean'})

df_filled1=melt_data(df_filled)
df_filled1.head()
```


```
{"summary":"{\n  \"name\": \"df_filled1\",\n  \"rows\": 265,\n
\"fields\": [\n    {\n      \"column\": \"time\",\n
\"properties\": {\n        \"dtype\": \"date\",\n        \"min\":
\"1996-04-01 00:00:00\",\n        \"max\": \"2018-04-01 00:00:00\",\n
\"num_unique_values\": 265,\n        \"samples\": [\n          \"2011-
03-01 00:00:00\",\n          \"2005-11-01 00:00:00\",\n
\"2004-04-01 00:00:00\"\n        ],\n        \"semantic_type\": \"\",\
n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"value\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 47951.50634487974,\n        \"min\":
117348.28499626435,\n        \"max\": 288039.9443048292,\n
\"num_unique_values\": 265,\n        \"samples\": [\n
205358.9417917544,\n          257485.77056306458,\n
211695.59872308632\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    }\n  ]\
n}","type":"dataframe","variable_name":"df_filled1"}
```

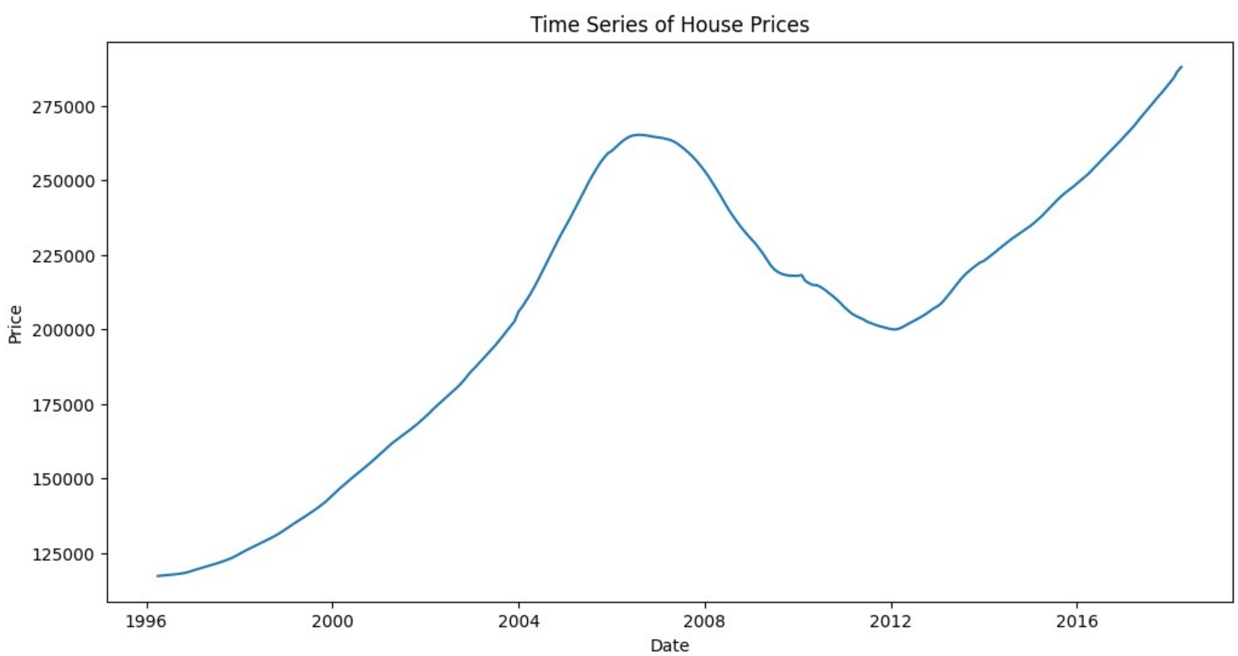

ARIMA Modeling

Stationarity Check and Differencing

1st Differencing

```
# Plot the time series
plt.figure(figsize=(12, 6))
```

```
plt.plot(df_filled1)
plt.title('Time Series of House Prices')
plt.xlabel('Date')
plt.ylabel('Price')
plt.show()

# Perform the Augmented Dickey-Fuller test
result = adfuller(df_filled1)
print(f'ADF Statistic: {result[0]}')
print(f'p-value: {result[1]}')

# If p-value > 0.05, the series is non-stationary and we need to
difference it
if result[1] > 0.05:
    ts_diff = df_filled1.diff().dropna()
    result = adfuller(ts_diff)
    print(f'ADF Statistic after differencing: {result[0]}')
    print(f'p-value after differencing: {result[1]}')
else:
    ts_diff = df_filled1
```




```
ADF Statistic: -1.923520236111722
p-value: 0.3210786192180571
ADF Statistic after differencing: -1.4088495412230682
p-value after differencing: 0.5780262607919437
```

Second Differencing

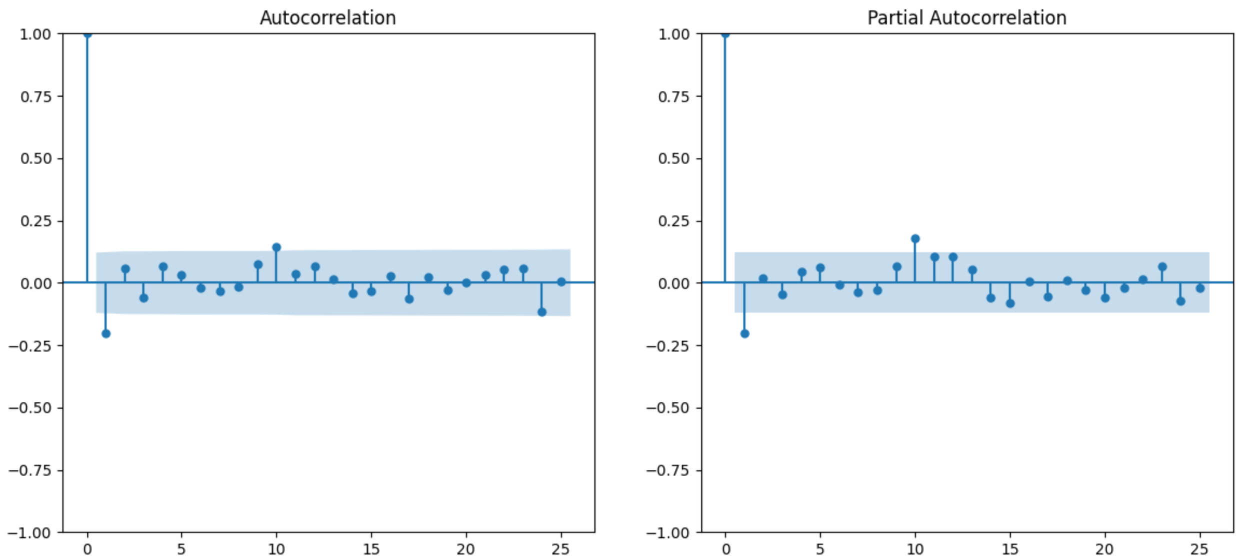
```
# Perform second differencing
ts_diff2 = df_filled1.diff().diff().dropna()

# Perform the Augmented Dickey-Fuller test on the twice differenced
series
result = adfuller(ts_diff2)
print(f'ADF Statistic after second differencing: {result[0]}')
print(f'p-value after second differencing: {result[1]}')

# Plot the twice differenced series
plt.figure(figsize=(12, 6))
plt.plot(ts_diff2)
plt.title('Twice Differenced Time Series of House Prices')
plt.xlabel('Date')
plt.ylabel('Price')
plt.show()

ADF Statistic after second differencing: -19.622168841937764
p-value after second differencing: 0.0
```

The 2nd differencing is as close to white noise model as we get.

Determine ARIMA Parameters (p, d, q)

```
# Plot ACF and PACF to determine p and q
plt.figure(figsize=(14, 6))
plt.subplot(121)
plot_acf(ts_diff2, ax=plt.gca())
plt.subplot(122)
```

```
plot_pacf(ts_diff2, ax=plt.gca())
plt.show()
```



Fit the ARIMA Model

```
# Convert the index to quarterly frequency ('Q')
df_filled1.index = pd.DatetimeIndex(df_filled1.index)
df_filled1 = df_filled1.to_period('Q')

# Define the ARIMA model with the correct order and frequency
model = ARIMA(df_filled1, order=(1, 2, 1))
model_fit = model.fit()

# Summary of the model
print(model_fit.summary())
```

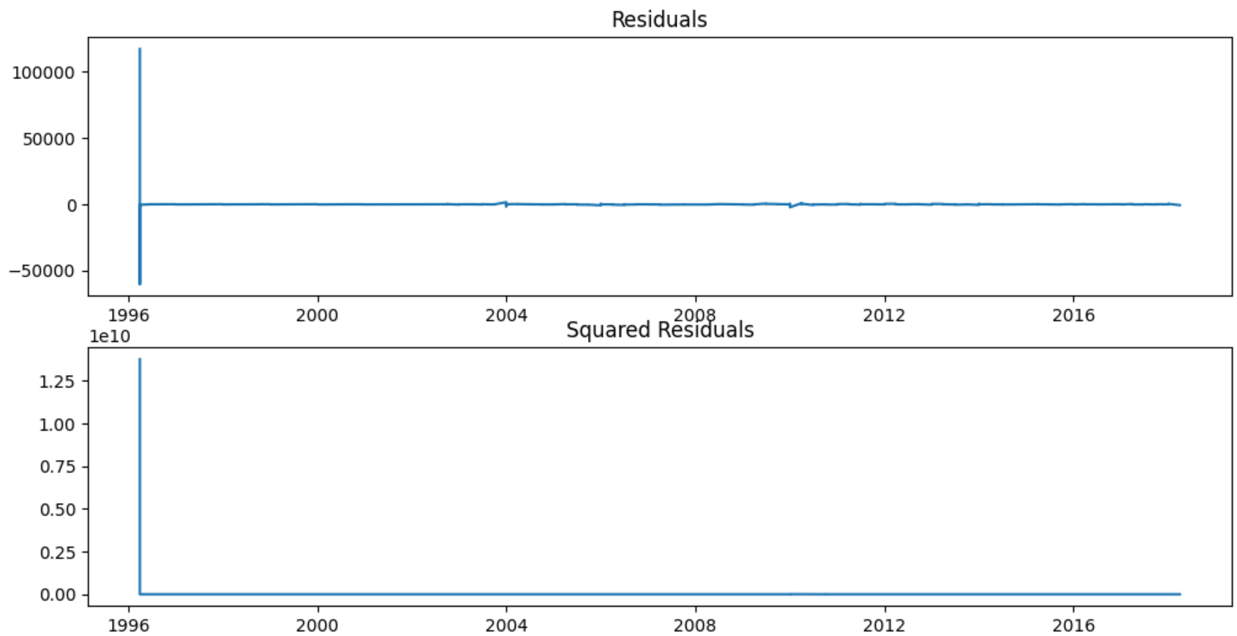
```
                               SARIMAX Results

======================================================================
========
Dep. Variable:                  value   No. Observations:
265
Model:                 ARIMA(1, 2, 1)   Log Likelihood                 -
1829.066
Date:                Wed, 03 Jul 2024   AIC
3664.131
Time:                        00:55:22   BIC
3674.848
Sample:                    06-30-1996   HQIC
3668.438
                         - 06-30-2018
```

```
Covariance Type:                  opg

======================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
----------------------------------------------------------------------
--------
ar.L1         -0.3828      0.615     -0.622      0.534      -1.589
0.823
ma.L1          0.3319      0.621      0.534      0.593      -0.885
1.549
sigma2      6.332e+04   1490.737     42.477      0.000    6.04e+04
6.62e+04
======================================================================
=============
Ljung-Box (L1) (Q):                   5.89   Jarque-Bera (JB):
10222.19
Prob(Q):                              0.02   Prob(JB):
0.00
Heteroskedasticity (H):               4.53   Skew:
-2.10
Prob(H) (two-sided):                  0.00   Kurtosis:
33.25
======================================================================
=============

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).
```

Model Diagnostics

```
# Plot residuals to check for white noise
residuals = model_fit.resid

# Convert PeriodIndex to datetime
residuals.index = residuals.index.to_timestamp()

# Ensure that residuals are numeric
residuals_numeric = residuals.astype(float)

# Plotting
plt.figure(figsize=(12, 6))

# Plot the residuals
plt.subplot(211)
plt.plot(residuals.index, residuals_numeric)
plt.title('Residuals')
```

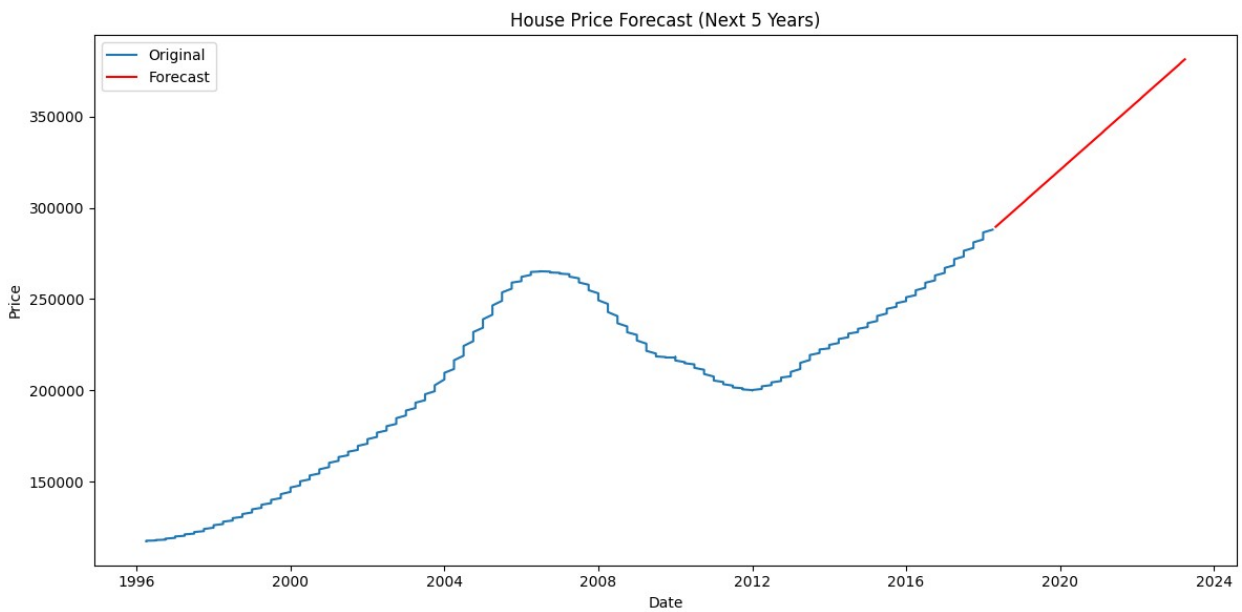
```python
# Plot the squared residuals
plt.subplot(212)
plt.plot(residuals.index, residuals_numeric**2)
plt.title('Squared Residuals')

plt.show()
```



Forecasting

```python
forecast_steps = 5 * 12  # 5 years * 12 months/year

# Generate the forecast
forecast = model_fit.forecast(steps=forecast_steps)

# Create a date range for the forecast that starts immediately after
the last historical date
# Convert last_historical_date to Timestamp before adding Timedelta
last_historical_date = df_filled1.index[-1].to_timestamp()
forecast_index = pd.date_range(start=last_historical_date +
pd.Timedelta(days=1), periods=forecast_steps, freq='M')

# Assign the new index to the forecast
forecast.index = forecast_index

# Ensure that dataframes are numeric if necessary
forecast_numeric = forecast.astype(float)
df_filled1_numeric = df_filled1.astype(float)

# Plot the forecast with adjusted figure size and layout
plt.figure(figsize=(12, 6))
```

```
# Convert the PeriodIndex to a datetime index for plotting
plt.plot(df_filled1_numeric.index.to_timestamp(), df_filled1_numeric,
label='Original')
plt.plot(forecast_numeric, label='Forecast', color='red')
plt.title('House Price Forecast (Next 5 Years)')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.tight_layout()
plt.show()
```



Interpretation

Model Summary The SARIMAX model was applied to the real estate price data, and the results are summarized below:

1. Model: ARIMA(1, 2, 1)
2. Number of Observations: 265
3. Log Likelihood: -1829.066
4. AIC (Akaike Information Criterion): 3664.131
5. BIC (Bayesian Information Criterion): 3674.848
6. HQIC (Hannan-Quinn Information Criterion): 3668.438 **Parameter Estimates**
7. AR(1) Coefficient: -0.3828 (std err: 0.615, z: -0.622, p: 0.534)
8. MA(1) Coefficient: 0.3319 (std err: 0.621, z: 0.534, p: 0.593)
9. Sigma^2 (Variance of the error term): 6.332e+04 (std err: 1490.737, z: 42.477, p: 0.000) **Diagnostic Tests**
10. Ljung-Box (L1) Test (Q): 5.89 (Prob(Q): 0.02)
11. Jarque-Bera (JB) Test: 10222.20 (Prob(JB): 0.00)

12. Heteroskedasticity (H) Test: 4.53 (Prob(H) (two-sided): 0.00)
13. Skewness: -2.10
14. Kurtosis: 33.25

Model Fit:

1. The model is an ARIMA(1, 2, 1), indicating that it includes one autoregressive term, two differencing steps, and one moving average term.
2. The low p-values for the Ljung-Box and Jarque-Bera tests indicate that there are some issues with residual autocorrelation and non-normality of residuals, respectively. This may suggest that the model could be improved.

Parameter Significance:

1. The p-values for the AR(1) and MA(1) coefficients are 0.534 and 0.593, respectively, suggesting that these parameters are not statistically significant. This could imply that the model might not be capturing the underlying patterns effectively.

Variance:

1. The sigma^2 value is high, indicating significant variability in the data, which is to be expected in real estate prices.

Diagnostics:

1. Ljung-Box Test: The p-value of 0.02 suggests that there may be some autocorrelation in the residuals, indicating that the model does not fully capture the time series structure.
2. Jarque-Bera Test: The high JB statistic with a p-value of 0.00 indicates that the residuals are not normally distributed, which might affect forecast accuracy.
3. Heteroskedasticity Test: The significant p-value suggests heteroskedasticity in the residuals, meaning that the variance of the residuals changes over time.
4. Skewness and Kurtosis: The negative skewness and high kurtosis indicate that the residuals are not normally distributed and have heavy tails.

```
# Identify date columns
date_columns = [col for col in df_filled.columns if '-' in col and
col[:4].isdigit()]

# Melt the DataFrame to long format using date columns
df_melted1 = pd.melt(df_filled, id_vars=['RegionName', 'City',
'State', 'CountyName', 'Metro'],
                    value_vars=date_columns, var_name='Date',
value_name='Price')

# Drop rows where 'Price' is NaN
df_melted1 = df_melted1.dropna(subset=['Price'])

# Convert 'Price' to numeric if needed
df_melted1['Price'] = pd.to_numeric(df_melted1['Price'],
errors='coerce')
```

```python
# Convert 'Date' to datetime format
df_melted1['Date'] = pd.to_datetime(df_melted1['Date'], format='%Y-
%m')

# Group by 'State' and calculate the average price, filtering out
states with zero average price
state_avg_prices = df_melted1.groupby('State')
['Price'].mean().loc[lambda x: x != 0]

# Sort the states by average price in descending order and get the top
5
top_5_states = state_avg_prices.sort_values(ascending=False).head(5)

print(top_5_states)
```

```
State
DC    487971.048218
CA    452726.608090
HI    427385.934267
NJ    325240.623919
MA    321933.198498
Name: Price, dtype: float64
```

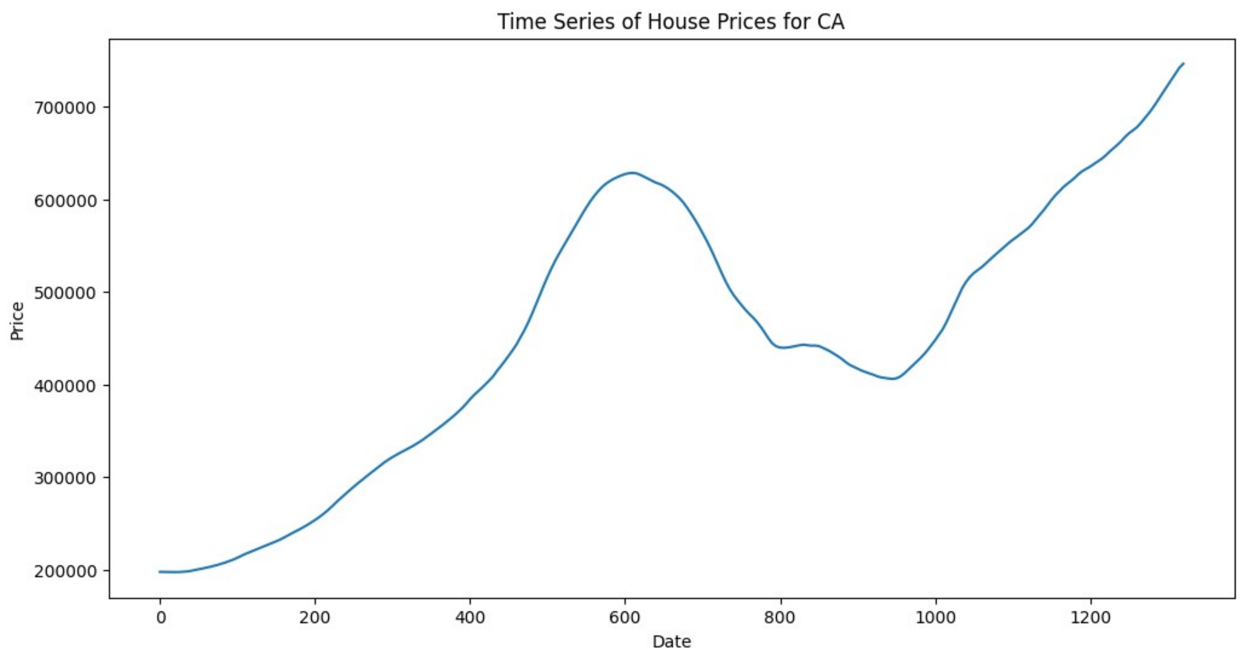
```python
import pandas as pd

def melt_state_data(df_filled, state=None):
    """
    Takes the zillow_data dataset in wide form or a subset of the
zillow_dataset.
    Returns a long-form datetime dataframe with the datetime column
names as the index
    and the values as the 'values' column, aggregated by mean price
for each city.

    If 'state' is provided, it filters the dataframe for those states
before melting.

    Parameters:
    - df_filled: DataFrame, the original wide-form dataset.
    - cities: list of str, optional. List of states to include. If
None, includes all states.

    Returns:
    - DataFrame: A long-form DataFrame with columns ['time', 'State',
'value'].
    """

    if state:
        df_filled = df_filled[df_filled['State'].isin(state)]

    melted = pd.melt(df_filled, id_vars=['RegionName', 'RegionID',
```

```
'SizeRank', 'City', 'State', 'Metro', 'CountyName'], var_name='time')
    melted['time'] = pd.to_datetime(melted['time'])
    melted = melted.dropna(subset=['value'])
    return melted.groupby(['time', 'State']).aggregate({'value':
'mean'}).reset_index()

top_5_states

State
DC    487971.048218
CA    452726.608090
HI    427385.934267
NJ    325240.623919
MA    321933.198498
Name: Price, dtype: float64

cities_to_plot = ['DC', 'CA', 'HI', 'NJ', 'MA']
melted_state_data = melt_state_data(df_filled, state=cities_to_plot)

melted_state_data.head()
```

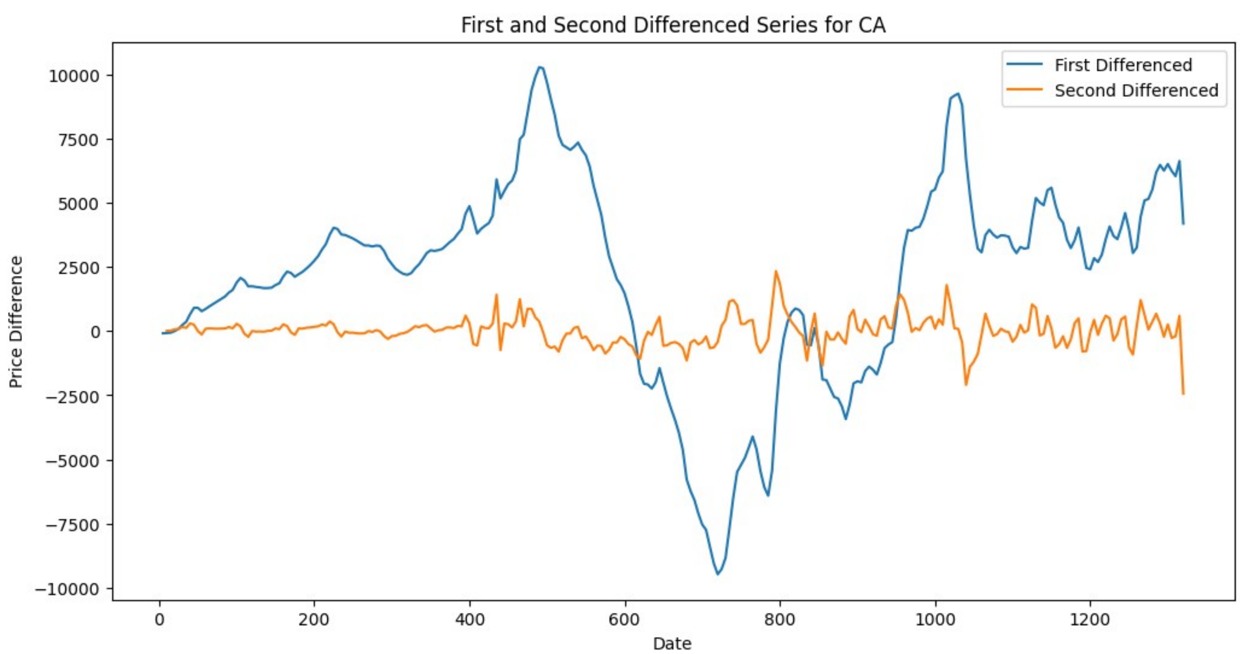

{"summary":"{\n \"name\": \"melted_state_data\",\n \"rows\": 1325,\n \"fields\": [\n   {\n   \"column\": \"time\",\n   \"properties\": {\n   \"dtype\": \"date\",\n   \"min\": \"1996-04-01 00:00:00\",\n   \"max\": \"2018-04-01 00:00:00\",\n   \"num_unique_values\": 265,\n   \"samples\": [\n   \"2011-03-01 00:00:00\",\n   \"2005-11-01 00:00:00\",\n   \"2004-04-01 00:00:00\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"State\",\n   \"properties\": {\n   \"dtype\": \"category\",\n   \"num_unique_values\": 5,\n   \"samples\": [\n   \"DC\",\n   \"NJ\",\n   \"HI\"\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"value\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 152169.58983009265,\n   \"min\": 159461.1510791367,\n   \"max\": 806166.6666666666,\n   \"num_unique_values\": 1325,\n   \"samples\": [\n   354777.09163346613,\n   372745.1612903226,\n   455414.51612903224\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   }\n]\n}","type":"dataframe","variable_name":"melted_state_data"}

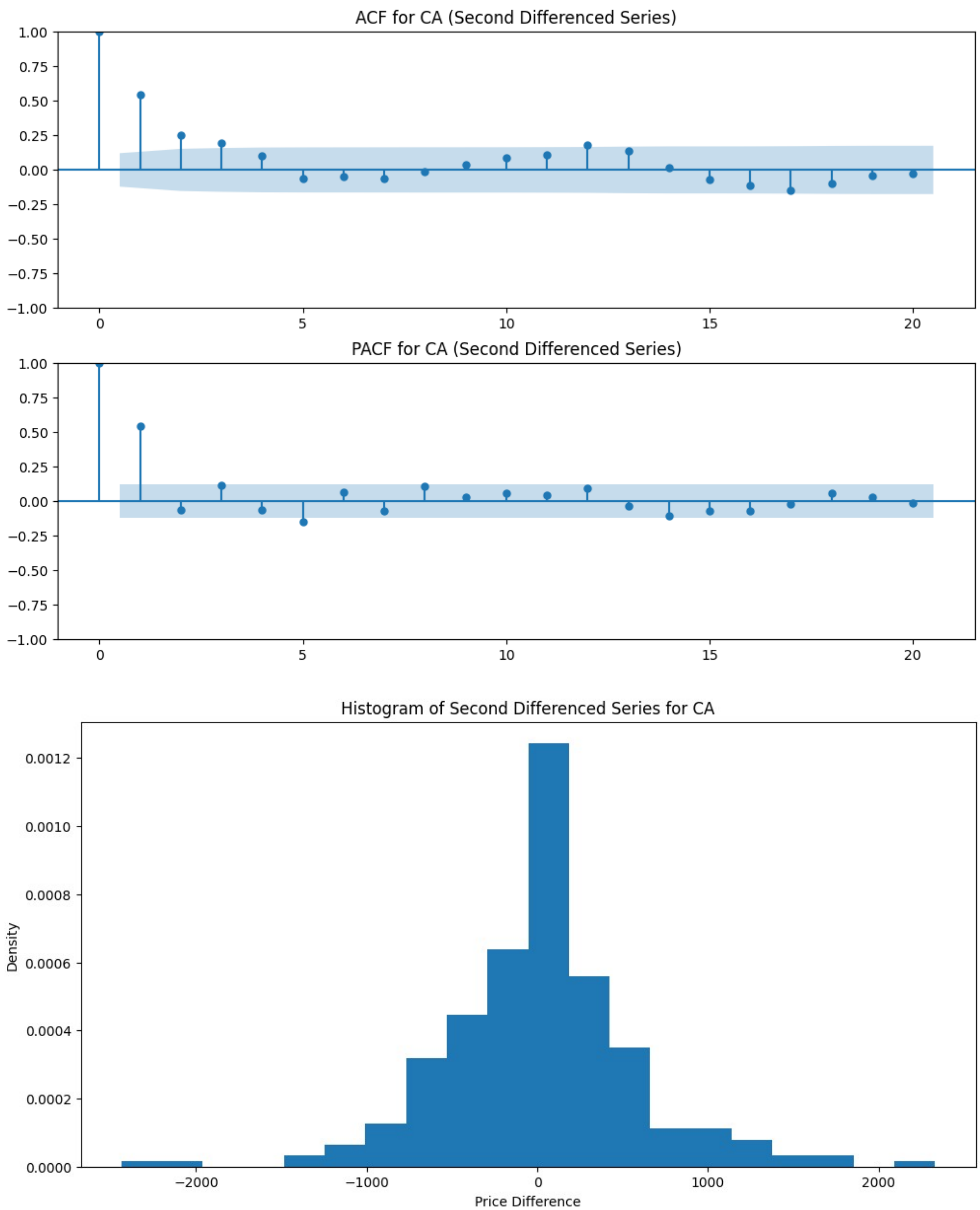

```
# Function to plot, test, and inspect differencing for states
def plot_test_differencing(melted_state_data):
    states = melted_state_data['State'].unique()

    for state in states:
        state_data = melted_state_data[melted_state_data['State'] ==
state]['value']
```

```
        # Plot the time series of house prices
        plt.figure(figsize=(12, 6))
        plt.plot(state_data)
        plt.title(f'Time Series of House Prices for {state}')
        plt.xlabel('Date')
        plt.ylabel('Price')
        plt.show()

        # Perform the Augmented Dickey-Fuller test for the second
differencing
        ts_diff_2 = state_data.diff().diff().dropna()
        result = adfuller(ts_diff_2)
        print(f'ADF Statistic for {state} (Second Differenced Series):
{result[0]}')
        print(f'p-value for {state} (Second Differenced Series):
{result[1]}')

        # Plot overlapping first and second differenced series
        plt.figure(figsize=(12, 6))
        plt.plot(state_data.diff(), label='First Differenced')
        plt.plot(ts_diff_2, label='Second Differenced')
        plt.title(f'First and Second Differenced Series for {state}')
        plt.xlabel('Date')
        plt.ylabel('Price Difference')
        plt.legend()
        plt.show()

        # Plot ACF and PACF for the second differenced series
        fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))
        sm.graphics.tsa.plot_acf(ts_diff_2, lags=20, ax=ax1)
        sm.graphics.tsa.plot_pacf(ts_diff_2, lags=20, ax=ax2)
        ax1.set_title(f'ACF for {state} (Second Differenced Series)')
        ax2.set_title(f'PACF for {state} (Second Differenced Series)')
        plt.show()

        # Plot histogram of the second differenced series to check for
white noise
        plt.figure(figsize=(12, 6))
        plt.hist(ts_diff_2, bins=20, density=True)
        plt.title(f'Histogram of Second Differenced Series for
{state}')
        plt.xlabel('Price Difference')
        plt.ylabel('Density')
        plt.show()

# Example usage:
plot_test_differencing(melted_state_data)
```

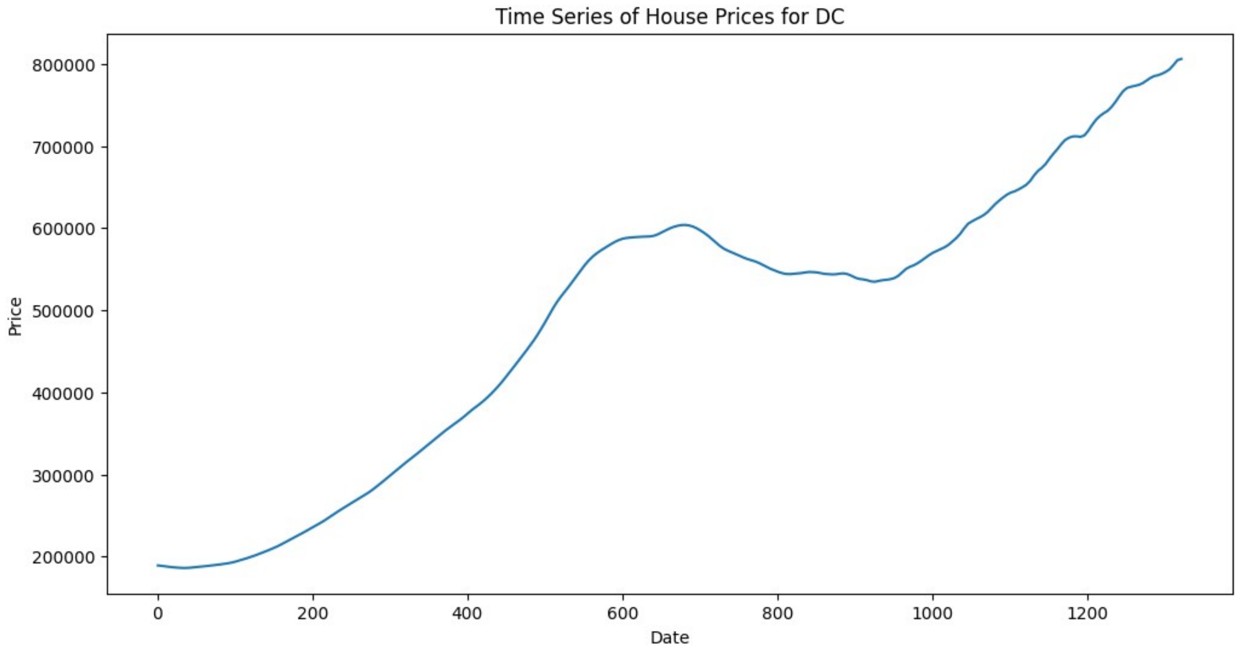



```
ADF Statistic for CA (Second Differenced Series): -4.356987723312447
p-value for CA (Second Differenced Series): 0.00035341359503088596
```

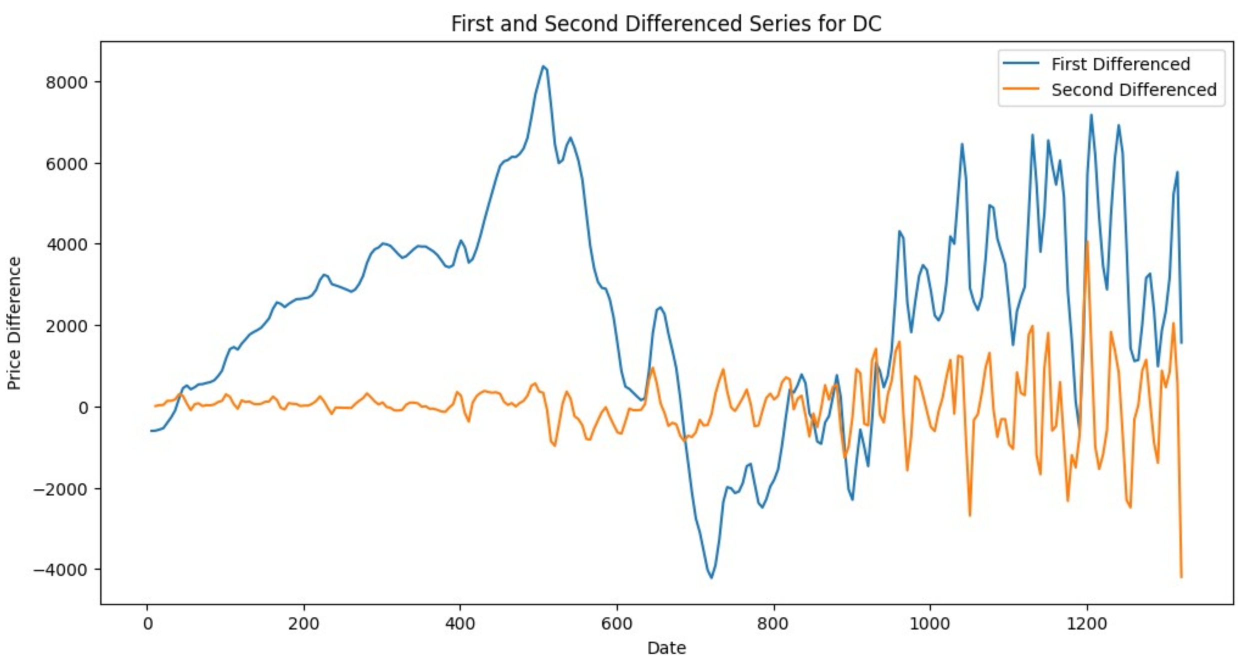


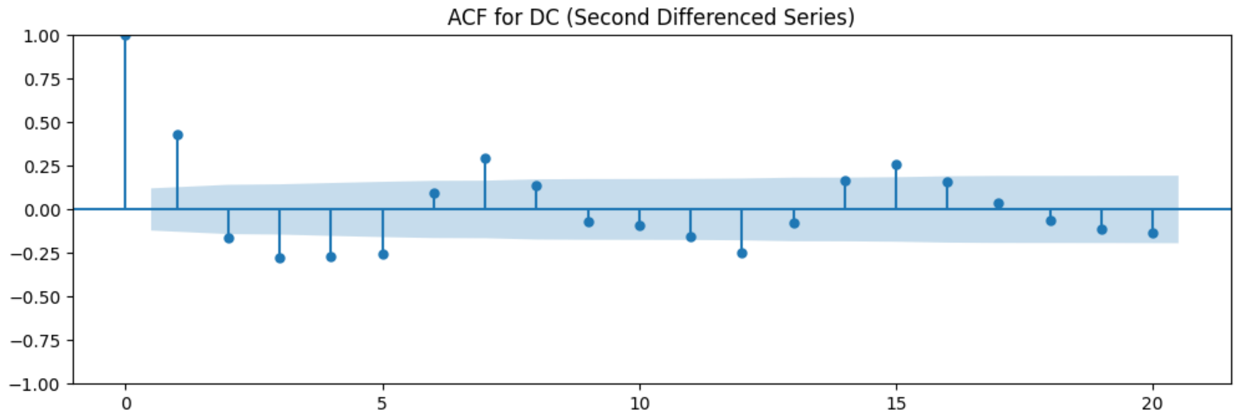


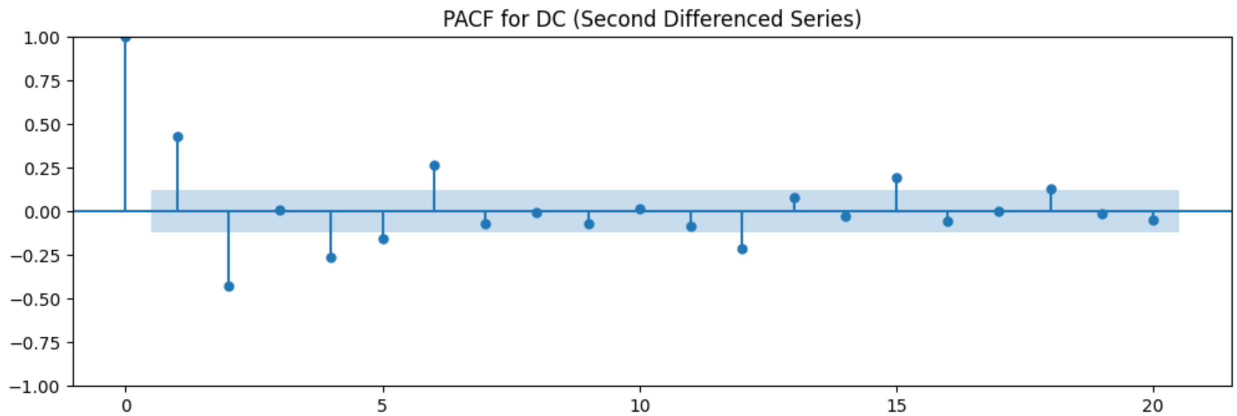
ACF for CA (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
PACF for CA (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
Histogram of Second Differenced Series for CA
Density
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
−2000
−1000
0
1000
2000
Price Difference

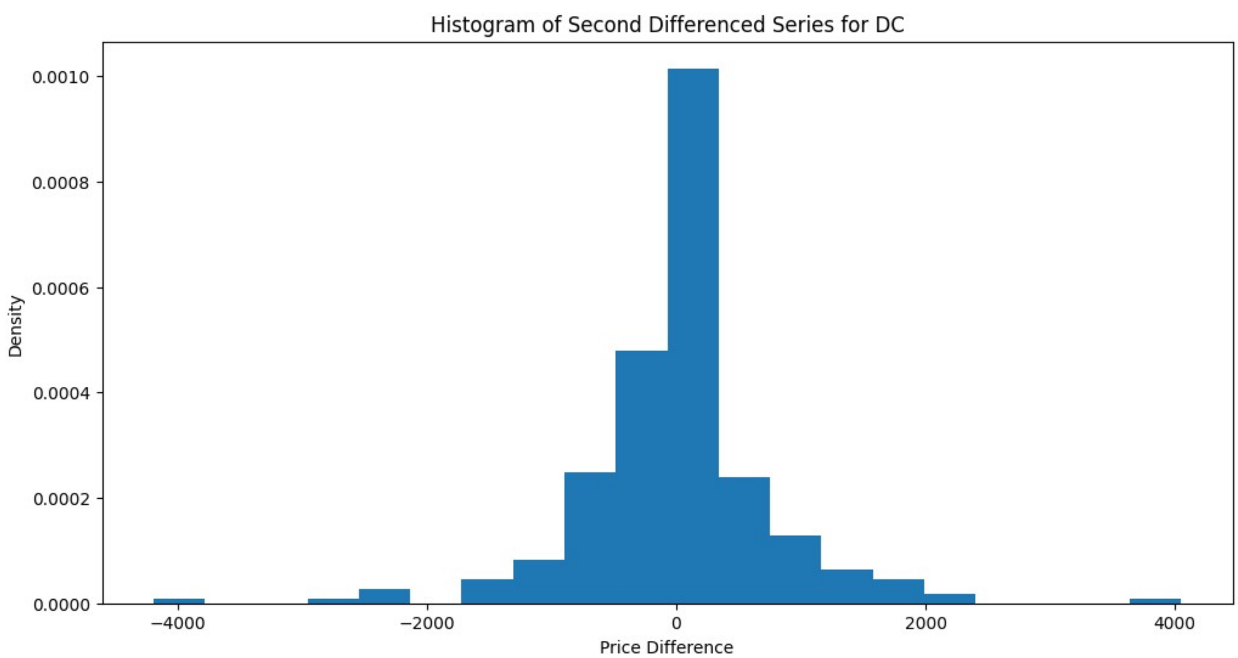


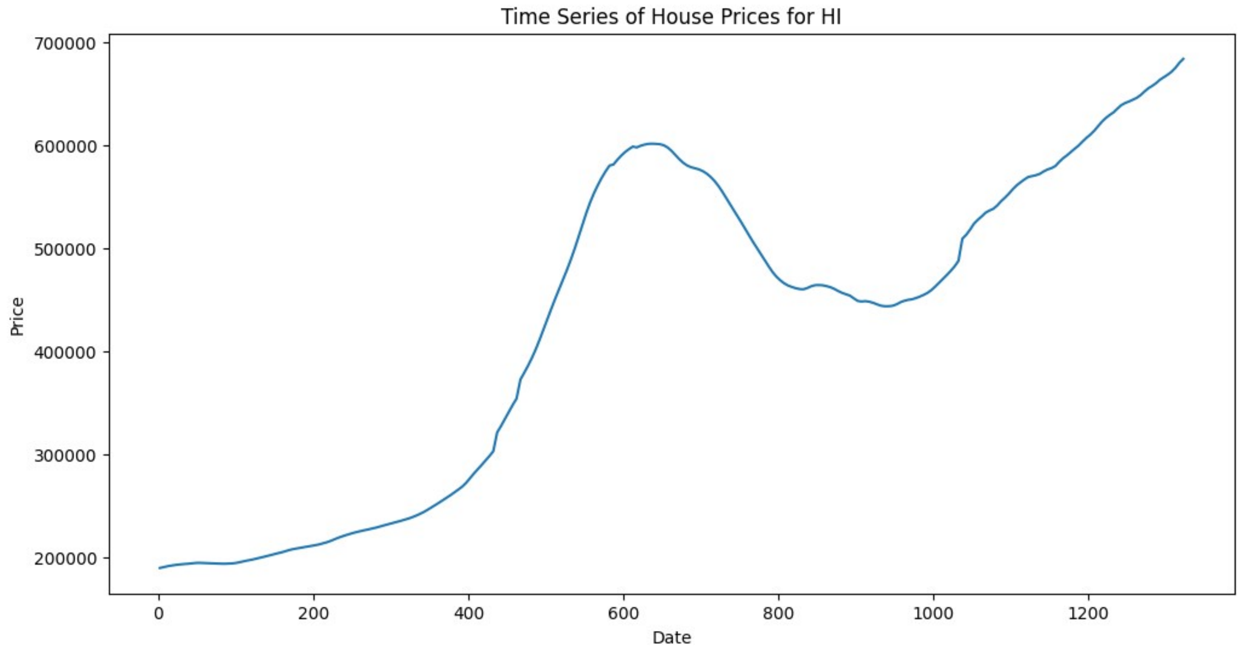

```
ADF Statistic for DC (Second Differenced Series): -3.9273389779423145
p-value for DC (Second Differenced Series): 0.0018426103473226565
```



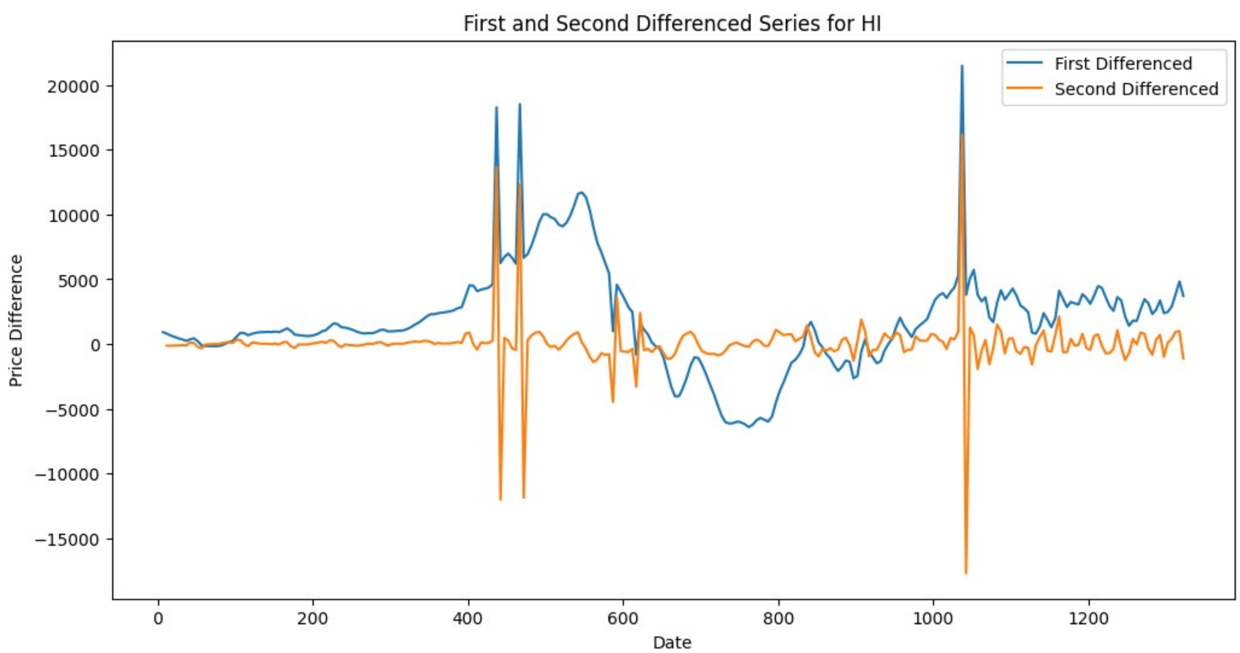

ACF for DC (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20

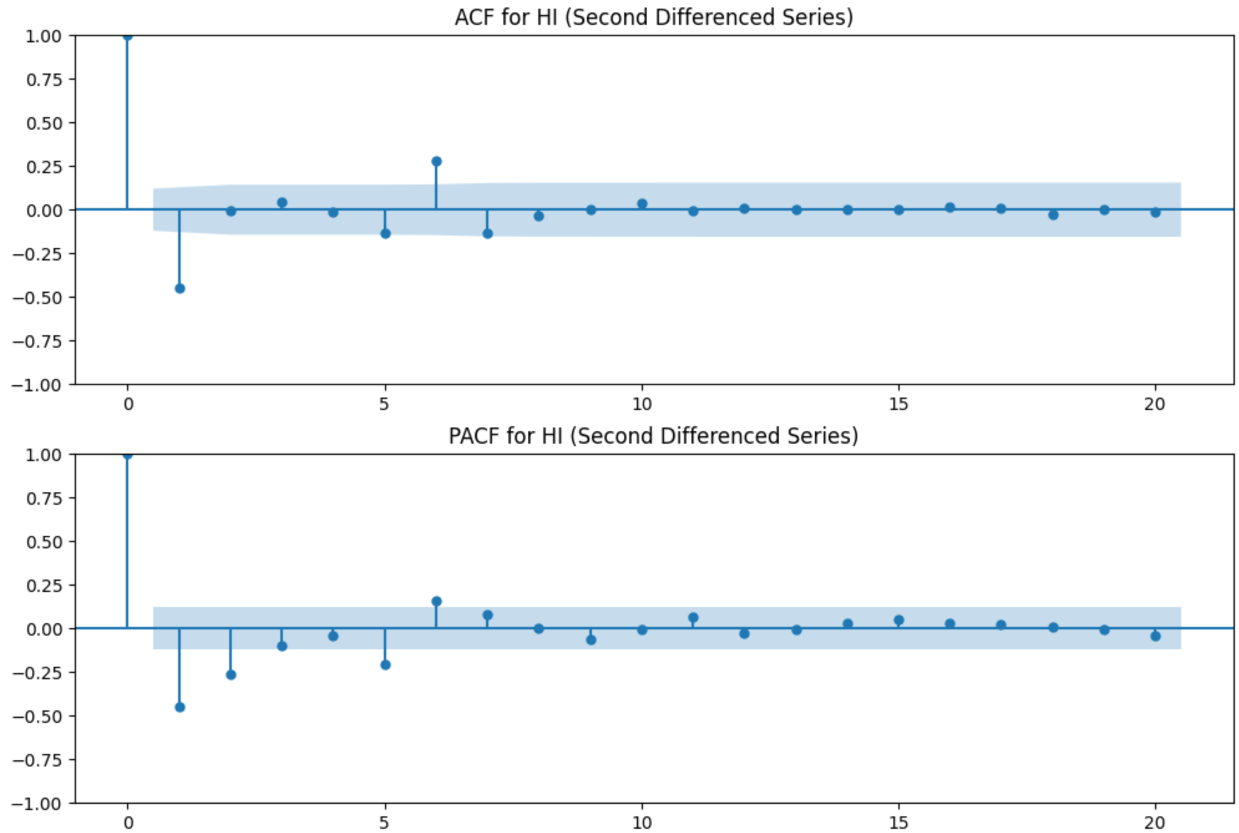

PACF for DC (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20

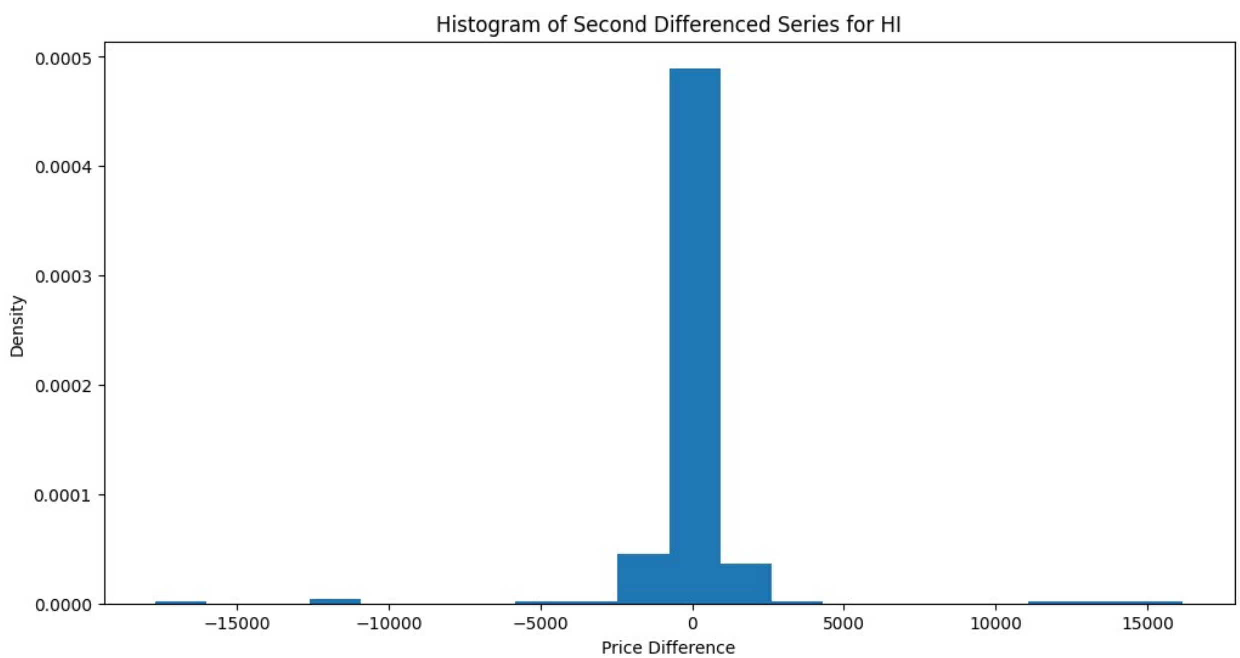

Histogram of Second Differenced Series for DC
Density
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
−4000
−2000
0
2000
4000
Price Difference



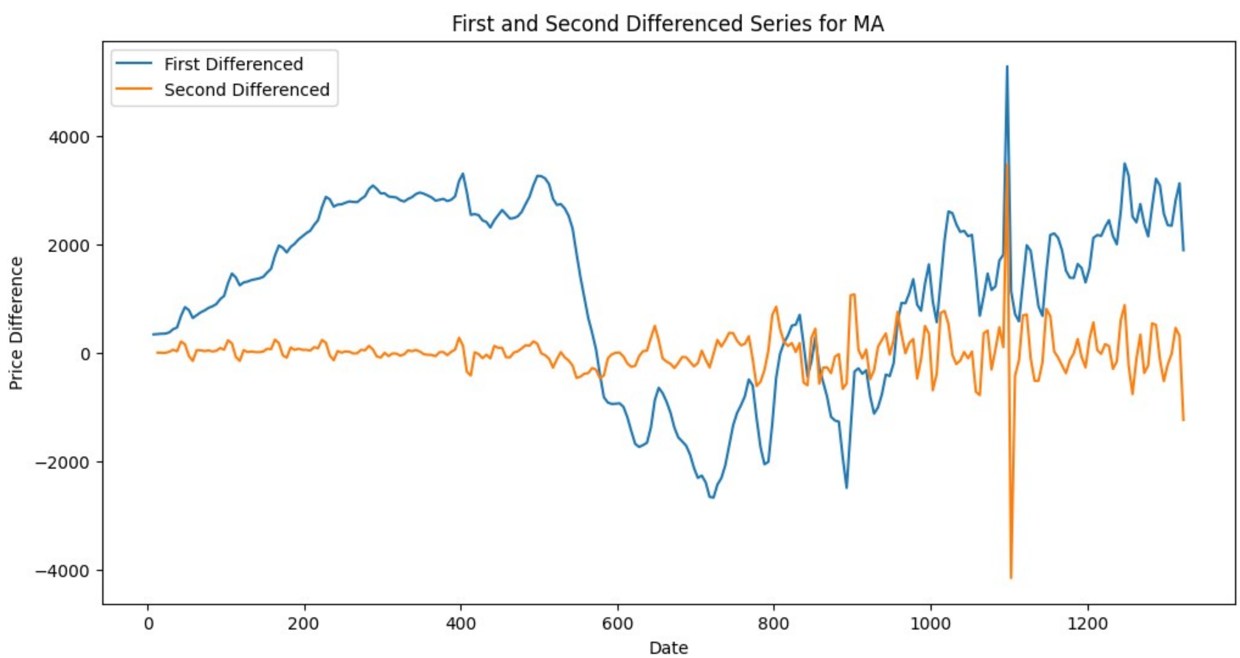

```
ADF Statistic for HI (Second Differenced Series): -7.573301241435548
p-value for HI (Second Differenced Series): 2.804039174744741e-11
```

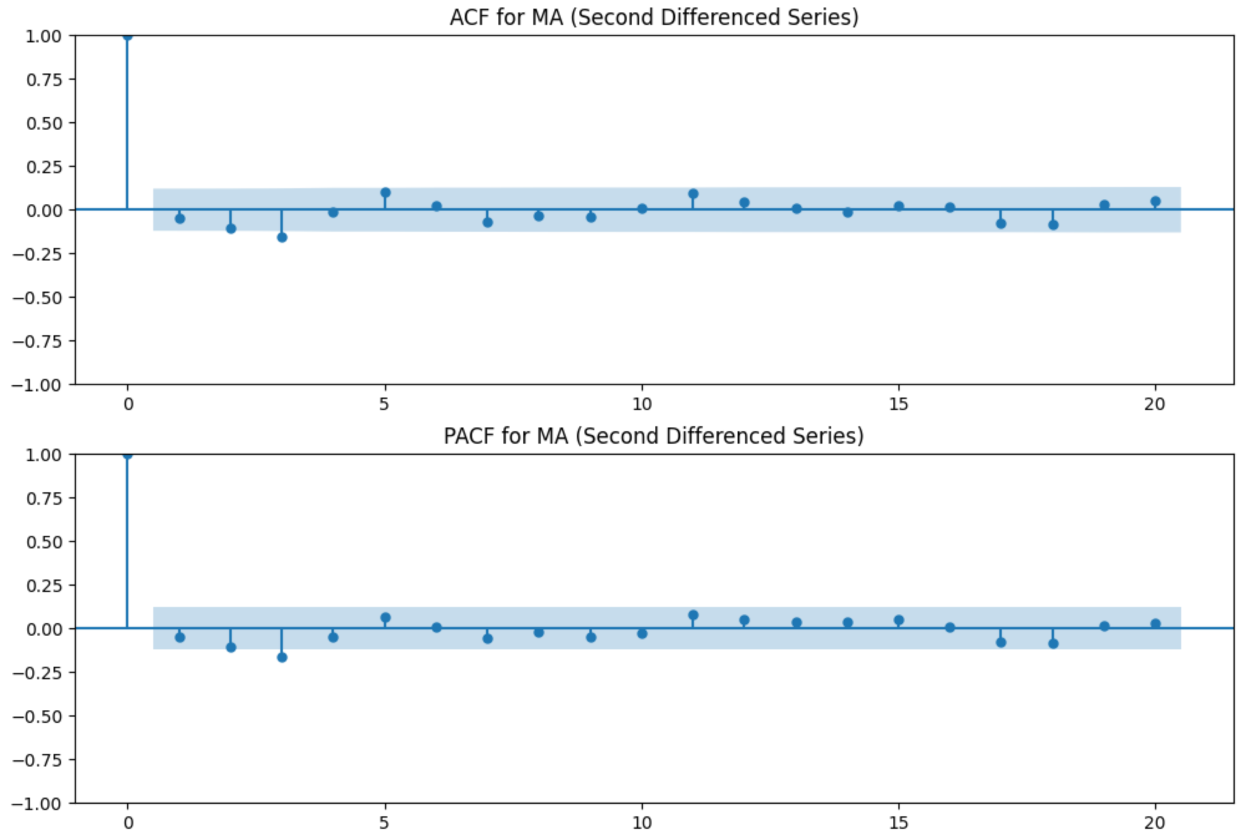


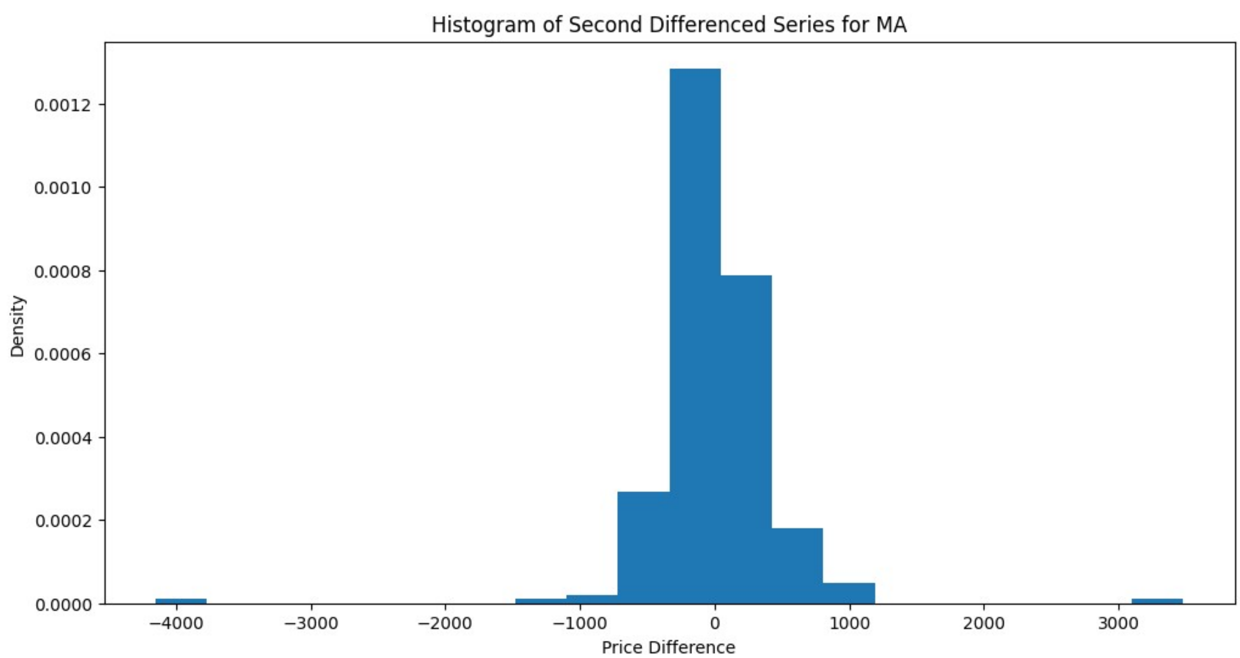

ACF for HI (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
PACF for HI (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20

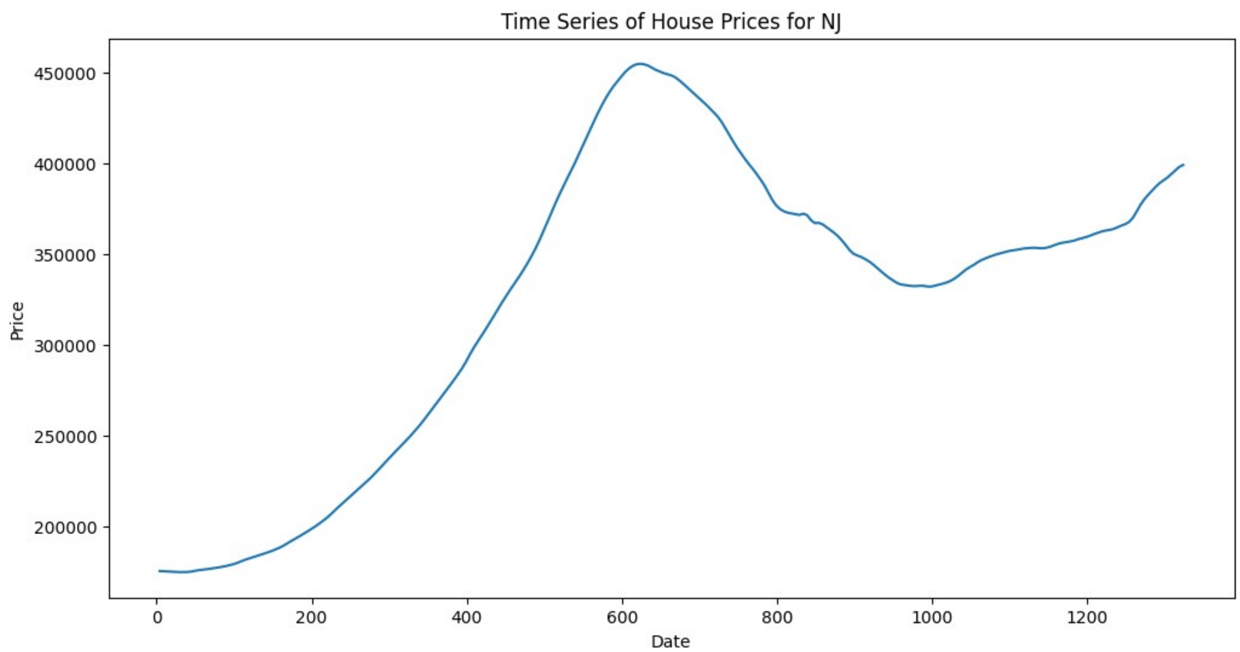

Histogram of Second Differenced Series for HI
Density
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
−15000
−10000
−5000
0
5000
10000
15000
Price Difference


```
ADF Statistic for MA (Second Differenced Series): -11.82283256170377
p-value for MA (Second Differenced Series): 8.335969810337181e-22
```

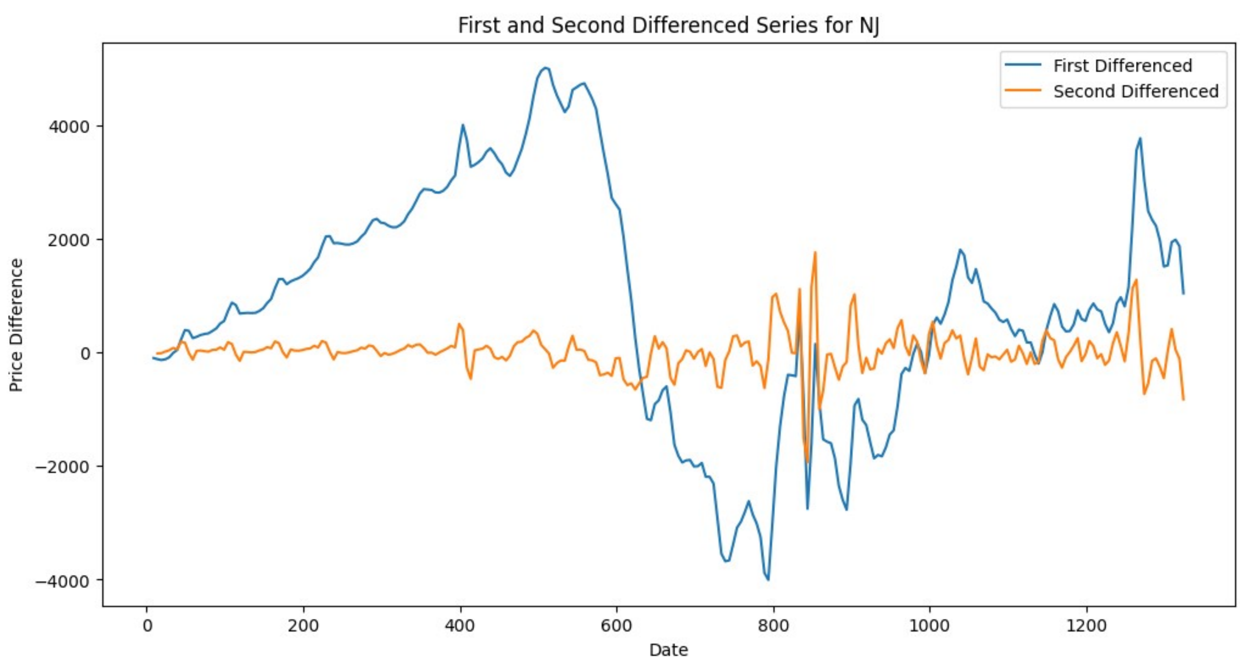


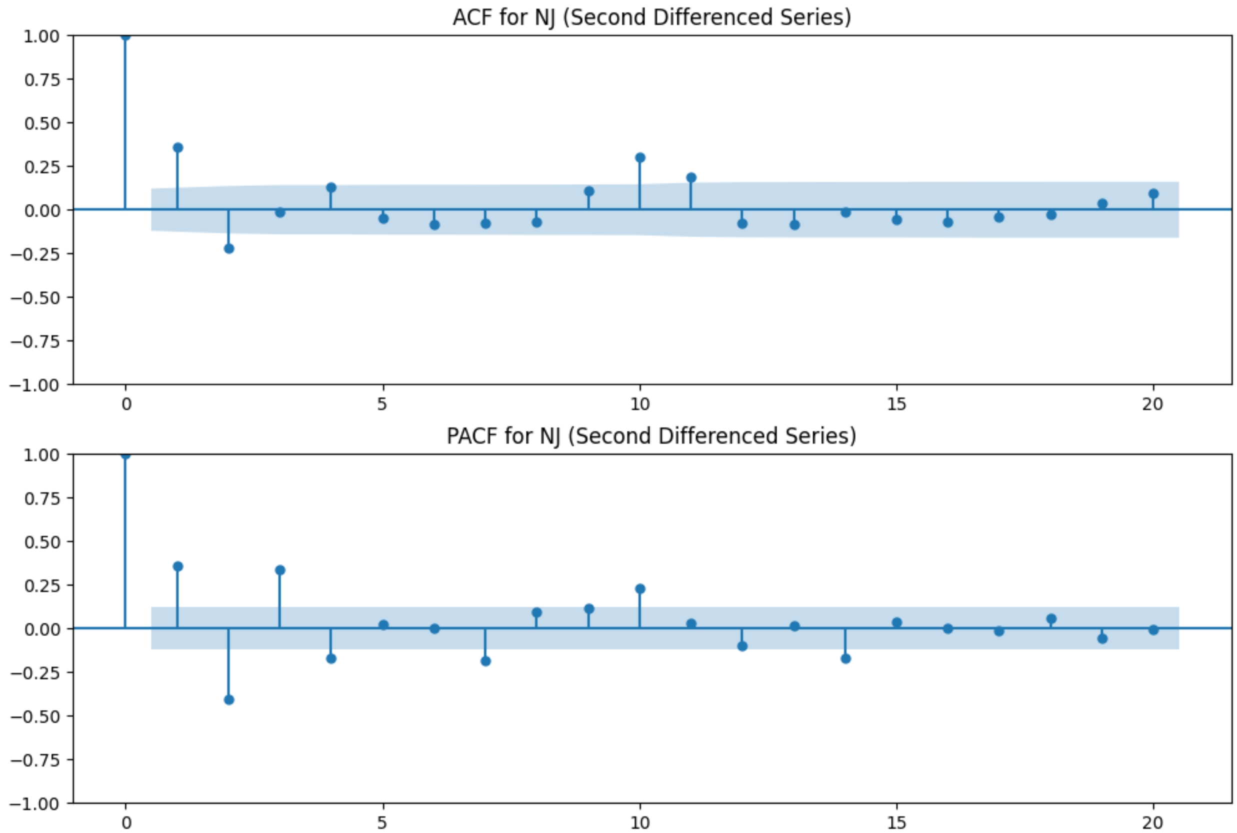

ACF for MA (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20
PACF for MA (Second Differenced Series)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
5
10
15
20

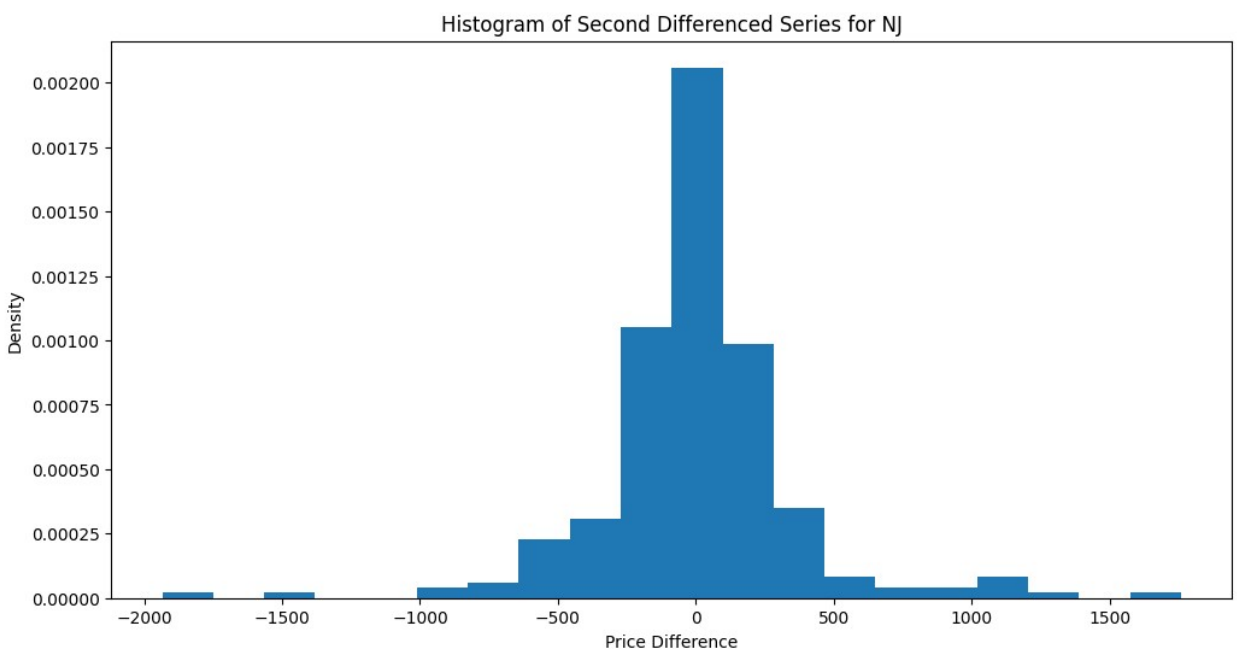

Histogram of Second Differenced Series for MA
Density
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
−4000
−3000
−2000
−1000
0
1000
2000
3000
Price Difference




```
ADF Statistic for NJ (Second Differenced Series): -3.986448478831105
p-value for NJ (Second Differenced Series): 0.001483157184679738
```








```
states = melted_state_data['State'].unique()

train_test_data = {}
```

```python
# Loop through each state
for state in states:
    state_data = melted_state_data[melted_state_data['State'] ==
state].set_index('time')['value']

    # Split into train/test sets
    train_size = int(len(state_data) * 0.8)
    train_data, test_data = state_data.iloc[:train_size],
state_data.iloc[train_size:]

    train_test_data[state] = {
        'train': train_data,
        'test': test_data
    }



# Dictionary to store fitted models
fitted_models = {}

# ARIMA parameters (p, d, q) for each state
state_params = {
    'CA': (3, 2, 1),
    'DC': (5, 2, 2),
    'HI': (1, 2, 2),
    'MA': (1, 2, 1),
    'NJ': (2, 2, 4)
}

# Function to fit ARIMA model and summarize
def fit_arima_and_summary(train_data, order):
    model = ARIMA(train_data, order=order)
    fitted_model = model.fit()
    print(fitted_model.summary())
    return fitted_model

# Loop through each state and fit ARIMA models
for state, params in state_params.items():
    train_data = train_test_data[state]['train']
    fitted_model = fit_arima_and_summary(train_data, order=params)
    fitted_models[state] = fitted_model
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/
tsa_model.py:473: ValueWarning: No frequency information was provided,
so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)

                               SARIMAX Results

======================================================================
========
Dep. Variable:                  value   No. Observations:
212
Model:                 ARIMA(3, 2, 1)   Log Likelihood               -
1622.839
Date:                Wed, 03 Jul 2024   AIC
3255.679
Time:                        00:55:33   BIC
3272.414
Sample:                    04-01-1996   HQIC
3262.444
                         - 11-01-2013

Covariance Type:                  opg

======================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
----------------------------------------------------------------------
--------
ar.L1          1.0302      0.023     44.116      0.000       0.984
1.076
ar.L2         -0.0114      0.011     -1.087      0.277      -0.032
0.009
ar.L3         -0.0219      0.010     -2.198      0.028      -0.041
-0.002
ma.L1         -0.9986      0.027    -36.656      0.000      -1.052
-0.945
sigma2      2.906e+05   1.99e+04     14.602      0.000    2.52e+05
3.3e+05
======================================================================
=============
Ljung-Box (L1) (Q):                  83.99   Jarque-Bera (JB):
62.32
Prob(Q):                              0.00   Prob(JB):
0.00
Heteroskedasticity (H):              13.22   Skew:
0.40
Prob(H) (two-sided):                  0.00   Kurtosis:
5.55
======================================================================
```

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

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).

/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/
tsa_model.py:473: ValueWarning: No frequency information was provided,
so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/
sarimax.py:966: UserWarning: Non-stationary starting autoregressive
parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/
sarimax.py:978: UserWarning: Non-invertible starting MA parameters
found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'

                               SARIMAX Results

======================================================================
========
Dep. Variable:                  value   No. Observations:
212
Model:                 ARIMA(5, 2, 2)   Log Likelihood               -
1597.743
Date:                Wed, 03 Jul 2024   AIC
3211.486
Time:                        00:55:34   BIC
3238.263
Sample:                    04-01-1996   HQIC
3222.311
                         - 11-01-2013

Covariance Type:                  opg

======================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
----------------------------------------------------------------------
```

```
--------
ar.L1          0.0274      4.164      0.007      0.995      -8.135
8.189
ar.L2          0.9997      2.982      0.335      0.737      -4.845
6.844
ar.L3         -0.0346      0.081     -0.425      0.671      -0.194
0.125
ar.L4         -0.0002      0.056     -0.004      0.997      -0.109
0.109
ar.L5          0.0077      0.034      0.226      0.821      -0.059
0.074
ma.L1         -0.0005      4.194     -0.000      1.000      -8.220
8.220
ma.L2         -0.9995      2.878     -0.347      0.728      -6.640
4.641
sigma2      2.322e+05      0.000   1.06e+09      0.000    2.32e+05
2.32e+05
======================================================================
=============
Ljung-Box (L1) (Q):                  57.01   Jarque-Bera (JB):
209.91
Prob(Q):                              0.00   Prob(JB):
0.00
Heteroskedasticity (H):              20.61   Skew:
-0.57
Prob(H) (two-sided):                  0.00   Kurtosis:
7.76
======================================================================
=============

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).
[2] Covariance matrix is singular or near-singular, with condition
number 2.33e+25. Standard errors may be unstable.

/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/
tsa_model.py:473: ValueWarning: No frequency information was provided,
so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/
sarimax.py:966: UserWarning: Non-stationary starting autoregressive
```

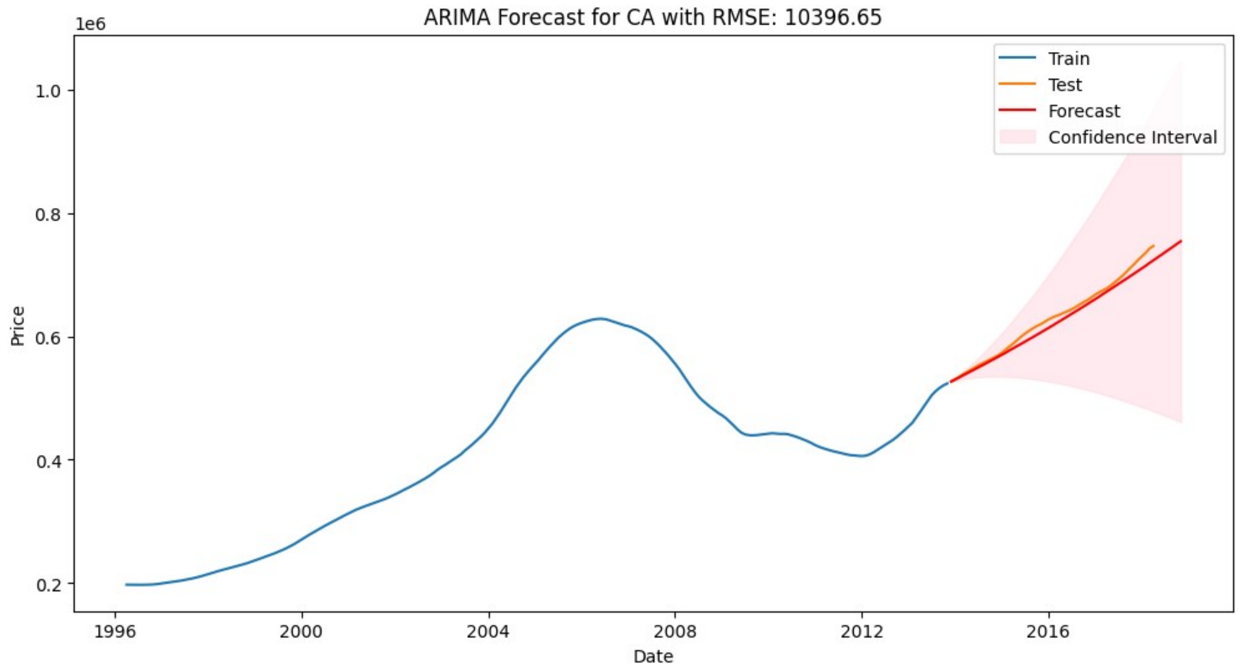
```
parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/statespace/
sarimax.py:978: UserWarning: Non-invertible starting MA parameters
found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
/usr/local/lib/python3.10/dist-packages/statsmodels/base/model.py:607:
ConvergenceWarning: Maximum Likelihood optimization failed to
converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)

                               SARIMAX Results

======================================================================
========
Dep. Variable:                  value   No. Observations:
212
Model:                 ARIMA(1, 2, 2)   Log Likelihood                -
1936.779
Date:                Wed, 03 Jul 2024   AIC
3881.559
Time:                        00:55:35   BIC
3894.947
Sample:                    04-01-1996   HQIC
3886.971
                         - 11-01-2013

Covariance Type:                  opg

======================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
----------------------------------------------------------------------
--------
ar.L1          0.8350      0.182      4.577      0.000       0.477
1.193
ma.L1         -0.9107      0.170     -5.348      0.000      -1.245
```

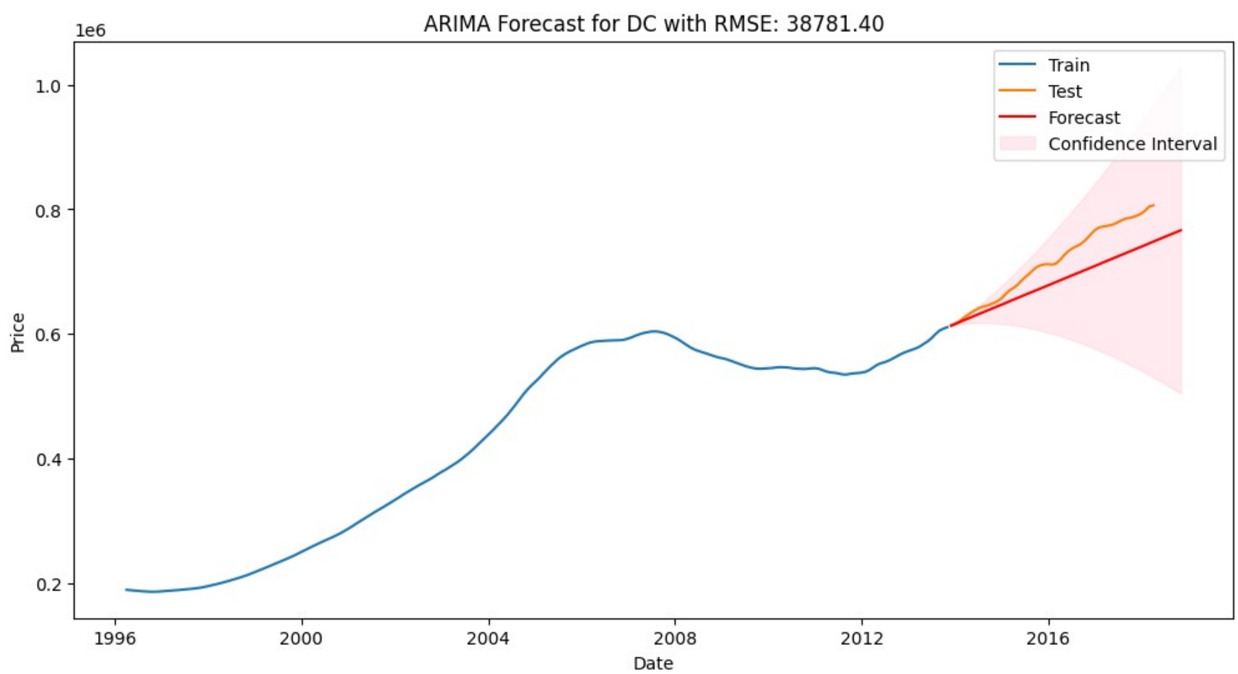
```
-0.577
ma.L2          0.0065      0.021      0.309      0.757      -0.035
0.048
sigma2      5.962e+06   1.87e+05     31.882      0.000     5.6e+06
6.33e+06
======================================================================
=============
Ljung-Box (L1) (Q):                  34.42   Jarque-Bera (JB):
6698.07
Prob(Q):                              0.00   Prob(JB):
0.00
Heteroskedasticity (H):              16.12   Skew:
0.69
Prob(H) (two-sided):                  0.00   Kurtosis:
30.63
======================================================================
=============

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).
                               SARIMAX Results

======================================================================
========
Dep. Variable:                  value   No. Observations:
212
Model:                 ARIMA(1, 2, 1)   Log Likelihood               -
1473.183
Date:                Wed, 03 Jul 2024   AIC
2952.365
Time:                        00:55:35   BIC
2962.407
Sample:                    04-01-1996   HQIC
2956.425
                         - 11-01-2013

Covariance Type:                  opg

======================================================================
========
                 coef    std err          z      P>|z|      [0.025
0.975]
----------------------------------------------------------------------
--------
ar.L1         -0.1063      0.629     -0.169      0.866      -1.340
1.127
ma.L1          0.1772      0.631      0.281      0.779      -1.060
1.415
sigma2      6.843e+04   4839.527     14.139      0.000    5.89e+04
```

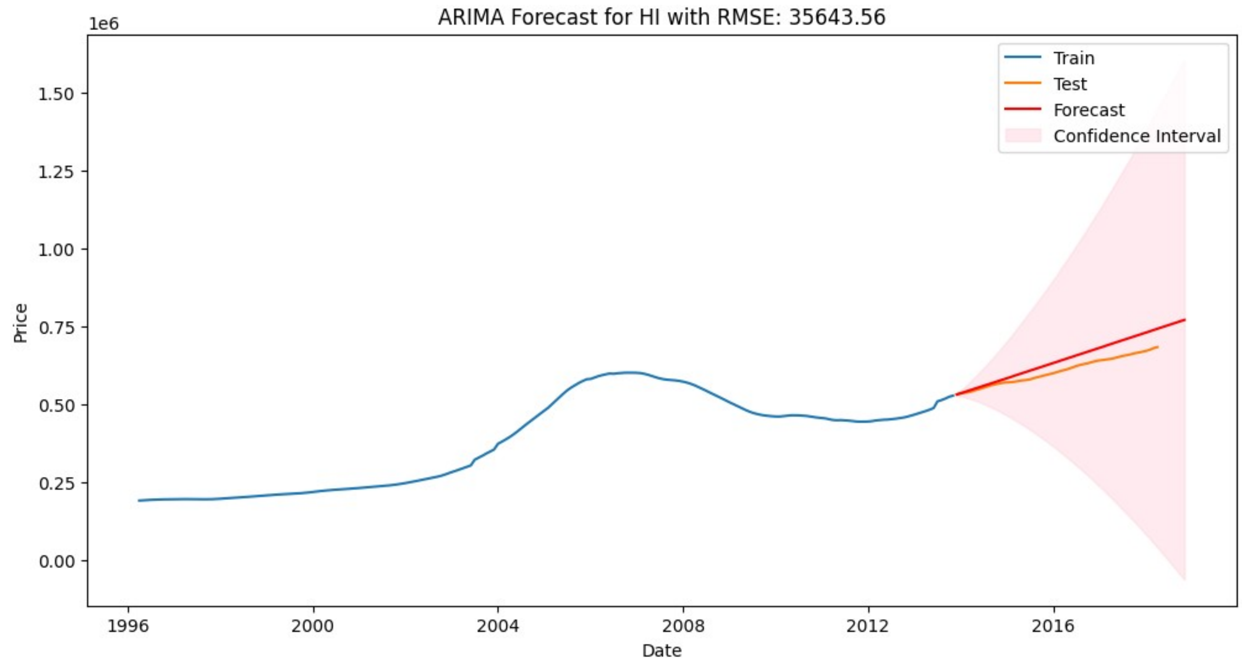
```
7.79e+04
==============================================================================
=============
Ljung-Box (L1) (Q):                  37.39   Jarque-Bera (JB):
71.45
Prob(Q):                              0.00   Prob(JB):
0.00
Heteroskedasticity (H):              11.36   Skew:
0.55
Prob(H) (two-sided):                  0.00   Kurtosis:
5.64
==============================================================================
=============

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).

/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/
tsa_model.py:473: ValueWarning: No frequency information was provided,
so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model
.py:473: ValueWarning: No frequency information was provided, so
inferred frequency MS will be used.
  self._init_dates(dates, freq)

                               SARIMAX Results

==============================================================================
========
Dep. Variable:                  value   No. Observations:
212
Model:                 ARIMA(2, 2, 4)   Log Likelihood               -
1533.225
Date:                Wed, 03 Jul 2024   AIC
3080.451
Time:                        00:55:37   BIC
3103.881
Sample:                    04-01-1996   HQIC
3089.923
                         - 11-01-2013

Covariance Type:                  opg

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

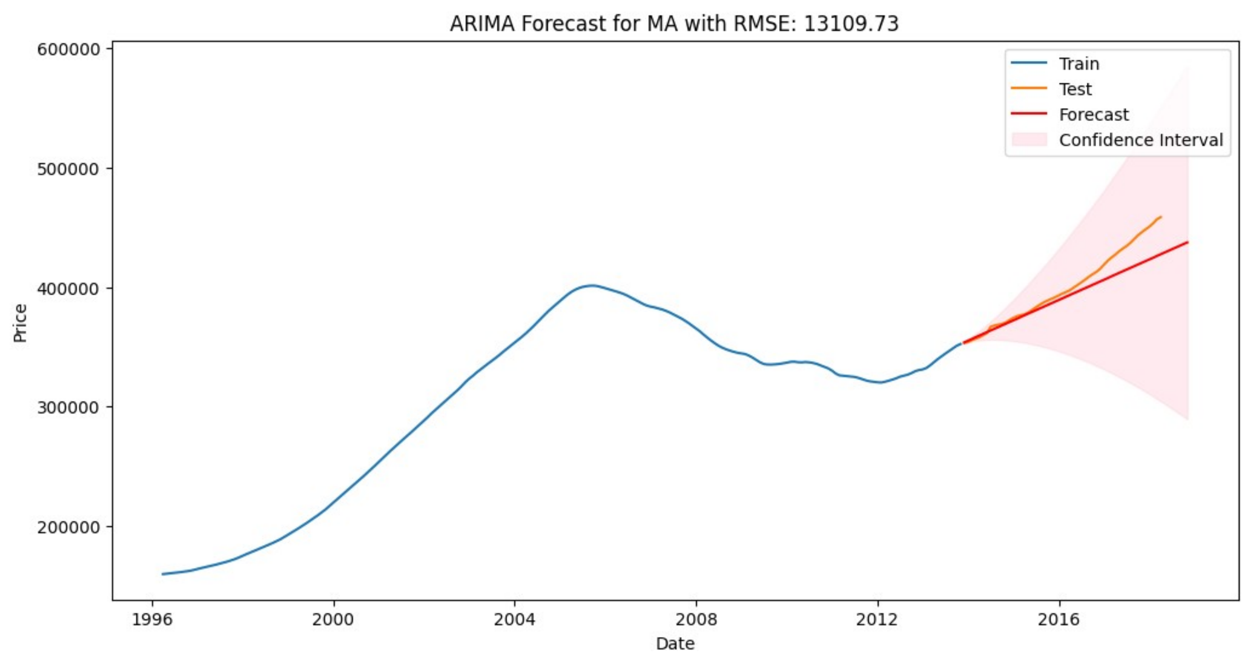
```
========
                  coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
--------
ar.L1          -0.8957      0.053    -17.011      0.000      -0.999
-0.793
ar.L2          -0.9996      0.013    -75.295      0.000      -1.026
-0.974
ma.L1           0.9240      0.057     16.268      0.000       0.813
1.035
ma.L2           1.0117      0.077     13.217      0.000       0.862
1.162
ma.L3           0.0162      0.025      0.636      0.525      -0.034
0.066
ma.L4          -0.0124      0.021     -0.578      0.563      -0.054
0.030
sigma2      1.233e+05   8.26e-07   1.49e+11      0.000    1.23e+05
1.23e+05
==============================================================================
=============
Ljung-Box (L1) (Q):                  21.90   Jarque-Bera (JB):
452.81
Prob(Q):                              0.00   Prob(JB):
0.00
Heteroskedasticity (H):              24.90   Skew:
-0.21
Prob(H) (two-sided):                  0.00   Kurtosis:
10.18
==============================================================================
=============

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).
[2] Covariance matrix is singular or near-singular, with condition
number 8.02e+25. Standard errors may be unstable.

/usr/local/lib/python3.10/dist-packages/statsmodels/base/model.py:607:
ConvergenceWarning: Maximum Likelihood optimization failed to
converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "

# Forecasting future periods
forecast_steps = 60

forecasts = {}
rmse_values = {}

for state, model in fitted_models.items():
```

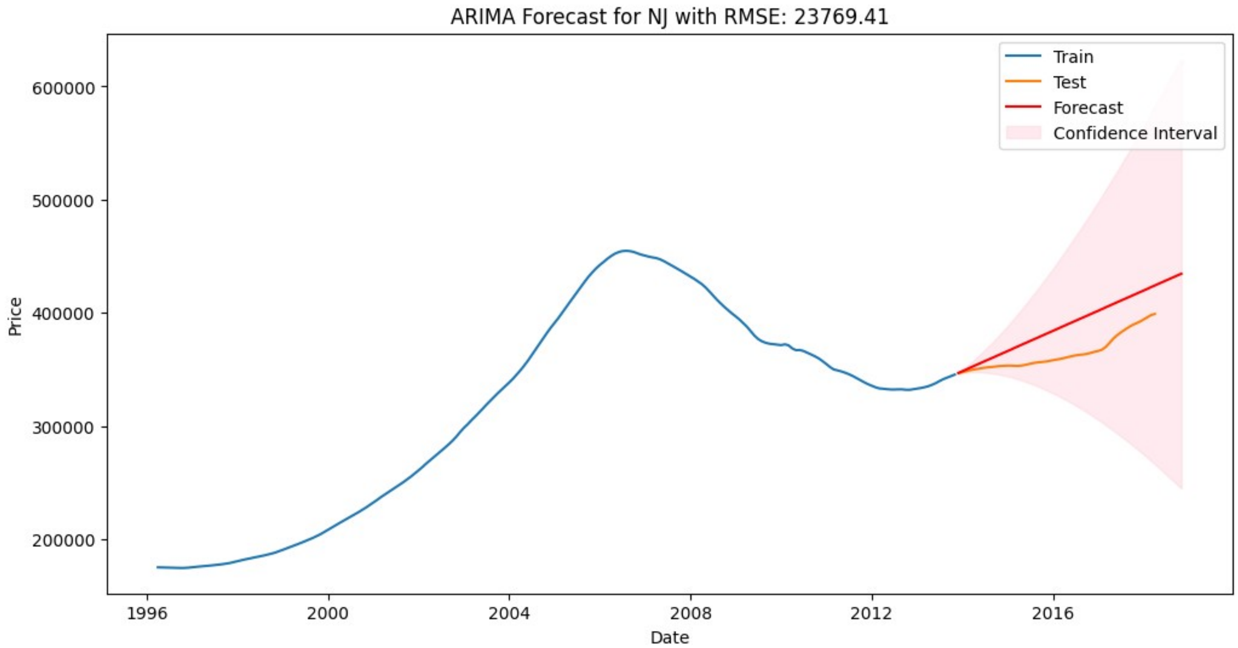
```
    train_data = train_test_data[state]['train']
    test_data = train_test_data[state]['test']

    # Forecast next 60 months
    forecast = model.get_forecast(steps=forecast_steps)
    forecasts[state] = forecast

    # Compute RMSE
    forecast_values = forecast.predicted_mean
    mse = mean_squared_error(test_data,
forecast_values[:len(test_data)])
    rmse = np.sqrt(mse)
    rmse_values[state] = rmse

    # Plot forecasts with RMSE
    plt.figure(figsize=(12, 6))
    plt.plot(train_data.index, train_data.values, label='Train')
    plt.plot(test_data.index, test_data.values, label='Test')
    plt.plot(forecast_values.index, forecast_values.values,
color='red', label='Forecast')

    # Plot confidence interval (CI)
    forecast_ci = forecast.conf_int()
    plt.fill_between(forecast_ci.index, forecast_ci.iloc[:, 0],
forecast_ci.iloc[:, 1], color='pink', alpha=0.3, label='Confidence
Interval')

    plt.title(f'ARIMA Forecast for {state} with RMSE: {rmse:.2f}')
    plt.xlabel('Date')
    plt.ylabel('Price')
    plt.legend()
    plt.show()
```


ARIMA Forecast for CA with RMSE: 10396.65
1e6
1.0
0.8
0.6
0.4
0.2
Price
1996
2000
2004
2008
2012
2016
Date
Train
Test
Forecast
Confidence Interval


ARIMA Forecast for DC with RMSE: 38781.40
1e6
1.0
0.8
0.6
0.4
0.2
Price
1996
2000
2004
2008
2012
2016
Date
Train
Test
Forecast
Confidence Interval


ARIMA Forecast for HI with RMSE: 35643.56
1e6
Train
Test
Forecast
Confidence Interval
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Price
1996
2000
2004
2008
2012
2016
Date


ARIMA Forecast for MA with RMSE: 13109.73
Train
Test
Forecast
Confidence Interval
600000
500000
400000
300000
200000
Price
1996
2000
2004
2008
2012
2016
Date



These are the top 5 states ranked by their average house price over the years with analysis done forecasting the predicted prices for 5 years. They are:

1. California
2. Washington, D.C
3. Hawaii
4. Massachussets
5. New Jersey

Conclusion

Investment Potential:

The top 5 identified states show high potential for real estate investment based on historical price trends and forecasted growth. These states balance high profit potential with manageable risk levels, making them attractive for investment.

Risk Assessment:

Volatility analysis suggests that while some high-value areas are riskier, they also offer higher returns. More stable areas provide safer investment options with moderate returns.

Next Steps

Recommendations

Detailed Risk Analysis: Conduct a more detailed risk analysis for the top 5 states, considering factors like economic conditions, employment rates, and local developments.

Scenario Planning: Develop scenario plans to understand the impact of different economic conditions on real estate prices. Use these scenarios to prepare for potential market downturns or booms.

Further Model Refinement: Refine the ARIMA model by incorporating additional variables such as interest rates, employment data, and economic indicators. Explore other time series forecasting models to compare performance and improve accuracy.

Diversified Investment:

1. Invest in a diversified portfolio that includes both high-growth, high-volatility areas and stable, moderate-growth areas.
2. This strategy will balance the potential for high returns with risk management. **Long-Term Investment:**
3. Focus on long-term investment strategies in areas with strong upward trends and stable growth.
4. Short-term investments can be considered in high-volatility areas with careful monitoring of market conditions. **Continuous Monitoring:**
5. Continuously monitor market conditions and economic indicators that could impact the real estate market.
6. Adjust investment strategies based on new data and forecasts to optimize returns and manage risks