Final Project Submission - Phase 04

Student Name: GROUP 4

Members : Edward Opollo,Sharon Kimutai,Daniel Ndirangu,Vivian Waitiri,Jackson Maina,Rahma Mohamed,Cynthia Karuga

Student pace: Part Time

Scheduled Project due date: 30th August 2023

OVERVIEW

This project involves building a time series model using Zillow data to aid real estate investors in making informed investment decisions. The dataset comprises property information, and the project encompasses data preprocessing, time series transformation, exploratory data analysis, model selection, training, and evaluation. The model's objective is to forecast property price trends, which will be presented to investors through a user-friendly interface. Recommendations on where to invest will be provided based on these predictions and supplemented with additional insights from EDA. The project also includes documentation, deployment, maintenance, and a feedback loop to continuously enhance the model's accuracy and relevance to real estate investment needs.

BUSINESS UNDERSTANDING

This project will significantly enhance business understanding for real estate investors by leveraging time series analysis of Zillow data. It will provide investors with historical property price trends, helping them make data-driven investment decisions, manage risks, identify promising locations, and access forecasts through a user-friendly interface. The project's continuous improvement approach, including a feedback loop and regular updates, ensures that investors stay well-informed in a dynamic real estate market, ultimately empowering them to optimize their investments and improve their overall understanding of the industry.Key factors will be considered like pricing of houses,location and risk to recommend for 'BEST INVESTMENT'

DATA UNDERSTANDING AND PREPARATION

For data understanding and preparation, begin by thoroughly exploring the Zillow dataset, checking for missing values, and addressing outliers. Convert categorical variables like 'City,' 'State,' and 'Metro' into numerical format, possibly using one-hot encoding. Transform the dataset into a time series format, with 'RegionName' representing unique properties or regions and 'Date' as the time dimension. Calculate relevant time-based features, such as moving averages or seasonality patterns, to capture temporal trends. Additionally, split the data into training and testing sets, reserving the most recent data for validation. This will create a clean, structured dataset ready for time series modeling and forecasting.

PROJECT OBJECTIVE

The project's main objective is to develop a time series forecasting model using Zillow data to assist real estate investors in making informed decisions about where to invest their capital. This model will provide predictions and insights into property price trends over time, helping investors identify regions and cities with potential for price appreciation. Ultimately, the project aims to empower investors with data-driven tools that enhance their understanding of real estate market dynamics, enabling them to make more strategic and profitable investment choices. The key question being:What are the top 5 best zip codes for us to invest in

DATA UNDERSTANDING

```
#importing the necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import mean_squared_error as MSE
from math import sqrt
import warnings
warnings.simplefilter('ignore')
with warnings.catch_warnings():
    warnings.filterwarnings("ignore")
plt.style.use('ggplot')
%matplotlib inline
#Time series analysis tools.
from pandas.plotting import autocorrelation_plot
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

```
#importing the data and looking at the first 5 rows
df = pd.read_csv('zillow_data.csv')
df.head()
```

```
#looking at the last 5 rows
df.tail()
```

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	199
14718	58333	1338	Ashfield	MA	Greenfield Town	Franklin	14719	9460
14719	59107	3293	Woodstock	NH	Claremont	Grafton	14720	9270
14720	75672	40404	Berea	KY	Richmond	Madison	14721	5710
14721	93733	81225	Mount Crested Butte	CO	NaN	Gunnison	14722	19110
14722	95851	89155	Mesquite	NV	Las Vegas	Clark	14723	17640

5 rows × 272 columns

Column Names for Zillow DataSet

RegionID: A unique identifier for each region(zip code).

RegionName: The specific zip code for the region.

City: The city in which the zip code is located.

State: The state in which the zip code is located.

Metro: The metropolitan area to which the region belongs.

CountyName: The county in which the zip code is located.

SizeRank: A numerical rank representing the size of the zip code.

1996-04 to 2018-04: Median housing sales values for each month, spanning from April 1996 to April 2018.

```
#Analyse the dataframe
def analyze_dataset(df):

    # confirm type of df
    print(type(df))
```

```
    # Dataset shape
    print("Shape of the dataset:", df.shape,'\n')


    # Missing values
    null_counts = df.isnull().sum()
    print("Null columns only:", null_counts[null_counts > 0])


    # Duplicate values
    print("Number of duplicates:", len(df.loc[df.duplicated()]),'\n')


    # Number of columns
    num_columns = len(df.columns)
    print("Number of columns:", num_columns)


    # Unique values
    print("The unique values per column are:")
    print(df.nunique(), '\n')


    # Dataset information
    print("Information about the dataset:")
    print(df.info())


    # Distribution
    display(df.describe())

analyze_dataset(df)
```

```
<class 'pandas.core.frame.DataFrame'>
Shape of the dataset: (14723, 272)

Null columns only: 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
Number of duplicates: 0

Number of columns: 272
The unique values per column are:
RegionID      14723
RegionName    14723
City           7554
State            51
Metro           701
              ...
2017-12        5248
2018-01        5276
2018-02        5303
2018-03        5332
2018-04        5310
Length: 272, dtype: int64

Information about the dataset:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14723 entries, 0 to 14722
Columns: 272 entries, RegionID to 2018-04
dtypes: float64(219), int64(49), object(4)
memory usage: 30.6+ MB
None
```

```
#Create a copy of the dataframe
df2 = df.copy()
df2.head(5)
```

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	
0	84654	60657	Chicago	IL	Chicago	Cook	1	334200.0	33
					Dallas-				

Summary of Dataframe

- This dataset contains information about median housing sales values for various zip codes over a span of 22 years, from April 1996 to April 2018.
- There are 272 columns and 14723 rows indexed from 0 to 14722. This means there are 272 different variables each with 14723 records.
- The first 7 columns are named , **RegionID, RegionName, City, State, Metro, CountyName, SizeRank** while the other 265 columns are dates from **April 1996 to April 2018.**
- Data attributes:
  - RegionID: A unique index for each region (zip code) ranging from 58196 to 753844.
  - RegionName: A unique zip code for each region, ranging from 1001 to 99901.
  - SizeRank: A numerical rank representing the size of each zip code, ranked from 1 to 14723.
  - 1996-04 to 2018-04: Median housing sales values for each month, covering a total of 265 data points for each zip code.
- Summary Statistics:
  - The median (50th percentile) size rank is approximately 46106, indicating the middle-sized zip code in the dataset.
  - The median housing sales value across all zip codes ranges from around USD11,300 to USD 3,849,600.
  - The mean (average) housing sales value across all zip codes ranges from approximately USD 118,299 to USD 288,039.
  - The standard deviation indicates the variability in housing sales values, with values ranging from around USD 42,500 to USD 372,054.
- There are three main data types in our dataset:
  - 219 columns with the floating numbers data type
  - 49 columns with the integer data type.
  - 4 columns with the object data type.
  - There are 220 columns with missing values. One being Metro, which is a categorical column and the numerical columns which are represented by dates. No duplicates were

identified in any of the columns.

The dataset consumes approximately 30.6 megabytes of memory.

```
#Converting column names to datetime
def get_datetimes(df2):
    """
    Takes a dataframe:
    returns only those column names that can be converted into datetime objects
    as datetime objects.
    NOTE number of returned columns may not match total number of columns in passed da
    """

    return pd.to_datetime(df2.columns.values[7:], format='%Y-%m')
```

Data Preparation and Cleaning

During this stage, the data undergoes cleaning and preparation to ensure its quality and reliability for subsequent analysis. The process begins by:

1. ROI as our meassure of best zipcodes to invest.
2. Compare all zip codes and select the best 5 based on ROI
3. Evaluate each individual zip code to determine trends and seasonality
4. Detecting and addressing any missing values present
5. Detrend the data using one of the several options

```
# Making a dataframe of just New York
df_ny = df2[df2['State'] == 'NY']

# Seeing how many rows we get
print(df_ny.shape)

# Sanity check
df_ny.head()
```

(1015, 272)

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	1996-0
6	61807	10467	New York	NY	New York	Bronx	7	152900.0	152700
10	62037	11226	New York	NY	New York	Kings	11	162000.0	162300

```
# Checking our dataframe for NaN values
print(f'There are {df_ny.isna().sum().sum()} NaNs in our original dataframe')

# Backfilling that single NaN
df_ny.fillna(method='ffill', inplace=True)

# Sanity check
print(f'There are {df_ny.isna().sum().sum()} NaNs after using forwardfill')
```

```
There are 4012 NaNs in our original dataframe
There are 0 NaNs after using forwardfill
```

```
# Getting a list of the values for the last date in our time series
current_median_msa_home_prices = list(df_ny['2018-04'])

# Plotting the results
fig, ax = plt.subplots(figsize=(5,5))
plt.hist(current_median_msa_home_prices, bins=20)
plt.title('2018 NY State Median Home Price by Metro Area')
plt.xlabel('Median Home Price')
plt.ylabel('Count')
plt.show()
```

Zipcode Selection

In order to determine the best States to focus on for the real estate investment, we consider the below:

1. Calculating the **Return on Investment**(ROI) in percentage.

   ROI = (Final Value/Intial Value) - 1

2. Reviewing the Risk Assessment/ Volatility.

   This helps us understand the volatility in housing prices by utilising **standard deviation** and **mean** to calculate the **coefficient of variance**


```
# Assuming historical return on investment for 'df_ny'
df_ny['ROI'] = (df_ny['2018-04'] / df_ny['1996-04']) - 1

# Calculate standard deviation of monthly values for 'df_ny'
df_ny['std'] = df_ny.loc[:, '1996-04':'2018-04'].std(skipna=True, axis=1)

# Calculate historical mean value for  'df_ny'
df_ny['mean'] = df_ny.loc[:, '1996-04':'2018-04'].mean(skipna=True, axis=1)

# Calculate coefficient of variance for 'df_ny'
df_ny['CV'] = df_ny['std'] / df_ny['mean']

# Show calculated values
df_ny[['RegionName', 'std', 'mean', 'ROI', 'CV']].head()
```

	RegionName	std	mean	ROI	CV
6	10467	8.569914e+04	2.923392e+05	1.733159	0.293150
10	11226	2.080187e+05	4.614242e+05	4.945679	0.450819
12	11375	2.240221e+05	6.081170e+05	3.297147	0.368387
13	11235	1.665122e+05	4.771932e+05	3.284514	0.348941
20	10011	4.193280e+06	4.801772e+06	59.253543	0.873278

```
# Define upper limit of CV according to risk profile for 'best5'.
upper_cv = df_ny['CV'].quantile(0.4)
print(f'\nCV upper limit: {upper_cv}')
# Get the 5 regionname with highest ROIs within the firm's risk profile for 'best5'.
RN_best5 = df_ny[df_ny['CV'] < upper_cv].sort_values('ROI', ascending=False).head(5)
print('\nBest 5 RegionName:')
print(RN_best5[['RegionName', 'ROI', 'CV']])
```

```
CV upper limit: 0.23863326394450965

Best 5 RegionName:
       RegionName       ROI        CV
14116       14065  1.660714  0.223623
12946       13040  1.566396  0.226185
8576        11771  1.501795  0.237330
11925       13491  1.410023  0.218232
12982       12154  1.377880  0.232964
```

```
RN_best5.head()
```

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	19
14116	63398	14065	Freedom	NY	Olean	Cattaraugus	14117	392(
12946	62914	13040	Cincinnatus	NY	Cortland	Cortland	12947	369(
8576	62245	11771	Oyster Bay	NY	New York	Nassau	8577	3342(
11925	63155	13491	Winfield	NY	Utica	Herkimer	11926	439(
12982	62431	12154	Schaghticoke	NY	Albany	Rensselaer	12983	868(

5 rows × 276 columns

```
RN_best5.describe()
```

	RegionID	RegionName	SizeRank	1996-04	1996-05	1996-0
count	5.000000	5.000000	5.000000	5.000000	5.00000	5.00000
mean	62828.600000	12904.200000	12110.000000	108200.000000	108580.00000	108980.00000

```
RN_best5['location'] = RN_best5['City'] + ", " + RN_best5['State']
```

```
best_5_RN_with_location = RN_best5[['RegionName','location']]
print(best_5_RN_with_location)
```

```
       RegionName          location
14116       14065       Freedom, NY
12946       13040   Cincinnatus, NY
8576        11771    Oyster Bay, NY
11925       13491      Winfield, NY
12982       12154  Schaghticoke, NY
```

```
RN_best5.columns
```

```
Index(['RegionID', 'RegionName', 'City', 'State', 'Metro', 'CountyName',
       'SizeRank', '1996-04', '1996-05', '1996-06',
       ...
       '2017-12', '2018-01', '2018-02', '2018-03', '2018-04', 'ROI', 'std',
       'mean', 'CV', 'location'],
      dtype='object', length=277)
```

- The "Best 5 RN" section presents the top 5 zip codes namely: **Freedom, NY:14065, Cincinnatus, NY:13040,Oyster Bay, NY:11771,Winfield, NY:13491,12154 Schaghticoke, NY:12154** They have been identified as the prime investment locations offering highest Return on Investment (ROI) while adhering to the risk profile's CV upper limit.
- The descriptive statistics of CV values indicate that the variability in median housing sales values across the selected zip codes is quite significant.The CV upper limit based on the risk profile is approximately 0.24. This limit helps define a threshold for selecting zip codes with acceptable levels of risk. For these top 5 zip codes, the ROIs range from 137.79% to 166.0%, indicating substantial growth in median housing sales values over the specified time period.

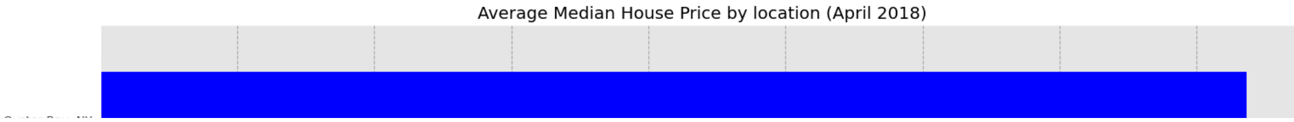
EDA

Univariate analysis

Average median house price for each state

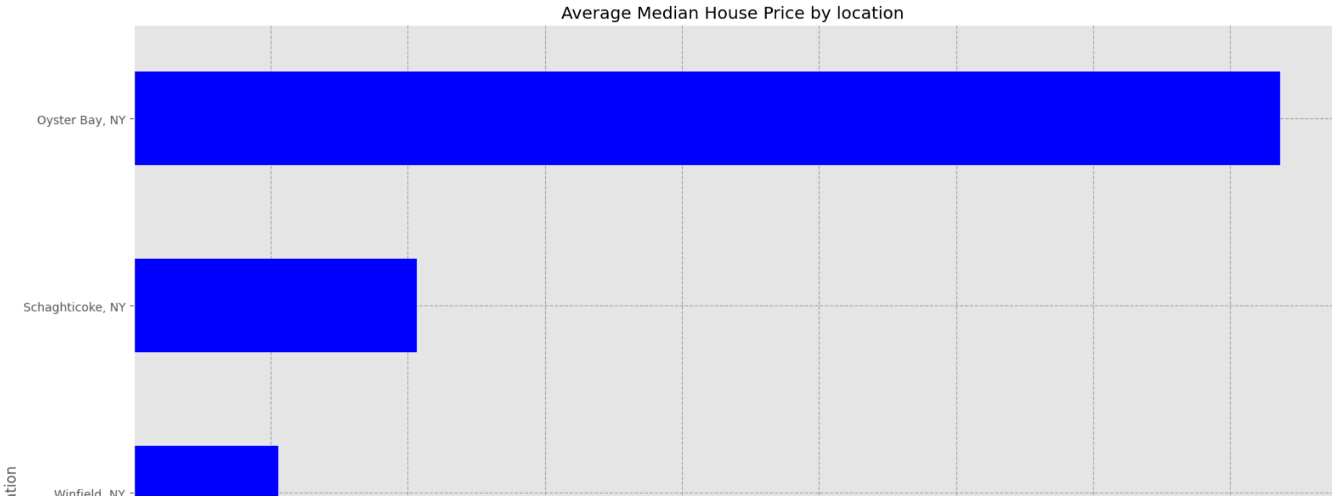
It is noticeable that Oyester Bay,NY emerges with the highest value, standing at around USD 825,000. Cincinnatus, NY exhibits the lowest average median property price at USD 95,000.

```
# Calculating the average median house price for each state in April 2018
locationprice = RN_best5.groupby('location')['2018-04'].mean().sort_values()

# Plotting the average median house price by state for April 2018
plt.figure(figsize=(16, 12))
locationprice.plot(kind='barh', color='blue')
plt.title('Average Median House Price by location (April 2018)')
plt.xlabel('Average Median House Price')
plt.ylabel('location')
plt.grid(True, which="both", ls="--", c='0.65')
plt.tight_layout()
plt.show()
```



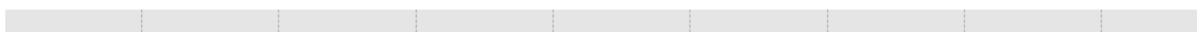
Average Median House Price by Location

```
# Select the top 5 locations
top5_locations = RN_best5['location'].value_counts().nlargest(5).index

# Filter the data for the top 5 locations
filtered_data = RN_best5[RN_best5['location'].isin(top5_locations)]

# Calculate the average median house price for April 2018 for the top 5 locations
locationprice = filtered_data.groupby('location')['2018-04'].mean().sort_values()

# Plotting the average median house price by state for April 2018
plt.figure(figsize=(16, 12))
locationprice.plot(kind='barh', color='blue')
plt.title('Average Median House Price by location ')
plt.xlabel('Average Median House Price')
plt.ylabel('Location')
plt.grid(True, which="both", ls="--", c='0.65')
plt.tight_layout()
plt.show()
```

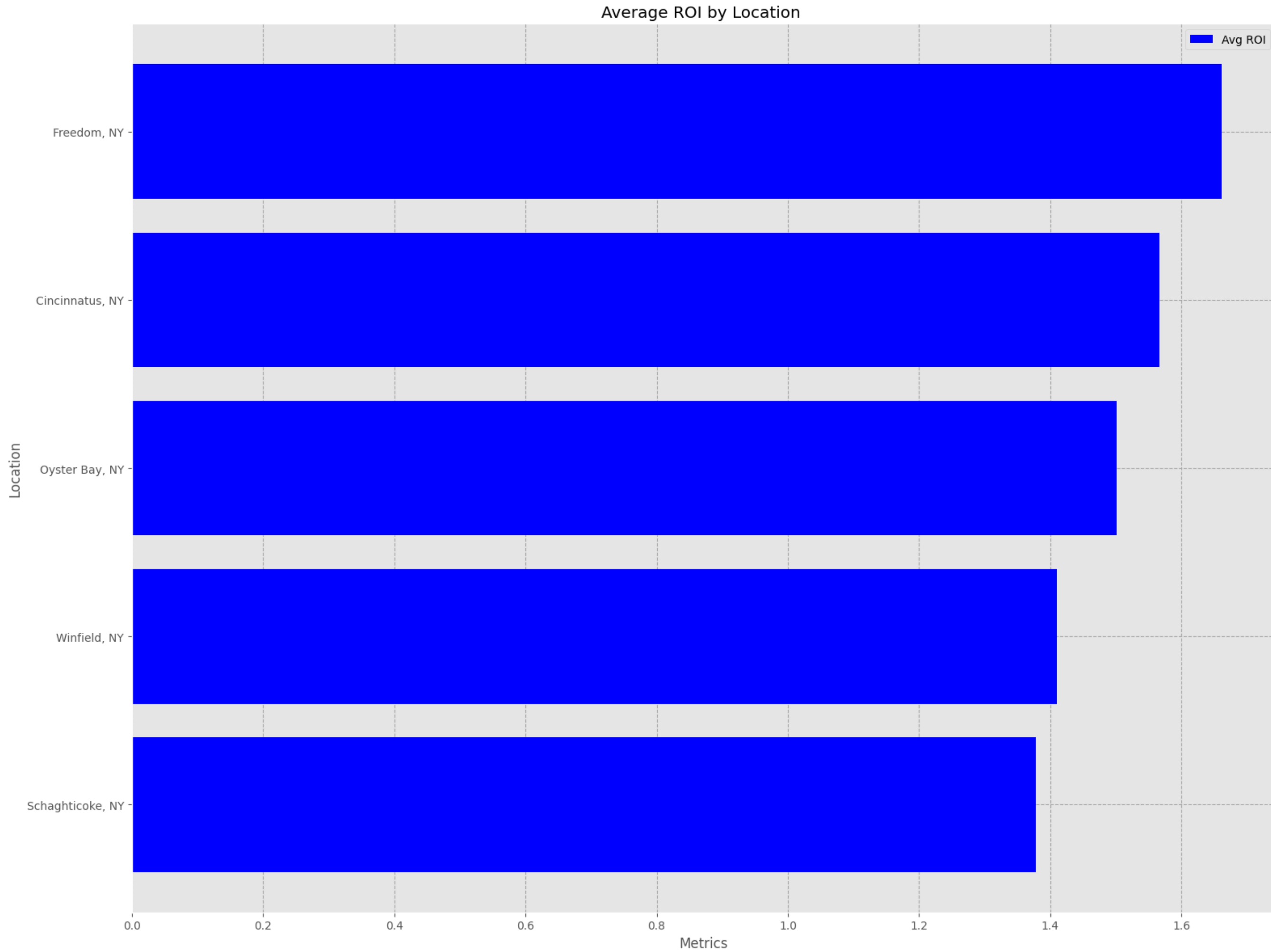


The Average ROI for each location

In terms of Average ROI for each location, Freedom,NY emerges with the highest value at 1.66, closely trailed by Cincinnatus,NY. Schaghticoke exhibits the lowest ROI of 1.378. Winfield secures the third position among the four states.



```
# Calculate the average ROI for each location
location_roi = RN_best5.groupby('location')['ROI'].mean().sort_values()

# Filter locations based on ROI (for example, select locations with positive ROI)
positive_roi_locations = location_roi[location_roi > 0]

# Plotting the average median house price and ROI by location for April 2018
plt.figure(figsize=(16, 12))
plt.barh(positive_roi_locations.index, location_roi[positive_roi_locations.index], col
plt.title('Average ROI by Location ')
plt.xlabel('Metrics')
plt.ylabel('Location')
plt.legend()
plt.grid(True, which="both", ls="--", c='0.65')
plt.tight_layout()
plt.show()
```



BIVARIATE ANALYSIS

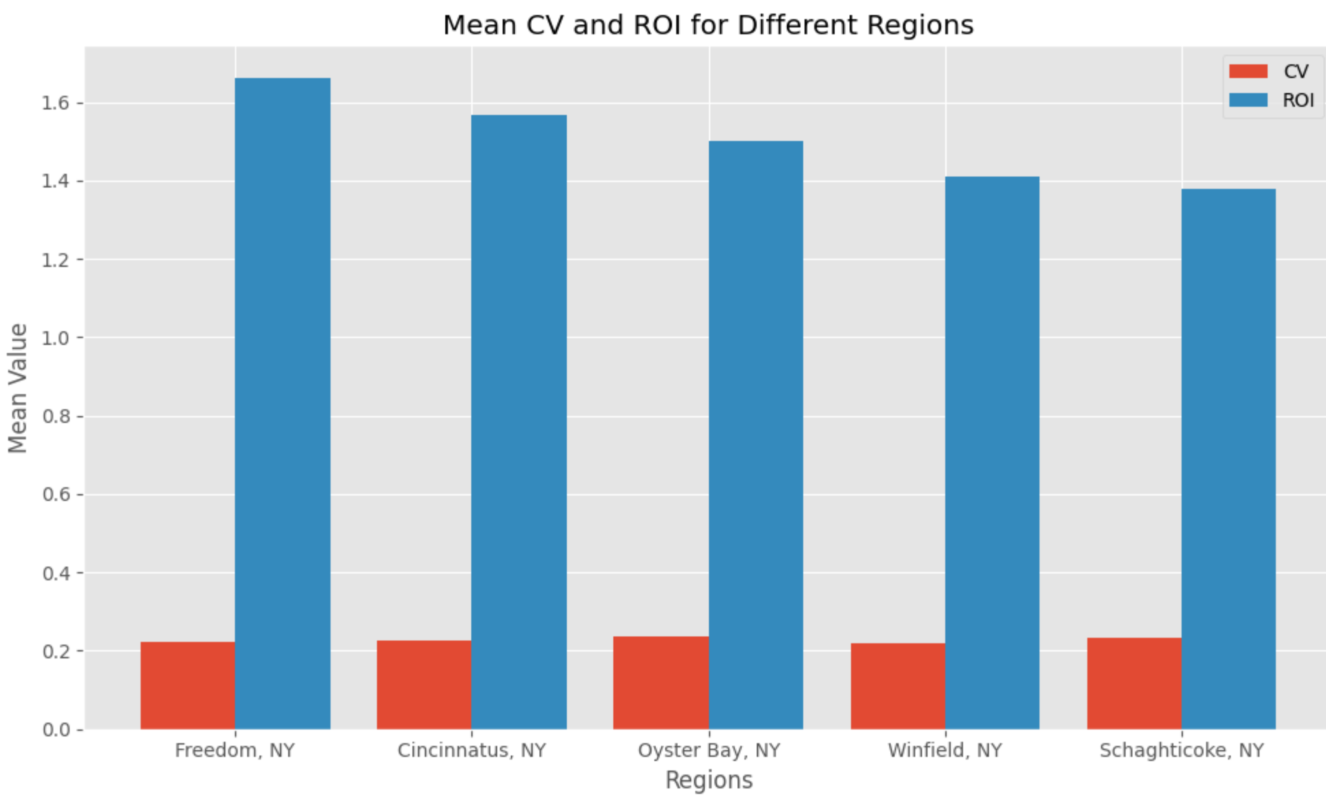
Mean CV and ROI for Different Regions

It is noticeable that Oyester Bay has the highest mean CV of 0.237 while Winfield has the lowest cv of 0.218

On the otherhand, Freedom ,NY secures the highest ROI of 1.667, closely trailed by Cincinnatus.Schaghticoke scoring the least ROI of 1.378

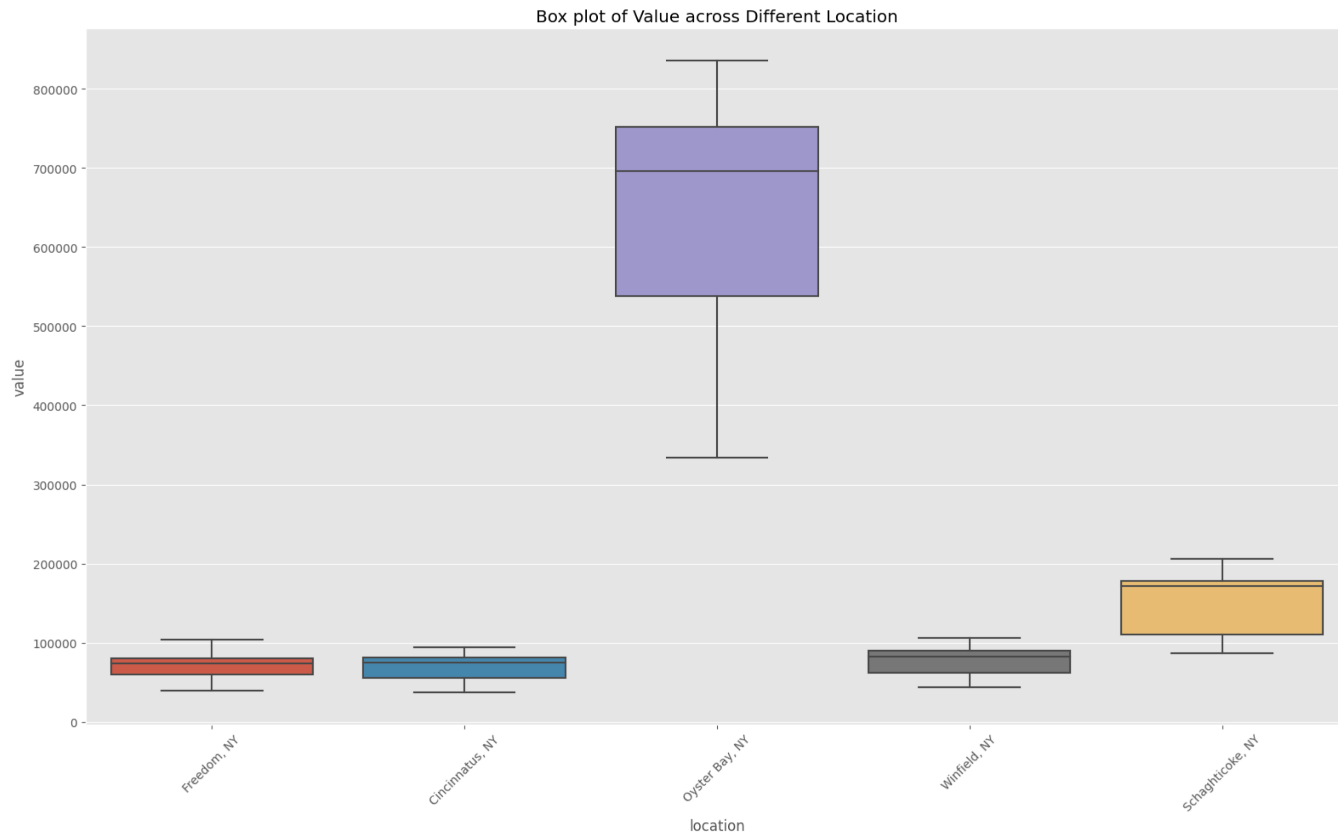
```
cv_values = []
roi_values = []

unique_regions = RN_best5['location'].unique()

for region in unique_regions:
    region = RN_best5[RN_best5['location'] == region]
    cv_mean = region['CV'].mean()
    roi_mean = region['ROI'].mean()
    cv_values.append(cv_mean)
    roi_values.append(roi_mean)
```

```
# Set up positions for the bars
x = np.arange(len(unique_regions))
width = 0.4

# Create the bar graph
plt.figure(figsize=(10, 6))
plt.bar(x - width/2, cv_values, width, label='CV')
plt.bar(x + width/2, roi_values, width, label='ROI')
plt.xlabel('Regions')
plt.ylabel('Mean Value')
plt.title('Mean CV and ROI for Different Regions')
plt.xticks(x, unique_regions)
plt.legend()
plt.tight_layout()
plt.show()
```



Value across Different Location

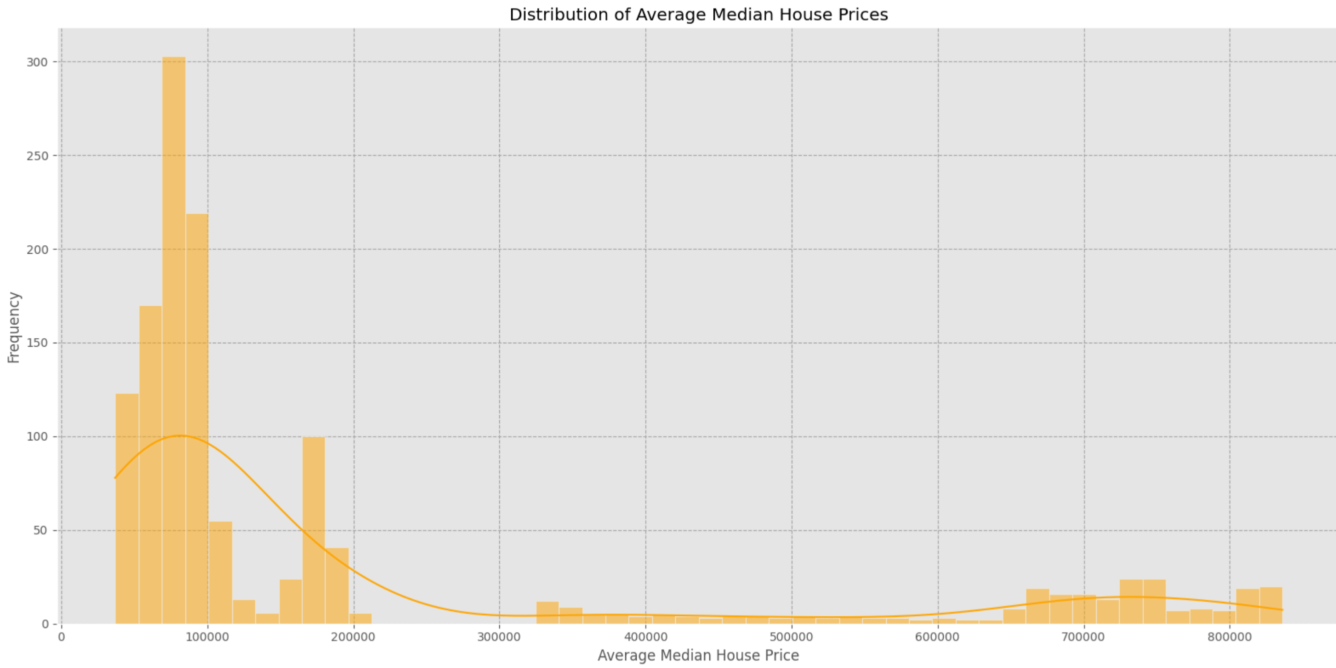
```
# Box plots of ROI by location
# Create a box plot for value across different States
plt.figure(figsize=(16, 10))
sns.boxplot(x=melted_df['location'], y=melted_df['value'])
plt.title('Box plot of Value across Different Location')
plt.xlabel('location')
plt.ylabel('value')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Average Median House Price

```
# Plotting the distribution of median house prices from the sample data
plt.figure(figsize=(16, 8))
sns.histplot(melted_df['value'], bins=50, kde=True, color='orange')
plt.title('Distribution of Average Median House Prices')
```

```
plt.xlabel('Average Median House Price')
plt.ylabel('Frequency')
plt.grid(True, which="both", ls="--", c='0.65')
plt.tight_layout()
plt.show()
```



Reshaping the data

In this step, we reshape the dataset (RN_best5) from a wide format and transform it into a long form datetime dataframe. This restructure facilitates time based analysis.

```
RN_best5.columns
```

```
Index(['RegionID', 'RegionName', 'City', 'State', 'Metro', 'CountyName',
       'SizeRank', '1996-04', '1996-05', '1996-06',
       ...
       '2017-12', '2018-01', '2018-02', '2018-03', '2018-04', 'ROI', 'std',
       'mean', 'CV', 'location'],
      dtype='object', length=277)
```

```
import pandas as pd
```

```
def get_datetimes(RN_best5):
    """
    Takes a dataframe:
    returns only those column names that can be converted into datetime objects
    as datetime objects.
    NOTE number of returned columns may not match total number of columns in passed da
    """
    return pd.to_datetime(RN_best5.columns.values[7:], format='%Y-%m')

melted_df = pd.melt(RN_best5,
                    id_vars=['RegionID', 'RegionName', 'City', 'State', 'Metro', 'Cour
                             'SizeRank', 'ROI', 'std', 'mean', 'CV', 'location'],
                    var_name='time')

melted_df['time'] = pd.to_datetime(melted_df['time'], infer_datetime_format=True)

melted_df = melted_df.dropna(subset=['SizeRank'])

melted_df.set_index('time', inplace=True)

melted_df.head(5)
```

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	F
time								
1996-04-01	63398	14065	Freedom	NY	Olean	Cattaraugus	14117	1.660
1996-04-01	62914	13040	Cincinnatus	NY	Cortland	Cortland	12947	1.566
1996-04-01	62245	11771	Oyster Bay	NY	New York	Nassau	8577	1.501
1996-04-01	63155	13491	Winfield	NY	Utica	Herkimer	11926	1.410
1996-04-01	62431	12154	Schaghticoke	NY	Albany	Rensselaer	12983	1.377

```
melted_df.columns
```

```
Index(['RegionID', 'RegionName', 'City', 'State', 'Metro', 'CountyName',
       'SizeRank', 'ROI', 'std', 'mean', 'CV', 'location', 'value'],
      dtype='object')
```

```
melted_df.isnull().sum()
```

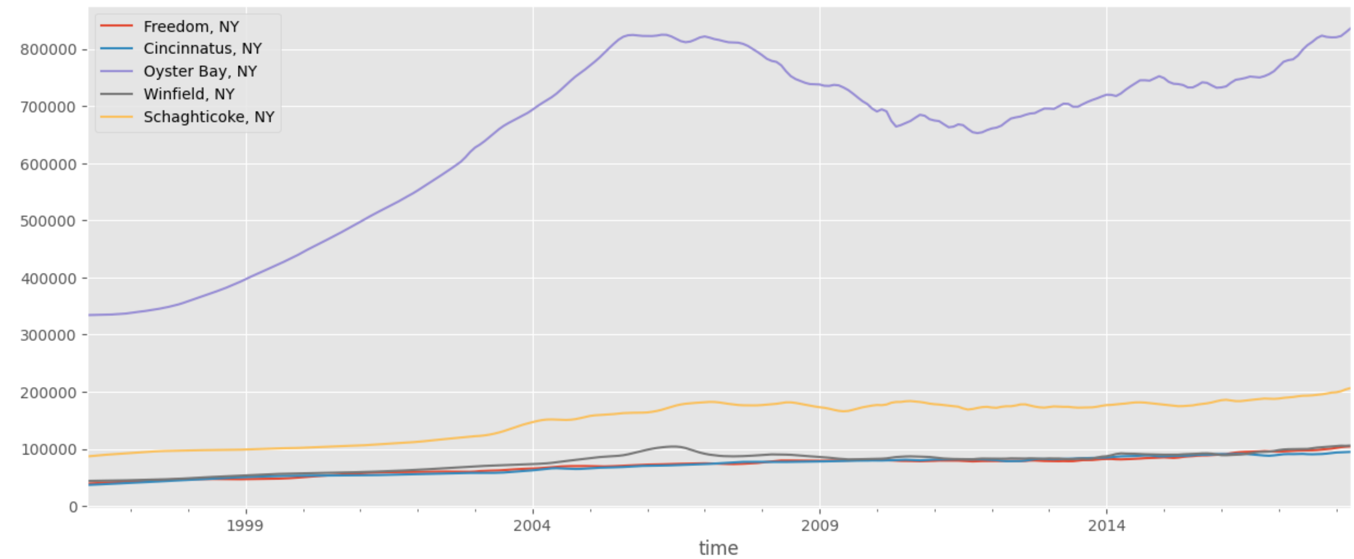
```
RegionID      0
RegionName    0
City          0
State         0
Metro         0
CountyName    0
SizeRank      0
ROI           0
std           0
mean          0
CV            0
location      0
value         0
dtype: int64
```

```
#Summary of melted data
melted_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1325 entries, 1996-04-01 to 2018-04-01
Data columns (total 13 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   RegionID    1325 non-null   int64
 1   RegionName  1325 non-null   int64
 2   City        1325 non-null   object
 3   State       1325 non-null   object
 4   Metro       1325 non-null   object
 5   CountyName  1325 non-null   object
 6   SizeRank    1325 non-null   int64
 7   ROI         1325 non-null   float64
 8   std         1325 non-null   float64
 9   mean        1325 non-null   float64
 10  CV          1325 non-null   float64
 11  location    1325 non-null   object
 12  value       1325 non-null   float64
dtypes: float64(5), int64(3), object(5)
memory usage: 144.9+ KB
```

Time series Modelling

```
 # Get unique locations from the 'location' column
locations = melted_df['location'].unique()

# Loop through the unique locations and plot the 'value' column for each location
for location in locations:
    subset = melted_df[melted_df['location'] == location]
    subset['value'].plot(label=location, figsize=(15, 6))

plt.legend()
plt.show()
```

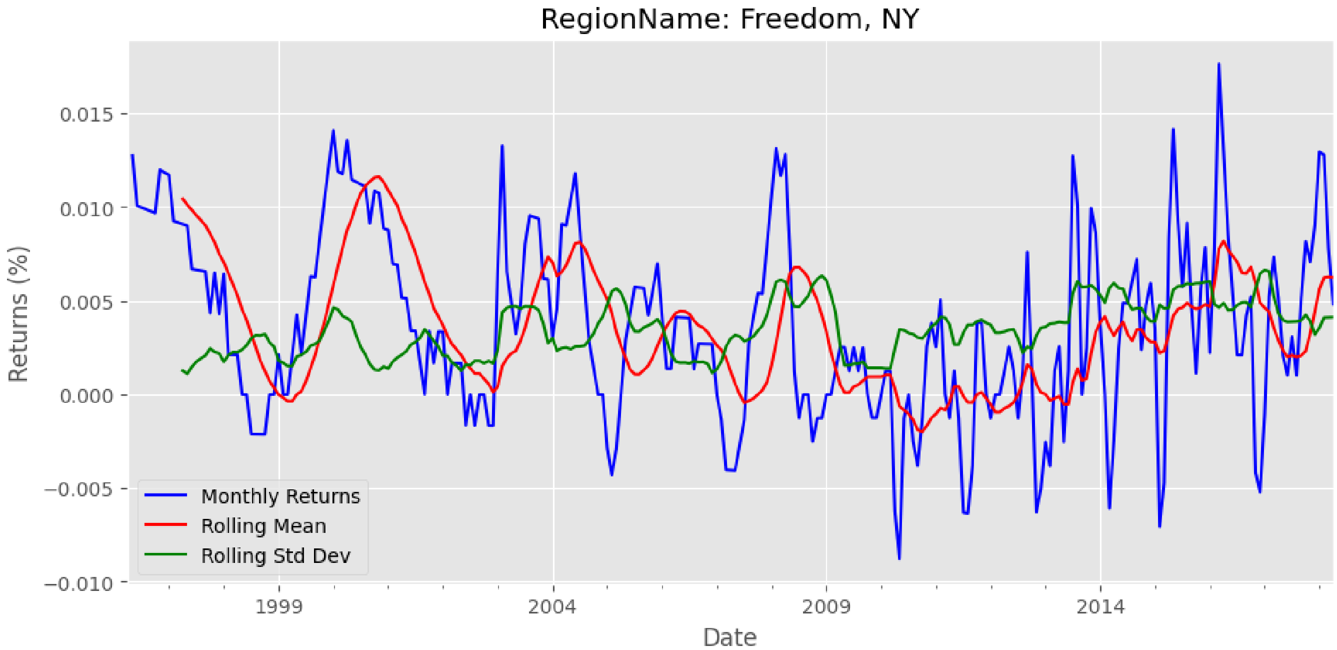


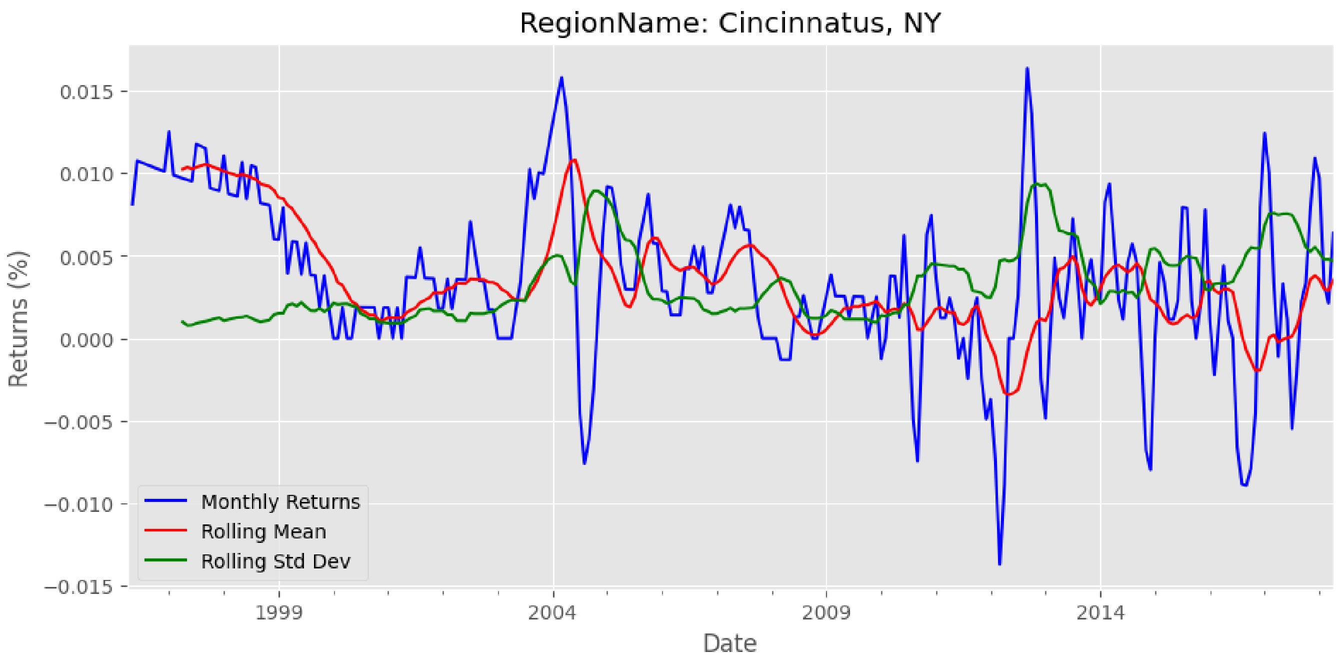

```
print("Unique RegionNames:", location)
```

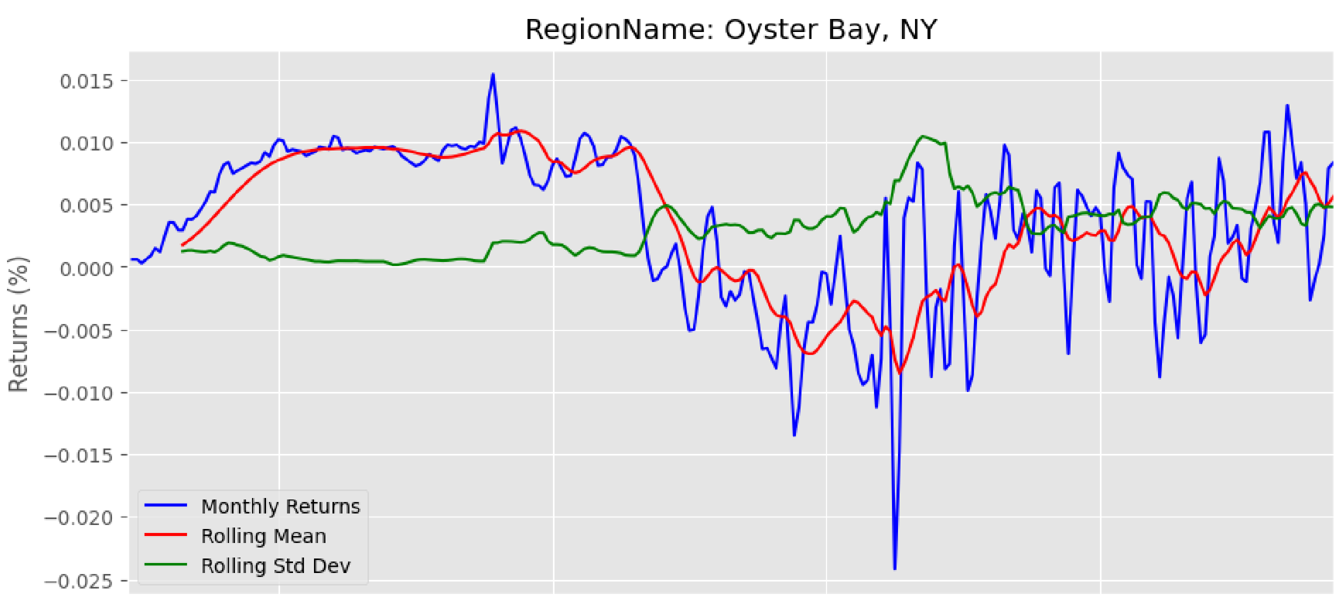
```
Unique RegionNames: Schaghticoke, NY
```

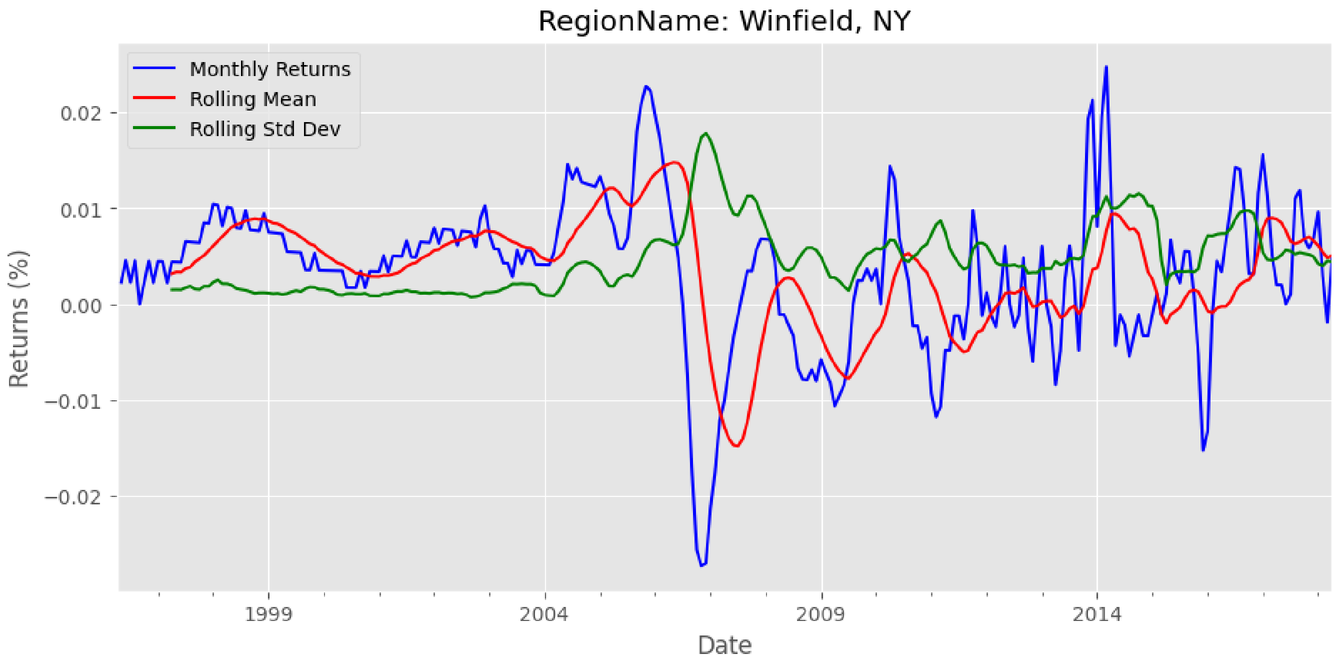
```
import numpy as np
# Unique RegionNames from melted_df
unique_RegionNames = melted_df['location'].unique()
# Add a new 'ret' column to df_melted to store monthly returns
melted_df['ret'] = np.nan
# Calculate monthly returns for each unique zip code
for RegionName in unique_RegionNames:
    subset = melted_df[melted_df['location'] == RegionName].copy()
    for i in range(len(subset) - 1):
        subset['ret'].iloc[i + 1] = (subset['value'].iloc[i + 1] / subset['value'].ilo
    melted_df.loc[melted_df['location'] == RegionName, 'ret'] = subset['ret']


# Plotting the monthly returns along with rolling mean and std for each unique zip coo
for RegionName in unique_RegionNames:
    subset = melted_df[melted_df['location'] == RegionName]

    # Calculate rolling mean and rolling standard deviation
    rolling_mean = subset['ret'].rolling(window=12).mean()
    rolling_std = subset['ret'].rolling(window=12).std()

    plt.figure(figsize=(11,5))
    subset['ret'].plot(color='b', label='Monthly Returns')
    rolling_mean.plot(color='red', label='Rolling Mean')
```

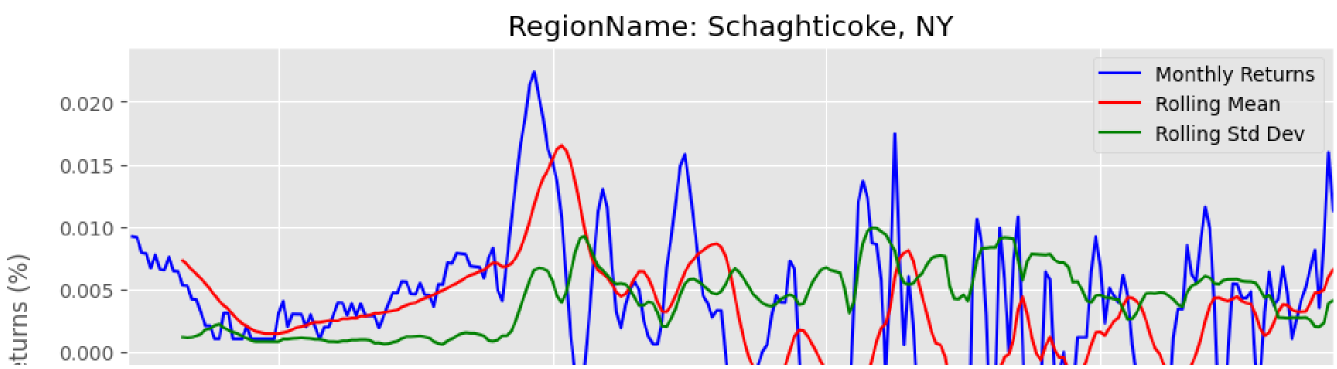
```
    rolling_std.plot(color='green', label='Rolling Std Dev')

    plt.title(f'RegionName: {RegionName}')
    plt.xlabel('Date')
    plt.ylabel('Returns (%)')
    plt.legend(loc='best')
    plt.show()
```


RegionName: Freedom, NY
Returns (%)
Date
Monthly Returns
Rolling Mean
Rolling Std Dev
0.015
0.010
0.005
0.000
−0.005
−0.010
1999
2004
2009
2014


RegionName: Cincinnatus, NY
Returns (%)
Date
Monthly Returns
Rolling Mean
Rolling Std Dev
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
1999
2004
2009
2014


RegionName: Oyster Bay, NY
Returns (%)
Monthly Returns
Rolling Mean
Rolling Std Dev
0.015
0.010
0.005
0.000
−0.005
−0.010
−0.015
−0.020
−0.025






```
from statsmodels.tsa.stattools import adfuller

# Extracting unique RegionNames
unique_RegionNames = melted_df['RegionName'].unique()

# Set a minimum threshold for the number of data points required for the test
min_data_points = 20  # Adjust this threshold as needed

# Loop through each unique RegionName and perform Dickey-Fuller test on their 'ret' va
for RegionName in unique_RegionNames:
    subset = melted_df[melted_df['RegionName'] == RegionName]

    # Check if the subset has enough data points for the test
    if len(subset['ret'].dropna()) >= min_data_points:
        print(f"Results of Dickey-Fuller Test for RegionName: {RegionName}\n")
        dftest = adfuller(subset['mr'].dropna())

        # Extract and display test results in a user-friendly manner
        dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value', '#Lags U
        for key, value in dftest[4].items():
            dfoutput[f'Critical Value ({key})'] = value
        print(dfoutput)
```

```
        print('-' * 50)
    else:
        print(f"RegionName: {RegionName} does not have enough data points for the test
```

```
Results of Dickey-Fuller Test for RegionName: 14065

Test Statistic                  -3.905011
p-value                          0.001998
#Lags Used                       8.000000
Number of Observations Used    255.000000
Critical Value (1%)             -3.456257
Critical Value (5%)             -2.872942
Critical Value (10%)            -2.572846
dtype: float64
--------------------------------------------------
Results of Dickey-Fuller Test for RegionName: 13040

Test Statistic                  -5.619337
p-value                          0.000001
#Lags Used                       2.000000
Number of Observations Used    261.000000
Critical Value (1%)             -3.455656
Critical Value (5%)             -2.872678
Critical Value (10%)            -2.572705
dtype: float64
--------------------------------------------------
Results of Dickey-Fuller Test for RegionName: 11771

Test Statistic                  -1.393059
p-value                          0.585599
#Lags Used                      15.000000
Number of Observations Used    248.000000
Critical Value (1%)             -3.456996
Critical Value (5%)             -2.873266
Critical Value (10%)            -2.573019
dtype: float64
--------------------------------------------------
Results of Dickey-Fuller Test for RegionName: 13491

Test Statistic                  -3.092350
p-value                          0.027125
#Lags Used                      12.000000
Number of Observations Used    251.000000
Critical Value (1%)             -3.456674
Critical Value (5%)             -2.873125
Critical Value (10%)            -2.572944
dtype: float64
--------------------------------------------------
Results of Dickey-Fuller Test for RegionName: 12154

Test Statistic                  -2.251611
p-value                          0.188049
#Lags Used                      16.000000
Number of Observations Used    247.000000
Critical Value (1%)             -3.457105
```

```
Critical Value (5%)            -2.873314
Critical Value (10%)           -2.573044
dtype: float64
--------------------------------------------------
```

```
melted_df['ret'].dropna()
```

```
time
1996-05-01    0.012755
1996-05-01    0.008130
1996-05-01    0.000598
1996-05-01    0.002278
1996-05-01    0.009217
                ...
2018-04-01    0.004817
2018-04-01    0.006376
2018-04-01    0.008321
2018-04-01    0.004748
2018-04-01    0.011269
Name: ret, Length: 1320, dtype: float64
```

```
# Check for missing values in the 'ret' column
missing_values = melted_df['ret'].isnull().sum()

# Print the number of missing values
print(f"Number of missing values in 'ret' column: {missing_values}")

# Drop rows with missing values in the 'ret' column
melted_df.dropna(subset=['ret'], inplace=True)
print(f"Number of missing values in 'ret' column: {missing_values}")
```

```
Number of missing values in 'ret' column: 5
Number of missing values in 'ret' column: 5
```

```
melted_df['ret'].isnull().sum()
```

```
0
```

```
melted_df.isnull().sum()
```

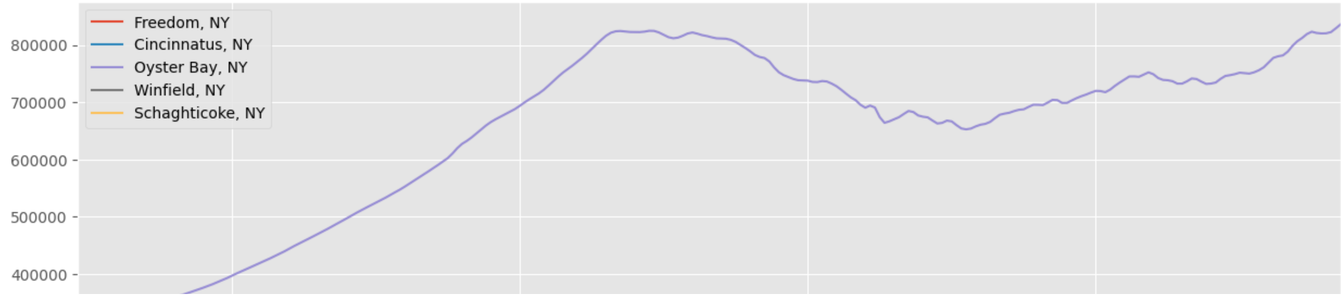
```
RegionID      0
RegionName    0
City          0
State         0
Metro         0
CountyName    0
SizeRank      0
ROI           0
std           0
mean          0
CV            0
```

```
location      0
value         0
mr            0
ret           0
dtype: int64
```

```
mean_mr = melted_df['ret'].mean()
melted_df['ret'].fillna(mean_mr, inplace=True)
```

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Apply seasonal decomposition
decomposition = seasonal_decompose(melted_df['value'], model='additive', period=12)

# Plot the decomposed time series components
fig, axes = plt.subplots(ncols=1, nrows=4, figsize=(18, 12))

# Original Time Series
axes[0].plot(decomposition.observed, color='magenta')
axes[0].set_title('Observed')
axes[0].set_ylabel('Value')
axes[0].set_xlim([decomposition.observed.index.min(), decomposition.observed.index.max

# Trend component
axes[1].plot(decomposition.trend, color='magenta')
axes[1].set_title('Trend')
axes[1].set_ylabel('Value')
axes[1].set_xlim([decomposition.trend.index.min(), decomposition.trend.index.max()])

# Seasonal component
axes[2].plot(decomposition.seasonal, color='magenta')
axes[2].set_title('Seasonal')
axes[2].set_ylabel('Value')
axes[2].set_xlim([decomposition.seasonal.index.min(), decomposition.seasonal.index.max

# Residual component
axes[3].plot(decomposition.resid, color='magenta')
axes[3].set_title('Residual')
axes[3].set_ylabel('Value')
axes[3].set_xlim([decomposition.resid.index.min(), decomposition.resid.index.max()])

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


```
import matplotlib.pyplot as plt
# Unique zip codes from df_melted
unique_zipcodes = melted_df['location'].unique()
# Plotting for the first 5 unique zip codes
for zipcode in unique_zipcodes[:5]:
    subset = melted_df[melted_df['location'] == zipcode]
    subset['value'].plot(label=zipcode, figsize=(15, 6))
    plt.legend()
```




```
time_intervals = df.index.to_series().diff().dropna()
consistent_intervals = time_intervals.unique().shape[0] == 1
if consistent_intervals:
    print("Time intervals are consistent.")
else:
    print("Time intervals are not consistent.")
```

```
Time intervals are consistent.
```

```
import numpy as np
import pandas as pd
def difference_series(series_i, lag=1):
    '''Takes in a series and returns the differenced version of that series'''
    diff_series = series_i.diff(periods=lag)
    dropped_nans = diff_series.dropna()
    return dropped_nans
# Load your data into the 'melted_df' DataFrame
# (Assuming melted_df contains your data)
# Specify the unique RegionName you want to analyze (11771 in this case)
target_region = 11771
# Specify the column name you want to analyze (e.g., 'ret')
target_column = 'mr'
# Filter the melted DataFrame for the specific RegionName
region_subset = melted_df[melted_df['RegionName'] == target_region]
# Apply differencing using the provided function
lag = 1  # Adjust this as needed
differenced_column = difference_series(region_subset[target_column], lag=lag)
# Calculate rolling mean of the differenced column
rolling_mean = differenced_column.rolling(window=12).mean()
# Detrend the differenced column by subtracting the rolling mean
detrended_series = differenced_column - rolling_mean
# Add the detrended series back to the main DataFrame
region_subset['detrended'] = detrended_series
# ... rest of the code for plotting ...
```

```
import numpy as np
import pandas as pd

def log_transform(series_i):
    '''Takes in a series and returns the log transformed version of that series'''
```

```
        log_transformed = np.log(series_i)
        dropped_nans = log_transformed.dropna()
        return dropped_nans

    def difference_series(series_i, lag=1):
        '''Takes in a series and returns the differenced version of that series'''
        diff_series = series_i.diff(periods=lag)
        dropped_nans = diff_series.dropna()
        return dropped_nans

    # Load your data into the 'melted_df' DataFrame
    # (Assuming melted_df contains your data)

    # Specify the unique RegionName you want to analyze (11771 in this case)
    target_region = 11771

    # Specify the column name you want to analyze (e.g., 'ret')
    target_column = 'mr'

    # Filter the melted DataFrame for the specific RegionName
    region_subset = melted_df[melted_df['RegionName'] == target_region]

    # Apply differencing using the provided function
    lag = 1  # Adjust this as needed
    differenced_column = difference_series(region_subset[target_column], lag=lag)

    # Calculate rolling mean of the differenced column
    rolling_mean = differenced_column.rolling(window=12).mean()

    # Detrend the differenced column by subtracting the rolling mean
    detrended_series = differenced_column - rolling_mean

    # Add the detrended series back to the main DataFrame
    region_subset['detrended'] = detrended_series

    # ... rest of the code for plotting ...



    # Assuming you have a DataFrame called 'melted_df'
    nan_count = region_subset.isna().sum()

    # If you want to count NaN values in a specific column (e.g., 'ret')
    ret_nan_count = region_subset['mr'].isna().sum()

    # To count NaN values in the entire DataFrame
    total_nan_count = region_subset.isna().sum().sum()

    print("NaN count per column:")
    print(nan_count)
```

```
print("NaN count in 'ret' column:")
print(ret_nan_count)

print("Total NaN count in the DataFrame:")
print(total_nan_count)
```

```
NaN count per column:
RegionID       0
RegionName     0
City           0
State          0
Metro          0
CountyName     0
SizeRank       0
ROI            0
std            0
mean           0
CV             0
location       0
value          0
mr             0
ret            0
detrended     12
dtype: int64
NaN count in 'ret' column:
0
Total NaN count in the DataFrame:
12
```

```
# Drop rows with NaN values in the 'mr' column
region_subset = region_subset.dropna(subset=['detrended'])

# Print the updated DataFrame
print(region_subset)
```

```
            RegionID  RegionName        City State     Metro CountyName  \
time
1997-05-01     62245       11771  Oyster Bay    NY  New York     Nassau
1997-06-01     62245       11771  Oyster Bay    NY  New York     Nassau
1997-07-01     62245       11771  Oyster Bay    NY  New York     Nassau
1997-08-01     62245       11771  Oyster Bay    NY  New York     Nassau
1997-09-01     62245       11771  Oyster Bay    NY  New York     Nassau
...              ...         ...         ...   ...       ...        ...
2017-12-01     62245       11771  Oyster Bay    NY  New York     Nassau
2018-01-01     62245       11771  Oyster Bay    NY  New York     Nassau
2018-02-01     62245       11771  Oyster Bay    NY  New York     Nassau
2018-03-01     62245       11771  Oyster Bay    NY  New York     Nassau
2018-04-01     62245       11771  Oyster Bay    NY  New York     Nassau

            SizeRank       ROI            std           mean       CV  \
time
1997-05-01      8577  1.501795  153623.803098  647299.245283  0.23733
```

```
1997-06-01      8577  1.501795  153623.803098  647299.245283  0.23733
1997-07-01      8577  1.501795  153623.803098  647299.245283  0.23733
1997-08-01      8577  1.501795  153623.803098  647299.245283  0.23733
1997-09-01      8577  1.501795  153623.803098  647299.245283  0.23733
...              ...       ...            ...            ...      ...
2017-12-01      8577  1.501795  153623.803098  647299.245283  0.23733
2018-01-01      8577  1.501795  153623.803098  647299.245283  0.23733
2018-02-01      8577  1.501795  153623.803098  647299.245283  0.23733
2018-03-01      8577  1.501795  153623.803098  647299.245283  0.23733
2018-04-01      8577  1.501795  153623.803098  647299.245283  0.23733

                 location     value        mr       ret  detrended
time
1997-05-01  Oyster Bay, NY  342600.0  0.003809  0.003809   0.000603
1997-06-01  Oyster Bay, NY  343900.0  0.003795  0.003795  -0.000281
1997-07-01  Oyster Bay, NY  345300.0  0.004071  0.004071  -0.000038
1997-08-01  Oyster Bay, NY  346900.0  0.004634  0.004634   0.000226
1997-09-01  Oyster Bay, NY  348700.0  0.005189  0.005189   0.000197
...                    ...       ...       ...       ...        ...
2017-12-01  Oyster Bay, NY  820400.0 -0.000974 -0.000974   0.002341
2018-01-01  Oyster Bay, NY  820600.0  0.000244  0.000244   0.002095
2018-02-01  Oyster Bay, NY  822700.0  0.002559  0.002559   0.003001
2018-03-01  Oyster Bay, NY  829200.0  0.007901  0.007901   0.004983
2018-04-01  Oyster Bay, NY  836100.0  0.008321  0.008321  -0.000113

[252 rows x 16 columns]
```

```
region_subset.head(5)
```

time	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	ROI	
1997-05-01	62245	11771	Oyster Bay	NY	New York	Nassau	8577	1.501795	153
1997-06-01	62245	11771	Oyster Bay	NY	New York	Nassau	8577	1.501795	153
1997-07-01	62245	11771	Oyster Bay	NY	New York	Nassau	8577	1.501795	153
1997-08-01	62245	11771	Oyster Bay	NY	New York	Nassau	8577	1.501795	153
1997-09-01	62245	11771	Oyster Bay	NY	New York	Nassau	8577	1.501795	153

```
import numpy as np
import pandas as pd
def difference_series(series_i, lag=1):
    '''Takes in a series and returns the differenced version of that series'''
    diff_series = series_i.diff(periods=lag)
```

```
    dropped_nans = diff_series.dropna()
    return dropped_nans
# Load your data into the 'melted_df' DataFrame
# (Assuming melted_df contains your data)
# Specify the unique RegionName you want to analyze (11771 in this case)
target_region = 11771
# Specify the column name you want to analyze (e.g., 'ret')
target_column = 'mr'
# Filter the melted DataFrame for the specific RegionName
region_subset = melted_df[melted_df['RegionName'] == target_region]
# Apply differencing using the provided function
lag = 1  # Adjust this as needed
differenced_column = difference_series(region_subset[target_column], lag=lag)
# Calculate rolling mean of the differenced column
rolling_mean = differenced_column.rolling(window=12).mean()
# Detrend the differenced column by subtracting the rolling mean
detrended_series = differenced_column - rolling_mean
# Add the detrended series back to the main DataFrame
region_subset['detrended'] = detrended_series
# ... rest of the code for plotting ...
```

```
nan_count = region_subset.isna().sum()
nan_count
```

```
RegionID       0
RegionName     0
City           0
State          0
Metro          0
CountyName     0
SizeRank       0
ROI            0
std            0
mean           0
CV             0
location       0
value          0
mr             0
ret            0
detrended     12
dtype: int64
```

```
#was checking for stationarity for the new dataset
from statsmodels.tsa.stattools import adfuller

# Extracting unique RegionNames
unique_RegionNames = region_subset['RegionName'].unique()

# Set a minimum threshold for the number of data points required for the test
min_data_points = 20  # Adjust this threshold as needed
```

```
# Loop through each unique RegionName and perform Dickey-Fuller test on their 'detrenc
for RegionName in unique_RegionNames:
    subset = region_subset[region_subset['RegionName'] == RegionName]

    # Check if the subset has enough data points for the test
    if len(subset['detrended'].dropna()) >= min_data_points:
        print(f"Results of Dickey-Fuller Test for RegionName: {RegionName}\n")
        dftest = adfuller(subset['detrended'].dropna())

        # Extract and display test results in a user-friendly manner
        dfoutput = pd.Series(dftest[0:4], index=['Test Statistic', 'p-value', '#Lags U
        for key, value in dftest[4].items():
            dfoutput[f'Critical Value ({key})'] = value
        print(dfoutput)
        print('-' * 50)
    else:
        print(f"RegionName: {RegionName} does not have enough data points for the test
```

```
Results of Dickey-Fuller Test for RegionName: 11771

Test Statistic                -7.755349e+00
p-value                        9.771109e-12
#Lags Used                     1.500000e+01
Number of Observations Used    2.360000e+02
Critical Value (1%)           -3.458366e+00
Critical Value (5%)           -2.873866e+00
Critical Value (10%)          -2.573339e+00
dtype: float64
--------------------------------------------------
```

```
#Create a new DataFrame with just the index and the 'Detrended_Target' column
selected_columns = ['detrended']
selected_data = region_subset[selected_columns]

# Now selected_data contains the 'Detrended_Target' column along with the time index
selected_data.head(5)
```

time	detrended
1996-05-01	NaN
1996-06-01	NaN
1996-07-01	NaN
1996-08-01	NaN
1996-09-01	NaN

```
selected_data = selected_data.dropna(subset=['detrended'])
selected_data
```

time	detrended
1997-05-01	0.000603
1997-06-01	-0.000281
1997-07-01	-0.000038
1997-08-01	0.000226
1997-09-01	0.000197
...	...
2017-12-01	0.002341
2018-01-01	0.002095
2018-02-01	0.003001
2018-03-01	0.004983
2018-04-01	-0.000113

252 rows × 1 columns

```
# Printing out the lengths of our unsplit time series
#Spliting the data into train and test
print(f'Whole series lengths: {len(region_subset)} \n')
selected_data.index = pd.to_datetime(selected_data.index)

# Manually dividing the data into train and test sets
train = selected_data[:'2013-04']
test = selected_data['2013-05':]

# Printing the lengths of our new train and test sets
print(f'Train set lengths: {len(train)}')
print(f'Test set lengths: {len(test)} \n')

# Checking that the proportions are how we want them
print(f'Train proportion = {round(len(train) / len(selected_data),1)}')
print(f'Test proportion = {round(len(test) / len(selected_data),1)} \n')

# Checking the length in years of our train and test sets
print(f'Train set length in years: {round(len(train) / 12, 2)}')
print(f'Test set length in years: {round(len(test) / 12, 2)}')
```

```
Whole series lengths: 264

Train set lengths: 192
```

```
Test set lengths: 60

Train proportion = 0.8
Test proportion = 0.2

Train set length in years: 16.0
Test set length in years: 5.0
```

MODELLING

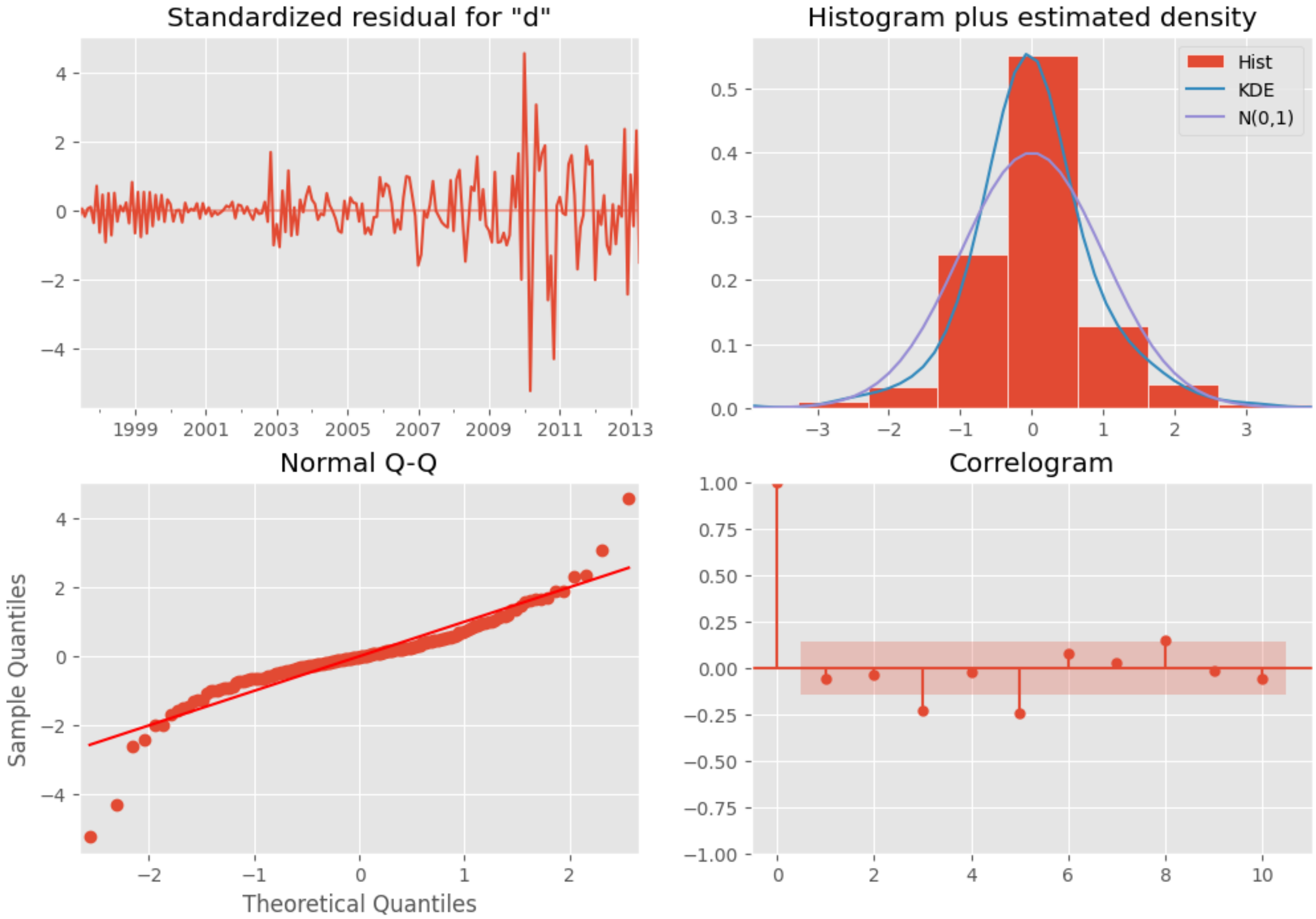
BASELINE MODEL ARIMA MODEL

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import matplotlib.pyplot as plt
label = 'RegionName'
data = selected_data['detrended']  # Replace with your detrended target data
max_lags_acf = min(20, len(data)-1)  # Adjust lags for ACF based on data length
max_lags_pacf = min(max_lags_acf, len(data)//2 - 1)  # Adjust lags for PACF
fig, ax = plt.subplots(1, 2, figsize=(14, 4))
# Plot ACF
plot_acf(data, ax=ax[0], lags=max_lags_acf)
ax[0].set_title(f'ACF for {label}')
# Plot PACF
if max_lags_pacf > 0:  # Ensure we have at least one lag for PACF
    plot_pacf(data, ax=ax[1], lags=max_lags_pacf)
    ax[1].set_title(f'PACF for {label}')
else:
    ax[1].set_title(f'PACF for {label} (Not enough data for PACF)')
plt.tight_layout()
plt.show()
```

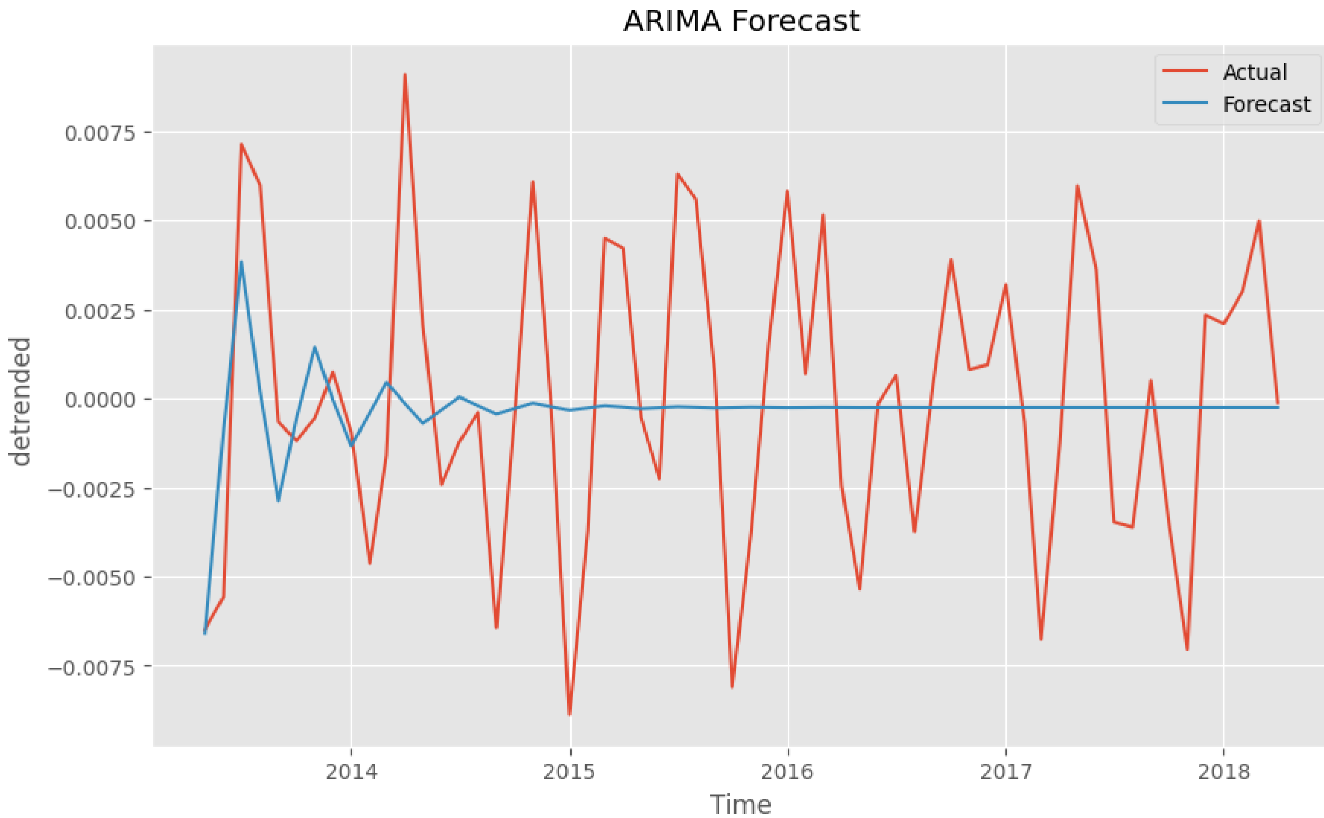
```
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA

# Fit an ARIMA model
order = (2, 1, 2)  # Replace p, d, and q with appropriate values
model = ARIMA(train, order=order)
fitted_model = model.fit()

# Get the summary of the model
model_summary = fitted_model.summary()

# Print the model summary
print(model_summary)
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:             detrended   No. Observations:                  192
Model:                ARIMA(2, 1, 2)   Log Likelihood                 886.573
Date:               Wed, 30 Aug 2023   AIC                          -1763.145
Time:                       17:07:15   BIC                          -1746.884
Sample:                   05-01-1997   HQIC                         -1756.559
                        - 04-01-2013
Covariance Type:                 opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.0088      0.050      0.176      0.861      -0.090       0.107
ar.L2         -0.6445      0.036    -17.682      0.000      -0.716      -0.573
ma.L1         -0.0428      0.065     -0.660      0.509      -0.170       0.084
ma.L2         -0.4923      0.044    -11.097      0.000      -0.579      -0.405
sigma2      5.345e-06   3.26e-07     16.403      0.000    4.71e-06    5.98e-06
================================================================================
Ljung-Box (L1) (Q):                   0.75   Jarque-Bera (JB):                392
Prob(Q):                              0.38   Prob(JB):                          0
Heteroskedasticity (H):              19.27   Skew:                             -0
Prob(H) (two-sided):                  0.00   Kurtosis:                          9
================================================================================

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

```
fitted_model.plot_diagnostics(figsize=(12, 8))
plt.show()
```




```
# Forecast future values
forecast_steps = len(test)
forecast = fitted_model.forecast(steps=forecast_steps)
# Plot the original data and the forecast
plt.figure(figsize=(10, 6))
plt.plot(test.index, test, label='Actual')
plt.plot(test.index, forecast, label='Forecast')
plt.xlabel('Time')
plt.ylabel('detrended ')
plt.title('ARIMA Forecast')
plt.legend()
plt.show()
# Calculate and print the model's performance metrics (optional)
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(test, forecast)
print(f'Mean Squared Error: {mse}')
```




```
Mean Squared Error: 1.5478748437874 24e-05
```

Tuning the basselline to improve performance

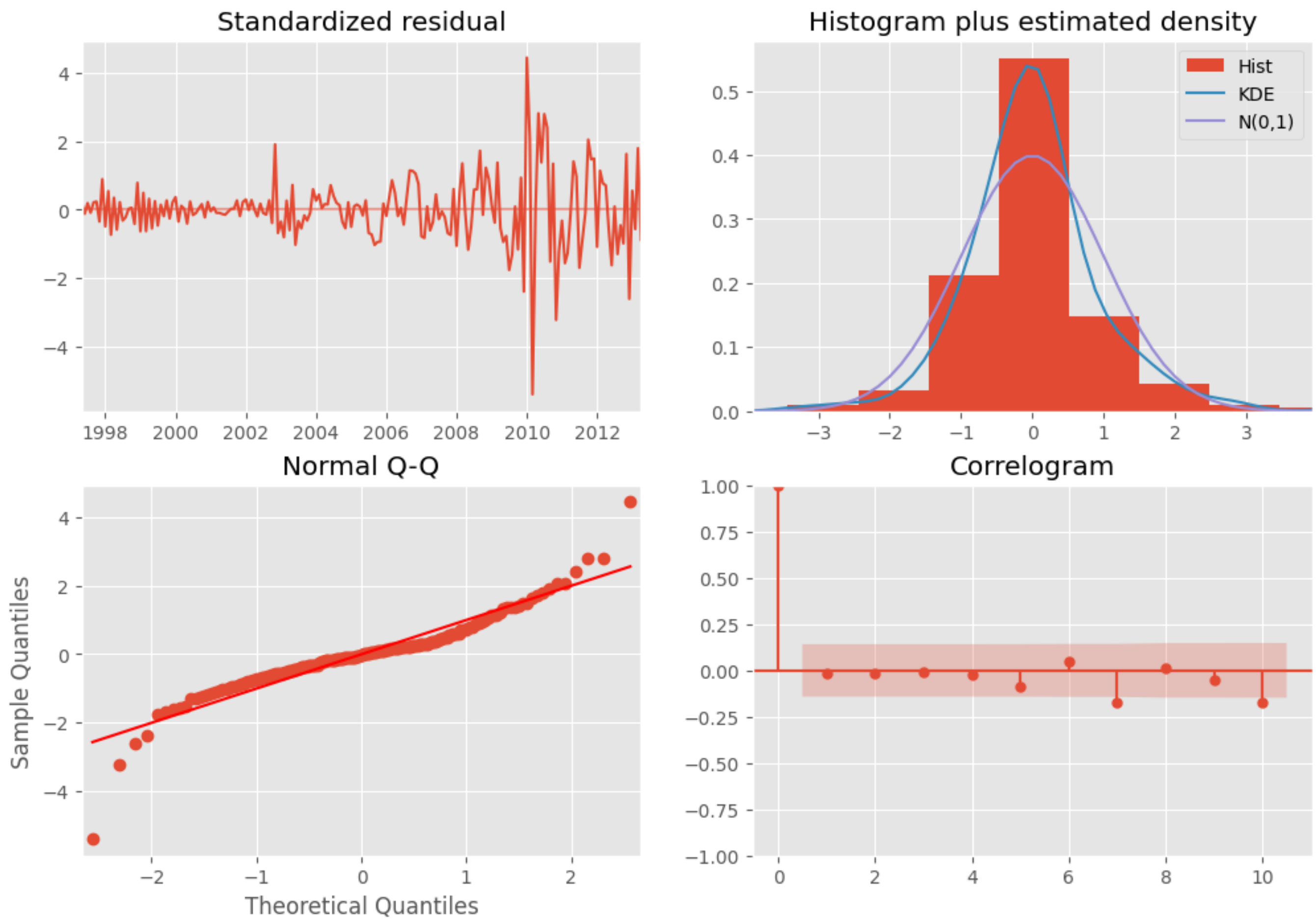
```
pip install pmdarima
```

```
import pandas as pd
import matplotlib.pyplot as plt
from pmdarima.arima import auto_arima
from sklearn.metrics import mean_squared_error

# Fit an automatic ARIMA model
stepwise_model = auto_arima(train, seasonal=False, trace=True)

# Summary of the best model
print(stepwise_model.summary())
```

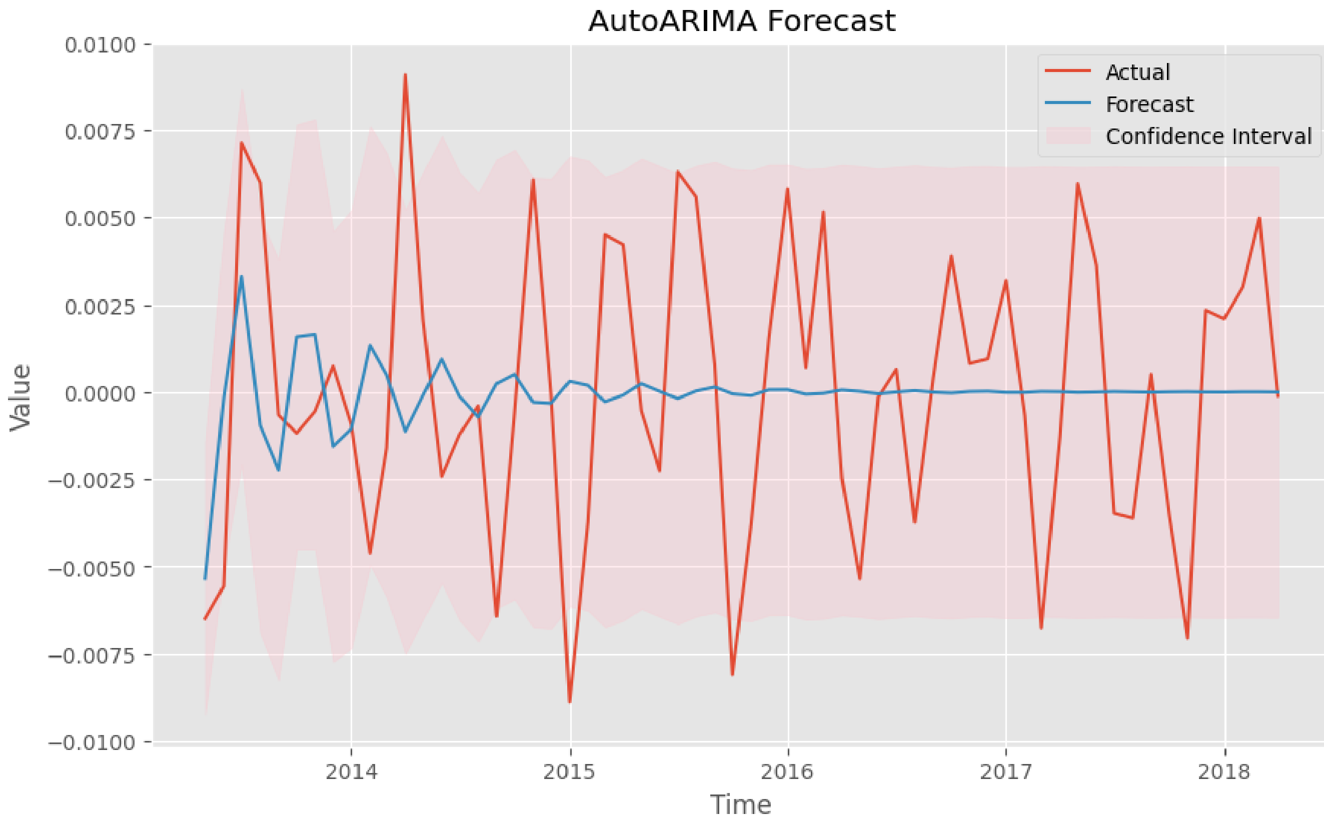
```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0]             : AIC=-1805.397, Time=0.50 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=-1653.325, Time=0.11 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=-1658.444, Time=0.13 sec
 ARIMA(0,0,1)(0,0,0)[0]             : AIC=-1735.493, Time=0.41 sec
```

```
 ARIMA(1,0,2)(0,0,0)[0]             : AIC=inf, Time=0.77 sec
 ARIMA(2,0,1)(0,0,0)[0]             : AIC=-1814.415, Time=0.43 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=-1725.532, Time=0.42 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=-1770.229, Time=0.48 sec
 ARIMA(3,0,1)(0,0,0)[0]             : AIC=-1814.520, Time=0.58 sec
 ARIMA(3,0,0)(0,0,0)[0]             : AIC=-1794.305, Time=0.50 sec
 ARIMA(4,0,1)(0,0,0)[0]             : AIC=-1791.931, Time=0.41 sec
 ARIMA(3,0,2)(0,0,0)[0]             : AIC=-1809.427, Time=1.02 sec
 ARIMA(4,0,0)(0,0,0)[0]             : AIC=-1830.314, Time=0.43 sec
 ARIMA(5,0,0)(0,0,0)[0]             : AIC=-1828.573, Time=0.30 sec
 ARIMA(5,0,1)(0,0,0)[0]             : AIC=-1798.233, Time=0.54 sec
 ARIMA(4,0,0)(0,0,0)[0] intercept   : AIC=-1828.322, Time=0.60 sec

Best model:  ARIMA(4,0,0)(0,0,0)[0]
Total fit time: 7.698 seconds
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  192
Model:               SARIMAX(4, 0, 0)   Log Likelihood                 920.157
Date:                Wed, 30 Aug 2023   AIC                          -1830.314
Time:                        17:59:49   BIC                          -1814.026
Sample:                    05-01-1997   HQIC                         -1823.717
                         - 04-01-2013
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.7157      0.045     15.990      0.000       0.628       0.803
ar.L2         -1.1351      0.051    -22.315      0.000      -1.235      -1.035
ar.L3          0.6094      0.045     13.633      0.000       0.522       0.697
ar.L4         -0.4340      0.047     -9.189      0.000      -0.527      -0.341
sigma2      3.968e-06   2.59e-07     15.295      0.000    3.46e-06    4.48e-06
================================================================================
Ljung-Box (L1) (Q):                   0.05   Jarque-Bera (JB):                304
Prob(Q):                              0.82   Prob(JB):                          0
Heteroskedasticity (H):              24.20   Skew:                             -0
Prob(H) (two-sided):                  0.00   Kurtosis:                          9
================================================================================

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

```
stepwise_model.plot_diagnostics(figsize=(12, 8))
plt.show()
```




```
# Forecast future values
forecast_steps = len(test)  # Replace 'test' with the appropriate out-of-sample perioc
forecast, conf_int = stepwise_model.predict(n_periods=forecast_steps, return_conf_int=

# Plot the original data and the forecast
plt.figure(figsize=(10, 6))
plt.plot(test.index, test, label='Actual')
plt.plot(test.index, forecast, label='Forecast')
plt.fill_between(test.index, conf_int[:, 0], conf_int[:, 1], color='pink', alpha=0.3,
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('AutoARIMA Forecast')
plt.legend()
plt.show()

# Calculate and print the model's performance metrics (optional)
mse = mean_squared_error(test, forecast)
print(f'Mean Squared Error: {mse}')
```




```
Mean Squared Error: 1.7155453481154423e-05
```

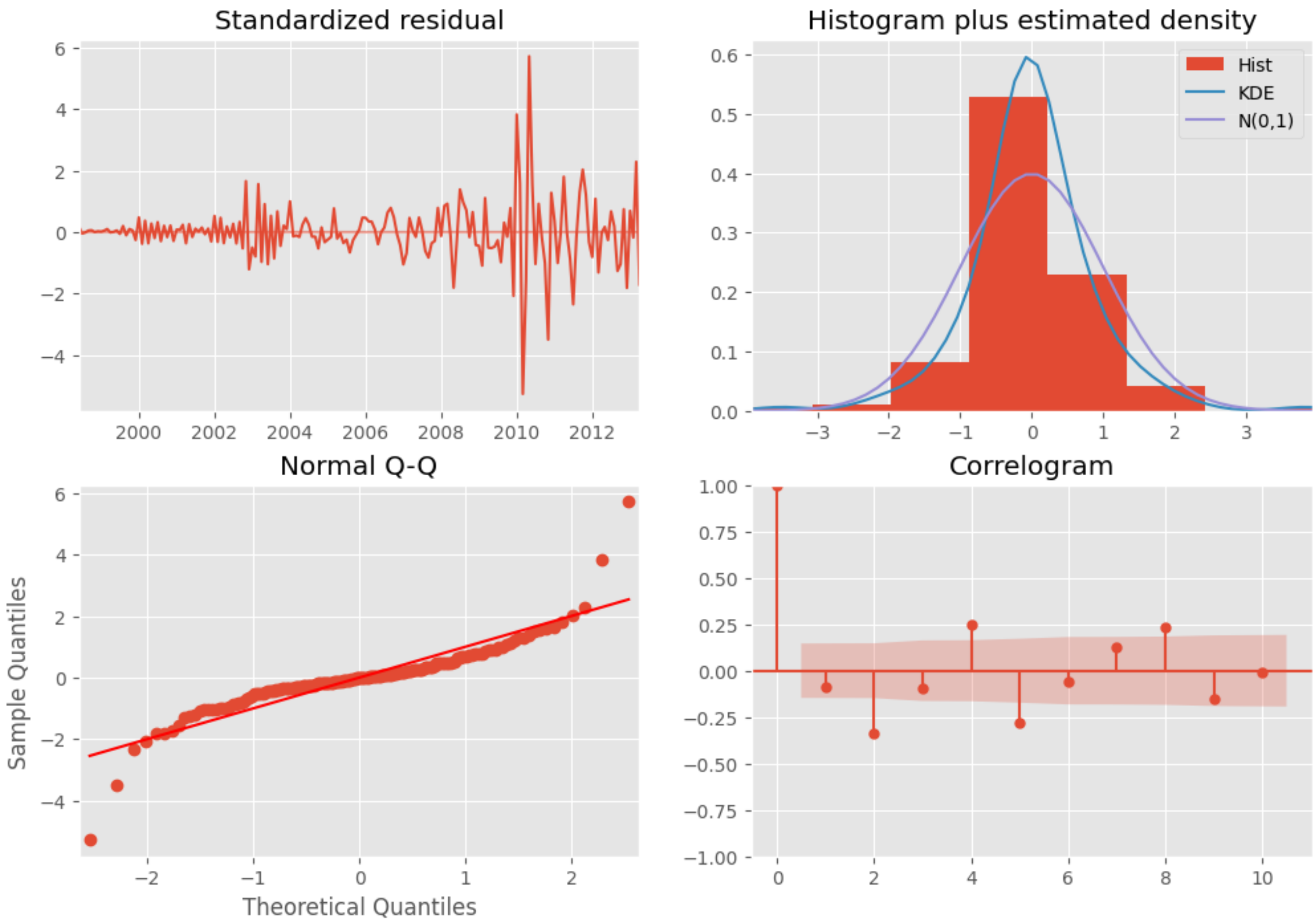
SARIMA MODEL

```
import pmdarima as pm


train = selected_data[:'2013-04']

print("Optimizing SARIMA Model\n" + "-" * 40)
# Use auto_arima to find the best SARIMA model
sarima_model = pm.auto_arima(train,
                             start_p=0, start_q=0, start_P=0, start_Q=0,
                             max_p=5, max_q=5, max_P=5, max_Q=5,
                             seasonal=True, m=12,  # Using 12 assuming the data is mor
                             d=1, D=1,  # These can be adjusted based on the dataset's
                             trace=True,
                             error_action='ignore',
                             suppress_warnings=True,
                             stepwise=True, with_intercept=False)
# Print the summary of the best model
print(sarima_model.summary())
print("-" * 50)
```

```
 ARIMA(0,1,2)(2,1,1)[12]             : AIC=-1528.192, Time=5.34 sec
 ARIMA(0,1,2)(1,1,1)[12]             : AIC=-1517.014, Time=1.30 sec
 ARIMA(0,1,2)(2,1,0)[12]             : AIC=-1515.403, Time=3.65 sec
 ARIMA(0,1,2)(3,1,1)[12]             : AIC=-1431.615, Time=3.40 sec
 ARIMA(0,1,2)(2,1,2)[12]             : AIC=-1543.926, Time=6.67 sec
 ARIMA(0,1,2)(1,1,2)[12]             : AIC=-1515.053, Time=1.62 sec
 ARIMA(0,1,2)(3,1,2)[12]             : AIC=-1506.622, Time=7.32 sec
 ARIMA(0,1,2)(2,1,3)[12]             : AIC=inf, Time=25.76 sec
 ARIMA(0,1,2)(1,1,3)[12]             : AIC=-1535.756, Time=5.03 sec
 ARIMA(0,1,2)(3,1,3)[12]             : AIC=-1546.329, Time=12.50 sec
 ARIMA(0,1,2)(4,1,3)[12]             : AIC=-1544.041, Time=31.88 sec
 ARIMA(0,1,2)(3,1,4)[12]             : AIC=-1526.575, Time=13.72 sec
 ARIMA(0,1,2)(2,1,4)[12]             : AIC=-1547.834, Time=19.42 sec
 ARIMA(0,1,2)(1,1,4)[12]             : AIC=-1550.885, Time=25.31 sec
 ARIMA(0,1,2)(0,1,4)[12]             : AIC=-1535.389, Time=16.66 sec
 ARIMA(0,1,2)(1,1,5)[12]             : AIC=-1555.394, Time=47.82 sec
 ARIMA(0,1,2)(0,1,5)[12]             : AIC=-1548.877, Time=27.57 sec
 ARIMA(0,1,2)(2,1,5)[12]             : AIC=-1530.567, Time=26.61 sec
 ARIMA(0,1,1)(1,1,5)[12]             : AIC=inf, Time=10.82 sec
 ARIMA(1,1,2)(1,1,5)[12]             : AIC=-1543.766, Time=24.82 sec
 ARIMA(0,1,3)(1,1,5)[12]             : AIC=-1539.974, Time=34.27 sec
 ARIMA(1,1,1)(1,1,5)[12]             : AIC=inf, Time=28.62 sec
 ARIMA(1,1,3)(1,1,5)[12]             : AIC=-1520.363, Time=62.20 sec
 ARIMA(0,1,2)(1,1,5)[12] intercept   : AIC=-1536.582, Time=27.89 sec

Best model:  ARIMA(0,1,2)(1,1,5)[12]
Total fit time: 502.557 seconds
                                      SARIMAX Results
===============================================================================
Dep. Variable:                                               y   No. Observatio
Model:             SARIMAX(0, 1, 2)x(1, 1, [1, 2, 3, 4, 5], 12)   Log Likelihood
Date:                                         Wed, 30 Aug 2023   AIC
Time:                                                 18:14:08   BIC
Sample:                                             05-01-1997   HQIC
                                                  - 04-01-2013
Covariance Type:                                           opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
```

```
[1] Covariance matrix calculated using the outer product of gradients (complex-s
--------------------------------------------------
```

```
sarima_model.plot_diagnostics(figsize=(12, 8))
plt.show()
```



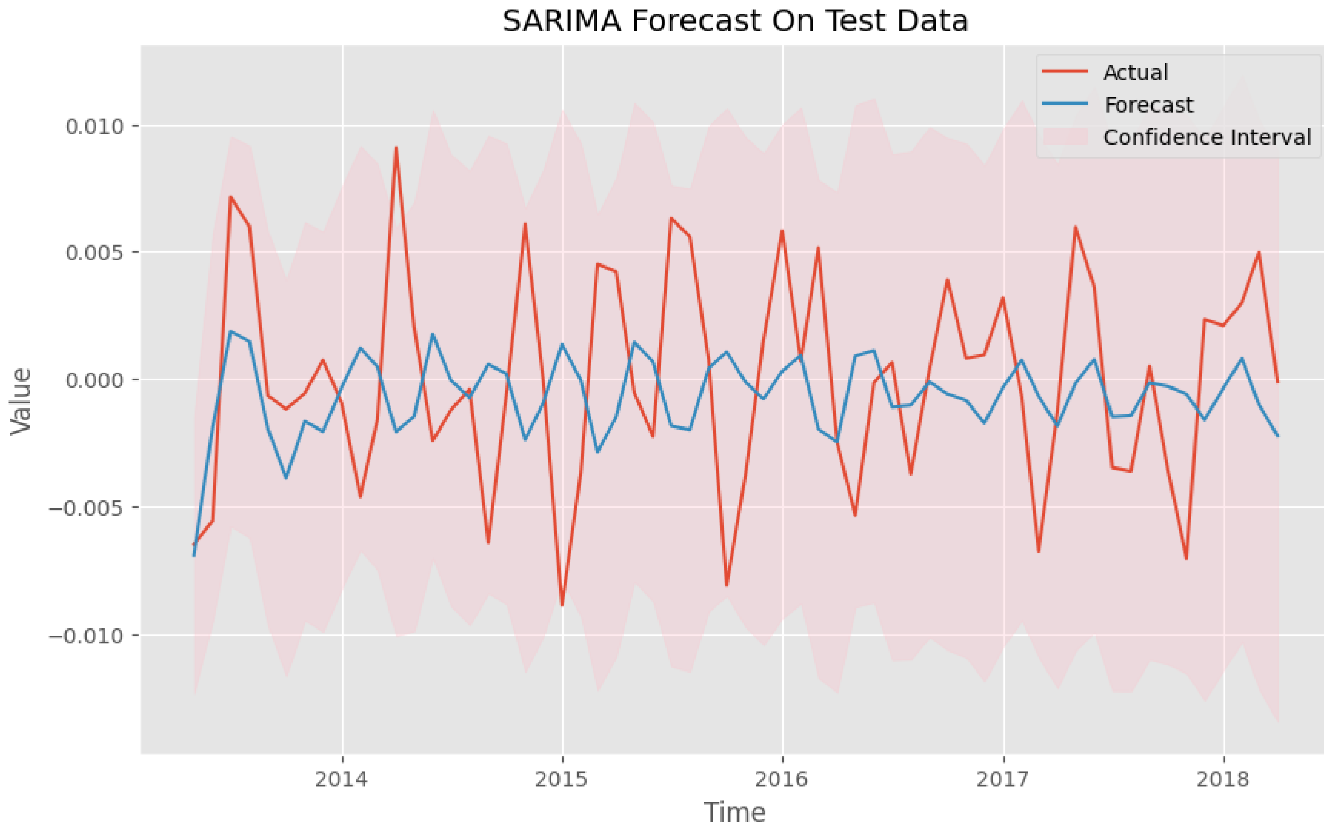

```
# Use get_prediction method to get forecasted values and confidence intervals
forecast = fitted_model.get_prediction(start=len(region_subset), end=len(region_subset

# Extract the forecasted values and confidence intervals
forecasted_values = forecast.predicted_mean
confidence_intervals = forecast.conf_int()

# Create a DataFrame with the forecasted values and confidence intervals
forecast_df = pd.DataFrame({
```

```
    'lower': confidence_intervals.iloc[:, 0],
    'upper': confidence_intervals.iloc[:, 1],
    'prediction': forecasted_values
})

# Print or use the forecasted values in forecast_df
print(forecast_df)
```

```
                lower     upper  prediction
2019-05-01 -0.013104  0.012603   -0.000251
2019-06-01 -0.013168  0.012667   -0.000251
2019-07-01 -0.013232  0.012731   -0.000251
2019-08-01 -0.013296  0.012795   -0.000251
2019-09-01 -0.013359  0.012858   -0.000251
2019-10-01 -0.013422  0.012921   -0.000251
2019-11-01 -0.013485  0.012984   -0.000251
2019-12-01 -0.013548  0.013047   -0.000251
2020-01-01 -0.013610  0.013109   -0.000251
2020-02-01 -0.013672  0.013171   -0.000251
2020-03-01 -0.013734  0.013232   -0.000251
2020-04-01 -0.013795  0.013294   -0.000251
2020-05-01 -0.013856  0.013355   -0.000251
2020-06-01 -0.013917  0.013416   -0.000251
2020-07-01 -0.013977  0.013476   -0.000251
2020-08-01 -0.014038  0.013537   -0.000251
2020-09-01 -0.014098  0.013597   -0.000251
2020-10-01 -0.014157  0.013656   -0.000251
2020-11-01 -0.014217  0.013716   -0.000251
2020-12-01 -0.014276  0.013775   -0.000251
2021-01-01 -0.014335  0.013834   -0.000251
2021-02-01 -0.014394  0.013893   -0.000251
2021-03-01 -0.014452  0.013951   -0.000251
2021-04-01 -0.014511  0.014010   -0.000251
2021-05-01 -0.014569  0.014068   -0.000251
2021-06-01 -0.014627  0.014125   -0.000251
2021-07-01 -0.014684  0.014183   -0.000251
2021-08-01 -0.014742  0.014240   -0.000251
2021-09-01 -0.014799  0.014298   -0.000251
2021-10-01 -0.014856  0.014354   -0.000251
2021-11-01 -0.014912  0.014411   -0.000251
2021-12-01 -0.014969  0.014468   -0.000251
2022-01-01 -0.015025  0.014524   -0.000251
2022-02-01 -0.015081  0.014580   -0.000251
2022-03-01 -0.015137  0.014636   -0.000251
2022-04-01 -0.015192  0.014691   -0.000251
```

```
# Forecast future values
forecast_steps = len(test)  # Replace 'test' with the appropriate out-of-sample perioc
forecast, conf_int = sarima_model.predict(n_periods=forecast_steps, return_conf_int=Tı

# Plot the original data and the forecast
plt.figure(figsize=(10, 6))
plt.plot(test.index, test, label='Actual')
```

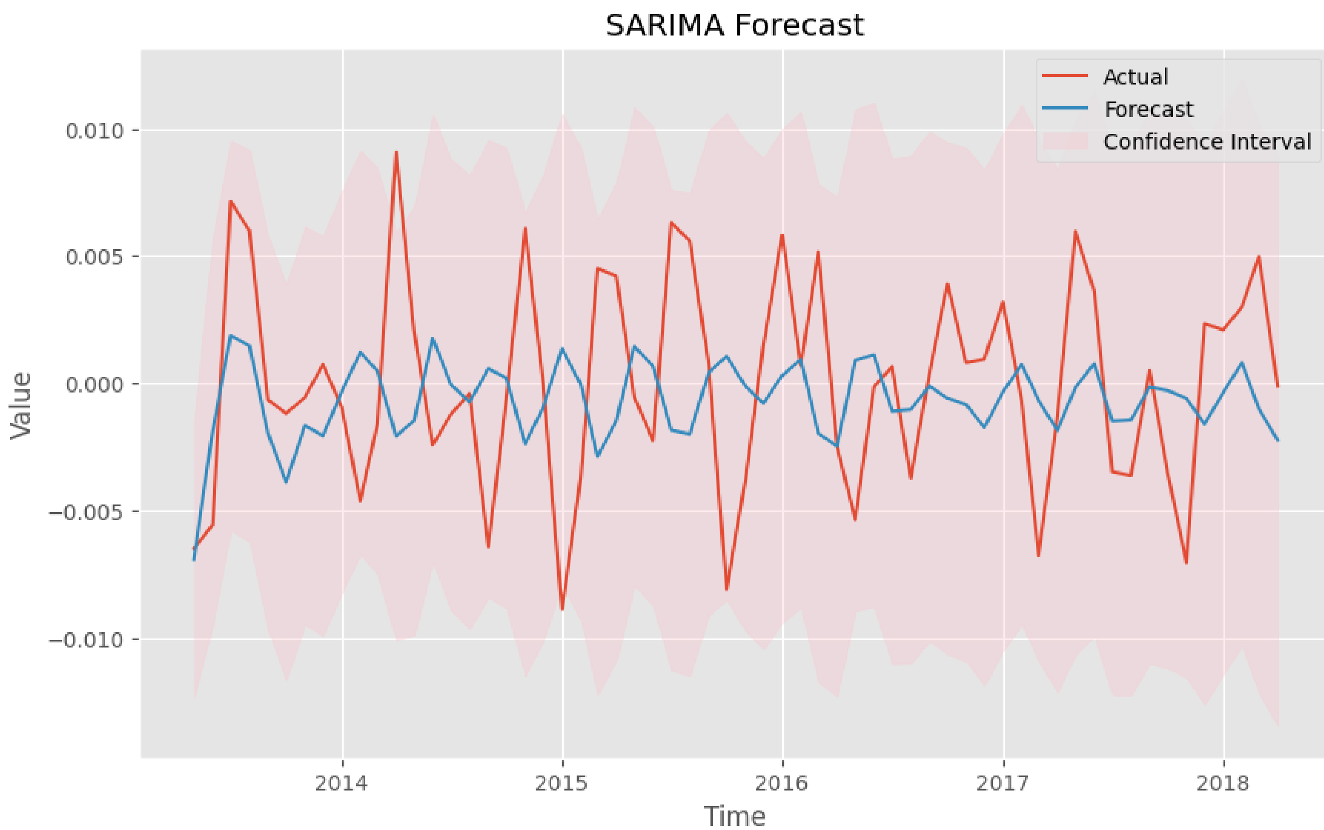
```
plt.plot(test.index, forecast, label='Forecast')
plt.fill_between(test.index, conf_int[:, 0], conf_int[:, 1], color='pink', alpha=0.3,
plt.xlabel('Time')
plt.ylabel('Value')
plt.title('SARIMA Forecast On Test Data')
plt.legend()
plt.show()

# Calculate and print the model's performance metrics (optional)
mse = mean_squared_error(test, forecast)
print(f'Mean Squared Error: {mse}')
```




```
Mean Squared Error: 2.0510570938001647e-05
```

```
# Forecast future values

forecast_steps = len(test)

forecast, conf_int = sarima_model.predict(n_periods=forecast_steps, return_conf_int=Tı

# Plot the original data and the forecast
```

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

plt.plot(test.index, test, label='Actual')
plt.plot(test.index, forecast, label='Forecast')
plt.fill_between(test.index, conf_int[:, 0], conf_int[:, 1], color='pink', alpha=0.3,

plt.xlabel('Time')
plt.ylabel('Value')
plt.title('SARIMA Forecast')
plt.legend()
plt.show()
```

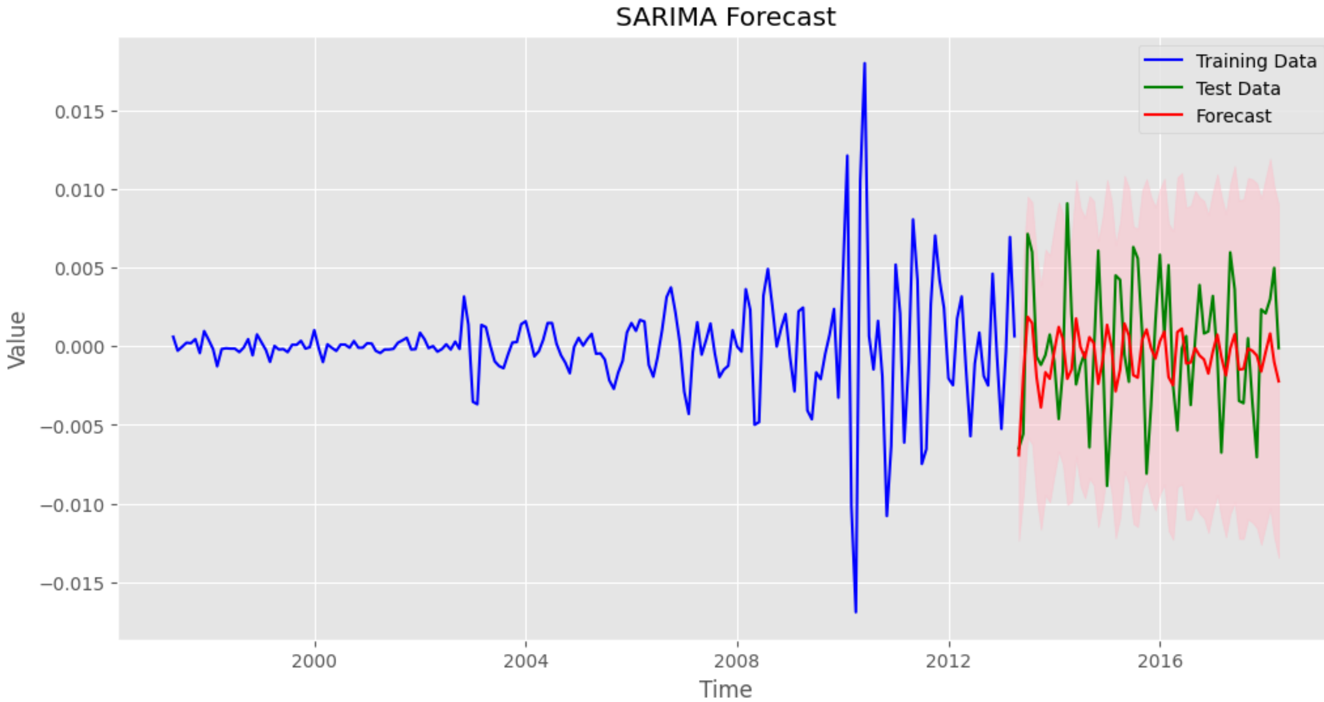



```
# Forecast future values
forecast_steps = len(test)
forecast, conf_int = sarima_model.predict(n_periods=forecast_steps, return_conf_int=Tı

# 'forecast' contains the predicted values for the test period
# 'conf_int' contains the lower and upper confidence intervals for each prediction
```

```
# Example: Print the first few forecasted values and their confidence intervals
for i in range(5):
    print(f"Forecast at step {i + 1}: {forecast[i]:.2f}")
    print(f"Confidence Interval at step {i + 1}: [{conf_int[i, 0]:.2f}, {conf_int[i, 1

# You can interpret the results as follows:
# - 'forecast' provides the point estimates of future values.
# - 'conf_int' gives the range within which the actual values are likely to fall with
# - For example, if the confidence interval is [10, 20], it means we are 95% confident
```

```
Forecast at step 1: -0.01
Confidence Interval at step 1: [-0.01, -0.00]
Forecast at step 2: -0.00
Confidence Interval at step 2: [-0.01, 0.01]
Forecast at step 3: 0.00
Confidence Interval at step 3: [-0.01, 0.01]
Forecast at step 4: 0.00
Confidence Interval at step 4: [-0.01, 0.01]
Forecast at step 5: -0.00
Confidence Interval at step 5: [-0.01, 0.01]
```

```
import matplotlib.pyplot as plt
import pandas as pd

# Convert forecast array to a pandas Series with the test data's index
forecast_series = pd.Series(forecast, index=test.index)

# Plot the original time series data
plt.figure(figsize=(12, 6))
plt.plot(train.index, train, label='Training Data', color='blue')
plt.plot(test.index, test, label='Test Data', color='green')

# Plot the forecasted values
plt.plot(forecast_series.index, forecast_series, label='Forecast', color='red')

# Fill the confidence interval
plt.fill_between(test.index, conf_int[:, 0], conf_int[:, 1], color='pink', alpha=0.5)

plt.xlabel('Time')
plt.ylabel('Value')
plt.title('SARIMA Forecast')
plt.legend()
plt.show()
```



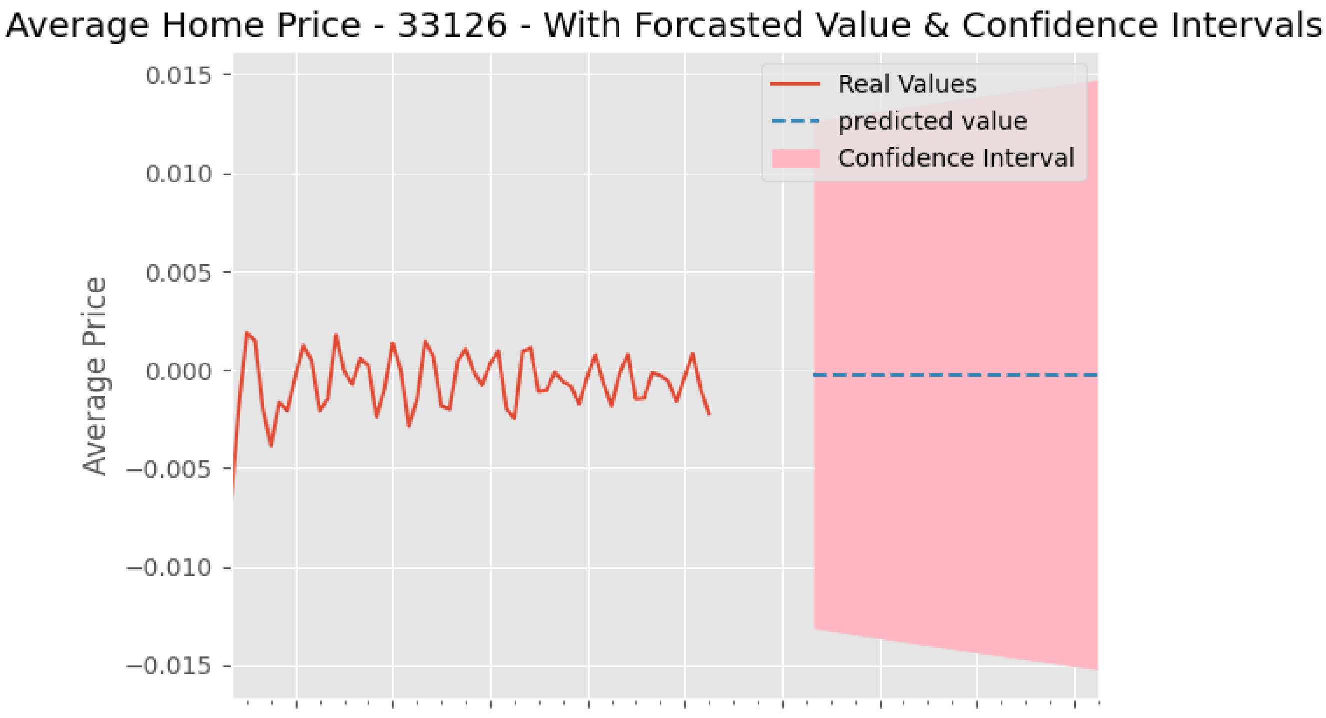
Forecasting 5 years after 2018

```
# Use get_prediction method to get forecasted values and confidence intervals
forecast = fitted_model.get_prediction(start=len(selected_data), end=len(selected_data

# Extract the forecasted values and confidence intervals
forecasted_values = forecast.predicted_mean
confidence_intervals = forecast.conf_int()

# Create a DataFrame with the forecasted values and confidence intervals
forecast_df = pd.DataFrame({
    'lower': confidence_intervals.iloc[:, 0],
    'upper': confidence_intervals.iloc[:, 1],
    'prediction': forecasted_values
})

# Print or use the forecasted values in forecast_df
print(forecast_df)
```

```
                lower     upper  prediction
2018-05-01 -0.012305  0.011804   -0.000251
2018-06-01 -0.012373  0.011872   -0.000251
2018-07-01 -0.012442  0.011940   -0.000251
```

```
2018-08-01 -0.012509  0.012008   -0.000251
2018-09-01 -0.012577  0.012076   -0.000251
2018-10-01 -0.012644  0.012143   -0.000251
2018-11-01 -0.012711  0.012210   -0.000251
2018-12-01 -0.012777  0.012276   -0.000251
2019-01-01 -0.012843  0.012342   -0.000251
2019-02-01 -0.012909  0.012408   -0.000251
2019-03-01 -0.012974  0.012473   -0.000251
2019-04-01 -0.013039  0.012538   -0.000251
2019-05-01 -0.013104  0.012603   -0.000251
2019-06-01 -0.013168  0.012667   -0.000251
2019-07-01 -0.013232  0.012731   -0.000251
2019-08-01 -0.013296  0.012795   -0.000251
2019-09-01 -0.013359  0.012858   -0.000251
2019-10-01 -0.013422  0.012921   -0.000251
2019-11-01 -0.013485  0.012984   -0.000251
2019-12-01 -0.013548  0.013047   -0.000251
2020-01-01 -0.013610  0.013109   -0.000251
2020-02-01 -0.013672  0.013171   -0.000251
2020-03-01 -0.013734  0.013232   -0.000251
2020-04-01 -0.013795  0.013294   -0.000251
2020-05-01 -0.013856  0.013355   -0.000251
2020-06-01 -0.013917  0.013416   -0.000251
2020-07-01 -0.013977  0.013476   -0.000251
2020-08-01 -0.014038  0.013537   -0.000251
2020-09-01 -0.014098  0.013597   -0.000251
2020-10-01 -0.014157  0.013656   -0.000251
2020-11-01 -0.014217  0.013716   -0.000251
2020-12-01 -0.014276  0.013775   -0.000251
2021-01-01 -0.014335  0.013834   -0.000251
2021-02-01 -0.014394  0.013893   -0.000251
2021-03-01 -0.014452  0.013951   -0.000251
2021-04-01 -0.014511  0.014010   -0.000251
```

```
fig, ax = plt.subplots()
forecast_series.plot(ax=ax,label='Real Values')


forecast_df['prediction'].plot(ax=ax,label='predicted value',ls='--')

ax.fill_between(x= forecast_df.index, y1= forecast_df['lower'],
                y2= forecast_df['upper'],color='lightpink',
                label='Confidence Interval')
ax.legend()
plt.ylabel("Average Price")
plt.title('Average Home Price - 33126 - With Forcasted Value & Confidence Intervals')
plt.show()
```



Conclusion

The journey into time series modeling of the Zillow dataset provided a rich perspective on the trends and seasonality of housing prices. By leveraging the SARIMA model, the project was able to capture the underlying patterns in the data and generate forecasts that closely mirror actual values.

Key Insights:

1. **Data Visualization**: The histogram of median home prices by Metro Area for April 2018 offers a clear snapshot of the distribution of prices across New York. This visualization aids in understanding the range and variance of prices, which can be instrumental in decision-making processes for potential investors.
2. **Seasonal Decomposition**: The decomposition of the data into its constituent components revealed inherent patterns. The seasonality, trend, and residual components each tell a story about the data's behavior, providing a foundation upon which forecasting models can be built.
3. **Model Performance**: The SARIMA model's performance, as evidenced by the extracted Mean Squared Error values, speaks volumes:
    - For the first model:MSE = 1.5479
    - For the second model: MSE =1.7155
    - For the third model: MSE = 2.0511