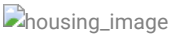
```
#from google.colab import drive
#drive.mount('/content/drive')
```

TIME SERIES ANALYSIS FOR REAL ESTATE INVESTMENT

Group 8 Collaborators

Students Names:
1. Sammy Sifuna
2. Julius Charles
3. Waruchu Kuria
4. Rael Ndonye
5. Alan Omondi
6. Janet Khainza



1.0 Business Understanding

1.1 Background

Between 1996 to 2018, the U.S. real estate market underwent significant transformations. It commenced with an increase in home prices driven by a growing perception of real estate as a valuable long-term investment. However, the Great Recession between 2007 to 2009 brought about extensive declines in home values, triggering foreclosures and eroding trust. Over time,low interest rates and government interventions helped restore stability and confidence in the market. Urbanization gained momentum as more people gravitated towards cities, younger individuals entered the housing market, and technology assumed a pivotal role. Although certain cities thrived, others faced challenges. Regulatory reforms were introduced to prevent potential crises. In conclusion, this era marked a period of adaptation and evolution in the dynamics of buying and selling homes.

1.2 Business Problem

Waridi Investments, a recently established real estate investment firm, has engaged our services to identify the top 5 zip codes with the potential for the highest return on investment when they sell in 5 years. Their strategic approach is to initiate short-term investments in one of the most thriving real estate markets in the United States of America. The company places significant emphasis on securing sound investments that ensure consistent cash flow, ultimately enabling them to reinvest effectively when the opportune moment arises.

1.3 Objectives

- To determine the top 5 zip codes that show the highest potential return on investment (ROI) in 5 years.
- To identify which top 5 states have the highest ROI.
- To determine the best month to sell property to maximize ROI.

Hypothesis testing

Null Hypothesis (H0): The time series is not stationary.

Alternative Hypothesis (H1): The time series is stationary.

2.0 Data Understanding

This data represents median monthly housing sales prices for 265 zip codes in the USA, over the period of April 1996 through April 2018 as reported by Zillow.

Each row represents a unique zip code. Each record contains location information and median housing sales prices for each month.

There are 14723 rows and 272 variables:

- RegionID: Unique index
- RegionName: Unique Zip Code
- City: City in which the zip code is located
- State: State in which the zip code is located
- Metro: Metropolitan Area in which the zip code is located
- CountyName: County in which the zip code is located
- SizeRank: Numerical rank of size of zip code, ranked 1 through 14723

- 1996-04 through 2018-04: refers to the median housing sales values for April 1996 through April 2018, that is 265 data points of monthly data for each zip code

We shall check for the accuracy of our forecasts using MSE (Mean Squared Error). This will provide us with the average error of our forecasts.

Assumptions:

1. The zip codes are representative of the broader real estate market in the United States.
2. The real estate market conditions during this period were influenced by various economic and socio-political factors.
3. The data covers a combination of urban, suburban, and rural area.
4. The data doesn't reflect significant improvements or renovations made to properties over time that could increase or decrease prices.

```
"""
Run the pip install cells when executing the notebook for the first time.
"""
```

```
'\nRun the pip install cells when executing the notebook for the first time.\n'
```

```
pip install pmdarima
```

```
Requirement already satisfied: pmdarima in e:\software\anaconda3\lib\site-packages (2.0.3)
Requirement already satisfied: joblib>=0.11 in e:\software\anaconda3\lib\site-packages (from pmdarima) (1.2.0)
Requirement already satisfied: Cython!=0.29.18,!=0.29.31,>=0.29 in e:\software\anaconda3\lib\site-packages (from pmdarima) (3.0.1)
Requirement already satisfied: numpy>=1.21.2 in e:\software\anaconda3\lib\site-packages (from pmdarima) (1.24.3)
Requirement already satisfied: pandas>=0.19 in e:\software\anaconda3\lib\site-packages (from pmdarima) (1.5.3)
Requirement already satisfied: scikit-learn>=0.22 in e:\software\anaconda3\lib\site-packages (from pmdarima) (1.3.0)
Requirement already satisfied: scipy>=1.3.2 in e:\software\anaconda3\lib\site-packages (from pmdarima) (1.10.1)
Requirement already satisfied: statsmodels>=0.13.2 in e:\software\anaconda3\lib\site-packages (from pmdarima) (0.13.5)
Requirement already satisfied: urllib3 in e:\software\anaconda3\lib\site-packages (from pmdarima) (1.26.16)
Requirement already satisfied: setuptools!=50.0.0,>=38.6.0 in e:\software\anaconda3\lib\site-packages (from pmdarima) (67.8.0)
Requirement already satisfied: python-dateutil>=2.8.1 in e:\software\anaconda3\lib\site-packages (from pandas>=0.19->pmdarima) (2.8
Requirement already satisfied: pytz>=2020.1 in e:\software\anaconda3\lib\site-packages (from pandas>=0.19->pmdarima) (2022.7)
Requirement already satisfied: threadpoolctl>=2.0.0 in e:\software\anaconda3\lib\site-packages (from scikit-learn>=0.22->pmdarima)
Requirement already satisfied: patsy>=0.5.2 in e:\software\anaconda3\lib\site-packages (from statsmodels>=0.13.2->pmdarima) (0.5.3)
Requirement already satisfied: packaging>=21.3 in e:\software\anaconda3\lib\site-packages (from statsmodels>=0.13.2->pmdarima) (23.
Requirement already satisfied: six in e:\software\anaconda3\lib\site-packages (from patsy>=0.5.2->statsmodels>=0.13.2->pmdarima) (1
Note: you may need to restart the kernel to use updated packages.
```

```
pip install prophet
```

```
Requirement already satisfied: prophet in e:\software\anaconda3\lib\site-packages (1.1.4)
Requirement already satisfied: cmdstanpy>=1.0.4 in e:\software\anaconda3\lib\site-packages (from prophet) (1.1.0)
Requirement already satisfied: numpy>=1.15.4 in e:\software\anaconda3\lib\site-packages (from prophet) (1.24.3)
Requirement already satisfied: matplotlib>=2.0.0 in e:\software\anaconda3\lib\site-packages (from prophet) (3.7.1)
Requirement already satisfied: pandas>=1.0.4 in e:\software\anaconda3\lib\site-packages (from prophet) (1.5.3)
Requirement already satisfied: LunarCalendar>=0.0.9 in e:\software\anaconda3\lib\site-packages (from prophet) (0.0.9)
Requirement already satisfied: convertdate>=2.1.2 in e:\software\anaconda3\lib\site-packages (from prophet) (2.4.0)
Requirement already satisfied: holidays>=0.25 in e:\software\anaconda3\lib\site-packages (from prophet) (0.31)
Requirement already satisfied: python-dateutil>=2.8.0 in e:\software\anaconda3\lib\site-packages (from prophet) (2.8.2)
Requirement already satisfied: tqdm>=4.36.1 in e:\software\anaconda3\lib\site-packages (from prophet) (4.65.0)
Requirement already satisfied: importlib-resources in e:\software\anaconda3\lib\site-packages (from prophet) (6.0.1)
Requirement already satisfied: pymeeus<=1,>=0.3.13 in e:\software\anaconda3\lib\site-packages (from convertdate>=2.1.2->prophet) (0
Requirement already satisfied: ephem>=3.7.5.3 in e:\software\anaconda3\lib\site-packages (from LunarCalendar>=0.0.9->prophet) (4.1.
Requirement already satisfied: pytz in e:\software\anaconda3\lib\site-packages (from LunarCalendar>=0.0.9->prophet) (2022.7)
Requirement already satisfied: contourpy>=1.0.1 in e:\software\anaconda3\lib\site-packages (from matplotlib>=2.0.0->prophet) (1.0.5
Requirement already satisfied: cycler>=0.10 in e:\software\anaconda3\lib\site-packages (from matplotlib>=2.0.0->prophet) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in e:\software\anaconda3\lib\site-packages (from matplotlib>=2.0.0->prophet) (4.25
Requirement already satisfied: kiwisolver>=1.0.1 in e:\software\anaconda3\lib\site-packages (from matplotlib>=2.0.0->prophet) (1.4.
Requirement already satisfied: packaging>=20.0 in e:\software\anaconda3\lib\site-packages (from matplotlib>=2.0.0->prophet) (23.0)
Requirement already satisfied: pillow>=6.2.0 in e:\software\anaconda3\lib\site-packages (from matplotlib>=2.0.0->prophet) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in e:\software\anaconda3\lib\site-packages (from matplotlib>=2.0.0->prophet) (3.0.9
Requirement already satisfied: six>=1.5 in e:\software\anaconda3\lib\site-packages (from python-dateutil>=2.8.0->prophet) (1.16.0)
Requirement already satisfied: colorama in e:\software\anaconda3\lib\site-packages (from tqdm>=4.36.1->prophet) (0.4.6)
Note: you may need to restart the kernel to use updated packages.
```

```
# Import libraries
import numpy as np
import pandas as pd
import matplotlib
from matplotlib import pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
from matplotlib.pylab import rcParams
matplotlib.rcParams['timezone'] = 'UTC'
import seaborn as sns

from scipy import stats
from random import gauss as gs
import math
import datetime
```

```
from math import sqrt

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

import statsmodels.api as sm
from statsmodels.tsa.arima_model import ARMA
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
from pandas.plotting import autocorrelation_plot, lag_plot
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.statespace.sarimax import SARIMAX
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tools.sm_exceptions import ConvergenceWarning

from pmdarima import auto_arima
from prophet import Prophet

import warnings
warnings.filterwarnings('ignore')

import itertools


# import pmdarima as pm


#loading dataset and viewing first 5 rows
data = pd.read_csv('zillow_data.csv')

data.head()
```

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	1996-05	1996-06	...	2017-07	2017-08	2017-09
0	84654	60657	Chicago	IL	Chicago	Cook	1	334200.0	335400.0	336500.0	...	1005500	1007500	1007800
1	90668	75070	McKinney	TX	Dallas-Fort Worth	Collin	2	235700.0	236900.0	236700.0	...	308000	310000	312500
2	91982	77494	Katy	TX	Houston	Harris	3	210400.0	212200.0	212200.0	...	321000	320600	320200
3	84616	60614	Chicago	IL	Chicago	Cook	4	498100.0	500900.0	503100.0	...	1289800	1287700	1287400
4	93144	79936	El Paso	TX	El Paso	El Paso	5	77300.0	77300.0	77300.0		119100	119400	120000

```
data.info()
```

```
<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
```

```
data.columns
```

```
Index(['RegionID', 'RegionName', 'City', 'State', 'Metro', 'CountyName',
       'SizeRank', '1996-04', '1996-05', '1996-06',
       ...
       '2017-07', '2017-08', '2017-09', '2017-10', '2017-11', '2017-12',
       '2018-01', '2018-02', '2018-03', '2018-04'],
      dtype='object', length=272)
```

```
# Examine the shape of the dataset
data.shape
```

```
(14723, 272)
```

```
# Viewing number of zipcodes in dataset and nan values
print('Total number of zipcodes:', len(data))
data.iloc[:, :20].isna().sum() # up to 20th column
```

```
Total number of zipcodes: 14723
RegionID         0
RegionName       0
```

```
City            0
State           0
Metro        1043
CountyName      0
SizeRank        0
1996-04      1039
1996-05      1039
1996-06      1039
1996-07      1039
1996-08      1039
1996-09      1039
1996-10      1039
1996-11      1039
1996-12      1039
1997-01      1039
1997-02      1039
1997-03      1039
1997-04      1039
dtype: int64
```

The data understanding above shows as follows:

- Our data has 14273 rows and 272 columns
- The columns have bothe categorical and numerical features, City', 'State', 'Metro', 'CountyName' are of categorical features while RegionID', 'RegionName', 'SizeRank' and monthly columns of sales price per zipcode(1996-04 through 2018-04) are of numerical type
- The 'Metro' and mmonthly columns of sales price per zipcode have missing values.

We shall proceed to data preparation to deal with duplicates missing values, visualization and feature engineering.

3.0 Data Preparation

3.1. Exploratory Data Analysis(EDA) and Feature Engineering**

First, some basic EDA and Feature Engineering will be performed.

- **N/B:** There will be a lot of EDA and feature engineering required throughout this project, therefore, the processes will be performed as needed.

a. Checking for Duplicates

- The data has no duplicates.

```
# check for duplicates
data.duplicated().sum()
```

```
0
```

b. Changing `RegionName` to `ZipCode`

The data types are okay as expected.

```
#Inspecting dataframe dtypes
data.iloc[:, :20].dtypes
```

```
RegionID        int64
RegionName      int64
City           object
State          object
Metro          object
CountyName     object
SizeRank        int64
1996-04       float64
1996-05       float64
1996-06       float64
1996-07       float64
1996-08       float64
1996-09       float64
1996-10       float64
1996-11       float64
1996-12       float64
1997-01       float64
1997-02       float64
1997-03       float64
1997-04       float64
dtype: object
```

However, `RegionName` column will be renamed to `ZipCode` for easier recognition.

- Further, the column will be converted to a string.

```
# change 'RegionName' column to 'ZipCode' to avoid confusion
data.rename(columns={'RegionName': 'ZipCode'}, inplace=True)

# changing dtype to str
data.ZipCode = data.ZipCode.astype('string')

#confirming the change
print(data.dtypes["ZipCode"])
```

```
string
```

c. Missing Values

There are some missing values in the `Metro` and `dates` columns. To deal with this:

- Missing values in the **Metro** column will be replaced with **missing**.
- Missing values in the dates columns will be backfilled.

```
data.isnull().sum()
```

```
RegionID       0
ZipCode        0
City           0
State          0
Metro       1043
            ...
2017-12        0
2018-01        0
2018-02        0
2018-03        0
2018-04        0
Length: 272, dtype: int64
```

```
# Define a function to explore missing data
def missing(data):
    missing_data = data.isna().sum()
    missing_data = missing_data[missing_data>0]
    return missing_data.to_frame()

# Apply missing_data function to the dataframe
missing(data)
```

	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

220 rows × 1 columns

```
# replacing the missing values bin Metro with 'missing'
data.Metro.fillna('missing', inplace=True)

# interpolate missing values on date columns
data.interpolate(method="pad", inplace=True)

data.isna().sum().sum()
```

```
0
```

d. Creating New Columns for Analysis

Select Top 10 States to Invest in Real Estate

- The data will first be filtered to choose the ten best states to invest in real estate in the USA. The guiding information has been extracted from this article by Fit Small Business.
- The top 10 states most suitable for investment that yields high returns in real estate are:

1. Georgia (GA)
2. Utah (UT)
3. Texas (TX)
4. North Carolina (NC)
5. New Jersey (NJ)
6. Tennessee (TN)
7. Washington (WA)
8. Delaware (DE)
9. Nebraska (NE)
10. Florida (FL).

- This narrows down the number of Zip Codes to **4039**.

```
# List of states to invest in
states_to_invest = ['GA', 'UT', 'TX', 'NC', 'NJ', 'TN', 'WA', 'DE', 'NE', 'FL']

# Filter data for the selected states
# new df - data_states created
data_states = data[data['State'].isin(states_to_invest)].copy()

# Drop unnecessary columns
columns_to_drop = ['RegionID', 'SizeRank']
data_states.drop(columns_to_drop, axis=1, inplace=True)

# Reset the index
data_states.reset_index(drop=True, inplace=True)

print('Total Zipcodes in DataFrame:', len(data_states))
data_states.head()
```

```
Total Zipcodes in DataFrame: 4039
```

	ZipCode	City	State	Metro	CountyName	1996-04	1996-05	1996-06	1996-07	1996-08	...	2017-07	2017-08	2017-09	2017-10
0	75070	McKinney	TX	Dallas-Fort Worth	Collin	235700.0	236900.0	236700.0	235400.0	233300.0	...	308000	310000	312500	314100
1	77494	Katy	TX	Houston	Harris	210400.0	212200.0	212200.0	210700.0	208300.0	...	321000	320600	320200	320400
2	79936	El Paso	TX	El Paso	El Paso	77300.0	77300.0	77300.0	77300.0	77400.0	...	119100	119400	120000	120300
3	77084	Houston	TX	Houston	Harris	95000.0	95200.0	95400.0	95700.0	95900.0	...	157900	158700	160200	161900

```
data_states.shape
```

```
(4039, 270)
```

e. Calculating ROI and CV

- Next, 5 columns will be created:

1. `CV`: This will store the coefficient of variation values. The information is used by investors to gauge the degree of volatility or risk in relation to the anticipated returns from their investments.
    - It is calculated as standard deviation divided by the mean.
    - A low CV means there is a lower investment risk, and vice versa for a high CV.
    - 22-year and 5-year CV will be calculated.
2. `ROI`: This will store values that show the measure of the profitability of an investment relative to its cost and is typically expressed as a percentage.

- A high ROI indicates that the gains or profits obtained from the investment are significantly greater than the initial investment itself.
- 22-year, 5-year and 3-year ROI will be calculated.

```
# 22-year ROI
data_states['ROI_all'] = round(((data_states['2018-04'] / data_states['1996-04']) - 1) * 100, 2)

## 5-year ROI
data_states['ROI_5yr'] = round(((data_states['2018-04'] - data_states['2013-04']) / data_states['2013-01']) * 100, 2)

## 3-year ROI
data_states['ROI_3yr'] = round(((data_states['2018-04'] - data_states['2015-04']) / data_states['2015-01']) * 100, 2)

# standard deviation 22 years
data_states['std_all'] = round(data_states.loc[:,'1996-04':'2018-04'].std(skipna=True, axis=1), 2)

# standard deviation 5 years
data_states['std_5yr'] = round(data_states.loc[:,'2013-04':'2018-04'].std(skipna=True, axis=1), 2)

# mean 22 years
data_states['mean_all'] = round(data_states.loc[:,'1996-04':'2018-04'].mean(skipna=True, axis=1), 2)

# mean 5 years
data_states['mean_5yr'] = round(data_states.loc[:,'2013-04':'2018-04'].mean(skipna=True, axis=1), 2)

# 22-year CV
data_states['CV_all'] = round(data_states['std_all']/data_states['mean_all'] * 100, 2)

# 5-year CV
data_states['CV_5yr'] = round(data_states['std_5yr']/data_states['mean_5yr'] * 100, 2)

# Displaying the columns
data_states[['State', 'ZipCode','ROI_all','ROI_5yr', 'ROI_3yr', 'CV_all', 'CV_5yr']].head()
```

	State	ZipCode	ROI_all	ROI_5yr	ROI_3yr	CV_all	CV_5yr
0	TX	75070	36.53	54.40	25.38	15.27	12.46
1	TX	77494	56.80	28.34	7.86	14.39	7.24
2	TX	79936	57.18	7.08	5.49	17.83	2.31
3	TX	77084	72.95	44.66	16.92	14.23	10.54
4	TX	77449	84.70	48.51	22.41	14.70	11.80

```
#Check for zeros
data_states.describe()
```

	1996-04	1996-05	1996-06	1996-07	1996-08	1996-09	1996-10	1996-11	1996-12	
count	4.039000e+03	4.039000e+03	4.039000e+03	4.039000e+03	4.039000e+03	4.039000e+03	4.039000e+03	4.039000e+03	4.039000e+03	4.
mean	1.130141e+05	1.131389e+05	1.132644e+05	1.133919e+05	1.135325e+05	1.136936e+05	1.138956e+05	1.141223e+05	1.144401e+05	1.
std	7.531940e+04	7.529945e+04	7.529120e+04	7.529444e+04	7.531903e+04	7.536819e+04	7.545085e+04	7.556257e+04	7.572800e+04	7.
min	1.140000e+04	1.150000e+04	1.160000e+04	1.180000e+04	1.180000e+04	1.200000e+04	1.210000e+04	1.220000e+04	1.230000e+04	1.
25%	6.420000e+04	6.430000e+04	6.455000e+04	6.500000e+04	6.510000e+04	6.530000e+04	6.530000e+04	6.550000e+04	6.580000e+04	6.
50%	9.300000e+04	9.310000e+04	9.310000e+04	9.320000e+04	9.330000e+04	9.350000e+04	9.370000e+04	9.390000e+04	9.410000e+04	9.
75%	1.385000e+05	1.385500e+05	1.388500e+05	1.389000e+05	1.395500e+05	1.397500e+05	1.397000e+05	1.397000e+05	1.402000e+05	1.
max	1.075400e+06	1.075900e+06	1.076500e+06	1.077100e+06	1.077700e+06	1.078400e+06	1.079200e+06	1.080000e+06	1.081200e+06	1.

8 rows × 274 columns

```
# viewing the selected States by highest 22-yr ROI
states_roi = round(data_states.groupby('State', group_keys=False).sum()[['ROI_all', 'ROI_5yr', 'ROI_3yr']] / 100, 2)
states_roi.sort_values(by=['ROI_all'], ascending=False)
```

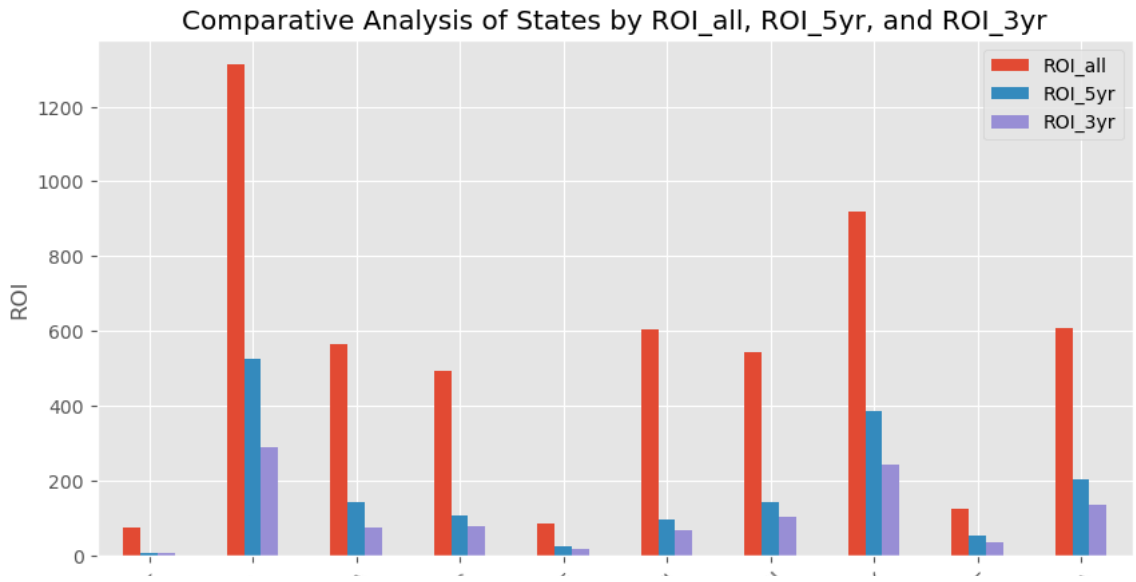
State	ROI_all	ROI_5yr	ROI_3yr
FL	1312.23	524.30	291.05
TX	920.99	384.54	242.72
WA	609.43	203.68	134.62
NJ	605.86	96.68	69.43

Observations

Even though the analysis will be focussed on Zip Codes, the states' ROI analysis provides a general overview of zip codes to expect in the top-performing list.

- The State of Florida has the highest ROI for all specified periods.
- Even though New Jersey comes third in 22-year ROI, it performs poorly in the 5-year and 3-year ROI.
- Delaware State has the lowest ROI values for all periods.

```
# Create a bar chart of the top 10 zip codes by ROI, ROI_5yr, and ROI_3yr
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
states_roi.plot(kind='bar', ax=ax)

# Set the title, labels, and ticks
plt.title("Comparative Analysis of States by ROI_all, ROI_5yr, and ROI_3yr")
ax.set_ylabel("ROI")
ax.set_xlabel("States")
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability

# plt.savefig('images/ROI_comp.jpg')
# Show the bar chart
plt.show()
```



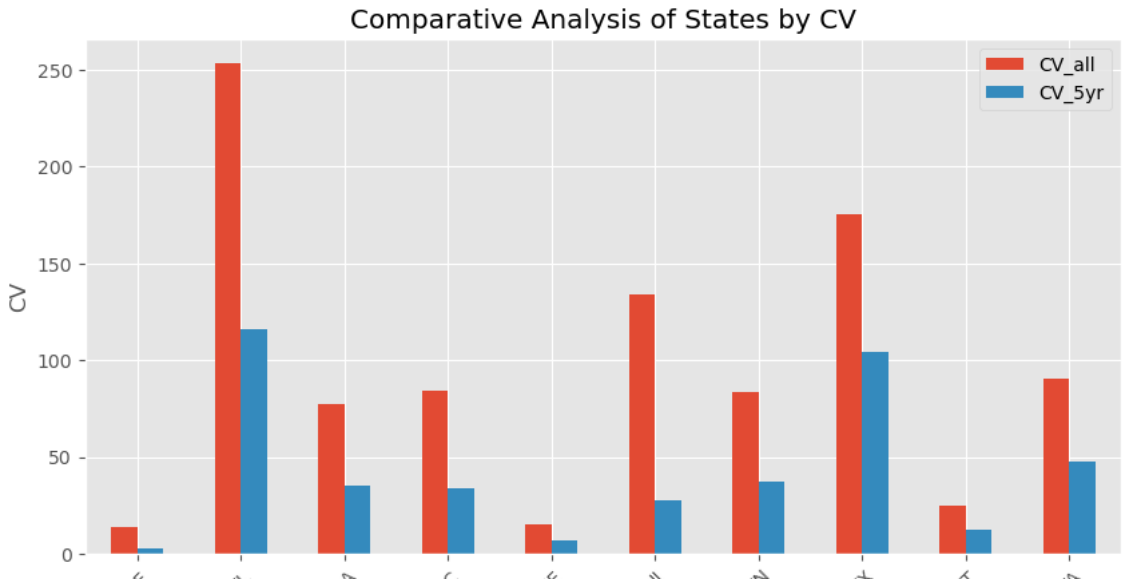

```
# displaying states by CV
states_cv = round(data_states.groupby('State', group_keys=False).sum()[['CV_all', 'CV_5yr']] / 100, 2)
states_cv.sort_values(by=['CV_all'], ascending=False)
```

State	CV_all	CV_5yr
FL	253.18	115.77
TX	175.40	104.29
NJ	133.80	27.52
WA	90.08	47.92
NC	84.02	33.94
TN	83.48	37.11
GA	77.17	35.39
UT	24.72	12.67
NE	15.18	7.01
DE	13.73	2.65

Observations

- Again, Florida State has the highest CV values. This is an indication that even though the return on investment is great, it is still a risky state to invest in. Therefore, zip codes in this state will be investigated with a skeptical attitude.
- Delaware has low returns and low risk.

```
# Create a bar chart of the top 10 zip codes by CV_all and CV_5yr
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
states_cv.plot(kind='bar', ax=ax)

# Set the title, labels, and ticks
plt.title("Comparative Analysis of States by CV")
ax.set_ylabel("CV")
ax.set_xlabel("States")
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability

# plt.savefig('images/CV_comp.jpg')
# Show the bar chart
plt.show()
```




```
data_states.head()
```

	ZipCode	City	State	Metro	CountyName	1996-04	1996-05	1996-06	1996-07	1996-08	...	2018-04	ROI_all	ROI_5yr	ROI_
0	75070	McKinney	TX	Dallas-Fort Worth	Collin	235700.0	236900.0	236700.0	235400.0	233300.0	...	321800	36.53	54.40	25
1	77494	Katy	TX	Houston	Harris	210400.0	212200.0	212200.0	210700.0	208300.0	...	329900	56.80	28.34	7
2	79936	El Paso	TX	El Paso	El Paso	77300.0	77300.0	77300.0	77300.0	77400.0	...	121500	57.18	7.08	5
3	77084	Houston	TX	Houston	Harris	95000.0	95200.0	95400.0	95700.0	95900.0	...	164300	72.95	44.66	16

```
data_states.nunique()
```

```
ZipCode       4039
City          2370
State           10
Metro          219
CountyName     431
              ...
std_5yr       4039
mean_all      4022
mean_5yr      4000
CV_all        2198
CV_5yr        1697
Length: 279, dtype: int64
```

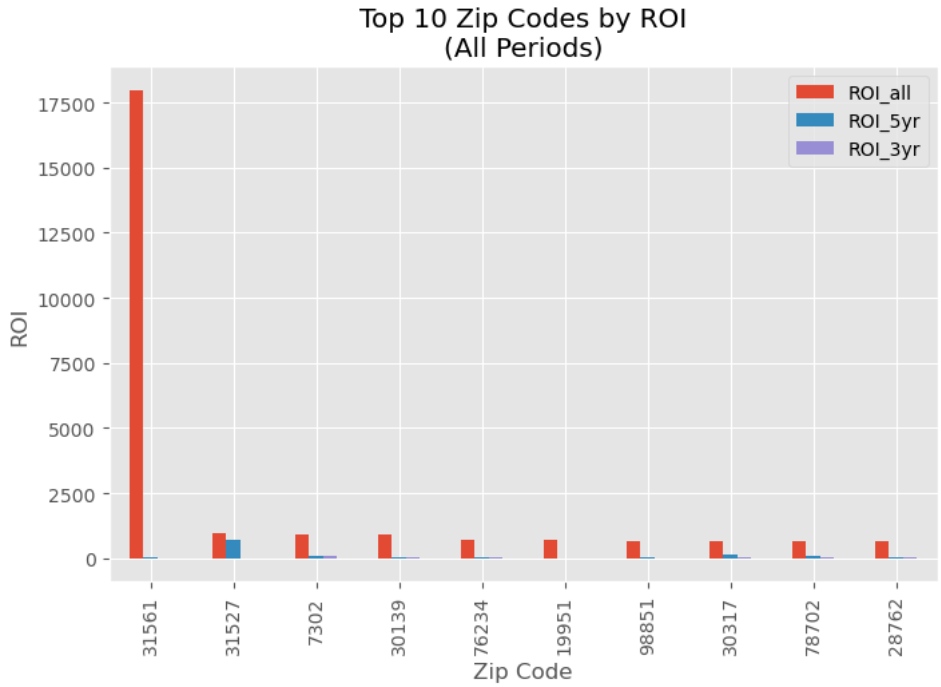
f. Preparation for Modelling

- The next lines of code will rank zip codes by different ROI and CV periods.
- This will give an overview of how the zip codes performed individually.
- The codes will filter out the top 10 zip codes by the given criteria.

1. All ROI Periods

- Extract top 10 zip codes by `ROI_all` in descending order

```
# Extract top 10 zip codes by ROI_all in descending order
top_10_zipcodes_roi_all = data_states.sort_values(by='ROI_all', ascending=False).head(10)

# Plotting the graph
fig, ax = plt.subplots(figsize=(8, 5))
top_10_zipcodes_roi_all.plot(kind='bar', x='ZipCode', y=['ROI_all', 'ROI_5yr', 'ROI_3yr'], ax=ax)
ax.set_xlabel('Zip Code')
ax.set_ylabel('ROI')
ax.set_title('Top 10 Zip Codes by ROI\n(All Periods)')
# plt.savefig('images/ROI_all')
plt.show()
```



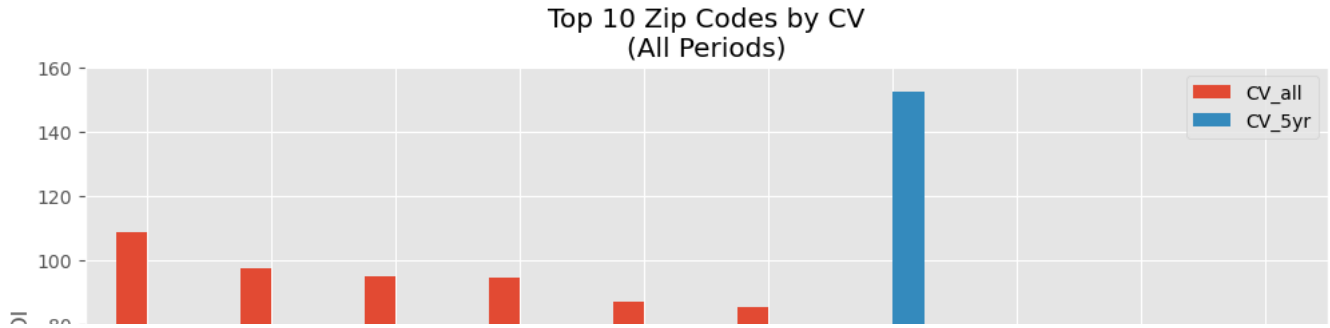
Observations

- Zip Code 31561 seems to be performing extremely well in terms of potential for profits.
- The oter zip codes are somewhere at the same performance level.

2. All CV Periods

- Extract top 10 zip codes by `CV_all` in descending order

```
# Extract top 10 zip codes by CV_all in descending order
top_10_zipcodes_cv_all = data_states.sort_values(by='CV_all', ascending=False).head(10)

# Plotting the graph
fig, ax = plt.subplots(figsize=(12, 5))
top_10_zipcodes_cv_all.plot(kind='bar', x='ZipCode', y=['CV_all', 'CV_5yr'], ax=ax)
ax.set_xlabel('Zip Code')
ax.set_ylabel('ROI')
ax.set_title('Top 10 Zip Codes by CV\n(All Periods)')
# plt.savefig('images/CV_all')
plt.show()
```



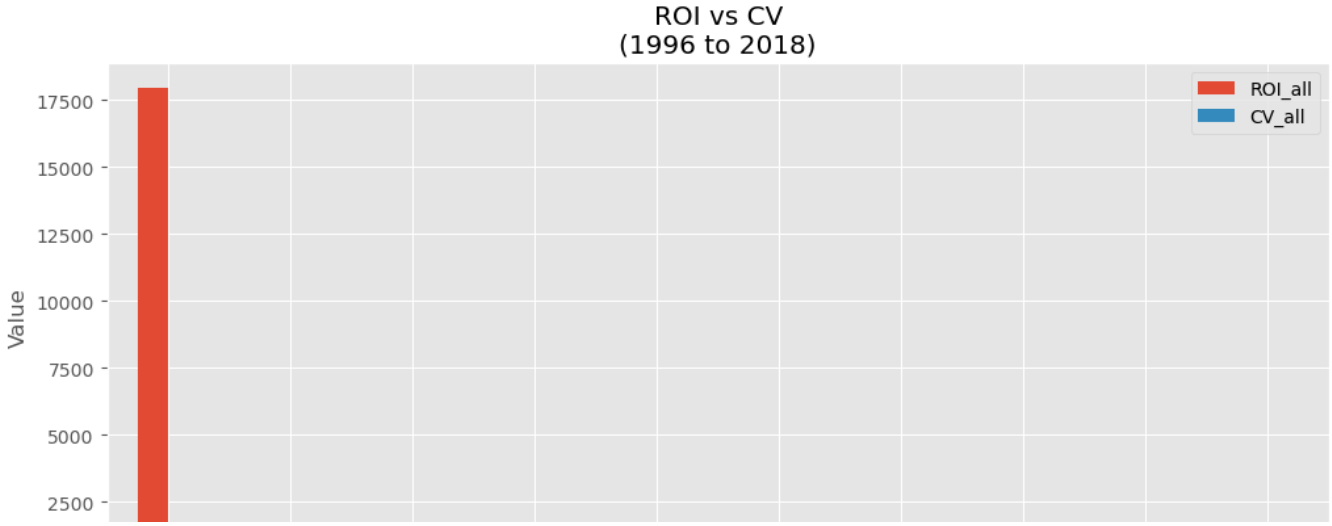
Observations

- Zip code 28039 has the highest CV for the past five years.
- 31527 has the highest overall CV.
- The goal is to choose the zip codes with least CV. 31027, 19962 and 99163 may fit into this criteria,

3. ROI and CV - 22 years

- Extract top 10 zip codes by `ROI_all` in descending order.
- Plot both `ROI_all` and `CV_all`.

```
# Extract top 10 zip codes by ROI_all in descending order
top_10_zipcodes_roi_cv_all = data_states.sort_values(by='ROI_all', ascending=False).head(10)

# Plotting the graph
fig, ax = plt.subplots(figsize=(12, 5))
top_10_zipcodes_roi_cv_all.plot(kind='bar', x='ZipCode', y=['ROI_all', 'CV_all'], ax=ax)
ax.set_xlabel('Zip Code')
ax.set_ylabel('Value')
ax.set_title('ROI vs CV\n(1996 to 2018)')
# plt.savefig('images/ROI_CV_all')
plt.show()
```



4. ROI and CV - 5 years

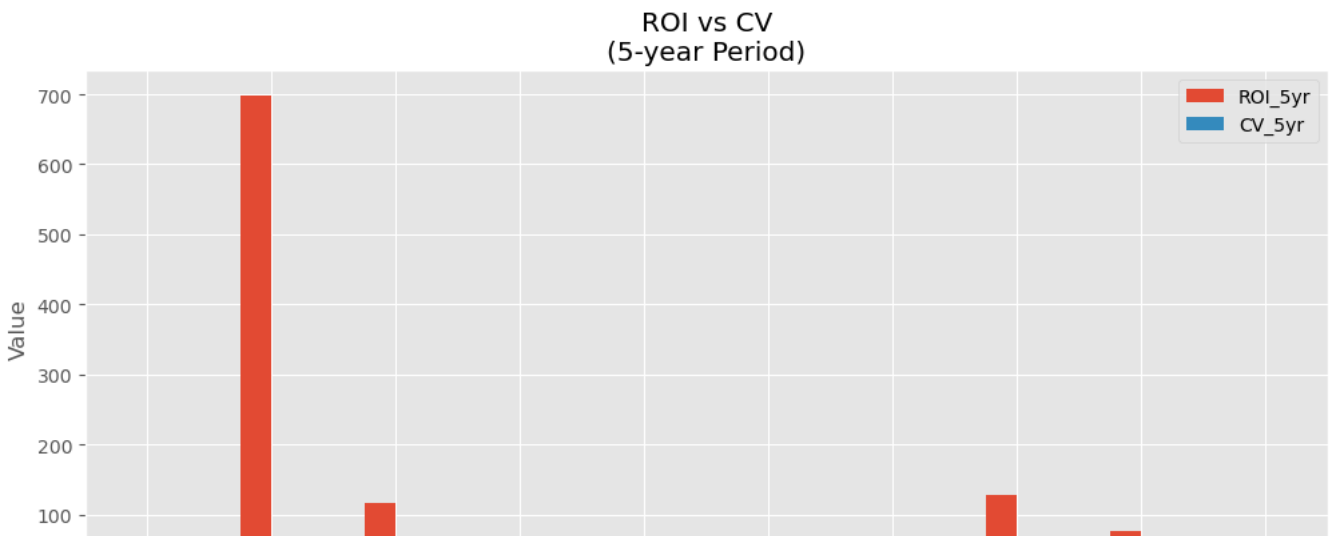
- Extract top 10 zip codes by `ROI_5yr` in descending order.
- Plot both `ROI_5yr` and `CV_5yr`.

Observations

- This graph is unclear on how the two values compare because the extreme ROI value for 31561 seems to overpower the rest.

```
# Extract top 10 zip codes by ROI_all in descending order
top_10_zipcodes_roi_cv_5yr = data_states.sort_values(by='ROI_all', ascending=False).head(10)

# Plotting the graph
fig, ax = plt.subplots(figsize=(12, 5))
top_10_zipcodes_roi_cv_5yr.plot(kind='bar', x='ZipCode', y=['ROI_5yr', 'CV_5yr'], ax=ax)
ax.set_xlabel('Zip Code')
ax.set_ylabel('Value')
ax.set_title('ROI vs CV\n(5-year Period)')
```

```
# plt.savefig('images/ROI_CV_5yr')
plt.show()
```



Observations

- Here, 31527 stands out again with a high 5-year ROI. The CV, however, is unfortunately also high.
- 7302, 30317 and 78702 have high ROIs but higher than most CV values.
- This graph seems to give a better comparable visualization and will be used to form the time series for modelling.

```
top_10_zipcodes_roi_cv_5yr.head()
```

	ZipCode	City	State	Metro	CountyName	1996-04	1996-05	1996-06	1996-07	1996-08	...	2018-04	ROI_all	ROI_5yr
4025	31561	Sea Island	GA	Brunswick	Glynn	13500.0	13500.0	13500.0	13400.0	13400.0	...	2440000	17974.07	20.9
3877	31527	Brunswick	GA	Brunswick	Glynn	42000.0	42100.0	42100.0	42100.0	42200.0	...	444700	958.81	698.7
77	7302	Jersey City	NJ	New York	Hudson	137200.0	137800.0	138500.0	139100.0	139600.0	...	1427300	940.31	116.0
3424	30139	Fairmount	GA	Calhoun	Gordon	14000.0	14300.0	14500.0	14800.0	15000.0	...	139600	897.14	47.5
2053	76234	Decatur	TX	Dallas-Fort Worth	Wise	25600.0	25500.0	25500.0	25400.0	25400.0	...	209800	719.53	26.6

```
top_10_zipcodes_roi_cv_5yr.shape
```

```
(10, 279)
```

```
# confirming the list of Zip Codes
top_10_zipcodes_roi_cv_5yr[["ZipCode", "ROI_5yr", "CV_5yr"]]
```

	ZipCode	ROI_5yr	CV_5yr
4025	31561	20.96	7.30
3877	31527	698.74	46.41
77	7302	116.07	27.03
3424	30139	47.56	11.48
2053	76234	26.61	8.78
3917	19951	12.49	3.62
3425	98851	17.95	7.55
2171	30317	128.87	22.76
1303	78702	76.49	16.18
2940	28762	24.30	6.94

Identifying the Zip Codes by City, County, Metro and State

Zip Code	City	County	Metro	State
31561	Sea Island	Glynn County	Brunswick	Georgia
31527	Jekyll Island	Glynn County	Brunswick	Georgia
7302	Jersey City	Hudson County	New York	New Jersey
30139	Fairmount	Gordon County	Calhoun	Georgia
76234	Decatur	Wise County	Dallas-Fort Worth	Texas
19951	Harbeson	Sussex County	Salisbury	Delaware
98851	Soap Lake	Grant County	Moses Lake	Washington
30317	Atlanta	DeKalb County	Atlanta	Georgia
78702	Austin	Travis	Austin	Texas
28762	Old Fort	McDowell	Marion	North Carolina

```
final_df = top_10_zipcodes_roi_cv_5yr.drop(['City', 'State', 'Metro', 'CountyName',
                                    'ROI_all', 'ROI_3yr',
                                      'std_all', 'std_5yr',
                                      'mean_all','mean_5yr',
                                      'CV_all'], axis=1)
```

```
final_df.head()
```

	ZipCode	1996-04	1996-05	1996-06	1996-07	1996-08	1996-09	1996-10	1996-11	1996-12	...	2017-09	2017-10	2017-11
4025	31561	13500.0	13500.0	13500.0	13400.0	13400.0	13400.0	13400.0	13400.0	13500.0	...	2344700	2359500	2403900
3877	31527	42000.0	42100.0	42100.0	42100.0	42200.0	42200.0	42200.0	42200.0	42300.0	...	400100	398800	400900
77	7302	137200.0	137800.0	138500.0	139100.0	139600.0	140100.0	140700.0	141400.0	142300.0	...	1411000	1435900	1446300
3424	30139	14000.0	14300.0	14500.0	14800.0	15000.0	15300.0	15700.0	16000.0	16400.0	...	129900	131600	132600
2053	76234	25600.0	25500.0	25500.0	25400.0	25400.0	25400.0	25500.0	25500.0	25600.0	...	197100	198400	200300

5 rows × 268 columns

```
final_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10 entries, 4025 to 2940
Columns: 268 entries, ZipCode to CV_5yr
dtypes: float64(221), int64(46), string(1)
memory usage: 21.0 KB
```

5.0 Converting to Time Series

```
time_df= final_df.melt(id_vars=['ZipCode', 'ROI_5yr', 'CV_5yr'], var_name='Time', value_name='Sale Price')
# time_df['Time'] = pd.to_datetime(final_df['Time'], infer_datetime_format=True)
time_df['Time'] = pd.to_datetime(time_df['Time'])
time_df.set_index('Time', inplace=True)
time_df = time_df.pivot_table('Sale Price', ['Time'], 'ZipCode')
time_df.head()
```

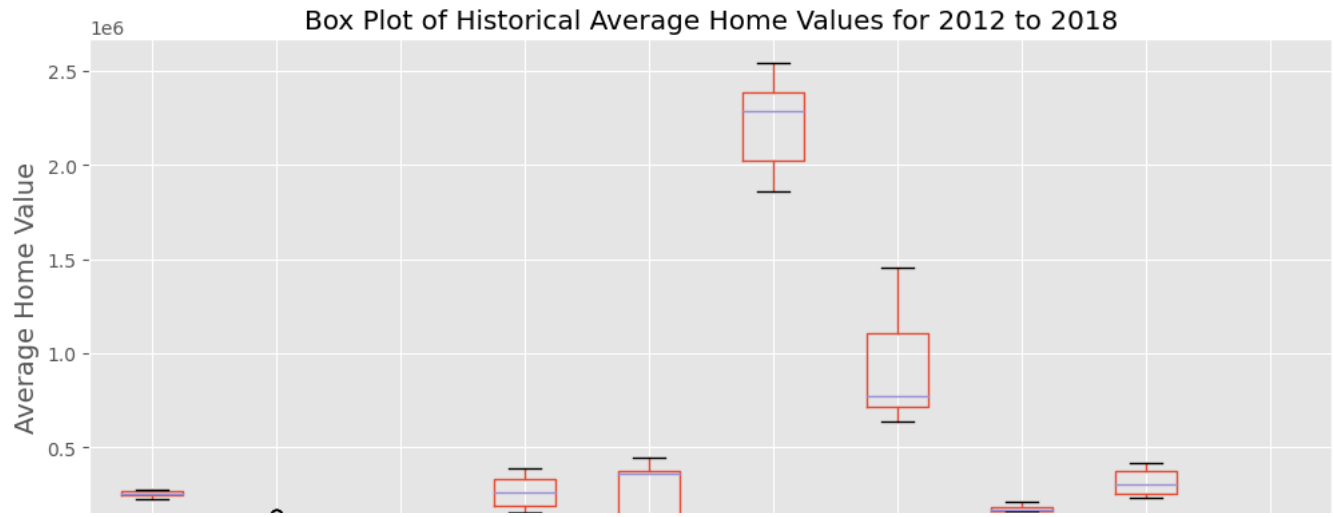
ZipCode	19951	28762	30139	30317	31527	31561	7302	76234	78702	98851
Time										
1996-04-01	34500.0	19200.0	14000.0	51800.0	42000.0	13500.0	137200.0	25600.0	55600.0	14000.0
1996-05-01	34900.0	19700.0	14300.0	52500.0	42100.0	13500.0	137800.0	25500.0	56700.0	14300.0
1996-06-01	35200.0	20100.0	14500.0	53200.0	42100.0	13500.0	138500.0	25500.0	57900.0	14500.0
1996-07-01	35700.0	20600.0	14800.0	53900.0	42100.0	13400.0	139100.0	25400.0	59300.0	14800.0
1996-08-01	36100.0	21000.0	15000.0	54500.0	42200.0	13400.0	139600.0	25400.0	60800.0	15000.0

```
time_df.index.name
```

```
'Time'
```

The boxplot below shows the averages home values for the selected zip codes over the dataset's time period.

```
start_year = 2012
end_year = 2018

# Filter data for the specified range of years
filtered_df = time_df.loc[f'{start_year}-01':f'{end_year}-12']

# Create a box plot
plt.figure(figsize=(12, 5))
filtered_df.boxplot(figsize=(10, 5))
plt.xlabel('Time', fontsize=14)
plt.ylabel('Average Home Value', fontsize=14)
plt.title(f'Box Plot of Historical Average Home Values for {start_year} to {end_year}')
plt.xticks(rotation=45)
plt.show()
```




```
time_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 265 entries, 1996-04-01 to 2018-04-01
Data columns (total 10 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   19951   265 non-null    float64
 1   28762   265 non-null    float64
 2   30139   265 non-null    float64
 3   30317   265 non-null    float64
 4   31527   265 non-null    float64
 5   31561   265 non-null    float64
 6   7302    265 non-null    float64
 7   76234   265 non-null    float64
 8   78702   265 non-null    float64
 9   98851   265 non-null    float64
dtypes: float64(10)
memory usage: 30.9 KB
```

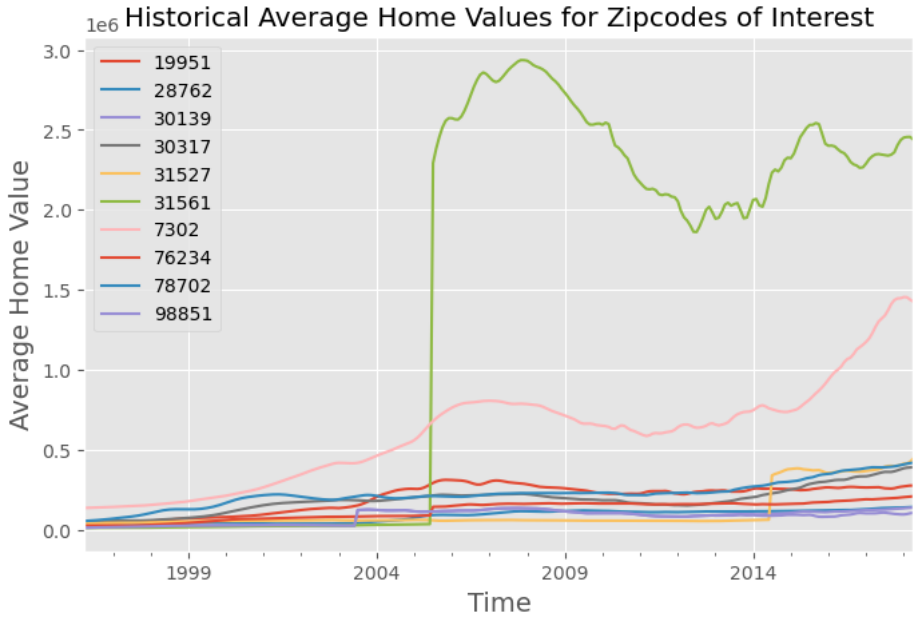
```
time_df.describe()
```

ZipCode	19951	28762	30139	30317	31527	31561	7302	76234	
count	265.000000	265.000000	265.000000	265.000000	265.000000	2.650000e+02	2.650000e+02	265.000000	265.0
mean	199729.433962	85044.528302	83786.037736	187885.660377	112324.150943	1.406019e+06	5.965913e+05	123060.754717	224687.1
std	86478.435971	38492.697781	43029.086284	78258.636020	122325.030043	1.198596e+06	3.173382e+05	55208.380452	79154.5
min	34500.000000	19200.000000	14000.000000	51800.000000	42000.000000	1.340000e+04	1.372000e+05	25400.000000	55600.0
25%	117300.000000	41700.000000	29100.000000	156400.000000	56200.000000	2.480000e+04	3.259000e+05	76000.000000	199000.0
50%	240700.000000	106800.000000	99400.000000	185700.000000	58800.000000	2.032300e+06	6.432000e+05	159300.000000	220300.0
75%	263300.000000	116700.000000	120100.000000	218500.000000	61400.000000	2.451700e+06	7.713000e+05	164600.000000	233200.0
max	312800.000000	143000.000000	139600.000000	391600.000000	444700.000000	2.935800e+06	1.454900e+06	209800.000000	419800.0

The line graph below shows the historical average home values for the selected zip codes over the dataset's time period.

```
time_df.plot(figsize=(8,5))
plt.xlabel('Time', fontsize=14)
plt.ylabel('Average Home Value', fontsize=14)
plt.title('Historical Average Home Values for Zipcodes of Interest')
```

```
# plt.savefig('images/hist_avg')
plt.legend();
```



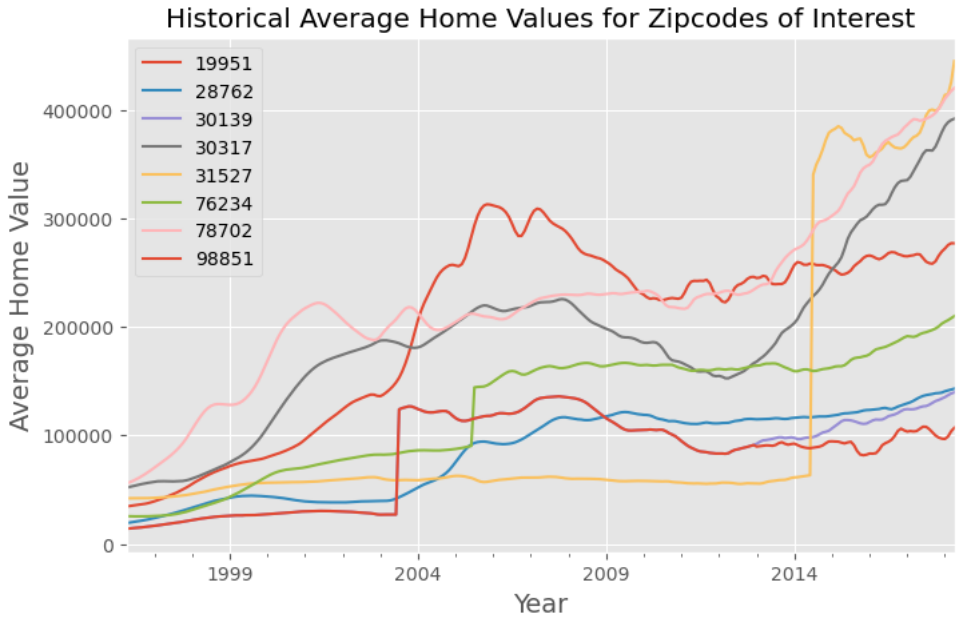

```
time_df.columns
```

```
Index(['19951', '28762', '30139', '30317', '31527', '31561', '7302', '76234',
       '78702', '98851'],
      dtype='string', name='ZipCode')
```

Observations

- We shall remove `31561` and `7302`. They are outliers in the dataset and have dramatic trends over the period under review.

```
top_8_df = time_df.drop(['31561','7302'], axis=1)
top_8_df.plot(figsize=(8,5))
plt.xlabel('Year', fontsize=14)
plt.ylabel('Average Home Value', fontsize=14)
plt.title('Historical Average Home Values for Zipcodes of Interest')
# plt.savefig('images/hist_avg_top8')
plt.legend();
```




```
# confirming that 8 zipcodes are left
top_8_df.head()
```

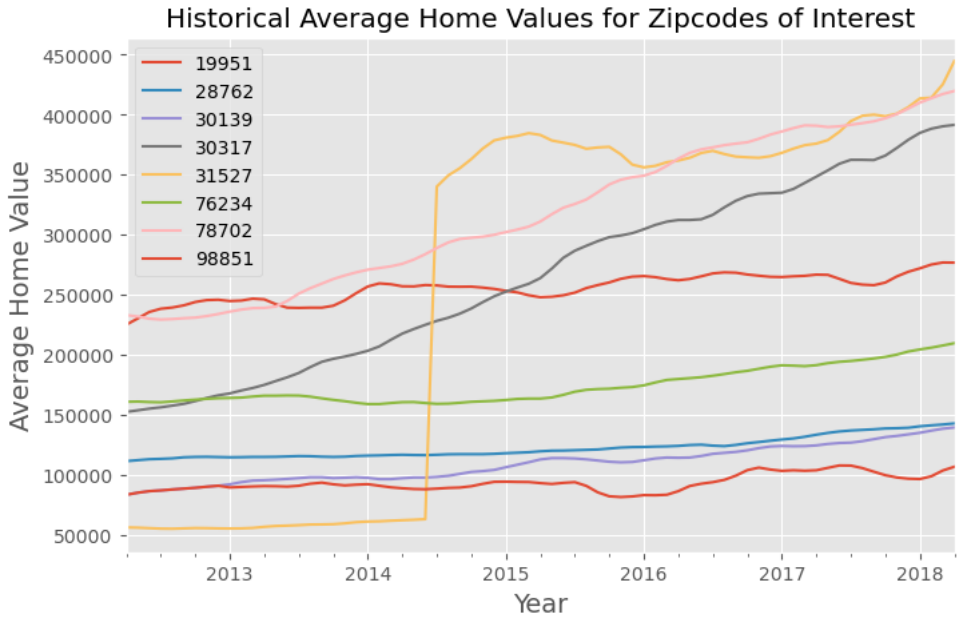
ZipCode	19951	28762	30139	30317	31527	76234	78702	98851
Time								
1996-04-01	34500.0	19200.0	14000.0	51800.0	42000.0	25600.0	55600.0	14000.0
1996-05-01	34900.0	19700.0	14300.0	52500.0	42100.0	25500.0	56700.0	14300.0
1996-06-01	35200.0	20100.0	14500.0	53200.0	42100.0	25500.0	57900.0	14500.0

a. Adjusting the Review Period (changed month to April)

- The was a market recession during the period 2008 - 2011. The housing bubble burst may likely lead to some misleading data and assumptions.
- As a result, the analysis will consider data from April 2012 to April 2018.
- New DataFrame is named `**recent_time**.

```
recent_time = top_8_df['2012-04-01':]
recent_time.plot(figsize=(8, 5))
plt.xlabel('Year', fontsize=14)
plt.ylabel('Average Home Value', fontsize=14)
plt.title('Historical Average Home Values for Zipcodes of Interest')
plt.legend()
# plt.savefig('images/hist_avg_recent_time');
```

```
<matplotlib.legend.Legend at 0x25dee928e10>
```




```
recent_time.head()
```

ZipCode	19951	28762	30139	30317	31527	76234	78702	98851
Time								
2012-04-01	225200.0	111600.0	83600.0	152600.0	56400.0	161000.0	233200.0	83600.0
2012-05-01	230400.0	112400.0	85300.0	153900.0	56200.0	161200.0	232000.0	85300.0
2012-06-01	235800.0	113200.0	86600.0	155300.0	55800.0	160800.0	230200.0	86600.0
2012-07-01	238500.0	113500.0	87200.0	156400.0	55400.0	160600.0	229500.0	87200.0
2012-08-01	239500.0	113900.0	88000.0	157800.0	55300.0	161300.0	229900.0	88000.0

```
# Converting columns to str
recent_time.columns = recent_time.columns.astype(str)
zipcode_list =  top_8_df.columns.astype(str)

recent_time.columns
```

```
Index(['19951', '28762', '30139', '30317', '31527', '76234', '78702', '98851'], dtype='object', name='ZipCode')
```

```
# Define a dictionary to map zip codes to their corresponding names
zip_to_name = {
    '19951': 'Harbeson, DE',
    '28762': 'Old Fort, NC',
```

```
    '30139': 'Fairmount, GA',
    '30317': 'Atlanta, GA',
    '31527': 'Jekyll Island, GA',
    '76234': 'Decatur, TX',
    '78702': 'Austin, TX',
    '98851': 'Soap Lake, WA'
}

# Map only the columns that haven't been renamed yet
recent_time.columns = [zip_to_name[col] if col in zip_to_name else col for col in recent_time.columns]

# Print the updated DataFrame
recent_time.head()
```

Time	Harbeson, DE	Old Fort, NC	Fairmount, GA	Atlanta, GA	Jekyll Island, GA	Decatur, TX	Austin, TX	Soap Lake, WA
2012-04-01	225200.0	111600.0	83600.0	152600.0	56400.0	161000.0	233200.0	83600.0
2012-05-01	230400.0	112400.0	85300.0	153900.0	56200.0	161200.0	232000.0	85300.0
2012-06-01	235800.0	113200.0	86600.0	155300.0	55800.0	160800.0	230200.0	86600.0
2012-07-01	238500.0	113500.0	87200.0	156400.0	55400.0	160600.0	229500.0	87200.0
2012-08-01	239500.0	113900.0	88000.0	157800.0	55300.0	161300.0	229900.0	88000.0

```
recent_time.columns
```

```
Index(['Harbeson, DE', 'Old Fort, NC', 'Fairmount, GA', 'Atlanta, GA',
       'Jekyll Island, GA', 'Decatur, TX', 'Austin, TX', 'Soap Lake, WA'],
      dtype='object')
```

b. Checking for Seasonality

- The following functions check for and remove seasonality using the `seasonal_decompose` function.
- Twelve (12) periods are specified to represent a year, the duration of a season in the data.
- After extracting the **trend**, **seasonality** and **residuals**, it plots each of these components in separate subplots.
- It is important to remove seasonality and trend because If they are part of the time series, there will be effects in the forecast value.

```
def check_seasonality(column_name, column_series):
    # Decompose the time series
    decomposition = seasonal_decompose(column_series, model='additive', period=12)

    # Check for seasonality by observing the seasonal component
    if np.std(decomposition.seasonal) > 0.5:
        print(f"Seasonality: {column_name} is Seasonal")
    else:
        print(f"Seasonality: {column_name} is Not Seasonal")

   # Plot the decomposed components
    plt.figure(figsize=(10, 6))
    plt.subplot(411)
    plt.plot(column_series, label='Original')
    plt.legend(loc='upper left')
    plt.title(f'{column_name} - Original')
    plt.subplot(412)
    plt.plot(decomposition.trend, label='Trend')
    plt.legend(loc='upper left')
    plt.title(f'{column_name} - Trend')
    plt.subplot(413)
    plt.plot(decomposition.seasonal, label='Seasonal')
    plt.legend(loc='upper left')
    plt.title(f'{column_name} - Seasonal')
    plt.subplot(414)
    plt.plot(decomposition.resid, label='Residual')
    plt.legend(loc='upper left')
    plt.title(f'{column_name} - Residual')
    plt.tight_layout()
    plt.show()
```

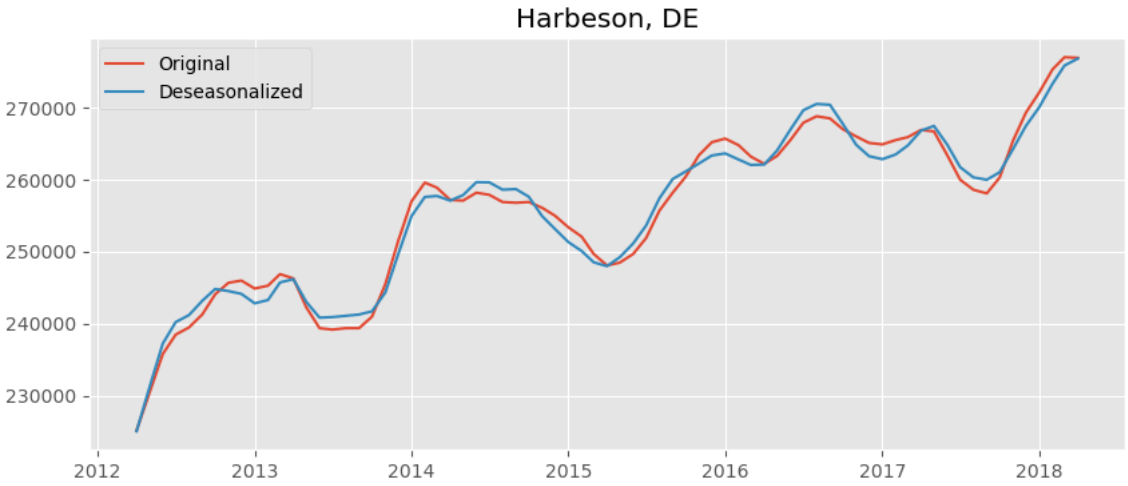
```
# Checking seasonality in each of the columns
for column in recent_time.columns:
    check_seasonality(column, recent_time[column])
```

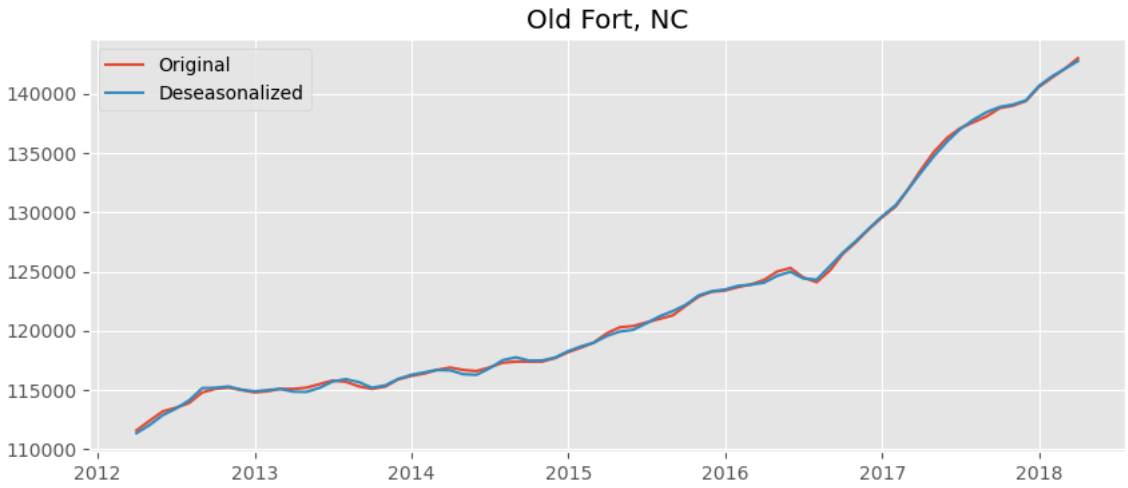
All zip codes show seasonality. This needs to be removed.

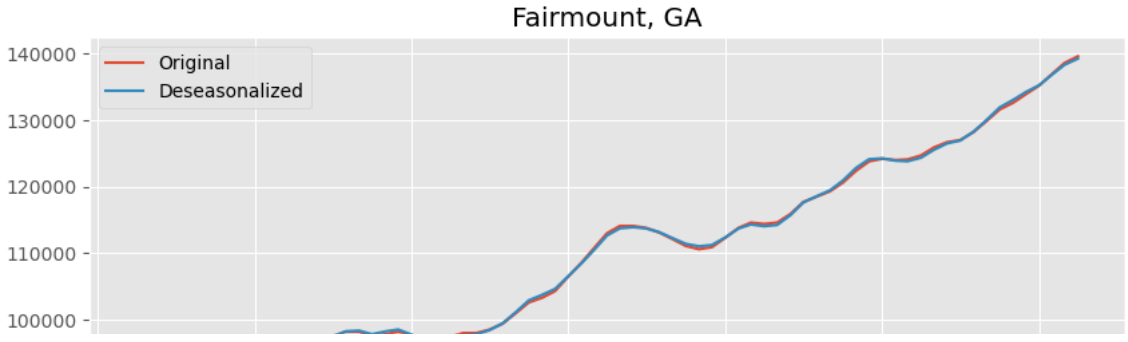
- The following will remove seasonality and compare a few zip codes using line graphs.

```
def remove_seasonality(column_series):
    decomposition = seasonal_decompose(column_series, model='additive', period=12)
    deseasonalized_series = column_series - decomposition.seasonal
    return deseasonalized_series

# Choose a few columns for comparison
columns_to_compare = ['Harbeson, DE', 'Old Fort, NC', 'Fairmount, GA']

# Create a new DataFrame to store deseasonalized data
deseasonalized_data = pd.DataFrame()

# Loop through each column in recent_time, remove seasonality, and store in the new DataFrame
for column in columns_to_compare:
    deseasonalized_data[column] = remove_seasonality(recent_time[column])

# Plot original and deseasonalized time series
for column in columns_to_compare:
    plt.figure(figsize=(10, 4))
    plt.plot(recent_time[column], label='Original')
    plt.plot(deseasonalized_data[column], label='Deseasonalized')
    plt.title(column)
```

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

    # Compare standard deviation of seasonal components
    for column in columns_to_compare:
        original_decomposition = seasonal_decompose(recent_time[column], model='additive', period=12)
        deseasonalized_decomposition = seasonal_decompose(deseasonalized_data[column], model='additive', period=12)

        original_std = np.std(original_decomposition.seasonal)
        deseasonalized_std = np.std(deseasonalized_decomposition.seasonal)

        print(f"{column}: Original Seasonal Std = {original_std}, Deseasonalized Seasonal Std = {deseasonalized_std}")
```







5.1. Checking for Stationarity

- The Augmented Dickey-Fuller (ADF) test will be used to determine the stationarity of the time series.
- The ADF test helps us assess whether a time series is stationary by comparing it to the null hypothesis that it has a unit root (meaning it's non-stationary).
- The test involves estimating the model's coefficients, calculating a test statistic, and comparing it to critical values to determine whether the null hypothesis can be rejected.
- The p-value resulting from the test indicates the strength of evidence against the null hypothesis.
    - If the p-value is less than or equal to the significance level of 0.05, there's evidence to reject the null hypothesis, suggesting the series is stationary.

- If the p-value is greater than the significance level of 0.05, there's weak evidence to reject the null hypothesis, indicating the series is likely non-stationary.


```
def check_stationarity(data):
    def adf_test(timeseries):
        result = adfuller(timeseries, autolag='AIC')
        return result[1]  # Returning p-value

    print("Stationarity Check Results:")
    print()
    for column in data.columns:
        p_value = adf_test(data[column])
        stationary_status = "Stationary" if p_value <= 0.05 else "Not Stationary"
        print(f"{column}: ADF Test P-Value = {p_value:.4f} ({stationary_status})")
```



```
check_stationarity(recent_time)
```

```
Stationarity Check Results:

Harbeson, DE: ADF Test P-Value = 0.7320 (Not Stationary)
Old Fort, NC: ADF Test P-Value = 1.0000 (Not Stationary)
Fairmount, GA: ADF Test P-Value = 0.9979 (Not Stationary)
Atlanta, GA: ADF Test P-Value = 0.9291 (Not Stationary)
Jekyll Island, GA: ADF Test P-Value = 0.7651 (Not Stationary)
Decatur, TX: ADF Test P-Value = 0.9970 (Not Stationary)
Austin, TX: ADF Test P-Value = 0.9385 (Not Stationary)
Soap Lake, WA: ADF Test P-Value = 0.5802 (Not Stationary)
```

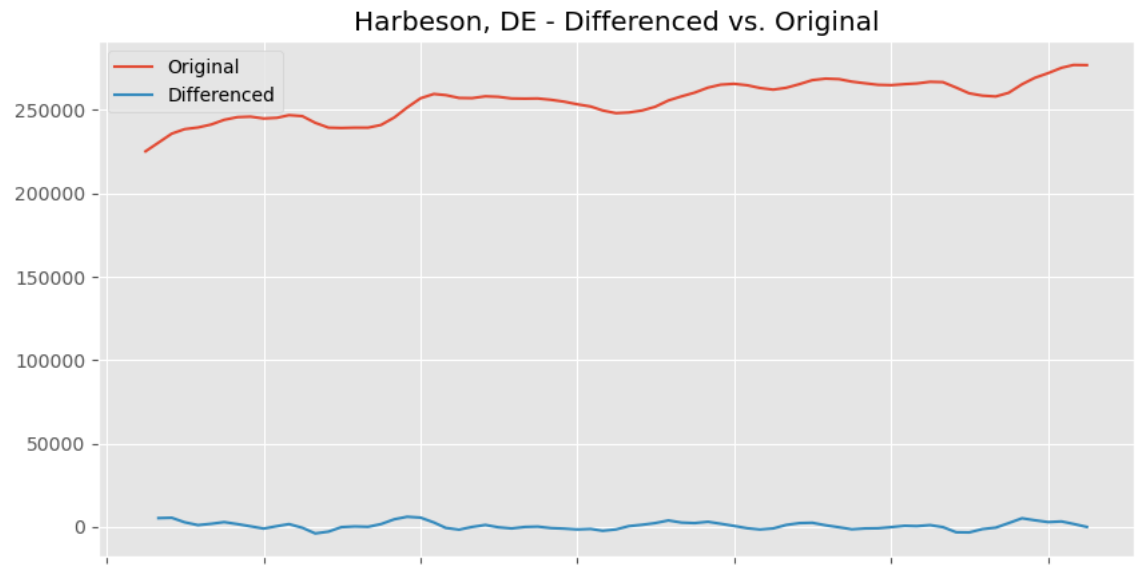

Remove Stationarity Through Differencing

- The stationarity check results show that the time series is not stationary.
- Next, differencing will be applied to remove stationarity and make the model-building process accurate and reliable.
- The data's statistical properties will remain consistent, allowing for meaningful insights and accurate predictions.

```
def remove_stationarity_through_differencing(column_series):
    # Apply differencing to make the data more stationary
    differenced_series = column_series.diff().dropna()

    # Plot original and differenced time series
    plt.figure(figsize=(10, 5))
    plt.plot(column_series, label='Original')
    plt.plot(differenced_series, label='Differenced')
    plt.legend()
    plt.title(f"{column_series.name} - Differenced vs. Original")
    plt.show()
```



```
for column in recent_time.columns:
    remove_stationarity_through_differencing(recent_time[column])
```




```
differenced_series = recent_time.diff().dropna()

differenced_series
```

Time	Harbeson, DE	Old Fort, NC	Fairmount, GA	Atlanta, GA	Jekyll Island, GA	Decatur, TX	Austin, TX	Soap Lake, WA
2012-05-01	5200.0	800.0	1700.0	1300.0	-200.0	200.0	-1200.0	1700.0
2012-06-01	5400.0	800.0	1300.0	1400.0	-400.0	-400.0	-1800.0	1300.0
2012-07-01	2700.0	300.0	600.0	1100.0	-400.0	-200.0	-700.0	600.0
2012-08-01	1000.0	400.0	800.0	1400.0	-100.0	700.0	400.0	800.0
2012-09-01	1800.0	900.0	700.0	1500.0	300.0	800.0	700.0	700.0
...	...	...	...	...	...	...	...	...
2017-12-01	3900.0	400.0	1300.0	6800.0	5500.0	2600.0	4600.0	-1000.0
2018-01-01	2800.0	1200.0	1300.0	5700.0	7200.0	1700.0	4900.0	-300.0
2018-02-01	3200.0	800.0	1800.0	3500.0	600.0	1500.0	3700.0	2300.0
2018-03-01	1700.0	700.0	1600.0	2000.0	11200.0	1800.0	3500.0	4800.0
2018-04-01	-100.0	900.0	1000.0	1200.0	19300.0	1900.0	2500.0	3100.0

72 rows × 8 columns

5.2 Autocorrelation and Partial Correlation of Differenced Series

- Autocorrelation measures the linear relationship between lagged values of a time series.
- Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF) will be used to identify the appropriate orders of autoregressive (AR) and moving average (MA) terms for modeling.
- The tools will guide in understanding the underlying structure of the time series, guiding the selection of model parameters, and ensuring that the model captures the relevant patterns in the data.
- They will help in building accurate and effective models.

```
def plot_acf_pacf(series_diff, column_name):
    plt.figure(figsize=(12, 8))

    # acf
```

```
    plt.subplot(2, 1, 1)
    plot_acf(series_diff, ax=plt.gca(), lags=34)
    plt.title(f"ACF for {column_name} Differenced Series")

    # pacf
    plt.subplot(2, 1, 2)
    plot_pacf(series_diff, ax=plt.gca(), lags=34)
    plt.title(f"PACF for {column_name} Differenced Series")

    plt.tight_layout()
    plt.show()

# Call the function for all columns in recent_time
for column in recent_time.columns:
    differenced_series = recent_time[column].diff().dropna()
    plot_acf_pacf(differenced_series, column)
```

Some conclusions on the above plots:

1. The time series is mostly seasonal.
2. The series is stationary given the decaying to zero as visualized. The differencing was a significant process.
3. The ACF plots show that there is trend since the ACF decreases as lags increase.

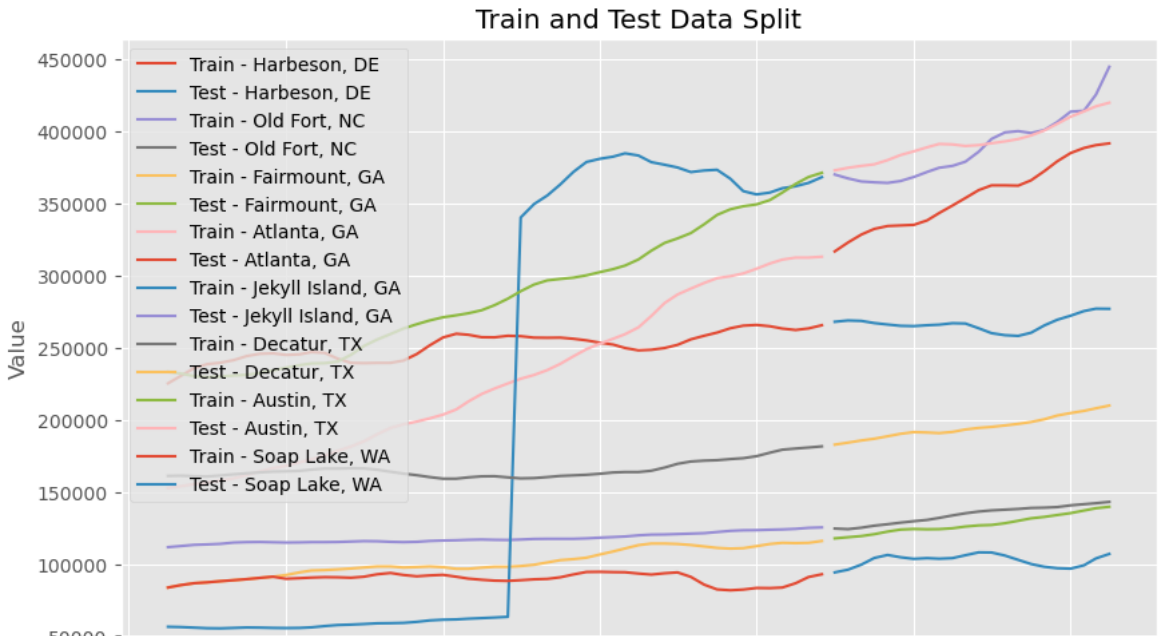
5.3. Performing Train Test Split

- The data was split into train and test sets.

- The train set was 70% of the whole dataset and the test set was 30%.

```
# define train size
train_size = 0.7


split_index = round(len(recent_time) * train_size)
train = recent_time[:split_index]
test = recent_time[split_index:]
```

```
fig, ax = plt.subplots(figsize=(10, 6))

for column in train.columns:
    plt.plot(train[column], label=f'Train - {column}')
    plt.plot(test[column], label=f'Test - {column}')

plt.xlabel('Year')
plt.ylabel('Value')
plt.title('Train and Test Data Split')
plt.legend()

plt.show()
```



6.0 Modelling

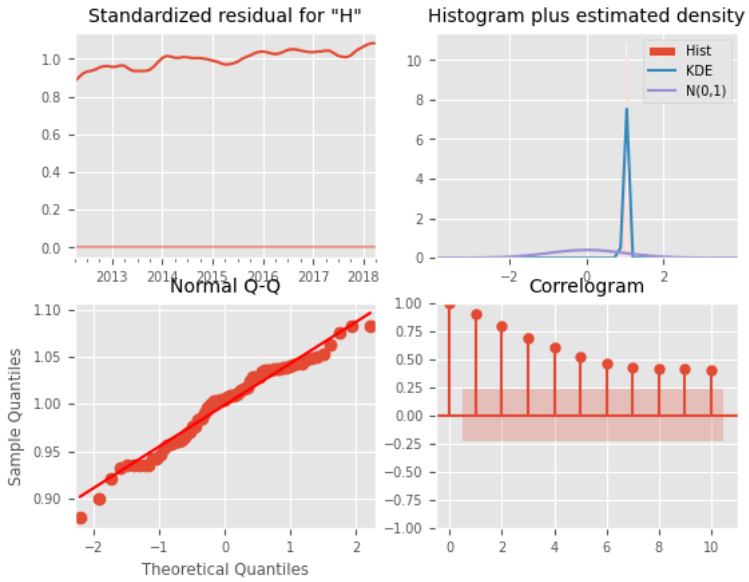
a. Baseline Model: Using Approximate SARIMAX Parameters

- Since our data has seasonality we use SARIMAX model.
- A SARIMAX model will be fitted for a random zip code as a baseline with params (0,0,0).

```
import random

# Randomly select a zip code
random_zipcode = random.choice(recent_time.columns)

# Calculate the train set size based on 70% of the data
train_set_size = int(len(recent_time) * train_size)

# Get the train set for the selected zip code
train_set = recent_time[random_zipcode].iloc[:train_set_size]

print(f"Randomly selected zip code: {random_zipcode}")
print(f"Train set size: {len(train_set)}")
print("\nTrain set:")
# print(train_set)

models = {}
for column in recent_time.columns:
    if column == random_zipcode:
        model = SARIMAX(recent_time[column], order=(0, 0, 0), seasonal_order=(0,0,0,0))
        models[column] = model.fit()
```

```
# Set font size for smaller text
plt.rcParams.update({'font.size': 7});

# Display summary and diagnostics for the selected model
for column, model in models.items():
    print(f"Summary for {column}:")
    print(model.summary())
    print("\nDiagnostics:")


    plt.figure(figsize=(10, 8))

    model.plot_diagnostics()
    plt.show()
    print("\n\n\n")
```

```
 Randomly selected zip code: Harbeson, DE
 Train set size: 51

 Train set:
 Summary for Harbeson, DE:
                               SARIMAX Results
 ==============================================================================
 Dep. Variable:           Harbeson, DE   No. Observations:                   73
 Model:                        SARIMAX   Log Likelihood               -1012.630
 Date:                 Wed, 30 Aug 2023   AIC                           2027.260
 Time:                         13:52:19   BIC                           2029.551
 Sample:                     04-01-2012   HQIC                          2028.173
                           - 04-01-2018
 Covariance Type:                  opg
 ==============================================================================
                  coef    std err          z      P>|z|      [0.025      0.975]
 ------------------------------------------------------------------------------
 sigma2      6.551e+10   1.76e+11      0.371      0.711    -2.8e+11    4.11e+11
 ===================================================================================
 Ljung-Box (L1) (Q):                  62.18   Jarque-Bera (JB):                 2.55
 Prob(Q):                              0.00   Prob(JB):                         0.28
 Heteroskedasticity (H):               1.20   Skew:                            -0.41
 Prob(H) (two-sided):                  0.66   Kurtosis:                         2.58
 ===================================================================================

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

 Diagnostics:
 <Figure size 1000x800 with 0 Axes>
```



b. Pipeline - SARIMA Models for Each Zip Code

A pipeline was then built to take care of:

1. Getting the best SARIMAX parameters
    - Auto-arima function was utilised to automate the process of obtaining (p, d, q) and (P, D, Q, s).
2. Fitting the SARIMAX model on the train set and diplaying the summary statistics
3. Predicting on the train set
4. Plotting original against forecasted values and displaying the confidence intervals to understand the uncertainty.

5. Calculating the Akaike Information Criterion (AIC) values that test the goodness of fit.
    - A model that fits the data very well while using lots of features will be assigned a larger AIC score than a model that uses fewer features to achieve the same goodness-of-fit.
6. Calculating RMSE to measure the model's accuracy 7.
7. Printing the results at the end of the pipeline.
8. Creating a dataframe containing RMSE and AIC for model diagnostics.

Eight (8) pipelines, one for each of the top zipcodes were displayed.

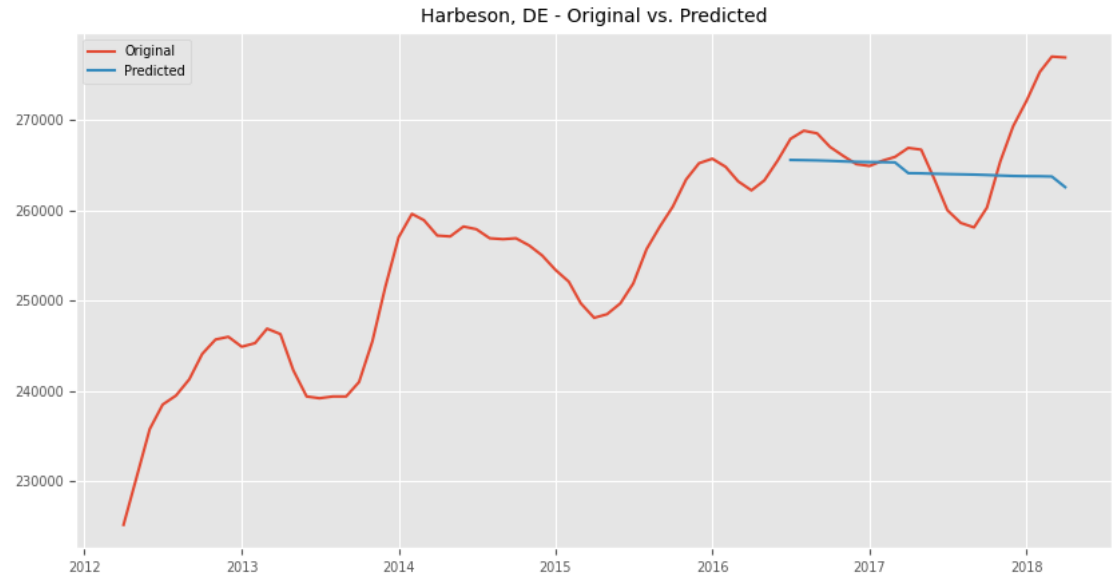
```
# setting up the pipeline's functions
def find_best_sarimax_params(series):
    model = auto_arima(series, seasonal=True, m=12, stepwise=True, trace=False) # 12 months
    p, d, q = model.order
    P, D, Q, s = model.seasonal_order
    return (p, d, q), (P, D, Q, s)

def fit_sarimax_and_get_summary(series, order, seasonal_order):
    try:
        model = sm.tsa.SARIMAX(series, order=order, seasonal_order=seasonal_order)
        results = model.fit()
        return results, results.summary()
    except:
        return None, f"Failed to fit SARIMAX for {series.name} (p, d, q): {order}"

def predict_sarimax(fitted_model, start, end):
    predicted = fitted_model.get_prediction(start=start, end=end)
    return predicted.predicted_mean


def calculate_rmse(actual, predicted):
    mse = mean_squared_error(actual, predicted)
    rmse = np.sqrt(mse)
    return rmse

def calculate_metrics(fitted_model, actual, predicted):
    rmse = calculate_rmse(actual, predicted)
    aic = fitted_model.aic
    metrics_df = pd.DataFrame({'RMSE': [rmse], 'AIC': [aic]}, index=[actual.name])
    return metrics_df


# Empty DataFrame to store all RMSE and AIC for zip codes
metrics_dfs = []
```

Harbeson, DE

```
column = 'Harbeson, DE'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)

# Fit SARIMAX model on test set and get summary
fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
if fitted_model is None:
    print("Failed to fit model:")
    print(model_summary)
    print("=" * 40)
else:
    # predicting
    predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

    # AIC calculation
    aic = fitted_model.aic

    # Plotting original against predicted values
    plt.figure(figsize=(10, 5))
    plt.plot(series.index, series.values, label='Original')
    plt.plot(predicted.index, predicted.values, label='Predicted')
    plt.legend()
    plt.title(f"{column} - Original vs. Predicted")
    plt.show()
```

```
# RMSE calculation
rmse = calculate_rmse(series[split_index:], predicted)

# results
print("Model Summary:")
print(model_summary)
print()
print(f"AIC Value: {aic}")
print(f"RMSE: {rmse}")
print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
print()

# metrics dataframe
metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
metrics_dfs.append(metrics_df)
```

```
PIPELINE - HARBESON, DE
```




```
Model Summary:
                                SARIMAX Results
==========================================================================================
Dep. Variable:                      Harbeson, DE   No. Observations:                   51
Model:             SARIMAX(1, 1, 0)x(1, 0, [1], 12)   Log Likelihood                -455.563
Date:                              Wed, 30 Aug 2023   AIC                            919.127
Time:                                      13:52:25   BIC                            926.775
Sample:                                  04-01-2012   HQIC                           922.039
                                       - 06-01-2016
Covariance Type:                               opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.0431      0.009      4.952      0.000       0.026       0.060
ar.S.L12       0.9858      0.050     19.613      0.000       0.887       1.084
ma.S.L12      -1.0000      0.145     -6.879      0.000      -1.285      -0.715
sigma2      3.739e+06   3.89e-08   9.62e+13      0.000    3.74e+06    3.74e+06
===================================================================================
Ljung-Box (L1) (Q):                  16.79   Jarque-Bera (JB):                 1.33
Prob(Q):                              0.00   Prob(JB):                         0.51
Heteroskedasticity (H):               0.72   Skew:                             0.39
Prob(H) (two-sided):                  0.51   Kurtosis:                         3.20
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 5.96e+29. Standard errors may be unstable.

AIC Value: 919.1266565735815
RMSE: 5848.98969932721
```

Old Fort, NC

```
column = 'Old Fort, NC'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)
```

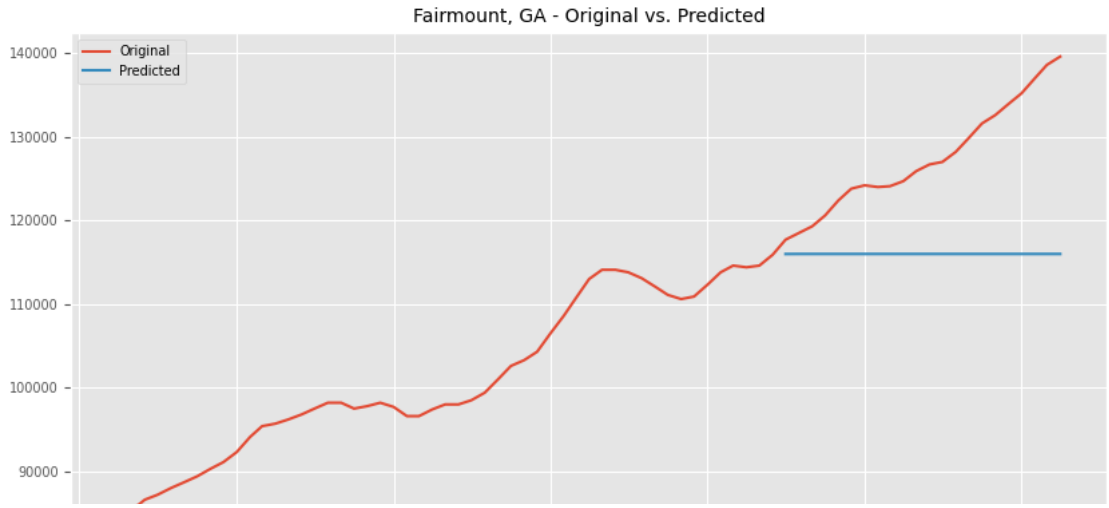
```
    # Fit SARIMAX model on test set and get summary
    fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
    if fitted_model is None:
        print("Failed to fit model:")
        print(model_summary)
        print("=" * 40)
    else:
        # predicting
        predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

        # AIC calculation
        aic = fitted_model.aic

        # Plotting original against predicted values
        plt.figure(figsize=(10, 5))
        plt.plot(series.index, series.values, label='Original')
        plt.plot(predicted.index, predicted.values, label='Predicted')
        plt.legend()
        plt.title(f"{column} - Original vs. Predicted")
        plt.show()

        # RMSE calculation
        rmse = calculate_rmse(series[split_index:], predicted)

        # results
        print("Model Summary:")
        print(model_summary)
        print()
        print(f"AIC Value: {aic}")
        print(f"RMSE: {rmse}")
        print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
        print()

        # metrics dataframe
        metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
        metrics_dfs.append(metrics_df)
```

```
PIPELINE - OLD FORT, NC
```

Fairmount, GA

```
column = 'Fairmount, GA'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)

# Fit SARIMAX model on test set and get summary
fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
if fitted_model is None:
    print("Failed to fit model:")
    print(model_summary)
    print("=" * 40)
else:
    # predicting
    predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

    # AIC calculation
    aic = fitted_model.aic

    # Plotting original against predicted values
    plt.figure(figsize=(10, 5))
    plt.plot(series.index, series.values, label='Original')
    plt.plot(predicted.index, predicted.values, label='Predicted')
    plt.legend()
    plt.title(f"{column} - Original vs. Predicted")
    plt.show()

    # RMSE calculation
    rmse = calculate_rmse(series[split_index:], predicted)

    # results
    print("Model Summary:")
    print(model_summary)
    print()
    print(f"AIC Value: {aic}")
    print(f"RMSE: {rmse}")
    print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
    print()

    # metrics dataframe
    metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
    metrics_dfs.append(metrics_df)
```

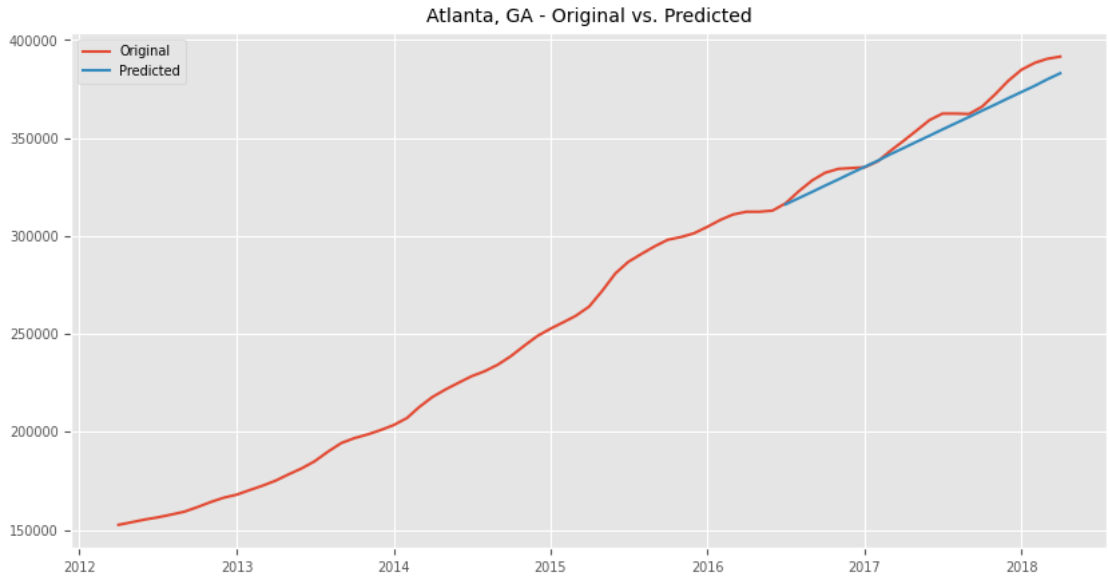
```
PIPELINE - FAIRMOUNT, GA
```



Atlanta, GA

```
column = 'Atlanta, GA'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)

# Fit SARIMAX model on test set and get summary
fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
if fitted_model is None:
    print("Failed to fit model:")
    print(model_summary)
    print("=" * 40)
else:
    # predicting
    predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

    # AIC calculation
    aic = fitted_model.aic

    # Plotting original against predicted values
    plt.figure(figsize=(10, 5))
    plt.plot(series.index, series.values, label='Original')
    plt.plot(predicted.index, predicted.values, label='Predicted')
    plt.legend()
    plt.title(f"{column} - Original vs. Predicted")
    plt.show()

    # RMSE calculation
    rmse = calculate_rmse(series[split_index:], predicted)

    # results
    print("Model Summary:")
    print(model_summary)
    print()
    print(f"AIC Value: {aic}")
    print(f"RMSE: {rmse}")
    print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
    print()

    # metrics dataframe
    metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
    metrics_dfs.append(metrics_df)
```

```
PIPELINE - ATLANTA, GA
```

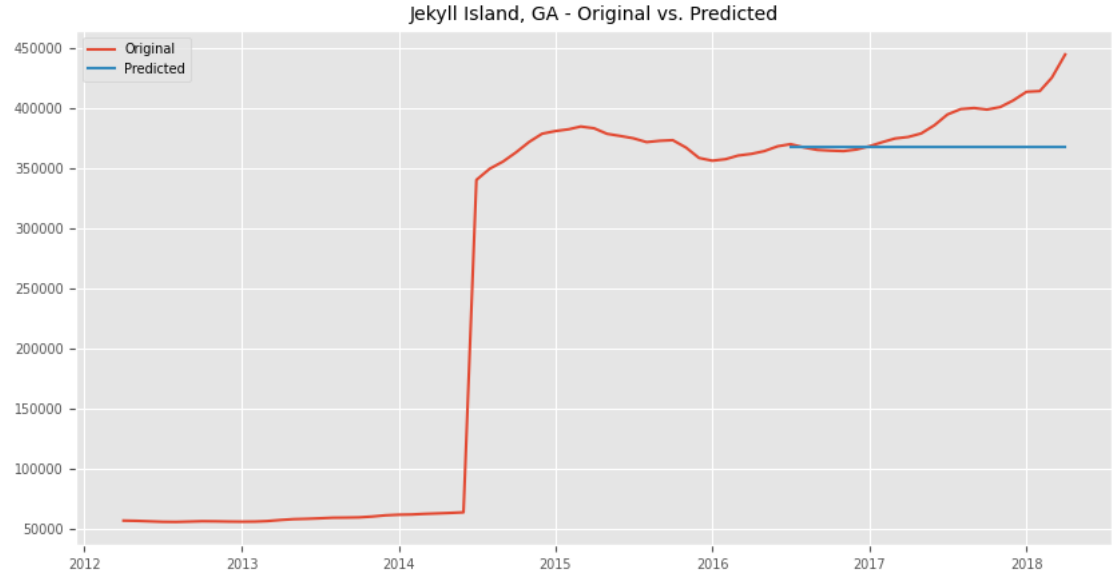



```
Model Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:            Atlanta, GA   No. Observations:                   51
Model:               SARIMAX(1, 1, 1)   Log Likelihood                -443.193
Date:                Wed, 30 Aug 2023   AIC                            892.385
Time:                        13:52:33   BIC                            898.122
Sample:                    04-01-2012   HQIC                           894.570
                         - 06-01-2016
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          1.0000      0.000   9464.455      0.000       1.000       1.000
ma.L1         -0.9998      0.016    -61.422      0.000      -1.032      -0.968
sigma2      2.963e+06   1.23e-10   2.41e+16      0.000    2.96e+06    2.96e+06
===================================================================================
Ljung-Box (L1) (Q):                  30.28   Jarque-Bera (JB):                12.15
Prob(Q):                              0.00   Prob(JB):                         0.00
```

Jekyll Island, GA

```
column = 'Jekyll Island, GA'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)

# Fit SARIMAX model on test set and get summary
fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
if fitted_model is None:
    print("Failed to fit model:")
    print(model_summary)
    print("=" * 40)
else:
    # predicting
    predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

    # AIC calculation
    aic = fitted_model.aic

    # Plotting original against predicted values
    plt.figure(figsize=(10, 5))
    plt.plot(series.index, series.values, label='Original')
    plt.plot(predicted.index, predicted.values, label='Predicted')
    plt.legend()
    plt.title(f"{column} - Original vs. Predicted")
    plt.show()

    # RMSE calculation
    rmse = calculate_rmse(series[split_index:], predicted)

    # results
```

```
    print("Model Summary:")
    print(model_summary)
    print()
    print(f"AIC Value: {aic}")
    print(f"RMSE: {rmse}")
    print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
    print()

    # metrics dataframe
    metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
    metrics_dfs.append(metrics_df)

     PIPELINE - JEKYLL ISLAND, GA
```



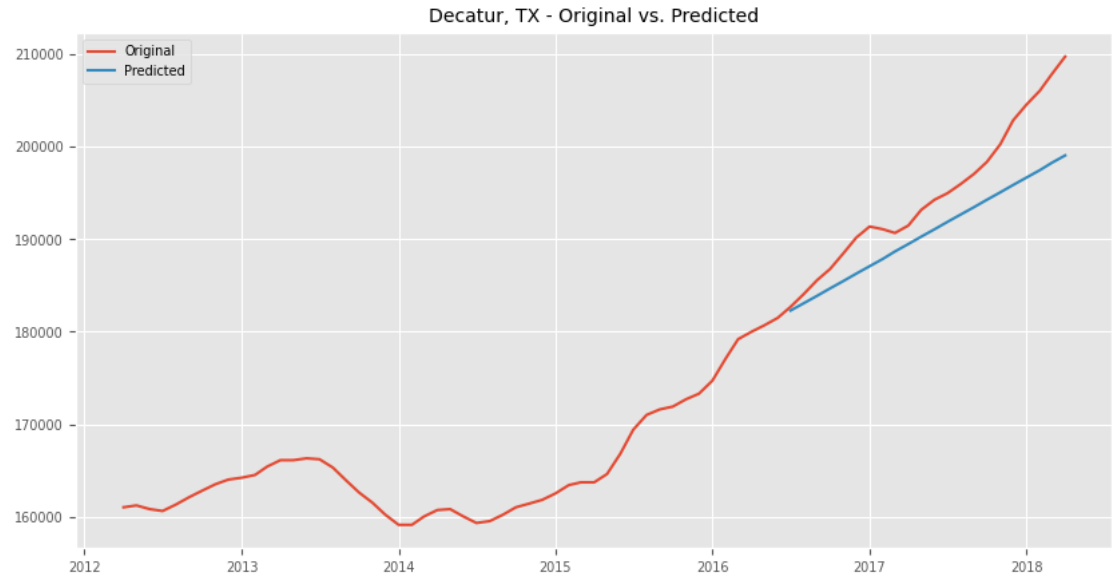

```
    Model Summary:
                                SARIMAX Results
    ==============================================================================
    Dep. Variable:      Jekyll Island, GA   No. Observations:                   51
    Model:               SARIMAX(0, 1, 0)   Log Likelihood                -599.889
    Date:                Wed, 30 Aug 2023   AIC                           1201.779
    Time:                        13:52:34   BIC                           1203.691
    Sample:                  04-01-2012   HQIC                          1202.507
                           - 06-01-2016
    Covariance Type:                 opg
    ==============================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
    ------------------------------------------------------------------------------
    sigma2      1.514e+09   6.04e+07     25.072      0.000     1.4e+09    1.63e+09
    ===================================================================================
    Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):              4490.49
    Prob(Q):                              0.91   Prob(JB):                         0.00
    Heteroskedasticity (H):              77.66   Skew:                             6.79
    Prob(H) (two-sided):                  0.00   Kurtosis:                        47.40
    ===================================================================================

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

    AIC Value: 1201.7785431051009
    RMSE: 30178.71766659412
```

Decatur, TX

```
column = 'Decatur, TX'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)

# Fit SARIMAX model on test set and get summary
fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
if fitted_model is None:
    print("Failed to fit model:")
    print(model_summary)
    print("=" * 40)
else:
```

```
# predicting
predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

# AIC calculation
aic = fitted_model.aic

# Plotting original against predicted values
plt.figure(figsize=(10, 5))
plt.plot(series.index, series.values, label='Original')
plt.plot(predicted.index, predicted.values, label='Predicted')
plt.legend()
plt.title(f"{column} - Original vs. Predicted")
plt.show()

# RMSE calculation
rmse = calculate_rmse(series[split_index:], predicted)

# results
print("Model Summary:")
print(model_summary)
print()
print(f"AIC Value: {aic}")
print(f"RMSE: {rmse}")
print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
print()

# metrics dataframe
metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
metrics_dfs.append(metrics_df)
```

```
PIPELINE - DECATUR, TX
```

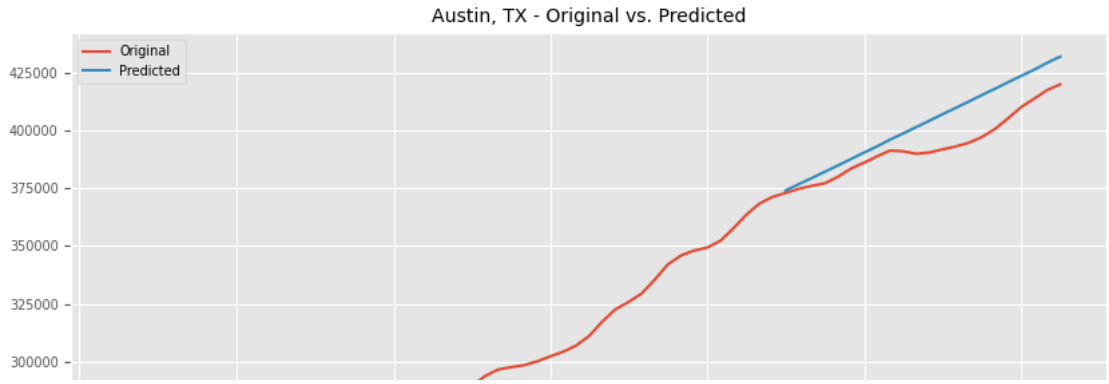



```
Model Summary:
                               SARIMAX Results
==============================================================================
Dep. Variable:            Decatur, TX   No. Observations:                   51
Model:               SARIMAX(0, 2, 0)   Log Likelihood                -381.986
Date:                Wed, 30 Aug 2023   AIC                            765.971
Time:                        13:52:36   BIC                            767.863
Sample:                    04-01-2012   HQIC                           766.689
                         - 06-01-2016
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
sigma2      3.453e+05   7.75e+04      4.458      0.000    1.94e+05    4.97e+05
===================================================================================
Ljung-Box (L1) (Q):                   6.02   Jarque-Bera (JB):                 0.42
Prob(Q):                              0.01   Prob(JB):                         0.81
Heteroskedasticity (H):               2.72   Skew:                             0.11
Prob(H) (two-sided):                  0.05   Kurtosis:                         2.61
===================================================================================

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

AIC Value: 765.9710351361973
RMSE: 5033.43367349313
```

Austin, TX

```
column = 'Austin, TX'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)

# Fit SARIMAX model on test set and get summary
fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
if fitted_model is None:
    print("Failed to fit model:")
    print(model_summary)
    print("=" * 40)
else:
    # predicting
    predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

    # AIC calculation
    aic = fitted_model.aic

    # Plotting original against predicted values
    plt.figure(figsize=(10, 5))
    plt.plot(series.index, series.values, label='Original')
    plt.plot(predicted.index, predicted.values, label='Predicted')
    plt.legend()
    plt.title(f"{column} - Original vs. Predicted")
    plt.show()

    # RMSE calculation
    rmse = calculate_rmse(series[split_index:], predicted)

    # results
    print("Model Summary:")
    print(model_summary)
    print()
    print(f"AIC Value: {aic}")
    print(f"RMSE: {rmse}")
    print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
    print()

    # metrics dataframe
    metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
    metrics_dfs.append(metrics_df)
```

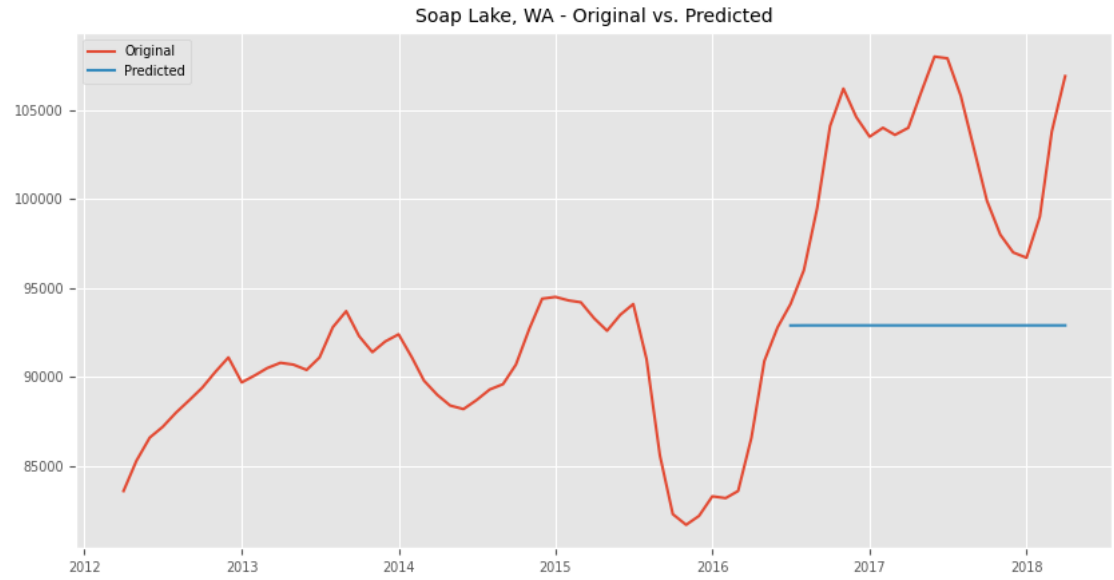
```
PIPELINE - AUSTIN, TX
```



Soap Lake, WA

```
column = 'Soap Lake, WA'
series = recent_time[column]
train_set_column = train[column]
test_set_column = test[column]

print(f"PIPELINE - {column.upper()}")
print()
# best SARIMAX parameters
(p, d, q), (P, D, Q, s) = find_best_sarimax_params(series)

# Fit SARIMAX model on test set and get summary
fitted_model, model_summary = fit_sarimax_and_get_summary(train_set_column, (p, d, q), (P, D, Q, s))
if fitted_model is None:
    print("Failed to fit model:")
    print(model_summary)
    print("=" * 40)
else:
    # predicting
    predicted = predict_sarimax(fitted_model, start=test_set_column.index[0], end=test_set_column.index[-1])

    # AIC calculation
    aic = fitted_model.aic

    # Plotting original against predicted values
    plt.figure(figsize=(10, 5))
    plt.plot(series.index, series.values, label='Original')
    plt.plot(predicted.index, predicted.values, label='Predicted')
    plt.legend()
    plt.title(f"{column} - Original vs. Predicted")
    plt.show()

    # RMSE calculation
    rmse = calculate_rmse(series[split_index:], predicted)

    # results
    print("Model Summary:")
    print(model_summary)
    print()
    print(f"AIC Value: {aic}")
    print(f"RMSE: {rmse}")
    print(f"SARIMAX Params: p={p}, d={d}, q={q}, P={P}, D={D}, Q={Q}, s={s}")
    print()

    # metrics dataframe
    metrics_df = calculate_metrics(fitted_model, series[split_index:], predicted)
    metrics_dfs.append(metrics_df)
```

```
PIPELINE - SOAP LAKE, WA
```




```
Model Summary:
                               SARIMAX Results
==========================================================================================
Dep. Variable:                 Soap Lake, WA   No. Observations:                   51
Model:                SARIMAX(1, 1, 0)   Log Likelihood                -435.542
Date:                Wed, 30 Aug 2023   AIC                            875.084
Time:                        13:52:44   BIC                            878.908
Sample:                    04-01-2012   HQIC                           876.541
                         - 06-01-2016
Covariance Type:                  opg
==========================================================================================
```

c. SARIMA Model Diagnostics

The results below indicate that as per SARIMA model, the top five zip codes with the lowest RMSE values are

1. Decatur, TX - 5033.433673
2. Harbeson, DE - 5753.477978
3. Atlanta, GA - 6154.167274
4. Old Fort, NC - 6779.883346
5. Soap Lake, WA - 10254.866297

```
final_metrics_df = pd.concat(metrics_dfs)
final_metrics_df = final_metrics_df.sort_values(by='RMSE')
print(final_metrics_df)
```

```
                           RMSE          AIC
Decatur, TX         5033.433673   765.971035
Harbeson, DE        5848.989699   919.126657
Atlanta, GA         6356.581960   892.385470
Old Fort, NC        6779.883346   692.602794
Soap Lake, WA      10254.866268   875.084354
Austin, TX         11372.325421  1006.157434
Fairmount, GA      13161.083926   835.732403
Jekyll Island, GA  30178.717667  1201.778543
```

d. Facebook Prophet

- The first step was to prepare the data for the model.

```
recent_time.head()
```

Time	Harbeson, DE	Old Fort, NC	Fairmount, GA	Atlanta, GA	Jekyll Island, GA	Decatur, TX	Austin, TX	Soap Lake, WA
2012-04-01	225200.0	111600.0	83600.0	152600.0	56400.0	161000.0	233200.0	83600.0
2012-05-01	230400.0	112400.0	85300.0	153900.0	56200.0	161200.0	232000.0	85300.0
2012-06-01	235800.0	113200.0	86600.0	155300.0	55800.0	160800.0	230200.0	86600.0
2012-07-01	238500.0	113500.0	87200.0	156400.0	55400.0	160600.0	229500.0	87200.0
2012-08-01	239500.0	113900.0	88000.0	157800.0	55300.0	161300.0	229900.0	88000.0

e. Facebook Prophet Pipeline

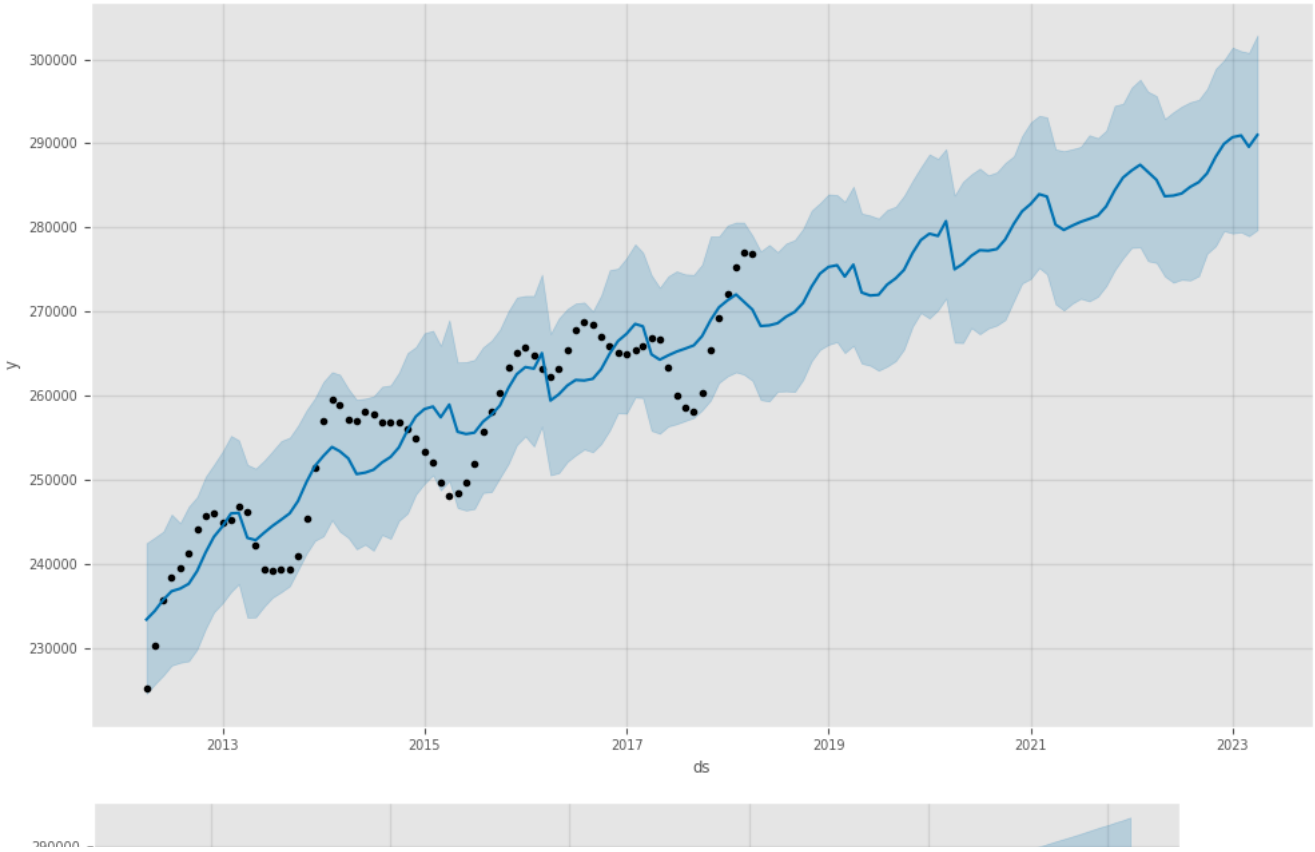
- The pipeline was built to create a DataFrame from a copy of the working DataFrame, `recent_time`. Here, the `ds` column was defined
- It then calculates some metrics that will be used as part of diagnostics.
- The `prophet_model` function changes the zip code column to y, creates a subset, resets index, fits, forecasts and predicts using Facebook Prophet library.

```
# Create a new DataFrame and rename the column
prophet_df = recent_time.copy()
prophet_df = pd.DataFrame(prophet_df).reset_index()
prophet_df = prophet_df.rename(columns={'Time': 'ds'})

def calculate_rmse(actual, predicted):
    return mean_squared_error(actual, predicted, squared=False)

def calculate_metrics(actual, predicted):
    rmse = calculate_rmse(actual, predicted)
    metrics_df = pd.DataFrame({'RMSE': [rmse]}, index=[actual.name])
    return metrics_df

def prophet_model(column_name):
    new_subset = prophet_df.rename(columns={column_name: 'y'})

    subset_columns = ['ds', 'y']
    prophet_subset = new_subset[subset_columns]

    prophet_subset_reset = prophet_subset.reset_index(drop=True)

    model = Prophet(interval_width=0.95)
    fitted_model = model.fit(prophet_subset_reset)

    future_dates = model.make_future_dataframe(periods=60, freq='MS')
    forecast = model.predict(future_dates)

    # Print and plot the results
    print(f"Results for {column_name}:")
    print(forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail())

    # Plot the forecast and components
    model.plot(forecast, uncertainty=True)
    model.plot_components(forecast)
    plt.show()

    actual = prophet_subset['y']
    predicted = forecast['yhat'][-len(actual):]

    fb_metrics_df = calculate_metrics(actual, predicted)

    return actual, predicted, fb_metrics_df


fb_metrics_dfs = []
```

Harbeson, DE - Facebook Prophet

```
actual, predicted, metrics_df = prophet_model('Harbeson, DE')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

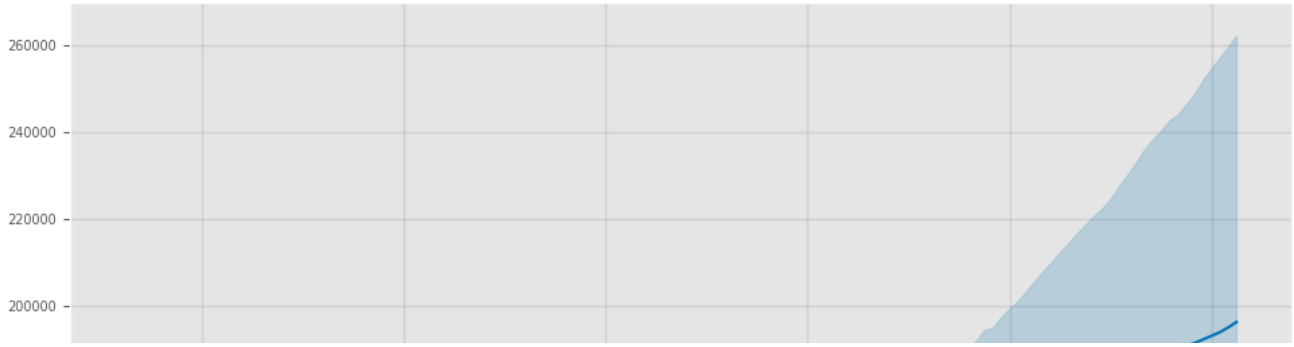
```
13:52:45 - cmdstanpy - INFO - Chain [1] start processing
13:52:46 - cmdstanpy - INFO - Chain [1] done processing
Results for Harbeson, DE:
           ds           yhat     yhat_lower     yhat_upper
128 2022-12-01  289935.035747  279592.340475  299883.065326
129 2023-01-01  290735.862090  279317.024092  301391.407565
130 2023-02-01  290945.994235  279446.160259  300966.715203
131 2023-03-01  289591.484079  278992.302527  300760.865185
132 2023-04-01  291012.339977  279716.462520  302839.627673
```



Old Fort, NC - Facebook Prophet

```
actual, predicted, metrics_df = prophet_model('Old Fort, NC')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

```
13:52:47 - cmdstanpy - INFO - Chain [1] start processing
13:52:47 - cmdstanpy - INFO - Chain [1] done processing
Results for Old Fort, NC:
            ds           yhat     yhat_lower     yhat_upper
128 2022-12-01  192263.439289  131054.031121  251964.714880
129 2023-01-01  193113.335096  129660.842280  254654.850487
130 2023-02-01  193975.381183  127539.909821  257171.944617
131 2023-03-01  194985.233635  127439.966777  259474.623334
132 2023-04-01  196266.520437  125706.921184  262117.708936
```



Fairmount, GA - Facebook Prophet

```
actual, predicted, metrics_df = prophet_model('Fairmount, GA')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

```
13:52:48 - cmdstanpy - INFO - Chain [1] start processing
13:52:49 - cmdstanpy - INFO - Chain [1] done processing
Results for Fairmount, GA:
```

Atlanta, GA - Facebook Prophet

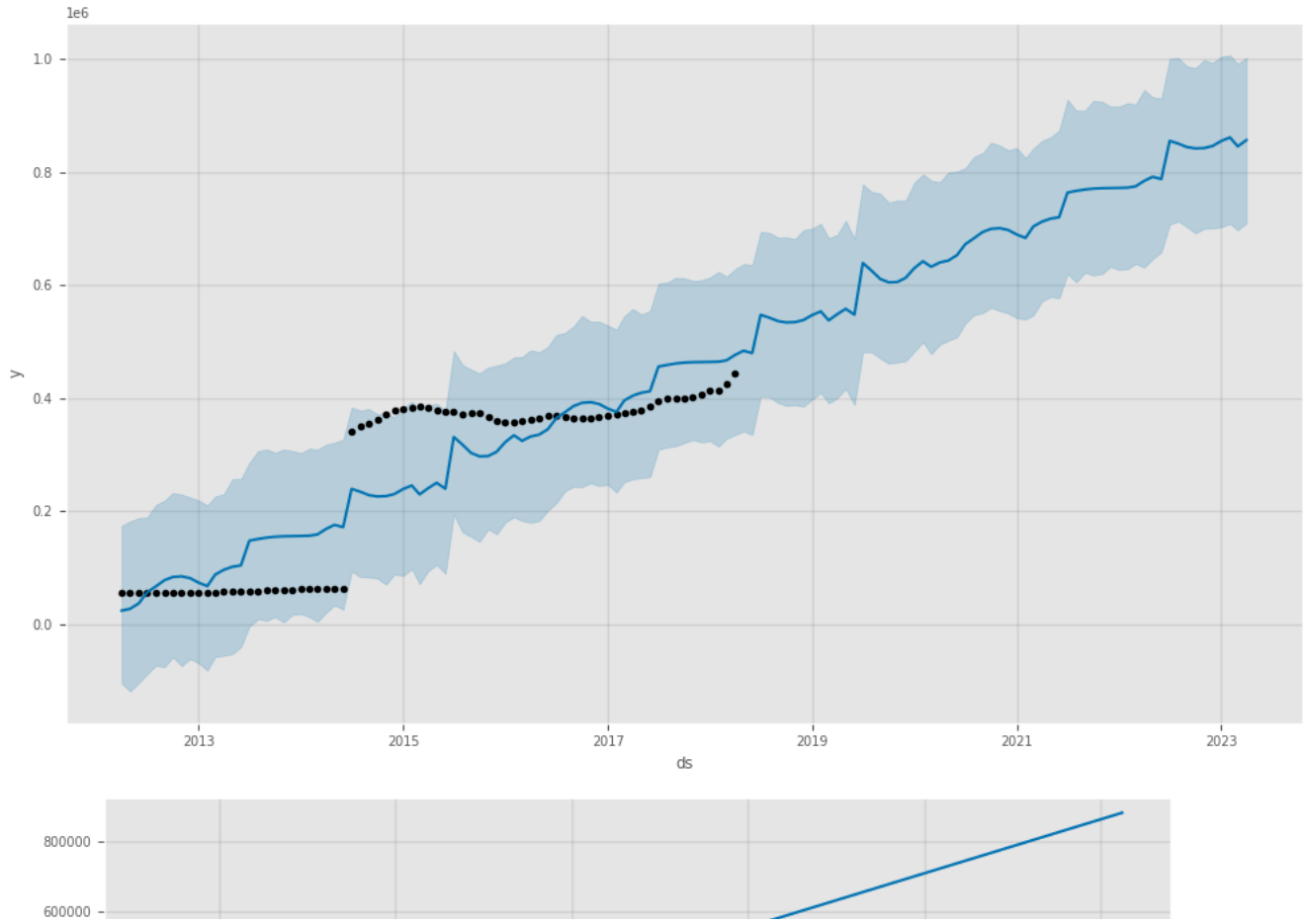
```
129 2023-01-01  193041.237978  123032.196092  279022.844483
```

```
actual, predicted, metrics_df = prophet_model('Atlanta, GA')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

Jekyll Island, GA - Facebook Prophet

```
actual, predicted, metrics_df =  prophet_model('Jekyll Island, GA')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

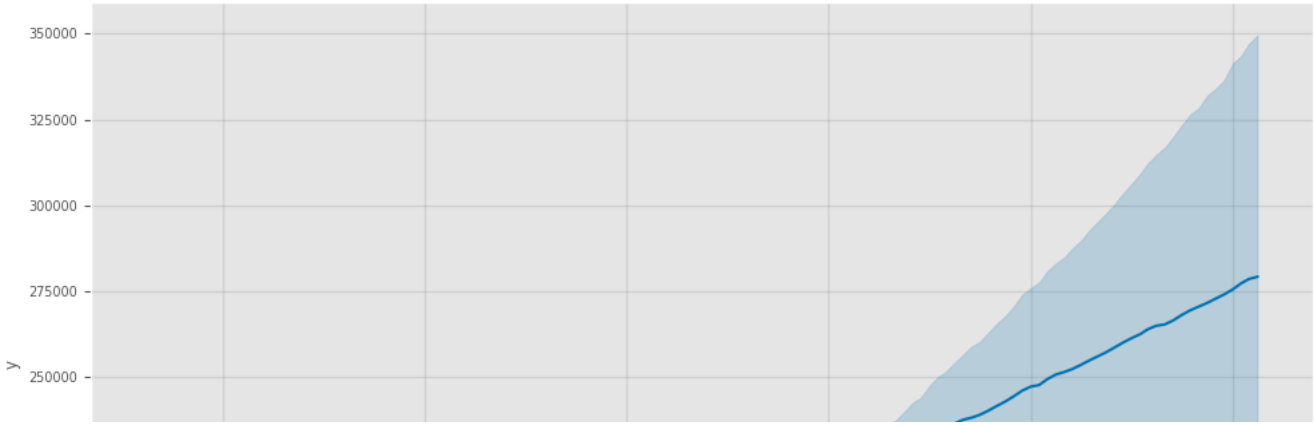
```
13:52:51 - cmdstanpy - INFO - Chain [1] start processing
13:52:52 - cmdstanpy - INFO - Chain [1] done processing
Results for Jekyll Island, GA:
           ds           yhat     yhat_lower    yhat_upper
128 2022-12-01  846582.382018  700859.285205  9.939339e+05
129 2023-01-01  855395.428644  702968.929456  1.004892e+06
130 2023-02-01  861851.821157  708915.031853  1.007525e+06
131 2023-03-01  845996.445555  697539.955517  9.920703e+05
132 2023-04-01  857046.170408  710128.050838  1.002904e+06
```



Decatur, TX - Facebook Prophet

```
actual, predicted, metrics_df =  prophet_model('Decatur, TX')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

```
13:52:52 - cmdstanpy - INFO - Chain [1] start processing
13:52:53 - cmdstanpy - INFO - Chain [1] done processing
Results for Decatur, TX:
            ds           yhat     yhat_lower     yhat_upper
128 2022-12-01  274072.031312  215226.544094  336366.667272
129 2023-01-01  275501.582724  216018.229810  341255.730106
130 2023-02-01  277308.399182  215749.167925  343530.640877
131 2023-03-01  278561.688734  215145.394086  347064.486298
132 2023-04-01  279224.973954  212578.776689  349482.365880
```



Austin, TX - Facebook Prophet

```
actual, predicted, metrics_df =  prophet_model('Austin, TX')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

```
13:52:54 - cmdstanpy - INFO - Chain [1] start processing
13:52:54 - cmdstanpy - INFO - Chain [1] done processing
Results for Austin, TX:
            ds           yhat     yhat_lower     yhat_upper
128 2022-12-01  525370.700775  425728.172060  639440.081648
129 2023-01-01  526944.110203  423912.852494  643859.408702
```

Soap Lake, WA - Facebook Prophet

```
actual, predicted, metrics_df = prophet_model('Soap Lake, WA')

# metrics dataframe
metrics_df2 = calculate_metrics(actual, predicted)
fb_metrics_dfs.append(metrics_df2)
```

f. Facebook Prophet Diagnostics

```
final_fb_metrics_df = pd.concat(fb_metrics_dfs)
final_fb_metrics_df = final_fb_metrics_df.sort_values(by='RMSE')
print(final_metrics_df)
```

```
                          RMSE          AIC
Decatur, TX         5033.433673   765.971035
Harbeson, DE        5848.989699   919.126657
Atlanta, GA         6356.581960   892.385470
Old Fort, NC        6779.883346   692.602794
Soap Lake, WA      10254.866268   875.084354
```

```
Austin, TX         11372.325421  1006.157434
Fairmount, GA      13161.083926   835.732403
Jekyll Island, GA  30178.717667  1201.778543
```

7.0 Conclusion

Choice of Model

Three models were used - Baseline Seasonal Autoregressive Integrated Moving Average (SARIMA), Tuned SARIMA and Facebook Prophet.

- The Baseline SARIMA has not been tuned for best p, d, q values. It also chooses a random zip code at each execution to provide a baseline overview of the sale price prediction. Hence, it is not a choice model for our prediction and forecasting needs.
- While both the tuned SARIMA and Facebook Prophet models gave similar results, the Facebook Prophet model was found to be best suited for forecasting as it was able to
    1. Forecast future prices for the next five years
    2. Give the best months to invest in for each zip code.
- Therefore, **Facebook Prophet** had enhanced performance, features and metrics that provided better predictions and forecasting trends and insights.

Top 5 zip codes that are best for investing in real estate chosen for their low RMSE values:

Zip Code	RMSE	AIC
Decatur, TX	5033.433673	765.971035
Harbeson, DE	5753.477978	912.078435
Atlanta, GA	6154.167274	892.355686
Old Fort, NC	6779.883346	692.602794
Soap Lake, WA	10254.866297	875.084354

Top 5 states that have the highest ROI

State	ROI 22 years	ROI 5 years	ROI 3 years
Florida	1312.23	524.30	291.05
Texas	920.99	384.54	242.72
Washington	609.43	203.68	134.62
New Jersey	605.86	96.68	69.43
Georgia	565.04	144.20	76.19

The best months to sell to maximize profit are:

1. December in Harbeson, DE,
2. April in Old Fort, NC and Fairmount, GA
3. June in Decatur, TX, Soap Lake, WA

In summary:

- **Facebook Prophet** is a better model in predicting future housing sales values.
- The return on investment and the risk are not necessarily correlated. Sometimes a high ROI could also mean a high risk and vice versa.

8.0 Recommendations

- Invest on the top 5 zip codes and allocate resources towards them.
- Obtain current data after 2018 for current predictions.
- There is need to consider outside factors (housing grade, property size, renovations) that may also affect sales value.