1 Real Estate ROI Analysis- Time Series Modeling

1.1 Project Overview

This project will look into a Real Estate database to find valuable information about the value of homes and return on investment. My company, Xabios data international, has been hired to determind the change of prices in this market over time and to determine which areas have the most potential to increase. Thus, this project is aimed at two very targeted goal using Time Series Modeling :

Jupyter Notebook A brief note on this Jupyter notebook. This notebook presents story line following this structure:

- Part 1: Overview, Business Understanding & Objective
- Part 2 : EDA
- Part 3 : Modeling
- Part 4 : Conclusion

1.2 Business Understanding

As important as current prices are to Real Estate companies, so it is the ability to know where prices will go higher in the future to make investement decisions. The key metric of leaving knowing how much money will an investment property return is known as Return on Investment or "ROI" as it is commonly abbreviated and known in finance. From a business perspective, the business would like to understand exactly what factors and zip codes contribute to ROI, and most importantly, post identification of those factors, define precise strategic initiatives to aim funds disbursement for the purchase of Real Estate property. This project, therefore aims to help the business first in identifying the past, how prices change over time in the zone in which this analysis occurs, and in turn, build time-series models that can help the business predict future value of property, so that in fact strategic business initiatives can be outlined for solution.

1.2.1 Business Modeling Objectives

1. Understand Real Estate Prices - how they have changed overtime
2. Find out Real Estate Areas of Growth Measured by ROI

1.2.2 Important Data Modeling Success Considerations

We are conducting a Time Series Analysis over the Zillow Home Value Index (ZHVI):A data set that tracks a sample of real estate prices over time. In particular, a time series allows one to see what factors influence certain variables from period to period. Time series analysis can be useful to see how a given asset, in this case Real Estate, or economic variable, in this case ROI, changes over time.

1.3 EDA - Loading, Modifying and Understanding DataSet

In [1]:
```
import pandas as pd
from pandas import Series
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt

import plotly.express as px
import plotly.io as pio

%matplotlib inline
```

In [2]:
```
#Loading data
df= pd.read_csv ('zillow_data.csv')
```

In [3]:
```
#Seeing the shape of the
df.shape
```

Out[3]:
```
(14723, 272)
```

In [4]:
```
#Finding more key information of the data set
```

In [5]:
```
df.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
```

In [6]:
```
#looking inside the data set
df.head()

```

Out[6]:

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	1996-05	
0	84654	60657	Chicago	IL	Chicago	Cook	1	334200.0	335400.0	3
1	90668	75070	McKinney	TX	Dallas-Fort Worth	Collin	2	235700.0	236900.0	2
2	91982	77494	Katy	TX	Houston	Harris	3	210400.0	212200.0	2
3	84616	60614	Chicago	IL	Chicago	Cook	4	498100.0	500900.0	5
4	93144	79936	El Paso	TX	El Paso	El Paso	5	77300.0	77300.0	

5 rows × 272 columns

Basically, this is a national database, that's showing us there are 14,723 ZIP codes in total. We'd need to focus on one market area specifically. For the purpose of forecasting within that market.

1.3.0.1 Selecting Market: State(s) & Indexing Columns

Focusing on getting an understanding of the Connecticut market zip codes since it is an area of interest to my client , and area of investment preference

In [7]:
```
#selecting zip codes within selected market
#making state Index and rename of RegionName and Zipcode
df_ct = df.loc[df['State']== 'CT'].reset_index()
df_ct.drop(['index', 'RegionID', 'City', 'State', 'CountyName'], axis=1
df_ct.rename(columns= {'RegionName' : 'Zipcode'}, inplace=True)
```

In [8]:
```
#finding unique values, in this case areas in Connecticut
df_ct.Metro.unique()
```

Out[8]:
```
array(['Hartford', 'New Haven', 'Stamford', 'New London', 'Torrington',
       'Worcester'], dtype=object)
```

In [9]:
```
#understand how many ZIP codes in the combined areas
print(f'Number of CT zip codes : {len(df_ct)}')
```

```
Number of CT zip codes : 124
```

In [10]:

```
#looking at new data set df_ct head
df_ct.head()
```

Out[10]:

	Zipcode	Metro	SizeRank	1996-04	1996-05	1996-06	1996-07	1996-08	1996-09	1996-1
0	6010	Hartford	113	120300.0	120000.0	119800.0	119400.0	119100.0	118800.0	118600
1	6516	New Haven	417	96500.0	96300.0	96100.0	95900.0	95600.0	95300.0	95000
2	6511	New Haven	546	89800.0	90000.0	90200.0	90300.0	90500.0	90700.0	90800
3	6810	Stamford	685	151100.0	150700.0	150200.0	149700.0	149100.0	148600.0	148200
4	6492	New Haven	899	146800.0	146600.0	146300.0	146100.0	145900.0	145700.0	145600

5 rows × 268 columns

1.3.1 Obtaining Return on Investment (ROI)

In [11]:

```
#Finding ROIs for 5 and 3 years
df_ct['ROI_5yr'] = round((df_ct['2018-04']- df_ct['2013-01'])/df_ct['20
df_ct['ROI_3yr'] = round((df_ct['2018-04']- df_ct['2015-01'])/df_ct['20

```

In [12]:

```
#Looking at the dataset to find ROI's were added correctly
df_ct.shape
df_ct.info()
df_ct.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 124 entries, 0 to 123
Columns: 270 entries, Zipcode to ROI_3yr
dtypes: float64(221), int64(48), object(1)
memory usage: 261.7+ KB
```

Out[12]:

	Zipcode	Metro	SizeRank	1996-04	1996-05	1996-06	1996-07	1996-08	1996-09	1996-1
0	6010	Hartford	113	120300.0	120000.0	119800.0	119400.0	119100.0	118800.0	118600
1	6516	New Haven	417	96500.0	96300.0	96100.0	95900.0	95600.0	95300.0	95000
2	6511	New Haven	546	89800.0	90000.0	90200.0	90300.0	90500.0	90700.0	90800
3	6810	Stamford	685	151100.0	150700.0	150200.0	149700.0	149100.0	148600.0	148200
4	6492	New Haven	899	146800.0	146600.0	146300.0	146100.0	145900.0	145700.0	145600

5 rows × 270 columns

The formatted shape of this data set needs to be re-formatted to follow a long format, where the 268 columns are rows, and the rows are columns. We do this by re-shaping.

1.3.2 Reshaping data set to Long Format calling it :melted_df

In [13]:
```
def melt_data(df):

    melted = pd.melt(df, id_vars=['Zipcode', 'Metro', 'SizeRank', 'ROI_
    melted['Date'] = pd.to_datetime(melted['Date'], infer_datetime_form
    melted = melted.dropna(subset=['value'])
    return melted

```

In [14]:
```
#passing the melt_data transformaiton into the dataset
melted_df=melt_data(df_ct)
```

1.3.2.1 melted_df (Long Format) view :

In [15]:
```
#looking at the long format dataset
melted_df.head()
```

Out[15]:

	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	Date	value
0	6010	Hartford	113	0.1063	0.0982	1996-04-01	120300.0
1	6516	New Haven	417	0.0752	0.0970	1996-04-01	96500.0
2	6511	New Haven	546	0.1462	0.1839	1996-04-01	89800.0
3	6810	Stamford	685	0.2219	0.1839	1996-04-01	151100.0
4	6492	New Haven	899	0.1182	0.0410	1996-04-01	146800.0

In [16]:
```
melted_df.isna().value_counts()
```

Out[16]:
```
Zipcode  Metro  SizeRank  ROI_5yr  ROI_3yr  Date   value
False    False  False     False    False    False  False    32860
dtype: int64
```

The melted_df is a much better looking data set with ROI on a per Zip Code basis. However we want to look at its information and types of data that define the columns, and perhaps modify the setting a new index given that we are workin with a Time-Series analysis. So I'll do this next.

1.3.2.2 Analyzing melted_df

In [17]:
```
#Looking at the information
melted_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 32860 entries, 0 to 32859
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Zipcode   32860 non-null  int64
 1   Metro     32860 non-null  object
 2   SizeRank  32860 non-null  int64
 3   ROI_5yr   32860 non-null  float64
 4   ROI_3yr   32860 non-null  float64
 5   Date      32860 non-null  datetime64[ns]
 6   value     32860 non-null  float64
dtypes: datetime64[ns](1), float64(3), int64(2), object(1)
memory usage: 2.0+ MB
```

1.3.2.3 Modifying melted_df for Time-Series Analysis

In [18]:
```
#Change Zipcode dtype to 'str'
melted_df['Zipcode'] = melted_df['Zipcode'].astype(str)

# Make sure the data type of the 'Date' column is datetime
melted_df['Date'] = pd.to_datetime(melted_df['Date'], format='%m/%y')

# Set the 'Date' column as index
melted_df.set_index('Date', inplace=True)
```

I now have changed the ZIP Code to a string type, made sure that the Date format is correct and have set Date as index.

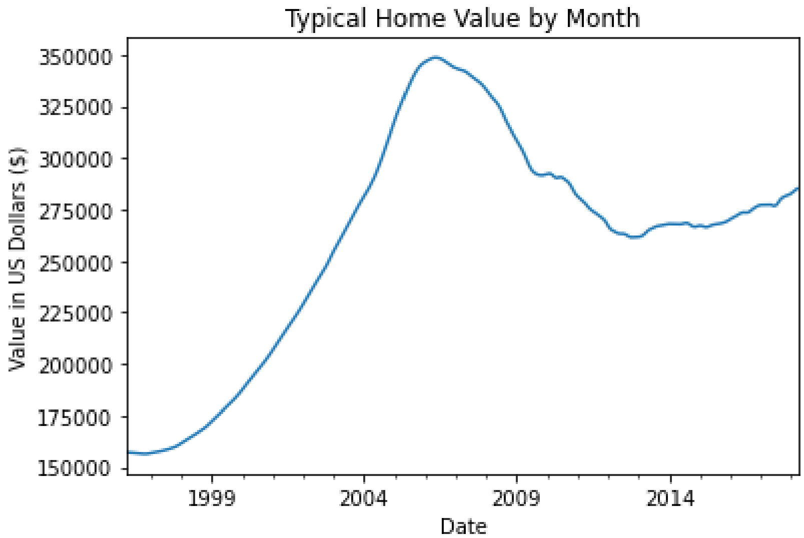
In [19]:
```
#checking for new date index
melted_df.index
```

Out[19]:
```
DatetimeIndex(['1996-04-01', '1996-04-01', '1996-04-01', '1996-04-01',
               '1996-04-01', '1996-04-01', '1996-04-01', '1996-04-01',
               '1996-04-01', '1996-04-01',
               ...
               '2018-04-01', '2018-04-01', '2018-04-01', '2018-04-01',
               '2018-04-01', '2018-04-01', '2018-04-01', '2018-04-01',
               '2018-04-01', '2018-04-01'],
              dtype='datetime64[ns]', name='Date', length=32860, freq=Non
e)
```

Looking at Trends: Home value in the last 10 years

The 2008 crash dropped the values of home until 2013, when it started to pick up

In [20]:
```
monthly_data = melted_df.resample('MS').mean()['value']
monthly_data.plot()
plt.title('Typical Home Value by Month')
plt.ylabel('Value in US Dollars ($)')
plt.show()
```



The Date index view and Trend Chart above gives insight into the data set and allows me to make choices on how is it going to be treated. First, we understand that this data set values are given from 1996 through 2018. Second, it shows an abrupt decline in prices in 2008. The abrupt decline can be explained by prices by the financial collapse that occurred nationally affecting CT obviously in 2008. Therefore, my focus on the analysis and modeling will be from 2008 through 2018 best ZIP Codes as determined by ROI.

1.3.2.4 Modifying data set for analysis (based on findings explained above)

The Melted_df will now hold values for the past 10 years

Now, I'll refrom melted_df to show the house prices for the past 10 years, and modify a few minor things in order to view my new data set, finally visualizing and ranking areas or ZIP Codes in preparation for modeling techniques.

In [21]:
```
#looking only at the last 10 years
melted_df = melted_df['2008-01-01':'2018-04-01']
```

In [22]:
```
#resampling
melted_df.resample('MS').mean().index
```

Out[22]:
```
DatetimeIndex(['2008-01-01', '2008-02-01', '2008-03-01', '2008-04-01',
               '2008-05-01', '2008-06-01', '2008-07-01', '2008-08-01',
               '2008-09-01', '2008-10-01',
               ...
               '2017-07-01', '2017-08-01', '2017-09-01', '2017-10-01',
               '2017-11-01', '2017-12-01', '2018-01-01', '2018-02-01',
               '2018-03-01', '2018-04-01'],
              dtype='datetime64[ns]', name='Date', length=124, freq='MS')
```

Resampling technique is used to gather more information about a sample.Retaking a sample or resampling often improves the overall accuracy and estimates any uncertainty within this picked population.

In [23]:
```
#looking at the melted data set that now will have information on CT ZI
melted_df.info()
melted_df.head()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 15376 entries, 2008-01-01 to 2018-04-01
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Zipcode   15376 non-null  object
 1   Metro     15376 non-null  object
 2   SizeRank  15376 non-null  int64
 3   ROI_5yr   15376 non-null  float64
 4   ROI_3yr   15376 non-null  float64
 5   value     15376 non-null  float64
dtypes: float64(3), int64(1), object(2)
memory usage: 840.9+ KB
```

Out[23]:

	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
Date						
2008-01-01	6010	Hartford	113	0.1063	0.0982	219300.0

1.3.2.5 Stats and Data Visualization

In [24]:
```

melted_df.Metro.unique()
```

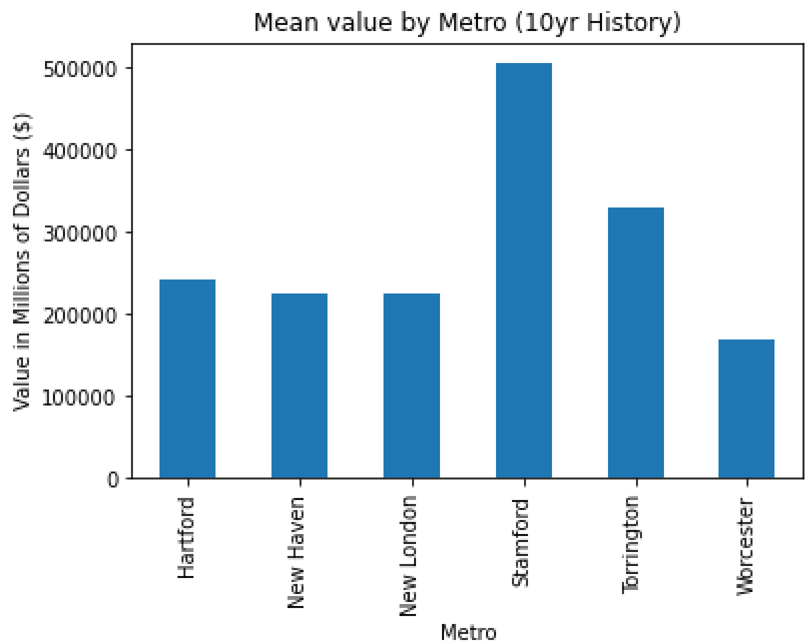
Out[24]:
```
array(['Hartford', 'New Haven', 'Stamford', 'New London', 'Torrington',
       'Worcester'], dtype=object)
```

In [25]:
```
#Grouping by Metro and Mean
metro_grp = melted_df.groupby('Metro')
metro_grp = metro_grp.value.mean()
metro_grp = metro_grp.sort_values(ascending=False).head(10)
```

We have 5 different areas by Metro area. I have grouped them and this will help visualize data set

In [26]:
```
#looking at the values over time by Metro area
metro_area= melted_df.groupby('Metro')
metro_area= metro_area.value.mean()
metro_ara= metro_area.sort_values(ascending=False).head()
```

In [27]:
```
metro_area.plot(x="Name", y=["Age", "Height(in cm)"], kind="bar")
plt.title('Mean value by Metro (10yr History)')
plt.ylabel('Value in Millions of Dollars ($)')
plt.show()
plt.savefig('10yr-metro-mean-value')
```




```
<Figure size 432x288 with 0 Axes>
```

Top 3 with higuest mean values are Stamford, Torrington, and Hartford

In [28]:
```
#Looking Stamford
stamford = melted_df.loc[melted_df['Metro']=='Stamford']
stamford.describe()
```

Out[28]:

	SizeRank	ROI_5yr	ROI_3yr	value
count	1984.000000	1984.000000	1984.000000	1.984000e+03
mean	5395.500000	0.117881	0.068744	5.050442e+05
std	2786.125208	0.108130	0.120613	3.820920e+05
min	685.000000	-0.019200	-0.076400	1.185000e+05
25%	3724.750000	0.040875	-0.000250	2.750750e+05
50%	5896.500000	0.090450	0.030600	3.450000e+05
75%	6677.500000	0.155375	0.109650	7.189750e+05
max	11245.000000	0.320200	0.316600	1.746000e+06

In [29]:
```
#Torringtong
torrington = melted_df.loc[melted_df['Metro']=='Torrington']
torrington.describe()
```

Out[29]:

	SizeRank	ROI_5yr	ROI_3yr	value
count	2480.000000	2480.000000	2480.000000	2480.000000
mean	11759.300000	0.150080	0.112490	331128.870968
std	2795.020029	0.099232	0.067996	123206.641312
min	3789.000000	0.015000	0.021000	135800.000000
25%	10302.250000	0.077700	0.062300	245900.000000
50%	13041.500000	0.146900	0.090500	307450.000000
75%	13663.000000	0.190975	0.168325	409825.000000
max	14552.000000	0.415800	0.248700	737800.000000

In [30]:
```
#Hartford
hartford = melted_df.loc[melted_df['Metro']=='Hartford']
hartford.describe()
```

Out[30]:

	SizeRank	ROI_5yr	ROI_3yr	value
count	4340.000000	4340.000000	4340.000000	4340.000000
mean	7921.885714	0.059234	0.071354	242469.493088
std	3897.308858	0.046046	0.042045	65698.245634
min	113.000000	-0.081200	-0.039600	103300.000000
25%	4011.000000	0.033900	0.040000	197475.000000
50%	8085.000000	0.063500	0.064900	242400.000000
75%	10809.000000	0.084400	0.096500	277900.000000
max	14655.000000	0.155800	0.158000	460500.000000

Note on Polulation Density as Size Rank

Looking at the population density

In [31]:
```
melted_df.SizeRank.unique()
```

Out[31]:
```
array([  113,   417,   546,   685,   899,  1145,  1332,  1362,  1599,
        1901,  2093,  2192,  2301,  2390,  2497,  3051,  3245,  3281,
        3368,  3532,  3565,  3581,  3696,  3760,  3778,  3789,  3978,
        4011,  4230,  4324,  4531,  4587,  4674,  4717,  5580,  5624,
        5687,  5756,  5852,  5870,  5941,  6037,  6063,  6084,  6176,
        6270,  6288,  6329,  6424,  6591,  6599,  6813,  6913,  7022,
        7031,  7144,  7259,  7317,  7382,  7471,  7496,  7595,  7704,
        7741,  7890,  8085,  8368,  8425,  8427,  8584,  8601,  8943,
        9032,  9136,  9160,  9275,  9298,  9457,  9577,  9681,  9715,
        9788,  9871,  9903,  9977, 10164, 10281, 10446, 10463, 10591,
       10651, 10663, 10786, 10809, 10893, 11245, 11629, 11907, 11929,
       12034, 12184, 12310, 12406, 12469, 12556, 12817, 12967, 13018,
       13065, 13214, 13215, 13266, 13465, 13564, 13567, 13646, 13714,
       13908, 13927, 14096, 14356, 14477, 14552, 14655])
```

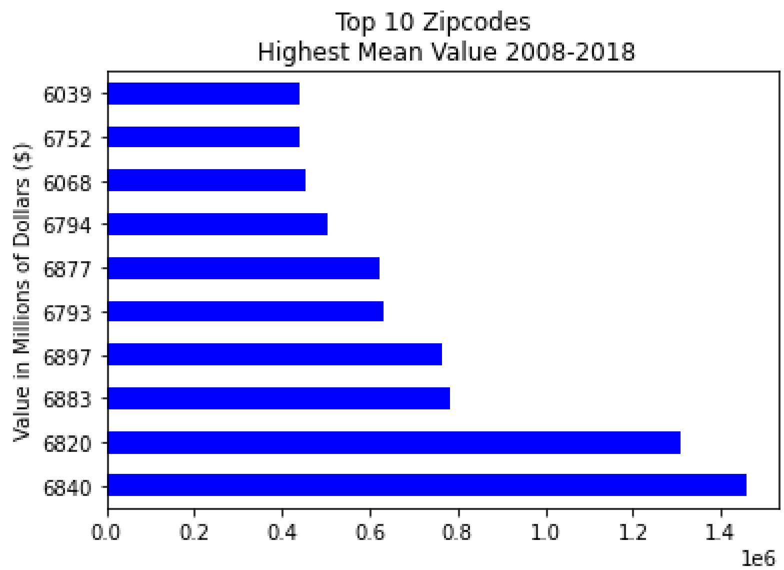
It is noted that the areas range between mid high and low densily populated.

1.3.2.6 Top Zip Codes Mean Value of Homes View

Now that we know the top areas by mean, we will look at the ZIP Codes themselves and determine which are exactly the best by looking at the average mean price

In [32]:

```
#group zip code
zipcode_group= melted_df.groupby('Zipcode')

#top 10 values
zip_mean= zipcode_group.value.mean()
zip_mean= zip_mean.sort_values(ascending=False).head(10)

zip_mean.plot.barh(color='blue')
plt.title('Top 10 Zipcodes  \n Highest Mean Value 2008-2018')
plt.ylabel('Value in Millions of Dollars ($)')
plt.show()
plt.savefig('Zipcodes')
```




```
<Figure size 432x288 with 0 Axes>
```

In [33]:

```
#looking at the higest mean value zip_cove
melted_df.loc[melted_df['Zipcode']=='6840']

```

Out[33]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6840	Stamford	5941	-0.0012	-0.0764	1746000.0
2008-02-01	6840	Stamford	5941	-0.0012	-0.0764	1736100.0
2008-03-01	6840	Stamford	5941	-0.0012	-0.0764	1729800.0
2008-04-01	6840	Stamford	5941	-0.0012	-0.0764	1728700.0
2008-05-01	6840	Stamford	5941	-0.0012	-0.0764	1727600.0
...	...	...	...	...	...	...
2017-12-01	6840	Stamford	5941	-0.0012	-0.0764	1355500.0
2018-01-01	6840	Stamford	5941	-0.0012	-0.0764	1361600.0
2018-02-01	6840	Stamford	5941	-0.0012	-0.0764	1374500.0
2018-03-01	6840	Stamford	5941	-0.0012	-0.0764	1381400.0
2018-04-01	6840	Stamford	5941	-0.0012	-0.0764	1379900.0

124 rows × 6 columns

In [34]:

```
#looking at the second higest mean value zip_cove
melted_df.loc[melted_df['Zipcode']=='6820']
```

Out[34]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6820	Stamford	6037	0.1149	0.0045	1401500.0
2008-02-01	6820	Stamford	6037	0.1149	0.0045	1399200.0
2008-03-01	6820	Stamford	6037	0.1149	0.0045	1395800.0
2008-04-01	6820	Stamford	6037	0.1149	0.0045	1392500.0
2008-05-01	6820	Stamford	6037	0.1149	0.0045	1391200.0
...	...	...	...	...	...	...
2017-12-01	6820	Stamford	6037	0.1149	0.0045	1375600.0
2018-01-01	6820	Stamford	6037	0.1149	0.0045	1374700.0
2018-02-01	6820	Stamford	6037	0.1149	0.0045	1379100.0
2018-03-01	6820	Stamford	6037	0.1149	0.0045	1385800.0
2018-04-01	6820	Stamford	6037	0.1149	0.0045	1388100.0

124 rows × 6 columns

In [35]:
```
#further understanding the values in this data set
melted_df.value.describe()
```

Out[35]:
```
count    1.537600e+04
mean     2.802331e+05
std      1.840456e+05
min      8.010000e+04
25%      1.838000e+05
50%      2.419000e+05
75%      3.145250e+05
max      1.746000e+06
Name: value, dtype: float64
```

The Max value of all time and the Min Value of all times are:

In [36]:
```
melted_df.loc[melted_df['value']== 1746000]
```

Out[36]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6840	Stamford	5941	-0.0012	-0.0764	1746000.0

In [37]:
```
melted_df.loc[melted_df['value']== 80100]
```

Out[37]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2015-04-01	6706	New Haven	7317	0.0274	0.3196	80100.0

1.3.3 Understanding 5 yr. ROI Homes & the Top 10% in this category

In [38]:
```
#There are 15,736 homes, we'll look at the top 10% in this set
melted_df['ROI_5yr'].describe()

```

Out[38]:
```
count    15376.000000
mean         0.098785
std          0.076585
min         -0.081200
25%          0.051025
50%          0.090000
75%          0.130150
max          0.415800
Name: ROI_5yr, dtype: float64
```

In [39]:
```
#getting the quantile .90 for the top 10%

#first getting 90% of these homes or the .90 quantile
ninety_perc_ROI_5yr = melted_df['ROI_5yr'].quantile(q=0.90)

#Now deriving the top 10% by taking the values greater than that 90th p
top_10_percent = melted_df.loc[melted_df['ROI_5yr']>=ninety_perc_ROI_5y
```

In [40]:
```
#looking at the first 10 values of the top 10% with best 5yr ROI
top_10_percent.head(5)
```

Out[40]:

	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
Date						
2008-01-01	6810	Stamford	685	0.2219	0.1839	321900.0
2008-01-01	6606	Stamford	1332	0.3202	0.2165	252300.0
2008-01-01	6513	New Haven	2301	0.2725	0.1802	187600.0
2008-01-01	6604	Stamford	3778	0.3141	0.3166	274700.0
2008-01-01	6610	Stamford	4717	0.2999	0.3111	216600.0

In [41]:
```
top_10_percent.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1612 entries, 2008-01-01 to 2018-04-01
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   Zipcode   1612 non-null   object 
 1   Metro     1612 non-null   object 
 2   SizeRank  1612 non-null   int64  
 3   ROI_5yr   1612 non-null   float64
 4   ROI_3yr   1612 non-null   float64
 5   value     1612 non-null   float64
dtypes: float64(3), int64(1), object(2)
memory usage: 88.2+ KB
```

1,612 homes are in that 10% with the best ROI. Doing the same for the 3 yrs ROI

1.3.4 Top 10% Zip Codes 3 yr. ROI

In [42]:
```
#getting the quantile .90 for the top 10%
ninety_perc_ROI_3yr = melted_df['ROI_3yr'].quantile(q=0.90)
top_10_percent_3yr = melted_df.loc[melted_df['ROI_3yr']>=ninety_perc_RO
```

In [43]:
```
#looking at the first 10 values of the top 10% with best 3yr ROI
top_10_percent_3yr.info()
top_10_percent_3yr.head(5)
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1612 entries, 2008-01-01 to 2018-04-01
Data columns (total 6 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   Zipcode   1612 non-null   object 
 1   Metro     1612 non-null   object 
 2   SizeRank  1612 non-null   int64  
 3   ROI_5yr   1612 non-null   float64
 4   ROI_3yr   1612 non-null   float64
 5   value     1612 non-null   float64
dtypes: float64(3), int64(1), object(2)
memory usage: 88.2+ KB
```

Out[43]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6606	Stamford	1332	0.3202	0.2165	252300.0
2008-01-01	6705	New Haven	3696	0.0056	0.1964	152500.0
2008-01-01	6604	Stamford	3778	0.3141	0.3166	274700.0
2008-01-01	6610	Stamford	4717	0.2999	0.3111	216600.0
2008-01-01	6351	New London	6591	0.1183	0.2234	217300.0

1.3.5 Data Set Prepared For Modeling

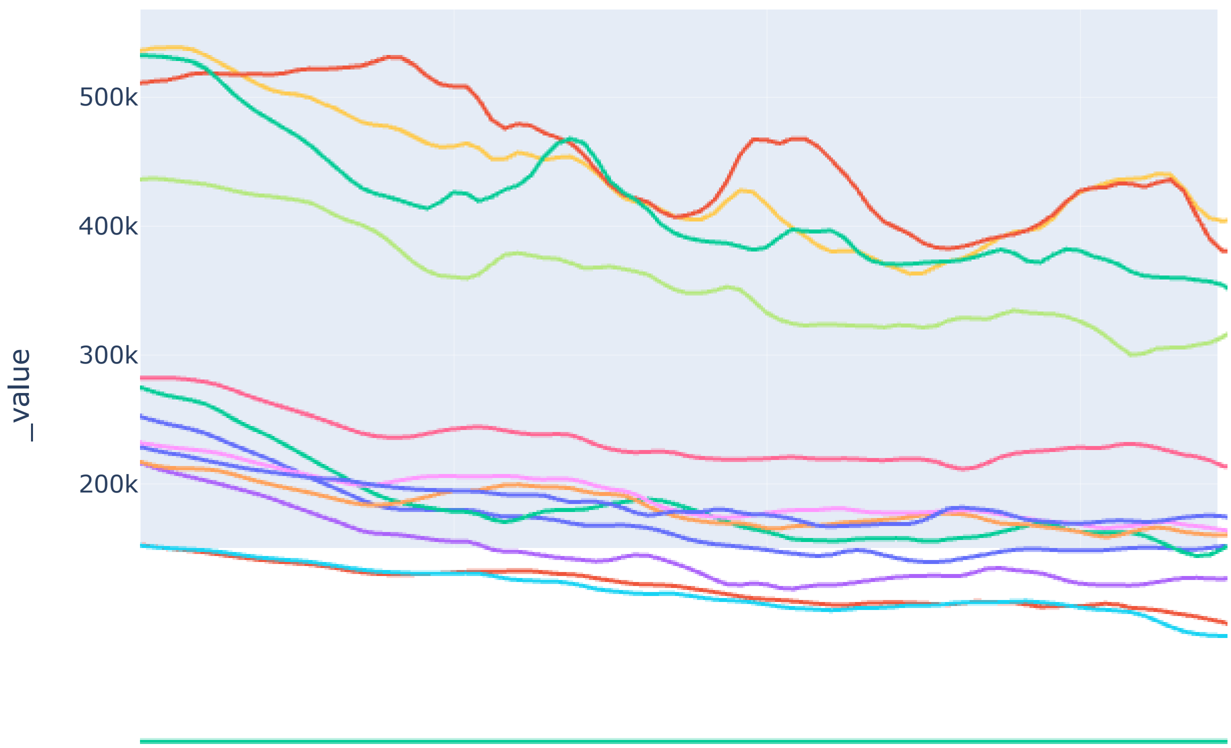
I'll therefore use the top 10% ROI ZIP Codes for in the past 3yrs, create a new data set called "df", store it and use it for ARIMA Modeling preditions.

In [44]:
```
#New data set to drop the ROI 3yr data set Metro and SizeRank to prepar
df = top_10_percent_3yr.drop(columns=['Metro', 'SizeRank'])

#Looking at it
df.head()
```

Out[44]:

Date	Zipcode	ROI_5yr	ROI_3yr	value
2008-01-01	6606	0.3202	0.2165	252300.0
2008-01-01	6705	0.0056	0.1964	152500.0
2008-01-01	6604	0.3141	0.3166	274700.0
2008-01-01	6610	0.2999	0.3111	216600.0
2008-01-01	6351	0.1183	0.2234	217300.0

In [45]:
```
#Showing the top 13 ZIP Codes ROI For the past 3yrs
px.line(df, color='Zipcode')
```



In [46]:
```
%store melted_df
```

Stored 'melted_df' (DataFrame)

In [47]:
```
%store df
```

Stored 'df' (DataFrame)

In [ ]:
```

```

1 ARIMA CODING

I'll be implementing and autoregressive integrated moving average model. An autoregressive integrated moving average mode, or ARIMA, is a statistical analysis model that uses time series data to either better understand the data set or to predict future trends. A statistical model is autoregressive if it predicts future values based on past values. I'll load the libraries and start the process of modeling. I'll build a base model and a second model choosing a best model. As the objective is to find the ZIP Codes in a particular market that predict the best ROI.

1.1 Loading data sets and libraries

In [1]:
```
#!pip install
%store -r df
%matplotlib inline
```

In [2]:
```
#importing libraries
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np


## Project Notebook Settings
pd.set_option('display.max_columns',0)

import warnings
warnings.filterwarnings('ignore')

plt.style.use('seaborn-notebook')
```

In [3]:
```
#checking the data set
df.head(5)
```

Out[3]:

Date	Zipcode	ROI_5yr	ROI_3yr	value
2008-01-01	6606	0.3202	0.2165	252300.0
2008-01-01	6705	0.0056	0.1964	152500.0
2008-01-01	6604	0.3141	0.3166	274700.0
2008-01-01	6610	0.2999	0.3111	216600.0
2008-01-01	6351	0.1183	0.2234	217300.0

1.1.1 Time Series Process

The data set has to be prepared to for modeling. The correct process for managing Time Series correctly includes:

1. Grouping the data set and creating a Time Series (TS)
2. Converting to Pandas DataFrame and Visualization
3. Conceptual Soundness -Set up for Modeling (Understanding Time Series-ACF & PACF, Differencing & Decomposition)
4. Stats Models- Visualizing ACF & PACF
5. Stats Models- Differencing Detrending Transformation
6. Modeling - Predictions and Results

1.1.2 1. Grouping the data set

In [4]:
```
#Making a Zipcode list
#taking the unique ZIP Code values and fitting them to the list
zipcode_list = df['Zipcode'].unique().tolist()
```

In [5]:
```
#Creating a TS (time series) dictionary and loop
TS = {}
for zipcode in zipcode_list:
    temp_df = df.groupby('Zipcode').get_group(zipcode).sort_index()['va
    TS[zipcode] = temp_df #df.loc[district]
```

In [6]:
```
#Looking at the keys in the TS dictionary
TS.keys()
```

Out[6]:
```
dict_keys(['6606', '6705', '6604', '6610', '6351', '6706', '6359', '606
9', '6330', '6039', '6235', '6068', '6796'])
```

In [7]:
```
#Looking at ZIP Code 6706 in the dictionary now that ZIP Codes are grou
#note that Date is index, group is the the ZIP Code by descending value
TS['6706']
```

Out[7]:
```
Date
2008-01-01    152200.0
2008-02-01    151200.0
2008-03-01    150300.0
2008-04-01    149600.0
2008-05-01    149200.0
                 ...
2017-12-01    108900.0
2018-01-01    109800.0
2018-02-01    109500.0
2018-03-01    109000.0
2018-04-01    108600.0
Name: value, Length: 124, dtype: float64
```

1.1.3 Converting (To Pandas DataFrame)

Now we are going to convert and visualize the TS dictionary created and put it into a Pandas DataFrame. The Pandas DataFrame for time series shows from 2008 through 2018 montly values under each of the ZIP Codes.

In [8]:

```
#Converting TS dictionary into Pandas DataFrame calling it ts_df
ts_df = pd.DataFrame(TS)
ts_df.head()
```

Out[8]:

	6606	6705	6604	6610	6351	6706	6359	6069	6330	
Date										
2008-01-01	252300.0	152500.0	274700.0	216600.0	217300.0	152200.0	282100.0	436300.0	232100.0	53
2008-02-01	249500.0	151300.0	271300.0	213100.0	214400.0	151200.0	282000.0	437000.0	230400.0	53
2008-03-01	247000.0	150200.0	268600.0	210100.0	212800.0	150300.0	282100.0	436300.0	228900.0	53
2008-04-01	244700.0	149100.0	266600.0	207500.0	212100.0	149600.0	281600.0	434900.0	227800.0	53
2008-05-01	242400.0	148100.0	264600.0	205100.0	211900.0	149200.0	280500.0	433700.0	226700.0	53

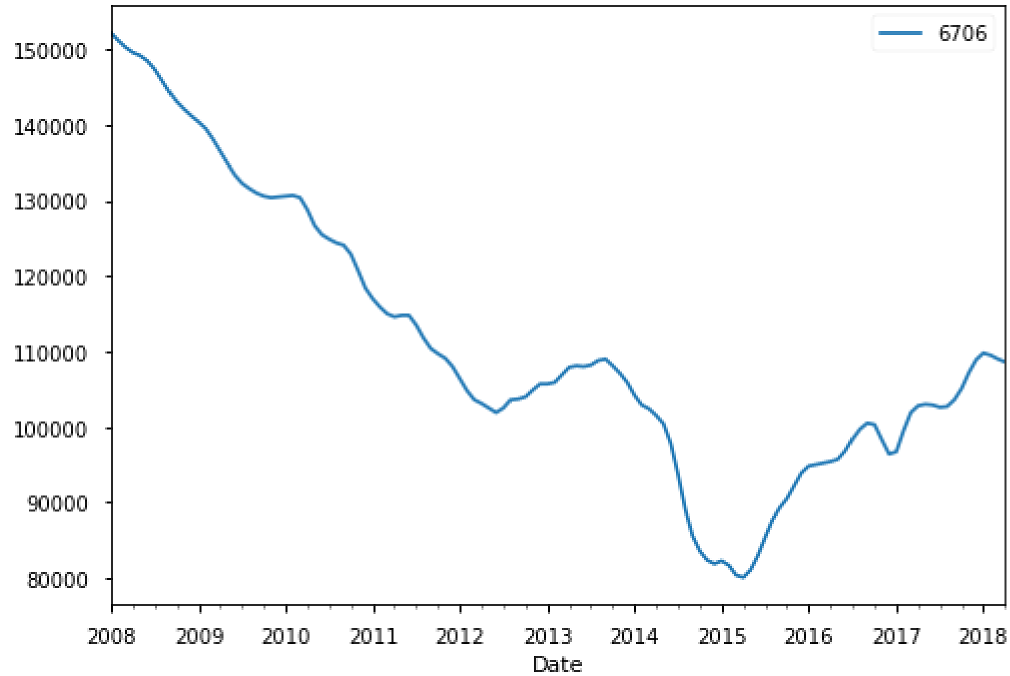
Visualizing individual ZIP Code

In [9]:

```
#Zip Code # 6706
zipcode = '6706'
```

Creating 'ts' data set containing 'ts_df' which is the Pandas Data Frame of the time-series

In [10]:
```
#Observing the Time Series
ts = ts_df[zipcode].copy()
ax = ts.plot()
ax.legend()
plt.show()
```



With the visualization above we can understand how home values in a ZIP Code have moved month over month and year over year. We are ready to navigate Time-Series deeper and set up for our predicting models.

1.1.4 Set up for Modeling - Conceptual Soundness

1.1.4.1 Understanding Time Series, Lags, Components, ACF & PACF

Components of Time-Series

Time Series have major components that can affect the lags or how the lags are shown, such as:

- Trend component.
- Seasonal component.
- Cyclical component.
- Irregular component.

To make sure that these don't affect the model, the goal is make the trend on these lags more even. In modeling I can then difference the trend or transform the trend to make the lags more even.

Lags

To understand Time-Series we are introduced first to the concept of lags. A lag is the period of time between one time series index and another one. A lag is value of time gap being considered.

A lag 1 autocorrelation is the correlation between values that are one time period apart. More generally, a lag k autocorrelation is the correlation between values that are k time periods apart. The number of lags is typically small of 1 or 2 lags. For the purpose of this project,given that this is montly data, my approach is 20 lags (usually the appropriate lags for monthly data is 6, 12 or 24 lags, depending on sufficient data points and for quarterly data, 1 to 8 lags). This concept will play a part in understanding the components of Time-Series.

Time series data can exhibit a variety of patterns, and it is often helpful to split a time series into the components explained above, each representing an underlying pattern category. To do this, we have to understand Auto Correlations and Partial Auto Correlations.

1.1.4.2 ACF & PACF of Time-Series

- ACF

Autocorrelation is a measure of how much the data sets at one point in time influences data sets at a later point in time- ACF seeks to identify how correlated the values in a time series are with each other.

The ACF starts at a lag of 0, which is the correlation of the time series with itself and therefore results in a correlation of 1. The ACF plots the correlation coefficient against the lag, which is measured in terms of a number of periods or units. In essence, its a measure of the link between the present and the past, therefore it helps us identify the moving average.

- PACF

Partial Autocorrelation (PACF) is a measure, that can plot the partial correlation coefficients between the series and lags of itself. In general, the "partial" correlation between two variables is the amount of correlation between them, which is not explained by their mutual correlations with a specified set of other variables.

In general, the "partial" correlation between two variables is the amount of correlation between them which is not explained by their mutual correlations with a specified set of other variables. PACF therefore helps us identify the Auto regressive order. PACF measures directs effects a.k.a Auto Regressive.

WHAT IS THE DIFFERENCE BETWEEN ACF and PACF ?

Partial autocorrelation function (PACF) gives the partial correlation of a stationary time series with its own lagged values, regressed the values of the time series at all shorter lags. It contrasts with the autocorrelation function, which does not control for other lags.

Both, ACF and PACF can provide valuable insights into the behaviour of time series data. They are often used to decide the number of Autoregressive (AR) and Moving Average (MA) lags for the ARIMA models. Moreover, they can also help detect any seasonality within the data. The correct application and interpretation are essential in extracting useful information from the ACF and PACF plots.

ACF and PACF can provide valuable insights into the behaviour of time series data. They are often used to decide the number of Autoregressive (AR) and Moving Average (MA) lags for the ARIMA models. Moreover, they can also help detect any seasonality within the data. The correct application and interpretation are essential in extracting useful information from the ACF and PACF plots.

1.1.4.3 STATS MODELS- Coding for ACF & PACF

First, we'll be importing libraries that will help up to this effect followed by plotting and visualization to understand the time-series data.

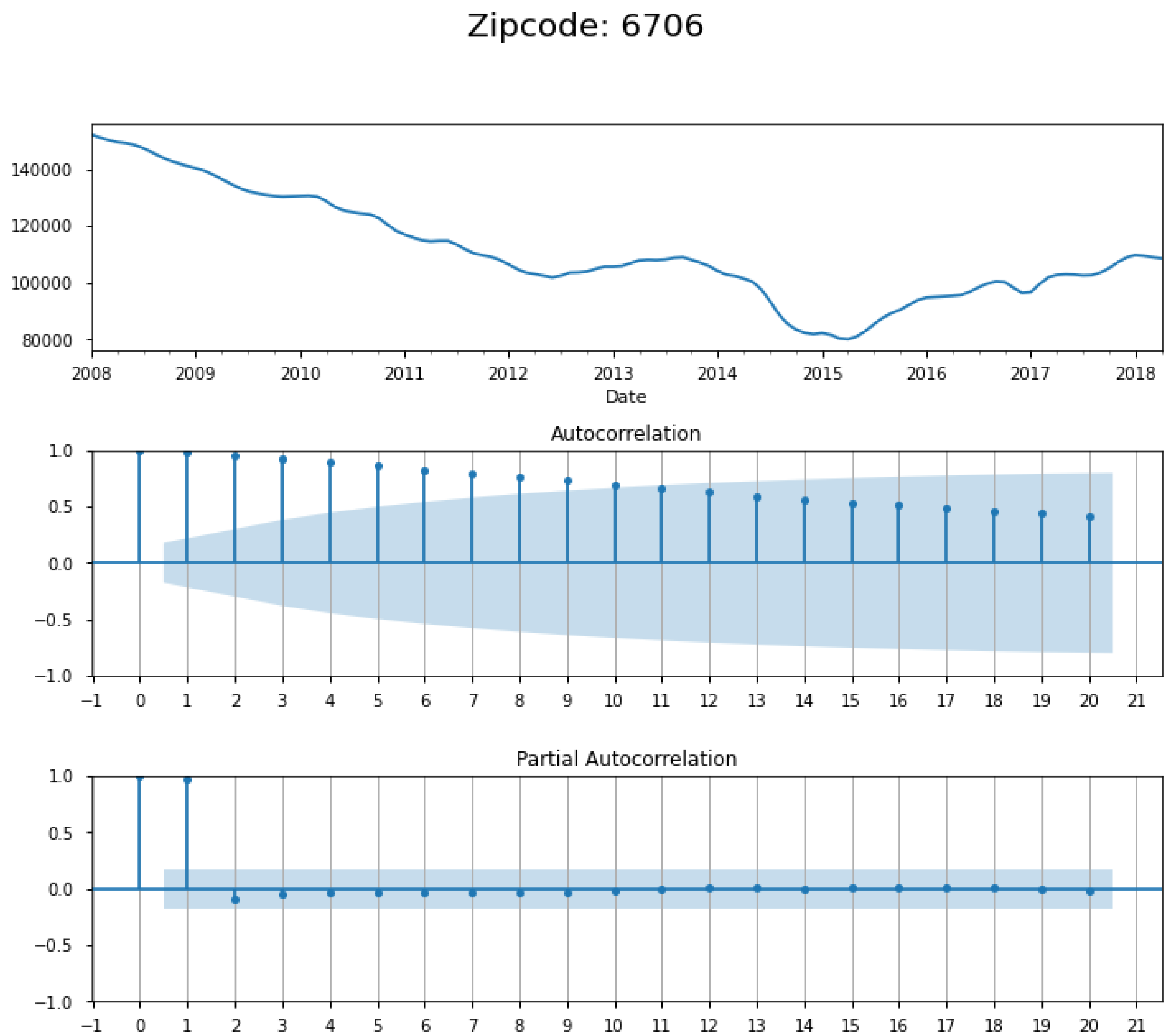
1.1.4.4 Importing relevant libraries

In [11]:
```
#For time series decomposition season decompose
from statsmodels.tsa.seasonal import seasonal_decompose
#Statsmodels for plotting the acf and pacf
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
#Pandas plotting import
from pandas.plotting import autocorrelation_plot,lag_plot
#Defining plot
def plot_acf_pacf(ts, figsize=(10,8),lags=24):

    fig,ax = plt.subplots(nrows=3,
                          figsize=figsize)

    ## Plot ts
    ts.plot(ax=ax[0])

    ## Plot acf, pavf
    plot_acf(ts,ax=ax[1],lags=lags)
    plot_pacf(ts, ax=ax[2],lags=lags)
    fig.tight_layout()

    fig.suptitle(f"Zipcode: {ts.name}",y=1.1,fontsize=20)

    for a in ax[1:]:
        a.xaxis.set_major_locator(mpl.ticker.MaxNLocator(min_n_ticks=la
        a.xaxis.grid()
    return fig,ax
```

1.1.4.5 Plots ACF & PACF

My approach with annual data is 20 lags. The number of lags is typically small, 1 or 2 lags. For quarterly data, 1 to 8 lags is appropriate, and for monthly data, 6, 12 or 24 lags can be used given sufficient data points.

In [12]:
```
plot_acf_pacf(ts,lags=20);
```



Next, Differencing- technique to transform time-series

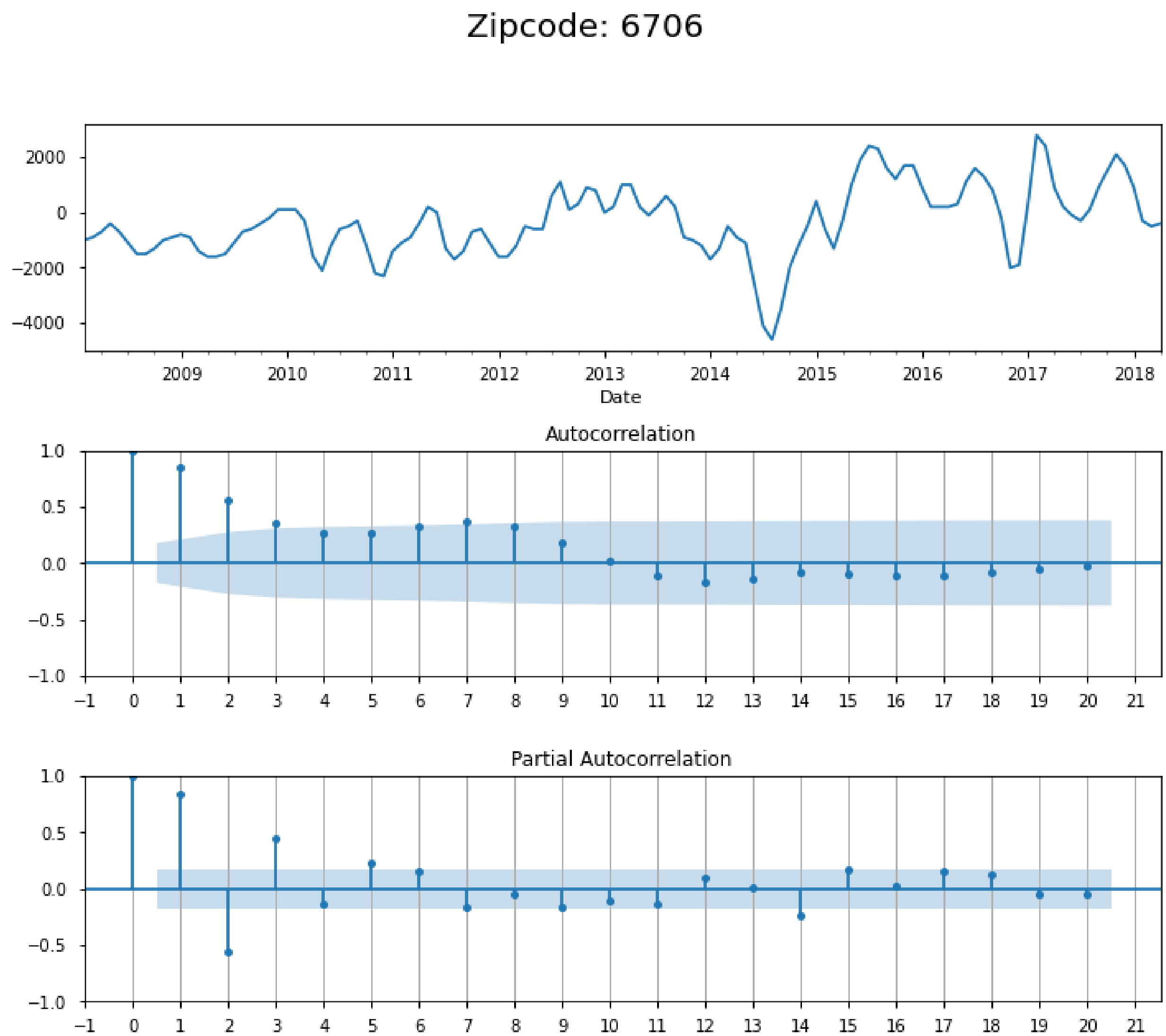
1.1.5 Differencing Transformation

Differencing is a technique to transform a non-stationary time series into a stationary one. It involves subtracting the current value of the series from the previous one, or from a lagged value. It can be used to remove the series dependence on time like trends and seasonality. This is an important step in preparing the data used in ARIMA Modeling. To do this we can code a new plot showing the differencing applied. Let's also understand the sub-components of Auto Correlation and Partial Autocorrelation.

It can be used to remove the series dependence on time like trends and seasonality. This is an important step in preparing the data used in ARIMA Modeling. To do this we can code a new plot showing the differencing applied.

d= 1 below, is a parameter that refers to the number of differencing transformations required by the time series to get stationary. By making the time series stationary I have basically made the mean and variance constant over time. It is easier to predict when the series is stationary.

In [13]:
```
#Coding ts.diff Differencing
d = 1
plot_acf_pacf(ts.diff(d).dropna(),lags=20);
```



Above we have detrended the series. I can now select parameters and run the first model.

Both of these functions (ACF and PACF) measure how correlated the data at time t is to its past values t-1,t-2,… There is one crucial difference, however. The ACF also measures indirect correlation up to the lag in question, while PCAF does not.

1.1.6 Modeling

I'll be doing 2 models, one with selected parameters (I'll explain those parameters), and the second one with parameters provided by the model finding best parameters.

1.1.6.1 Selecting Parameters

ARIMA models are made up of three different parameters or terms:

- d: The degree of differencing. Is a parameter that refers to the number of differencing transformations required by the time series to get stationary.
- p: The order of the auto-regressive (AR) model (i.e., the number of lag observations). A time series is considered AR when previous values in the time series are very predictive of later values.
- q: The order of the moving average (MA) model. This is essentially the size of the "window" function over time series data.

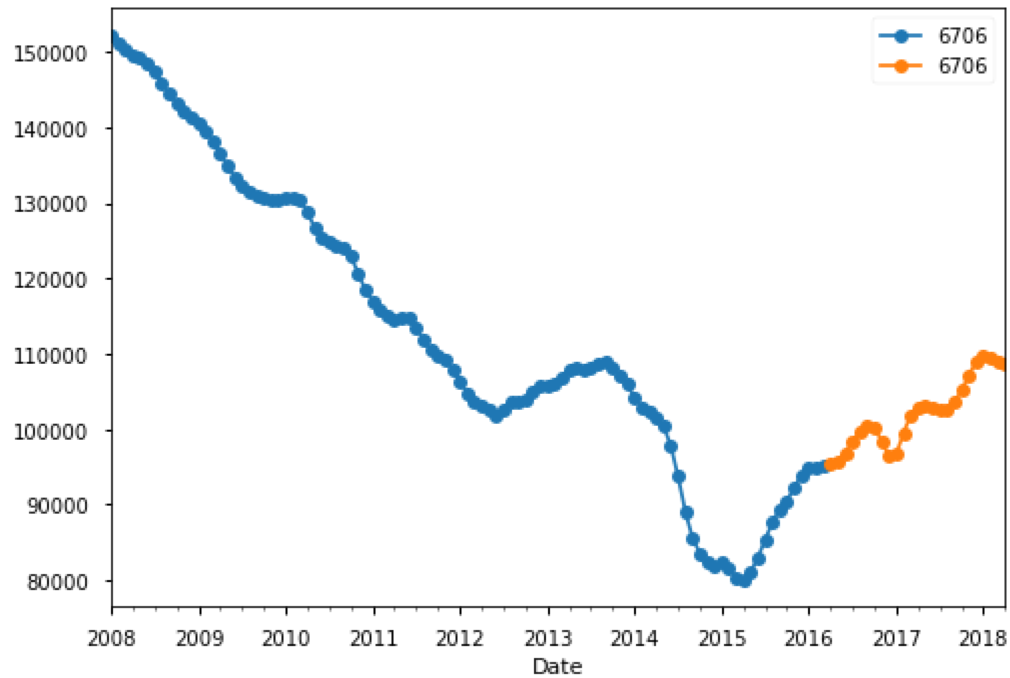
1.1.7 Model 1

Selecting parameters

In [14]:
```
# selected params
#d of 1. Number of differencing transformations to make the ts stationa
d = 1
#p of 1. Order of Auto Regressive model (number of lags)
p = 1
#q of 1. Order of Moving Average. The "window sizes" of the time series
q = 1
```

Train Test Split

In [15]:
```
#selecting a training size
train_size = 0.8
#multiply train size by len of ts
split_idx = round(len(ts)* train_size)
split_idx

## split train/test for train 80% and test 20%
train = ts.iloc[:split_idx]
test = ts.iloc[split_idx:]

## Visualize split
fig,ax= plt.subplots()
kws = dict(ax=ax,marker='o')
train.plot(**kws)
test.plot(**kws)
ax.legend(bbox_to_anchor=[1,1])
plt.show()
```



Above, we see this ZIP Code's train test split

Running Model on Statsmodels (SARIMAX)

SARIMAX, is an extension of the ARIMA class of models. ARIMA models compose 2 parts: the autoregressive term (AR) and the moving-average term (MA). AR views the value at one time just as a weighted sum of past values. The MA model takes that same value also as a weighted sum but of past residuals. Overall, ARIMA is a very good model. However, it cannot handle seasonality, thus SARIMAX is used in this model.

Applying SARIMAX Model to Train data

In [16]:
```
#Using SARIMAX because it is better to use on seasonal data
from statsmodels.tsa.statespace.sarimax import SARIMAX

## Baseline model from eye-balled params
model = SARIMAX(train,order=(p,d,q),).fit()
display(model.summary())
model.plot_diagnostics();
plt.show()
```

```
/Users/jonax/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sta
tsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency informatio
n was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/Users/jonax/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sta
tsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency informatio
n was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
```

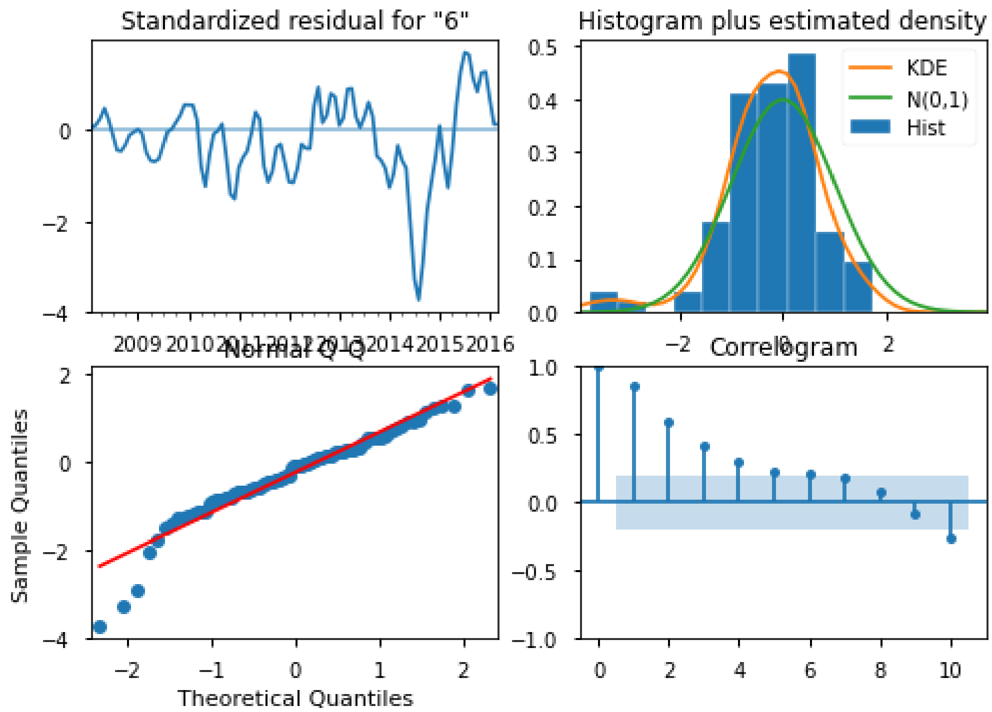
SARIMAX Results

Dep. Variable:	6706	**No. Observations:**	99
Model:	SARIMAX(1, 1, 1)	**Log Likelihood**	-833.096
Date:	Thu, 06 Jul 2023	**AIC**	1672.192
Time:	14:38:28	**BIC**	1679.947
Sample:	01-01-2008	**HQIC**	1675.329
	- 03-01-2016		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.9769	0.010	97.349	0.000	0.957	0.997
ma.L1	-0.9999	0.101	-9.903	0.000	-1.198	-0.802
sigma2	1.558e+06	6.13e-08	2.54e+13	0.000	1.56e+06	1.56e+06

Ljung-Box (L1) (Q):	74.21	**Jarque-Bera (JB):**	38.95
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	9.74	**Skew:**	-0.95
Prob(H) (two-sided):	0.00	**Kurtosis:**	5.44

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 1.99e+28. Standard errors may be unstable.



Understanding Charts Above

- Quantile Plots:

Commonly known as Q-Q Plots, It helps answer the question: "if the set of observations approximately normally distributed?". It is a plot of the quantiles of the first data set against the quantiles of the second data set (Sample vs. Theoritical in this case). Shows you how reliable predictions are within standard deviations. Our Mean Price, is fairly good at predictions within value.

- Histogram plus Estimated Density (KDE)

Undelying distribution for this data. Created bins for the data, and count the number of values creating a histogram. The KDE is the smooth out continous version of that data distribution. Allowing to estimate the probability density function. And the PDF, allows us to find the chances that the value of a random variable will occur within a range of values that you specify. More specifically, a PDF is a function where its integral for an interval provides the probability of a value occurring in that interval.

- Correlogram

A correlogram is a plot of autocorrelations . In time series data, looking at correlations between succesive correlations over time, that are periods apart (it can be 1 period or several periods apart)/For example a data group or point that you observe a month ago or a point you observed two months ago. The horizontal axis is the timeline. The blue shadows are the thresholds. The bars above the shadows are autocorrelations that are statistically significant, it is not 0 and they are

It answers the question: 1) Is that Data Random? It is when not all points are above threshold. 2) Is there a trend in the data? There will be a trend, when the autocorrelations coeffiecient do not fall below the critical upper limit (upper limit) at any lag . If there is a trend the data is not stationary.

In [17]:

```
#Checking for the len of test
len(test)
```

Out[17]: 25

Forecasting on Test Data

In [18]:

```
#Importing Library
from sklearn import metrics

#Obtaining forecast tools and set forecast code on the testing data
forecast = model.get_forecast(steps=len(test))
```

In [19]:

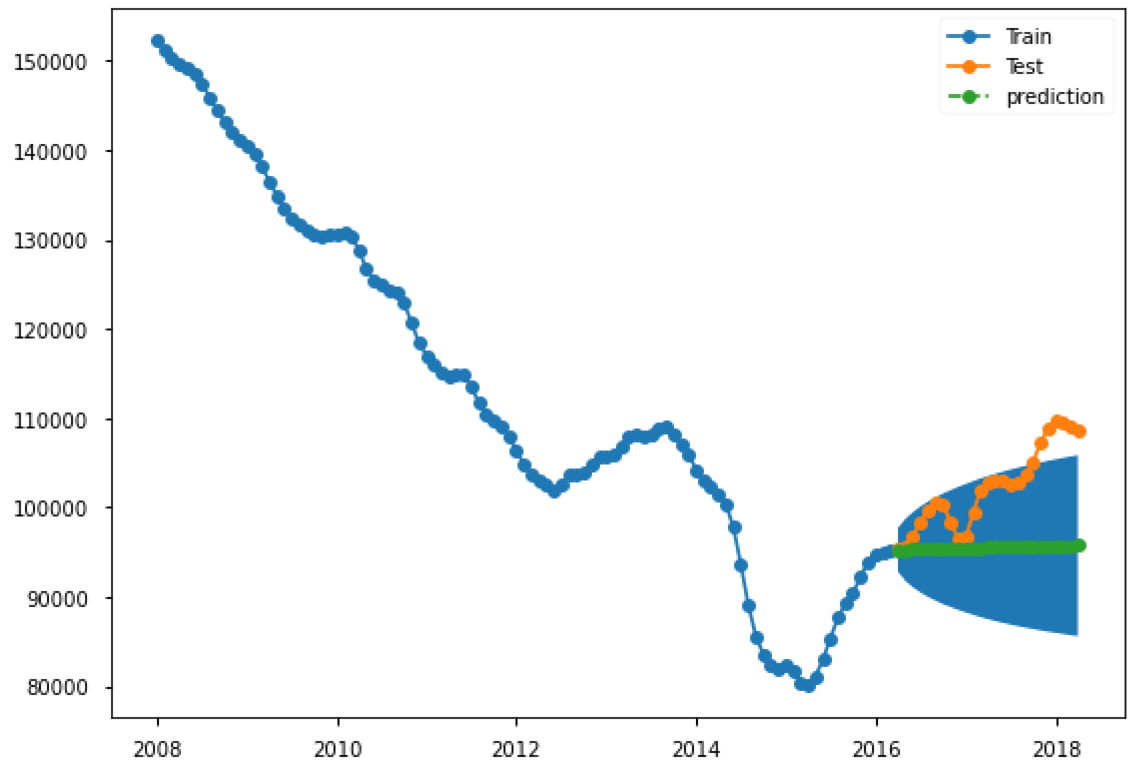
```
#Defining Forecast and applying it to data set test on predicted mean
def forecast_to_df(forecast,zipcode):
    test_pred = forecast.conf_int()
    test_pred[zipcode] = forecast.predicted_mean
    test_pred.columns = ['lower','upper','prediction']
    return test_pred


pred_df = forecast_to_df(forecast,zipcode)
```

Plotting Model 1

In [20]:

```
#Defining Plot
def plot_train_test_pred(train,test,pred_df):
    fig,ax = plt.subplots()
    kws = dict(marker='o')

    ax.plot(train,label='Train',**kws)
    ax.plot(test,label='Test',**kws)
    ax.plot(pred_df['prediction'],label='prediction',ls='--',**kws)

    ax.fill_between(x=pred_df.index,y1=pred_df['lower'],y2=pred_df['upp
    ax.legend(bbox_to_anchor=[1,1])
    fig.tight_layout()
    return fig,ax
```

In [21]:
```
plot_train_test_pred(train,test,pred_df)
plt.show()
```



1.1.7.1 Conclusion on Model 1

A flat prediction line once we run the model. Let's update the parameters and find optimal parameters for prediction.

1.1.8 Model 2

Finding optimal parameters

I want to identify the optimal parameters for my model. Pmdarima's auto_arima function is very useful when building an ARIMA model as it helps us identify the most optimal p,d,q parameters and return a fitted model.

In [22]:
```
#@!pip install pmdarima
```

In [23]:
```
#importing libraries
from pmdarima.arima import auto_arima
```

Applying to Train Data

In [24]:
```
auto_model = auto_arima(train,start_p=0,start_q=0)
display(auto_model.summary())
auto_model.plot_diagnostics();
```

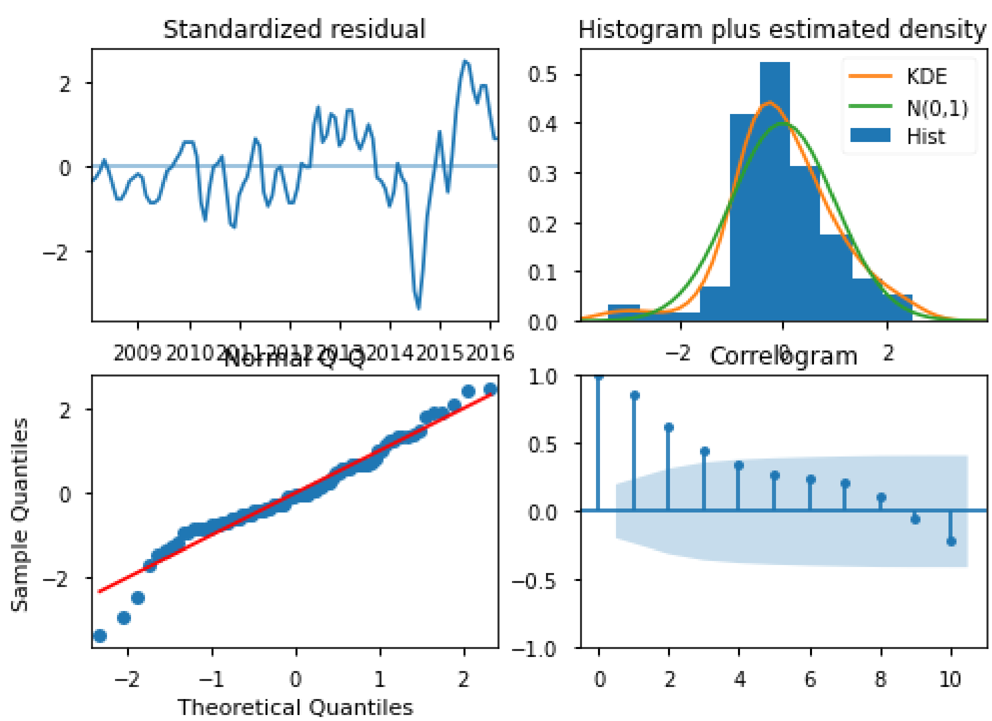
SARIMAX Results

Dep. Variable:	y	**No. Observations:**	99
Model:	SARIMAX(0, 1, 0)	**Log Likelihood**	-833.565
Date:	Thu, 06 Jul 2023	**AIC**	1671.131
Time:	14:38:35	**BIC**	1676.301
Sample:	01-01-2008	**HQIC**	1673.222
	- 03-01-2016		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	-581.6327	121.477	-4.788	0.000	-819.723	-343.542
sigma2	1.431e+06	1.6e+05	8.956	0.000	1.12e+06	1.74e+06

Ljung-Box (L1) (Q):	75.43	**Jarque-Bera (JB):**	7.58
Prob(Q):	0.00	**Prob(JB):**	0.02
Heteroskedasticity (H):	7.58	**Skew:**	-0.19
Prob(H) (two-sided):	0.00	**Kurtosis:**	4.31

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



Type *Markdown* and LaTeX: α^2

In [25]:
```
#organize panda data frame into date range , with 10 periods, frequency
pd.date_range(train.index[-1], periods=10,freq='M')

```

Out[25]:
```
DatetimeIndex(['2016-03-31', '2016-04-30', '2016-05-31', '2016-06-30',
               '2016-07-31', '2016-08-31', '2016-09-30', '2016-10-31',
               '2016-11-30', '2016-12-31'],
              dtype='datetime64[ns]', freq='M')
```

Date ranges (pd.date_range) is like a "date ruler". You have a start and end time, the frequency of how I'd like to split the dates by.

In [26]:
```
#showing the index of train data
train.index
```

Out[26]:
```
DatetimeIndex(['2008-01-01', '2008-02-01', '2008-03-01', '2008-04-01',
               '2008-05-01', '2008-06-01', '2008-07-01', '2008-08-01',
               '2008-09-01', '2008-10-01', '2008-11-01', '2008-12-01',
               '2009-01-01', '2009-02-01', '2009-03-01', '2009-04-01',
               '2009-05-01', '2009-06-01', '2009-07-01', '2009-08-01',
               '2009-09-01', '2009-10-01', '2009-11-01', '2009-12-01',
               '2010-01-01', '2010-02-01', '2010-03-01', '2010-04-01',
               '2010-05-01', '2010-06-01', '2010-07-01', '2010-08-01',
               '2010-09-01', '2010-10-01', '2010-11-01', '2010-12-01',
               '2011-01-01', '2011-02-01', '2011-03-01', '2011-04-01',
               '2011-05-01', '2011-06-01', '2011-07-01', '2011-08-01',
               '2011-09-01', '2011-10-01', '2011-11-01', '2011-12-01',
               '2012-01-01', '2012-02-01', '2012-03-01', '2012-04-01',
               '2012-05-01', '2012-06-01', '2012-07-01', '2012-08-01',
               '2012-09-01', '2012-10-01', '2012-11-01', '2012-12-01',
               '2013-01-01', '2013-02-01', '2013-03-01', '2013-04-01',
               '2013-05-01', '2013-06-01', '2013-07-01', '2013-08-01',
               '2013-09-01', '2013-10-01', '2013-11-01', '2013-12-01',
               '2014-01-01', '2014-02-01', '2014-03-01', '2014-04-01',
               '2014-05-01', '2014-06-01', '2014-07-01', '2014-08-01',
               '2014-09-01', '2014-10-01', '2014-11-01', '2014-12-01',
               '2015-01-01', '2015-02-01', '2015-03-01', '2015-04-01',
               '2015-05-01', '2015-06-01', '2015-07-01', '2015-08-01',
               '2015-09-01', '2015-10-01', '2015-11-01', '2015-12-01',
               '2016-01-01', '2016-02-01', '2016-03-01'],
              dtype='datetime64[ns]', name='Date', freq=None)
```

In [27]:
```
#predictive models mean
pred_mean,pred_conf_int = auto_model.predict(return_conf_int=True)
pred_mean
```

Out[27]:
```
2016-04-01    94618.367347
2016-05-01    94036.734694
2016-06-01    93455.102041
2016-07-01    92873.469388
2016-08-01    92291.836735
2016-09-01    91710.204082
2016-10-01    91128.571429
2016-11-01    90546.938776
2016-12-01    89965.306122
2017-01-01    89383.673469
Freq: MS, dtype: float64
```

1.1.8.1 GridSearch Hyperparameter

Grid search The traditional way of performing hyperparameter optimization has been grid search, or a parameter sweep, which is simply an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm. A grid search algorithm must be guided by some performance metric, typically measured by cross-validation on the training set or evaluation on a hold-out validation set.

In [28]:
```
#Grid Parameters
pred_df = pd.DataFrame({'pred':pred_mean,
                        'conf_int_lower':pred_conf_int[:,0],
                        'conf_int_upper':pred_conf_int[:,1]},
                        index= pd.date_range(test.index[0],
                                             periods=10,freq='M'))
# auto_model.conf_int()
pred_df
```

Out[28]:

	pred	conf_int_lower	conf_int_upper
2016-04-30	NaN	92274.031616	96962.703077
2016-05-31	NaN	90721.343309	97352.126079
2016-06-30	NaN	89394.593446	97515.610636
2016-07-31	NaN	88184.797927	97562.140849
2016-08-31	NaN	87049.742679	97533.930790
2016-09-30	NaN	85967.777756	97452.630407
2016-10-31	NaN	84926.042096	97331.100761
2016-11-30	NaN	83916.156006	97177.721545
2016-12-31	NaN	82932.298931	96998.313314
2017-01-31	NaN	81970.232961	96797.113978

In [29]:
```
train.index[-1]
```

Out[29]:
```
Timestamp('2016-03-01 00:00:00')
```

In [30]:
```
auto_model.get_params()
```

Out[30]:
```
{'maxiter': 50,
 'method': 'lbfgs',
 'order': (0, 1, 0),
 'out_of_sample_size': 0,
 'scoring': 'mse',
 'scoring_args': {},
 'seasonal_order': (0, 0, 0, 0),
 'start_params': None,
 'suppress_warnings': True,
 'trend': None,
 'with_intercept': True}
```

Mean Square Error (Test)

In [31]:
```
from sklearn.metrics import mean_squared_error
from math import sqrt
```

In [32]:
```
auto_model = auto_arima(test, m=12)
```

In [33]:
```
test.info()
```

```
<class 'pandas.core.series.Series'>
DatetimeIndex: 25 entries, 2016-04-01 to 2018-04-01
Series name: 6706
Non-Null Count  Dtype
--------------  -----
25 non-null     float64
dtypes: float64(1)
memory usage: 400.0 bytes
```

In [34]:
```
#organize panda data frame into date range , with 25 periods, frequency
pd.date_range(test.index[0], periods=25,freq='M')
```

Out[34]:
```
DatetimeIndex(['2016-04-30', '2016-05-31', '2016-06-30', '2016-07-31',
               '2016-08-31', '2016-09-30', '2016-10-31', '2016-11-30',
               '2016-12-31', '2017-01-31', '2017-02-28', '2017-03-31',
               '2017-04-30', '2017-05-31', '2017-06-30', '2017-07-31',
               '2017-08-31', '2017-09-30', '2017-10-31', '2017-11-30',
               '2017-12-31', '2018-01-31', '2018-02-28', '2018-03-31',
               '2018-04-30'],
              dtype='datetime64[ns]', freq='M')
```

In [35]:
```
#showing test index
test_df = test.reset_index()
test_df= test_df.rename(columns= {'index': 'Date'})
test_df.head()
test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25 entries, 0 to 24
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    25 non-null     datetime64[ns]
 1   6706    25 non-null     float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 528.0 bytes
```

In [36]:
```
test.head()
```

Out[36]:
```
Date
2016-04-01    95400.0
2016-05-01    95700.0
2016-06-01    96800.0
2016-07-01    98400.0
2016-08-01    99700.0
Name: 6706, dtype: float64
```

In [37]:
```
test.index[-1]
```

Out[37]:
```
Timestamp('2018-04-01 00:00:00')
```

In [38]:
```
test_df.set_index('Date')
```

Out[38]:

Date	6706
2016-04-01	95400.0
2016-05-01	95700.0
2016-06-01	96800.0
2016-07-01	98400.0
2016-08-01	99700.0
2016-09-01	100500.0
2016-10-01	100300.0
2016-11-01	98300.0
2016-12-01	96400.0
2017-01-01	96700.0
2017-02-01	99500.0
2017-03-01	101900.0
2017-04-01	102800.0
2017-05-01	103000.0
2017-06-01	102900.0
2017-07-01	102600.0
2017-08-01	102700.0
2017-09-01	103600.0
2017-10-01	105100.0
2017-11-01	107200.0
2017-12-01	108900.0
2018-01-01	109800.0
2018-02-01	109500.0
2018-03-01	109000.0
2018-04-01	108600.0

In [39]:
```
#predictive models mean
pred_mean_test,pred_conf_int = auto_model.predict(return_conf_int=True)
pred_test_df= pred_mean_test.reset_index()
pred_test_df= pred_test_df.rename(columns= {'index': 'Date'})
pred_test_df= pred_test_df.rename(columns= {0: '6706'})
pred_test_df.head()
```

Out[39]:

	Date	6706
0	2018-05-01	109150.0
1	2018-06-01	109700.0
2	2018-07-01	110250.0
3	2018-08-01	110800.0
4	2018-09-01	111350.0

In [40]:
```
print (pred_conf_int)
```

```
[[106855.22336598 111444.77663402]
 [106454.69576155 112945.30423845]
 [106275.33027786 114224.66972214]
 [106210.44673196 115389.55326804]
 [106218.72345316 116481.27654684]
 [106278.96817299 117521.03182701]
 [106378.59171195 118521.40828805]
 [106509.39152311 119490.60847689]
 [106665.67009795 120434.32990205]
 [106843.27911517 121356.72088483]]
```

In [41]:
```
pred_test_df.set_index('Date')
```

Out[41]:

	6706
Date	
2018-05-01	109150.0
2018-06-01	109700.0
2018-07-01	110250.0
2018-08-01	110800.0
2018-09-01	111350.0
2018-10-01	111900.0
2018-11-01	112450.0
2018-12-01	113000.0
2019-01-01	113550.0
2019-02-01	114100.0

In [42]:
```
pred_test_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    10 non-null     datetime64[ns]
 1   6706    10 non-null     float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 288.0 bytes
```

In [43]:
```
pd.date_range(pred_mean_test.index[0], periods=25,freq='M')
```

Out[43]:
```
DatetimeIndex(['2018-05-31', '2018-06-30', '2018-07-31', '2018-08-31',
               '2018-09-30', '2018-10-31', '2018-11-30', '2018-12-31',
               '2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
               '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
               '2019-09-30', '2019-10-31', '2019-11-30', '2019-12-31',
               '2020-01-31', '2020-02-29', '2020-03-31', '2020-04-30',
               '2020-05-31'],
              dtype='datetime64[ns]', freq='M')
```

In [44]:
```
test_df.merge(pred_test_df, how= 'outer', indicator= True)._merge.value
```

Out[44]:
```
_merge
left_only     25
right_only    10
both           0
Name: count, dtype: int64
```

In [45]:
```
mse_df= test_df.merge(pred_test_df, how= 'outer', indicator= True)
```

In [46]:
```
mse_df.set_index ('Date')
```

Out[46]:

	6706	_merge
Date		
2016-04-01	95400.0	left_only
2016-05-01	95700.0	left_only
2016-06-01	96800.0	left_only
2016-07-01	98400.0	left_only
2016-08-01	99700.0	left_only
2016-09-01	100500.0	left_only
2016-10-01	100300.0	left_only
2016-11-01	98300.0	left_only
2016-12-01	96400.0	left_only
2017-01-01	96700.0	left_only
2017-02-01	99500.0	left_only
2017-03-01	101900.0	left_only
2017-04-01	102800.0	left_only
2017-05-01	103000.0	left_only
2017-06-01	102900.0	left_only
2017-07-01	102600.0	left_only
2017-08-01	102700.0	left_only
2017-09-01	103600.0	left_only
2017-10-01	105100.0	left_only
2017-11-01	107200.0	left_only
2017-12-01	108900.0	left_only
2018-01-01	109800.0	left_only
2018-02-01	109500.0	left_only
2018-03-01	109000.0	left_only
2018-04-01	108600.0	left_only
2018-05-01	109150.0	right_only
2018-06-01	109700.0	right_only
2018-07-01	110250.0	right_only
2018-08-01	110800.0	right_only
2018-09-01	111350.0	right_only
2018-10-01	111900.0	right_only
2018-11-01	112450.0	right_only
2018-12-01	113000.0	right_only

	6706	_merge
Date		
2019-01-01	113550.0	right_only
2019-02-01	114100.0	right_only

In [ ]:

```

```

In [47]:

```
#y_forecasted = pred            #what is this vector ?
#y_truth = y['1998-01-01':]   # What is this vector ?
# Compute the mean square error
#mse = ((y_forecasted - y_truth) ** 2).mean()
#print('The Mean Squared Error of our forecasts is {}'.format(round(mse
```

In [48]:

```
#calculating the MSE and RMSE
expected =  test_df.tail(10)['6706']    # what is this vector value?
forecast =  pred_test_df ['6706']
mean_squared_error( expected, forecast, squared= False)
```

Out[48]: 5148.422088368436

In [49]:

```
#calculating the MSE and RMSE
expected =  test_df.tail(10)['6706']    # what is this vector value?
forecast =  pred_test_df ['6706']
mse= mean_squared_error( expected, forecast)
RMSE= sqrt (mse)
print (RMSE)
```

5148.422088368436

In [50]:

```
y_min=test_df.min()['6706']

```

In [51]:

```
y_max=test_df.max() ['6706']
```

In [52]:

```
Norm_divide= y_max-y_min
```

In [53]:

```
RMSE/Norm_divide
```

Out[53]: 0.3575293116922525

In [54]:

```
#another way of finding rmse
#def find_rmse(model, test=test):
#    y_hat = model.predict(type='levels')
#    return np.sqrt(mean_squared_error(train_data, y_hat))
```

In [ ]:

```

```

1.2 Best Model Plot

In [55]:
```
best_model = SARIMAX(ts,order=auto_model.order,
                     seasonal_order=auto_model.seasonal_order).fit()
display(best_model.summary())
best_model.plot_diagnostics();
```

```
/Users/jonax/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sta
tsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency informatio
n was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/Users/jonax/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sta
tsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency informatio
n was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
```

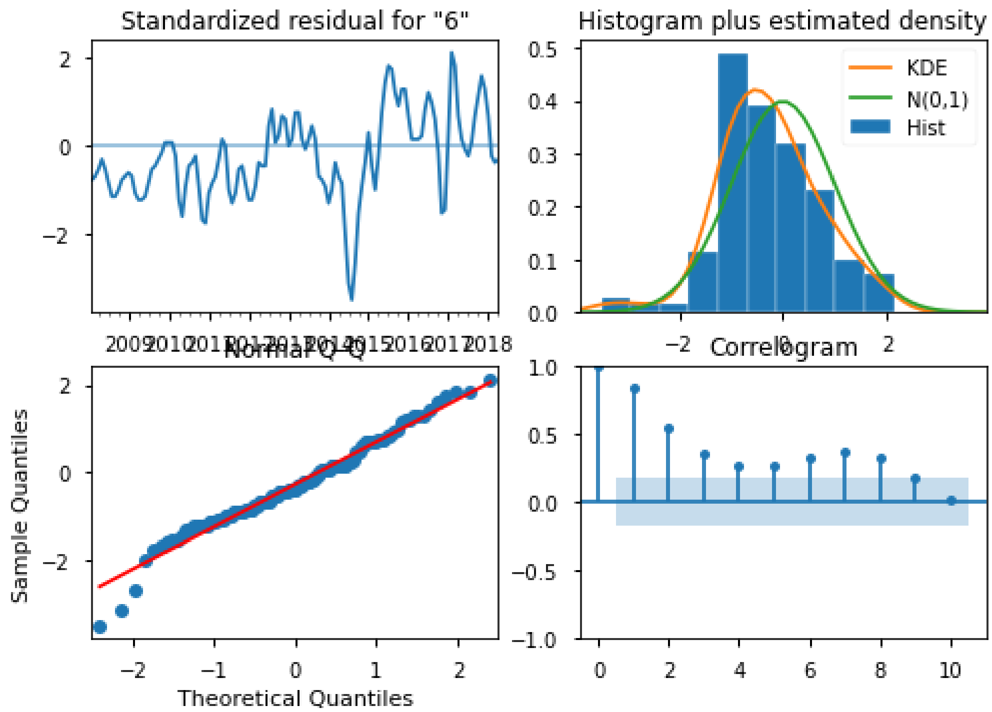
SARIMAX Results

Dep. Variable:	6706	**No. Observations:**	124
Model:	SARIMAX(0, 1, 0)	**Log Likelihood**	-1058.117
Date:	Thu, 06 Jul 2023	**AIC**	2118.234
Time:	14:38:44	**BIC**	2121.046
Sample:	01-01-2008	**HQIC**	2119.376
	- 04-01-2018		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
sigma2	1.722e+06	1.85e+05	9.330	0.000	1.36e+06	2.08e+06

Ljung-Box (L1) (Q):	90.11	**Jarque-Bera (JB):**	3.49
Prob(Q):	0.00	**Prob(JB):**	0.17
Heteroskedasticity (H):	1.38	**Skew:**	-0.12
Prob(H) (two-sided):	0.31	**Kurtosis:**	3.79

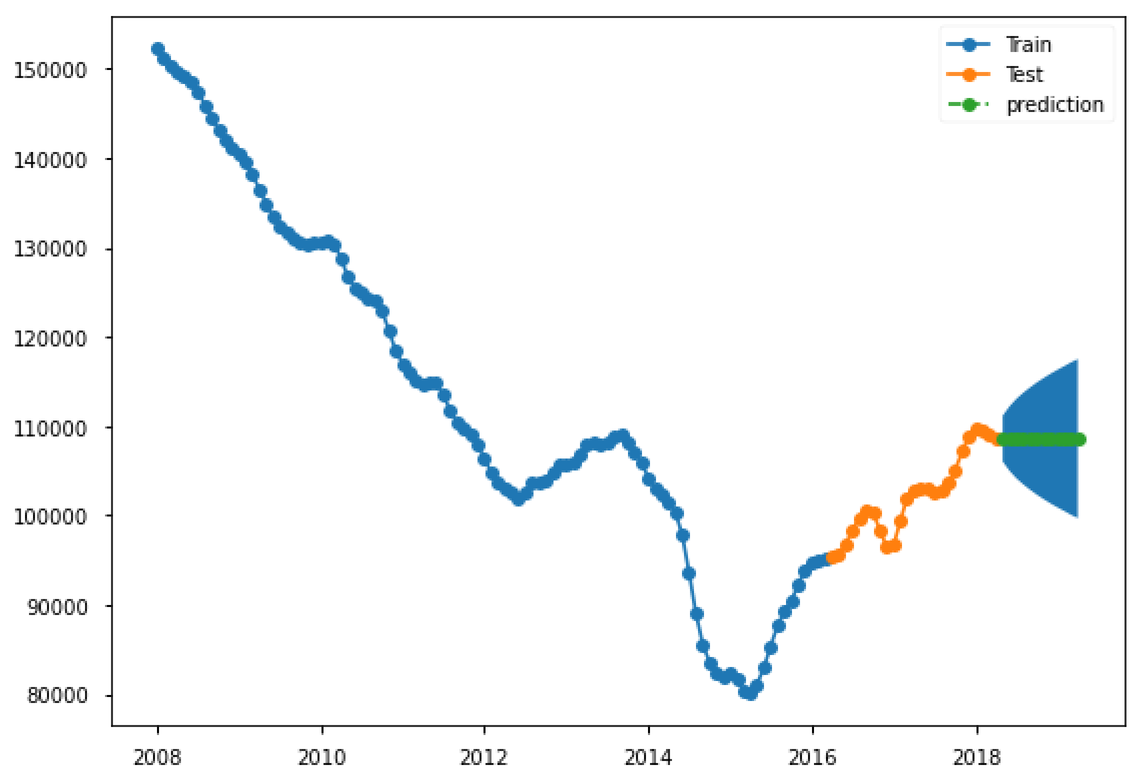
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).



1.2.1 Forecast

In [56]:
```
pred = best_model.get_forecast(steps=12)#start=test.index[0],end=test.i
pred_df = forecast_to_df(pred,zipcode)
display(plot_train_test_pred(train,test,pred_df));
plt.show()
```

(<Figure size 576x396 with 1 Axes>, <AxesSubplot:>)



1.2.1.1 Loop Best Model Forecast for the 13 best ROI ZIP Codes

In [57]:
```
RESULTS = {}

for zipcode in zipcode_list:
    print(zipcode)

    ## Make empty dict for ZIP data
    zipcode_d = {}

    ## Copy Time Series
    ts = ts_df[zipcode].copy()


    ## Train Test Split Index
    train_size = 0.8
    split_idx = round(len(ts)* train_size)

    ## Split
    train = ts.iloc[:split_idx]
    test = ts.iloc[split_idx:]


    ## Get best params using auto_arima
    gridsearch_model = auto_arima(ts,start_p=0,start_q=0)
    best_model = SARIMAX(ts,order=gridsearch_model.order,
                         seasonal_order=gridsearch_model.seasonal_order).fi



    ## Get predictions
    pred  = best_model.get_forecast(steps=36)#start=test.index[0],end=t
    pred_df = forecast_to_df(pred,zipcode)

    # RMSE Fitting into all ZIP Codes
    RMSE = mean_squared_error(ts.tail(36), pred_df['prediction'], squar
    y_min = ts.min()
    y_max = ts.max()
    Norm_divide = y_max-y_min
    Norm = RMSE/Norm_divide
    print (f'RMSE for this ZIP Code is {Norm}')


    ## Save info to dict
    zipcode_d['pred_df'] = pred_df
    zipcode_d['model'] = best_model
    zipcode_d['train'] = train
    zipcode_d['test'] = test

    ## Display Results
    display(best_model.summary())
    plot_train_test_pred(train,test,pred_df)
    plt.xlabel('Year')
    plt.ylabel('Value in US Dollars ($)')
    plt.show()


    ## Save district dict in RESULTS
    RESULTS[zipcode] = zipcode_d
```

```
58     print('---'*20,end='\n\n')
```

```
6606
RMSE for this ZIP Code is 0.1587860300220871

/Users/jonax/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sta
tsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency informatio
n was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
/Users/jonax/opt/anaconda3/envs/learn-env/lib/python3.8/site-packages/sta
tsmodels/tsa/base/tsa_model.py:473: ValueWarning: No frequency informatio
n was provided, so inferred frequency MS will be used.
  self._init_dates(dates, freq)
```

SARIMAX Results

Dep. Variable:	6606	**No. Observations:**	124
Model:	SARIMAX(0, 2, 0)	**Log Likelihood**	-981.444
Date:	Thu, 06 Jul 2023	**AIC**	1964.889
Time:	14:38:45	**BIC**	1967.693

1.3 Top 5 Zip Codes Recommendations

In [58]:
```
%store -r melted_df
```

Zip Code 6069

In [59]:
```
melted_df.loc[melted_df['Zipcode']=='6069']

```

Out[59]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6069	Torrington	12310	0.2366	0.2412	436300.0
2008-02-01	6069	Torrington	12310	0.2366	0.2412	437000.0
2008-03-01	6069	Torrington	12310	0.2366	0.2412	436300.0
2008-04-01	6069	Torrington	12310	0.2366	0.2412	434900.0
2008-05-01	6069	Torrington	12310	0.2366	0.2412	433700.0
...	...	...	...	...	...	...
2017-12-01	6069	Torrington	12310	0.2366	0.2412	365800.0
2018-01-01	6069	Torrington	12310	0.2366	0.2412	370900.0
2018-02-01	6069	Torrington	12310	0.2366	0.2412	375900.0
2018-03-01	6069	Torrington	12310	0.2366	0.2412	386500.0
2018-04-01	6069	Torrington	12310	0.2366	0.2412	397200.0

124 rows × 6 columns

Zip Code 6610

In [60]:
```
melted_df.loc[melted_df['Zipcode']=='6610']
```

Out[60]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6610	Stamford	4717	0.2999	0.3111	216600.0
2008-02-01	6610	Stamford	4717	0.2999	0.3111	213100.0
2008-03-01	6610	Stamford	4717	0.2999	0.3111	210100.0
2008-04-01	6610	Stamford	4717	0.2999	0.3111	207500.0
2008-05-01	6610	Stamford	4717	0.2999	0.3111	205100.0
...	...	...	...	...	...	...
2017-12-01	6610	Stamford	4717	0.2999	0.3111	160400.0
2018-01-01	6610	Stamford	4717	0.2999	0.3111	162200.0
2018-02-01	6610	Stamford	4717	0.2999	0.3111	163900.0
2018-03-01	6610	Stamford	4717	0.2999	0.3111	165900.0
2018-04-01	6610	Stamford	4717	0.2999	0.3111	167300.0

124 rows × 6 columns

Zip Code 6330

In [61]:
```
melted_df.loc[melted_df['Zipcode']=='6330']
```

Out[61]:

	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
Date						
2008-01-01	6330	New London	12817	0.1107	0.2078	232100.0
2008-02-01	6330	New London	12817	0.1107	0.2078	230400.0
2008-03-01	6330	New London	12817	0.1107	0.2078	228900.0
2008-04-01	6330	New London	12817	0.1107	0.2078	227800.0
2008-05-01	6330	New London	12817	0.1107	0.2078	226700.0
...	...	...	...	...	...	...
2017-12-01	6330	New London	12817	0.1107	0.2078	194700.0
2018-01-01	6330	New London	12817	0.1107	0.2078	195200.0
2018-02-01	6330	New London	12817	0.1107	0.2078	195800.0
2018-03-01	6330	New London	12817	0.1107	0.2078	197000.0
2018-04-01	6330	New London	12817	0.1107	0.2078	197600.0

124 rows × 6 columns

Zip Code 6039

In [62]:
```
melted_df.loc[melted_df['Zipcode']=='6039']
```

Out[62]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6039	Torrington	13065	0.3212	0.1861	535900.0
2008-02-01	6039	Torrington	13065	0.3212	0.1861	537700.0
2008-03-01	6039	Torrington	13065	0.3212	0.1861	538100.0
2008-04-01	6039	Torrington	13065	0.3212	0.1861	538300.0
2008-05-01	6039	Torrington	13065	0.3212	0.1861	536700.0
...	...	...	...	...	...	...
2017-12-01	6039	Torrington	13065	0.3212	0.1861	468200.0
2018-01-01	6039	Torrington	13065	0.3212	0.1861	471500.0
2018-02-01	6039	Torrington	13065	0.3212	0.1861	473700.0
2018-03-01	6039	Torrington	13065	0.3212	0.1861	476000.0
2018-04-01	6039	Torrington	13065	0.3212	0.1861	480000.0

124 rows × 6 columns

Zip Code 6058

In [63]:
```
melted_df.loc[melted_df['Zipcode']=='6058']
```

Out[63]:

Date	Zipcode	Metro	SizeRank	ROI_5yr	ROI_3yr	value
2008-01-01	6058	Torrington	13564	0.4158	0.1849	350900.0
2008-02-01	6058	Torrington	13564	0.4158	0.1849	345200.0
2008-03-01	6058	Torrington	13564	0.4158	0.1849	340000.0
2008-04-01	6058	Torrington	13564	0.4158	0.1849	336800.0
2008-05-01	6058	Torrington	13564	0.4158	0.1849	336500.0
...	...	...	...	...	...	...
2017-12-01	6058	Torrington	13564	0.4158	0.1849	286500.0
2018-01-01	6058	Torrington	13564	0.4158	0.1849	292100.0
2018-02-01	6058	Torrington	13564	0.4158	0.1849	298900.0
2018-03-01	6058	Torrington	13564	0.4158	0.1849	307700.0
2018-04-01	6058	Torrington	13564	0.4158	0.1849	314600.0

124 rows × 6 columns

2 Conclusion - Top 5 ZIP Codes

to Invest in these ZIP Codes due to Higher ROI

Highest projected ROI:

- 6069 = 24.12%,
- 6610 = 31%,
- 6330 = 21%,
- 6039 = 19%,
- 6058 = 19%.

In [ ]:

```
1
```