UNLOCKING REAL ESTATE SUCCESS WITH TIME SERIES MODELLING

This is a collaborative group project done at the end of Phase 4 of Moringa School's Data Science program. The team members of this group include:

- Abdideq Adan (https://github.com/AdanAbdideq)
- Clara Gatambia (https://github.com/claragatambia)
- Isaack Odera (https://github.com/derak-isaack)
- Mwiti Mwongo (https://github.com/M13Mwongo)
- Wilson Mutungu (https://github.com/mutungu)

Problem statement

Real estate represents a significant portion of most people's wealth, and this is especially true for many homeowners in the United States. A number of factors drive the real estate market including government policies, demographics of the potential buyers,affordability, disparity in housing access, location, cash flows and liquidity as well as the current economic climate. The many variables can make the process tedious for the buyers. Naruto consultants hopes to create a predictive time series model that can help to determine the top five zipcodes in which to invest in.

This forecasting model will help adress the anticipation of any other financial crisis with apt predictive capabilities. The best model shuld capitalize more on the **R2 score** which will be our optmization metric.

3. Objectives

As Naruto consultants, we seek out to:

1. Investigate and establish the 5 best ZipCodes(Regions) that are the best to invest in that guarantee a good Return on investment.
2. Build a time series regression modell to predict the average house prices in the US and top 5 best performing zipcodes.
3. Investigate impact of the 2008 recession on the housing market per ZipCode(Region).
4. Investigate the growth rates of the real estate industry in Metropolitan areas and Zipcodes.

The objectives of this project are as follows:

- Identify the key metrics that would be used to classify profitability.
- Identify the criteria that would classify a house as a "high-end" house or "affordable" house.

Potential Challenges

- Potentially large amounts of missing data with no conclusive reason, forcing assumptions to be made.
- Some trends may be hard to explain due to differences in locale and perceptions of some matters.
- High seasonality caused by the 2008 market crash.

Importing necessary libraries

These libraries help us in performing data manipulation, visualizing distributions and performing forecasts.

In [1]:
```
# Importing necessary libraries
# Basics
import pandas as pd
import numpy as np
import itertools
from io import StringIO

# Visualization libraries
import matplotlib.pyplot as plt
%matplotlib inline
import plotly.express as px
import seaborn as sns
import matplotlib.patches as mpatches
from matplotlib.pylab import rcParams
import time

# Modeling libraries
import statsmodels.api as sm
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error, r2_score
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import acf, pacf, adfuller
from sklearn.linear_model import LassoLarsCV
from sklearn.model_selection import TimeSeriesSplit
from pmdarima import auto_arima

from prophet import Prophet

#Model deployment libraries
import joblib


# Warnings
import warnings
from statsmodels.tools.sm_exceptions import ConvergenceWarning
warnings.simplefilter('ignore', ConvergenceWarning)
warnings.filterwarnings('ignore')

# Custom Options for displaying rows.
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns',100)
```

Defining custom classes/Functions

In line with the concepts of OOP, custom classes were created to speed up the opening and manipulation of data in this project. They are as follows:

In [2]:
```
# Function to plot the bar plots
def plot_barplot(dataframe,x,y,x_title,y_title):
  fig,ax = plt.subplots(figsize=(10,5))

  sns.barplot(data = dataframe,x=x, y=y,orient='h',errorbar=None)
  ax.set_title(f"{y_title} vs {x_title}")
  ax.set_xlabel(x_title)
  ax.set_ylabel(y_title)
  plt.show()

# Function to plot histograms
def plot_histogram(dataframe, x, y, x_title, y_title):
  fig, ax = plt.subplots(figsize=(10, 5))

  sns.histplot(data=dataframe, x=x, y=y, kde=True)
  ax.set_title(f"{y_title} vs {x_title}")
  ax.set_xlabel(x_title)
  ax.set_ylabel(y_title)
  plt.show()

# Function to plot boxplots
def plot_boxplot(dataframe,x,y,x_title,y_title):
  fig,ax = plt.subplots(figsize=(10,5))

  sns.boxplot(data = dataframe,x=x, y=y)
  ax.set_title(f"{y_title} vs {x_title}")
  ax.set_xlabel(x_title)
  ax.set_ylabel(y_title)
  plt.xticks(rotation=90)
  plt.show()
```

Data Understanding

In this section after the loading the data, we seek to find a brief overview of the data. This includes:

- the data information.
- The data distribution(shape)
- Check for missing values.

Reading data

In [3]:
```
df = pd.read_csv('./data/zillow_data.csv')
```

The `df.head()` function is used to get a rough look at a few of the records in the dataframe to understand the data better.

In [4]:
```
df.head(10)
```

Out[4]:

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	19
0	84654	60657	Chicago	IL	Chicago	Cook	1	334200.0	335
1	90668	75070	McKinney	TX	Dallas-Fort Worth	Collin	2	235700.0	236
2	91982	77494	Katy	TX	Houston	Harris	3	210400.0	212
3	84616	60614	Chicago	IL	Chicago	Cook	4	498100.0	500
4	93144	79936	El Paso	TX	El Paso	El Paso	5	77300.0	77
5	91733	77084	Houston	TX	Houston	Harris	6	95000.0	95
6	61807	10467	New York	NY	New York	Bronx	7	152900.0	152
7	84640	60640	Chicago	IL	Chicago	Cook	8	216500.0	216
8	91940	77449	Katy	TX	Houston	Harris	9	95400.0	95
9	97564	94109	San Francisco	CA	San Francisco	San Francisco	10	766000.0	771

10 rows × 272 columns

The `df.info()` and `df.dtypes` functions are both called to give a rough understanding of the dataframe, and the types of data held in each column. Normally, `df.info()` would be sufficient, but due to the sheer number of columns in the dataframe, that information isn't displayed. Thus, `df.dtypes` is called.

In [5]:
```
# Looping through data to display data information
for data in df.columns:
    print(df[data].info())
    print(df[data].dtypes)
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 14723 entries, 0 to 14722
Series name: RegionID
Non-Null Count  Dtype
--------------  -----
14723 non-null  int64
dtypes: int64(1)
memory usage: 115.1 KB
None
int64
<class 'pandas.core.series.Series'>
RangeIndex: 14723 entries, 0 to 14722
Series name: RegionName
Non-Null Count  Dtype
--------------  -----
14723 non-null  int64
dtypes: int64(1)
memory usage: 115.1 KB
None
```

As expected, all the columns representing time-series data will be storing either an integer or a float data type. It is also important to note that the `RegionID` & `RegionName` are stored as integers - to ensure they are unique - and the remaining 5 columns all contain strings.

The shape of the dataframe was observed as follows:

In [5]:
```
# Check for missing values
df.isnull().sum()
```

Out[5]:
```
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
```

Aside from the columns that contain time-series values, only the metro column has null values. This can be seen as inconsequential as there is enough data from the other columns to overlook this.

Taking a closer look at the missing time-series values, there seems to be a steady trend. The values do not appear to be random, as they steadily decrease from April 1996 to July 2014, from where they remain 0 throughout till April 2018.

The gradual change indicates that the presence of null values in these columns is anything but random. As such, when dealing with the null values in the time-series value columns, **all null values will be left as is**. No replacement or removal of records will be done. This is done as it is assumped that not all the houses were built at the same time, thus it is expected that there will be null values for some houses and not others. Furthermore, some of these null values are attributed to the differential times that the houses were put on the market.

When dealing with the null values in the metro column, all null values will be left as the data missing does not impact the objectives of the project.

In [6]:
```
# Check for duplicate values
df.duplicated().sum()
```

Out[6]:
```
0
```

In [7]:
```
# Check the shape of the data
df.shape
```

Out[7]:
```
(14723, 272)
```

The dataframe has 14723 records and 272 column. However, majority of those columns represent the time series values. The first seven columns - `RegionID`, `RegionName`, `City`, `State`, `Metro`, `CountyName` and `SizeRank` - are the columns that give us more information about the dataset.

Each of these columns is intended to hold a certain type of data as follows:

- *RegionID*: The unique ID of the region in question.
- *RegionName*: The name of the region in question.
- *City*: The name of the city within a given region.
- *State*: The state in which the RegionID is found.
- *Metro*: The metropolitan name within which the RegionID is found.
- *CountyName*: The name of the county within a given region.
- *SizeRank*: The region's area ranking vis-a-vis other regions, organised in descending order.

Going forward, these columns shall be referred to as the `columns of interest`, while the remaining columns shall be referred to as the `time_series_cols`.

In [8]:
```
# ['RegionID', 'RegionName', 'City', 'State', 'Metro',
'CountyName','SizeRank']

cols_of_interest = df.columns[:7]
cols_of_interest
```

Out[8]:
```
Index(['RegionID', 'RegionName', 'City', 'State', 'Metro', 'CountyName',
       'SizeRank'],
      dtype='object')
```

In [9]:
```
time_series_cols = df.columns[-265:]
time_series_cols
```

Out[9]:
```
Index(['1996-04', '1996-05', '1996-06', '1996-07', '1996-08', '1996-09',
       '1996-10', '1996-11', '1996-12', '1997-01',
       ...
       '2017-07', '2017-08', '2017-09', '2017-10', '2017-11', '2017-12',
       '2018-01', '2018-02', '2018-03', '2018-04'],
      dtype='object', length=265)
```

The last 265 columns all need to be datetime columns.

Data Preprocessing

In [10]:
```
# Ensuring all time series columns are converted to float64 i.e. the last
265 columns

for col in time_series_cols:
  if df[col].dtype is not 'float64':
    df[col] = df[col].astype('float64')


df.dtypes
```

Out[10]:
```
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
```

Data Handling

I| Data aggregation

As all values in the time-series columns are deemed important, aggregating the data to a yearly basis, for example, at such an early stage was deemed unnecessary. This is because a lot of vital information would be lost in the process that would be necessary later on. Thus, aggregation would be done on an as-needed basis.

II| Grouping data

Data will now be grouped by state, to get a better overview of the metropolitan housing distribution in various states.

In [11]:
```
metro_dist = df[['State',
'RegionID','RegionName','City','Metro','CountyName','SizeRank']].groupby(
"State").count().sort_values(by='Metro',

                                                ascending=False).head(10)
metro_dist
```

Out[11]:

State	RegionID	RegionName	City	Metro	CountyName	SizeRank
CA	1224	1224	1224	1182	1224	1224
NY	1015	1015	1015	963	1015	1015
TX	989	989	989	873	989	989
PA	831	831	831	816	831	831
FL	785	785	785	758	785	785
OH	588	588	588	570	588	588
IL	547	547	547	537	547	547
NJ	502	502	502	502	502	502
MI	499	499	499	457	499	499
MA	417	417	417	416	417	417

Particular focus will be placed on the `RegionID` as that column does not have any null values, thus an accurate count of properties can be done. This column will appropriately be renamed to `Count of Properties` as it signifies the count of properties in a particular state.

In [12]:
```
df_state_grouping = df[['State', 'RegionID']].groupby(
    "State").count().sort_values(by='RegionID', ascending=False).head(10)
df_state_grouping.reset_index(inplace=True)
df_state_grouping.rename(columns={'RegionID': 'Count of Properties'},
inplace=True)
df_state_grouping
```

Out[12]:

	State	Count of Properties
0	CA	1224
1	NY	1015
2	TX	989
3	PA	831
4	FL	785
5	OH	588
6	IL	547
7	NJ	502
8	MI	499
9	IN	428

California (CA) is seen to have the most listings with 1224, followed closely by New York, NY (1015) and Texas, TX (989). Vermont (VT), Washington DC (DC) and San Diego (SD) have the least listings at 16, 18 and 19 respectively.

III| Outliers

There will be outliers present in the data, that will be more apparent in the EDA section. These outliers may come around as a result of over-inflated house prices, or simply having houses marked below their market value by owners who are in a rush to sell. Whatever the case may be, the outliers in the dataset **will be kept throughout the analysis**.

These outliers play an important role in developing a more realistic model, as the world of real-estate will always have overpriced or underpriced properties that do not conform to the norms. Keeping these outliers in will serve to create a more realistic model.

Feature Engineering

| New Feature addittion

Relevant features that could affect house prices were noted and their creation was deemed necessary. These features are:

- *Return On Investment(ROI)(%)*: Calculated as $\left(\frac{\{\text{Last Price of Property}\}}{\{\text{Initial Price of Property}\}} - 1\right) \times 100\%$

As we know that the latter, and in particular the last time-series column, has no null values, this column will automatically be our numerator (i.e. `df.columns[271]`). As for the denominator, it's value will be occuppied by the oldest datetime that is not a null value. Therefore, the calculation of the ROI was done as follows:

In [13]:
```
# Create a copy of the original model to engineer new features
df_feature_engineered = df.copy()

# Loop through the columns
for index, row in df_feature_engineered.iterrows():
  # Establishing the last price of the property
  last_price = df_feature_engineered.columns[271]

  # Establishing the initial price of the property
  initial_price = 0

  # Loop through time_series_cols to find the first non-null value
  for col in time_series_cols:
      initial_price = df_feature_engineered[col].values[index]
      if not pd.isnull(initial_price):
        # print(f"Record {index} initial price: {initial_price} @ {col}")
        break
  df_feature_engineered.at[index, 'Initial Price'] = initial_price

  df_feature_engineered['Last Price'] = df[last_price]

  df_feature_engineered['ROI (%)'] = round(
      (((df_feature_engineered['Last Price'] /
df_feature_engineered['Initial Price']) - 1) * 100), 2)

df_feature_engineered.head(10)
```

Out[13]:

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	19
0	84654	60657	Chicago	IL	Chicago	Cook	1	334200.0	335
1	90668	75070	McKinney	TX	Dallas-Fort Worth	Collin	2	235700.0	236
2	91982	77494	Katy	TX	Houston	Harris	3	210400.0	212
3	84616	60614	Chicago	IL	Chicago	Cook	4	498100.0	500
4	93144	79936	El Paso	TX	El Paso	El Paso	5	77300.0	77
5	91733	77084	Houston	TX	Houston	Harris	6	95000.0	95
6	61807	10467	New York	NY	New York	Bronx	7	152900.0	152
7	84640	60640	Chicago	IL	Chicago	Cook	8	216500.0	216
8	91940	77449	Katy	TX	Houston	Harris	9	95400.0	95
9	97564	94109	San Francisco	CA	San Francisco	San Francisco	10	766000.0	771

10 rows × 275 columns

Normalization and Scaling

Having gone through the data, no different scales are noticed. However, there may be a need for normalization when it comes to comparing regions with vastly different quantities of properties listed that have a variety of prices. This is to ensure that the comparison is as accurate as possible and without an inherent bias or skew. This shall be done later on in the EDA stages on an as-needed basis

A new dataframe is now created which contains the melted data. This will convert the original dataframe `df` from wide to long format. This will be done by applying the custom function `melt_data`

In [14]:
```
# Function to convert data from wide to long formart.
def melt_data(df):
    """
    Takes the zillow_data dataset in wide form or a subset of the
zillow_dataset.
    Returns a long-form datetime dataframe
    with the datetime column names as the index and the values as the
'values' column.

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

    melted = pd.melt(df, id_vars=['RegionName', 'RegionID', 'SizeRank',
                      'City', 'State', 'Metro', 'CountyName'],
var_name='time')
    melted['time'] = pd.to_datetime(melted['time'],
infer_datetime_format=True)
    melted = melted.dropna()
    return melted
```

Converting the dataframe to long format:

In [15]:
```
# Create a variable to hold the long-format data
df_eda = melt_data(df)
df_eda.head()
```

Out[15]:

	RegionName	RegionID	SizeRank	City	State	Metro	CountyName	time	value
0	60657	84654	1	Chicago	IL	Chicago	Cook	1996-04-01	334200.(
1	75070	90668	2	McKinney	TX	Dallas-Fort Worth	Collin	1996-04-01	235700.(
2	77494	91982	3	Katy	TX	Houston	Harris	1996-04-01	210400.(
3	60614	84616	4	Chicago	IL	Chicago	Cook	1996-04-01	498100.(
4	79936	93144	5	El Paso	TX	El Paso	El Paso	1996-04-01	77300.(

In [16]:
```
# Engineer new features for the month and year.
df_eda['Year'] = df_eda['time'].dt.year
df_eda['Month'] = df_eda['time'].dt.month
```

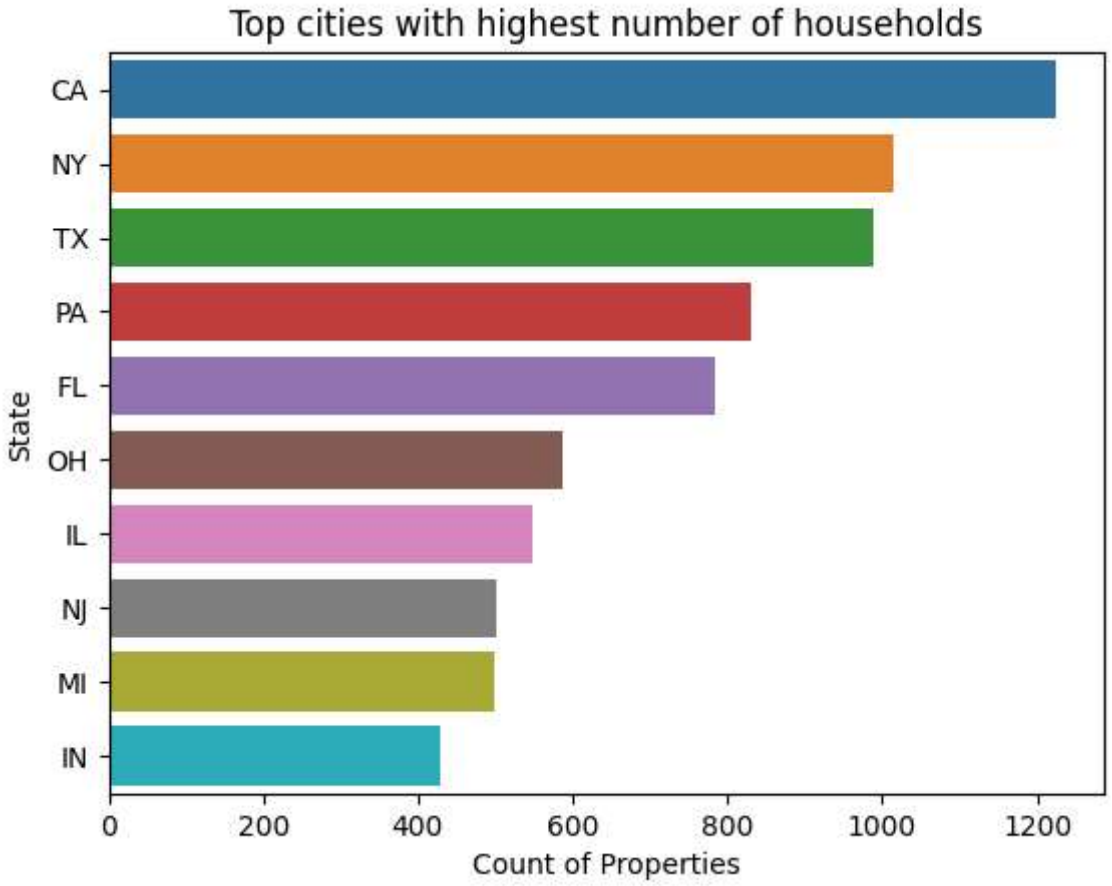
Explorative Data Analysis(Visualization)

This section involves visualizing the effects of the crash and analyzing the best performing regions/markets. Of consideration is also analyzing the housing distribution in major cities and counties.

The years to be considered to analyze the market trends durig the crisis are **2008, 2009 and 2010**.

The years to consider in analyzing the bounce back rates after the crisis are **2011, 2012, 2013, 2014 and 2015**.

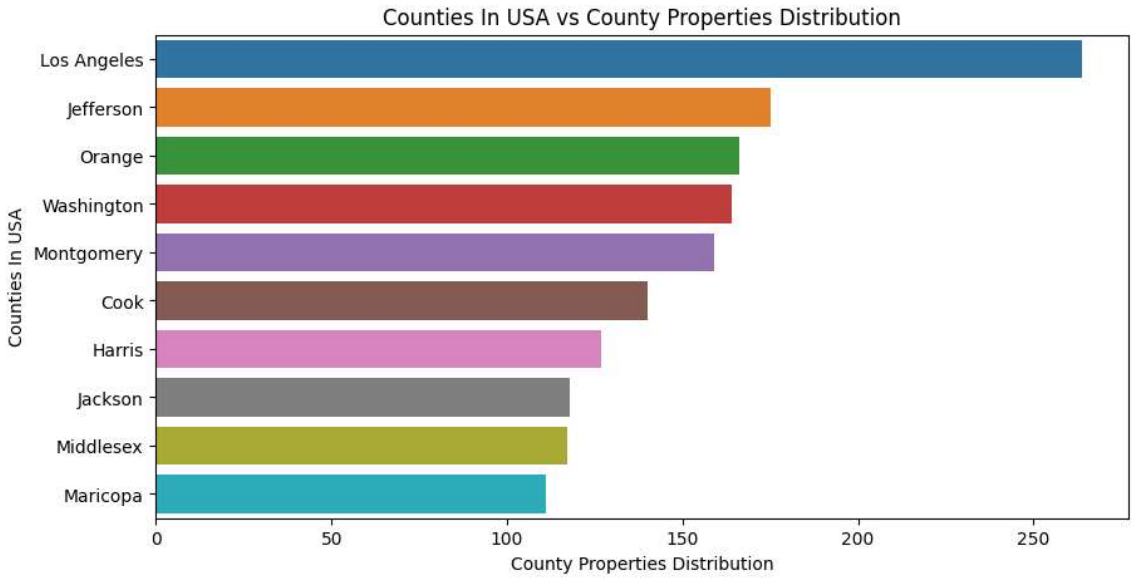
In [17]:
```
# Use the function to plot horizontal bar plots.
sns.barplot(data=df_state_grouping, y='State',x='Count of Properties')
plt.title("Top cities with highest number of households")
plt.show()
```



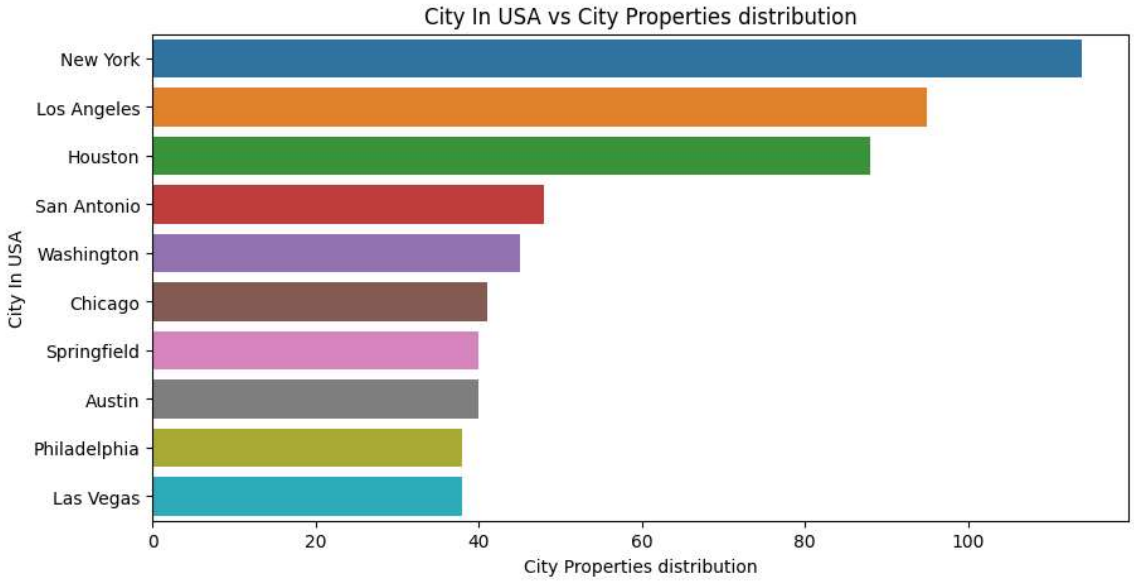
This same analysis can be done to observe the spread of property listings in the various cities and counties.

The State of California, New York and Texas have the highest number of property distributions.

In [18]:
```
# Apply the bar plot function.
plot_barplot(dataframe=df[['RegionID',
'CountyName']].groupby('CountyName').count().reset_index().sort_values(
    by='RegionID', ascending=False).head(10), y='CountyName',
x='RegionID', y_title='Counties In USA', x_title='County Properties
Distribution')
```



In [19]:
```
# Apply the barplot function to get property distribution accross cities.
plot_barplot(dataframe=df[['RegionID',
'City']].groupby('City').count().reset_index().sort_values(
    by='RegionID', ascending=False).head(10), y='City', x='RegionID',
y_title='City In USA', x_title='City Properties distribution')
```



In [21]:
```
# Group data to get the zipcodes with high property ratings.
group_metropolitan = pd.DataFrame(df_eda.groupby(['Metro','RegionName'])
['value'].agg('mean').sort_values(ascending=False).head(30))
group_metropolitan
```

Out[21]:

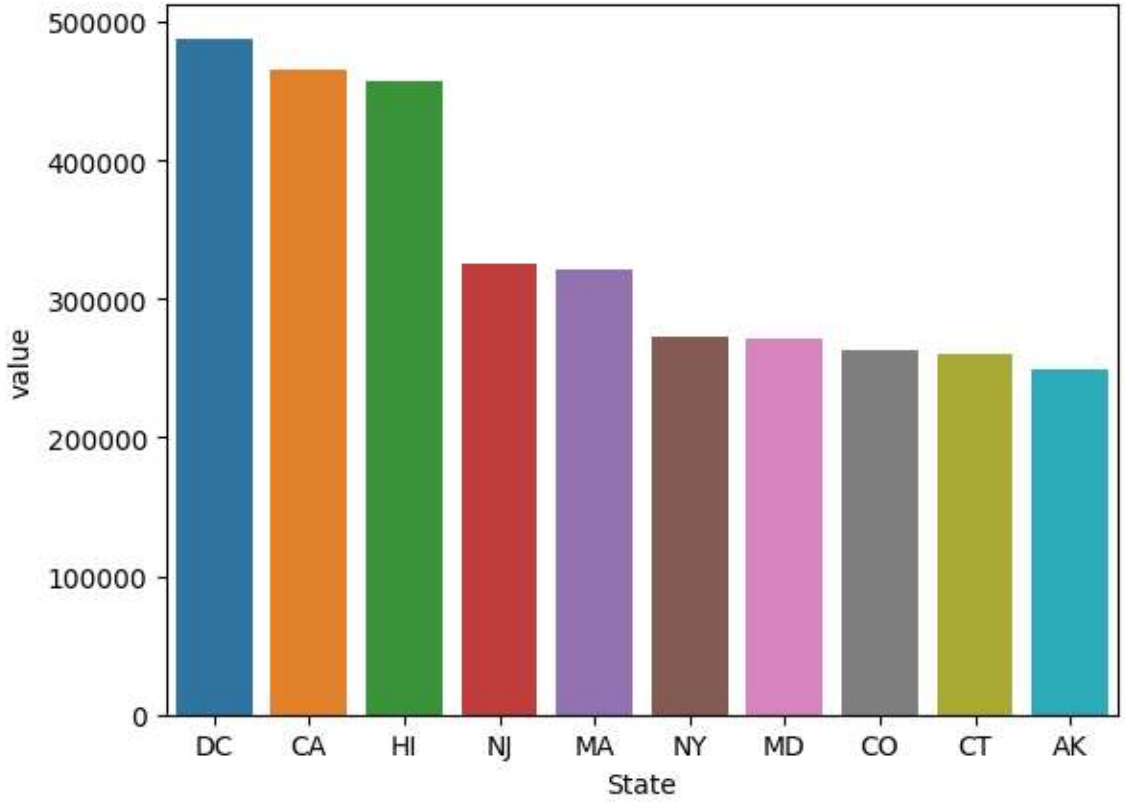
		value
Metro	**RegionName**	
New York	**10021**	1.285427e+07
	10011	7.755844e+06
	10014	6.836902e+06
	10128	5.085436e+06
San Francisco	**94027**	3.487129e+06
Glenwood Springs	**81611**	3.147124e+06
Los Angeles-Long Beach-Anaheim	**90210**	2.789977e+06
Miami-Fort Lauderdale	**33480**	2.634498e+06
San Francisco	**94123**	2.630977e+06
Brunswick	**31561**	2.403194e+06
San Francisco	**94115**	2.399030e+06
	94109	2.395636e+06
Glenwood Springs	**81615**	2.300179e+06
Los Angeles-Long Beach-Anaheim	**90402**	2.292232e+06
San Diego	**92067**	2.170122e+06
Los Angeles-Long Beach-Anaheim	**90020**	2.149644e+06
San Francisco	**94028**	2.131495e+06
San Jose	**94301**	2.084380e+06
New York	**7620**	2.080074e+06
Los Angeles-Long Beach-Anaheim	**92657**	2.070006e+06
San Jose	**94022**	2.066818e+06
San Francisco	**94957**	2.043843e+06
Santa Maria-Santa Barbara	**93108**	1.991682e+06
New York	**11217**	1.977407e+06
San Jose	**94305**	1.918675e+06
Boston	**2116**	1.887043e+06
Los Angeles-Long Beach-Anaheim	**90265**	1.876195e+06
San Francisco	**94133**	1.833388e+06
New York	**11976**	1.803404e+06
Los Angeles-Long Beach-Anaheim	**90049**	1.799291e+06

Average house prices by state

This is done using the groupby function on the house values.

In [22]:
```
# Average house prices by state
state_group = df_eda.groupby('State')
['value'].agg('mean').sort_values(ascending=False).head(10)

sns.barplot(data=state_group.reset_index(), x='State', y='value')
```

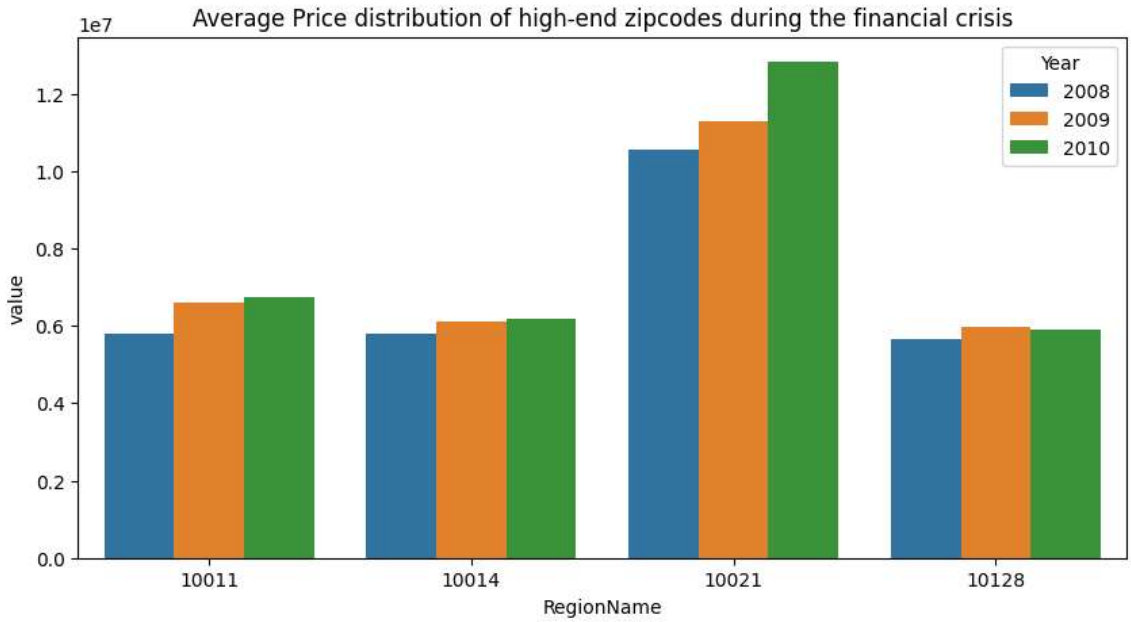
Out[22]:
```
<AxesSubplot: xlabel='State', ylabel='value'>
```



Market crash analysis

Filter down only on the specific zip codes and specific years for the market crash to obtain the average price distribution.

In [23]:
```
# Analysis of market crash crisis on regions.
regions = [10021, 10011, 10014, 10128, 94027, 81611, 90210, 33480, 94123,
31561,
                   94115, 94109, 81615, 90402, 92067, 90020, 94028,
94301, 7620, 92657,
                   94022, 94957, 93108, 11217, 94305, 2116, 90265, 94133,
11976, 90049]


years = [2008, 2009, 2010]

plt.figure(figsize=(10,5))
filtered_df = df_eda[(df_eda['Year'].isin(years)) &
df_eda['RegionName'].isin(regions)]
group_crisis = pd.DataFrame(filtered_df.groupby(['RegionName','Year'])
['value'].agg('mean').sort_values(ascending=False)).head(12)
sns.barplot(data=group_crisis.reset_index(), x='RegionName', y='value',
hue='Year')
plt.title("Average Price distribution of high-end zipcodes during the
financial crisis")
plt.show()
```

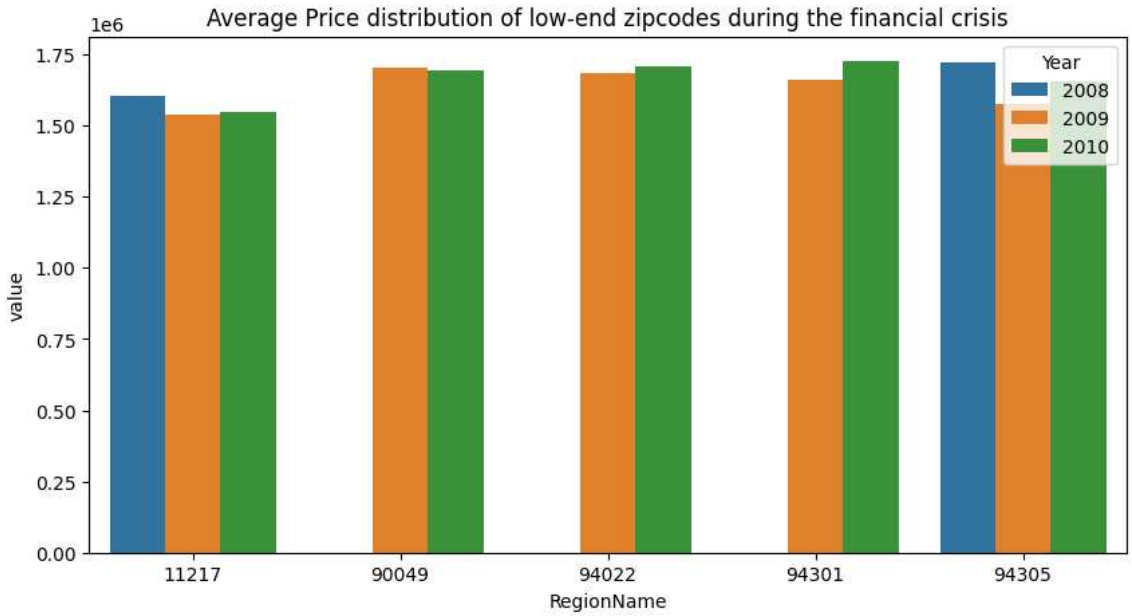


The crash ended in the year **June 2009** but the after effects could still be felt in the subsequent year(2010) and that's why we included it to analyze the average price distribution across the top 30 expensive zipcodes during the crisis.

Market crash analysis

We aim to obtain distribution of house prices in low-end zip codes during the crisis.

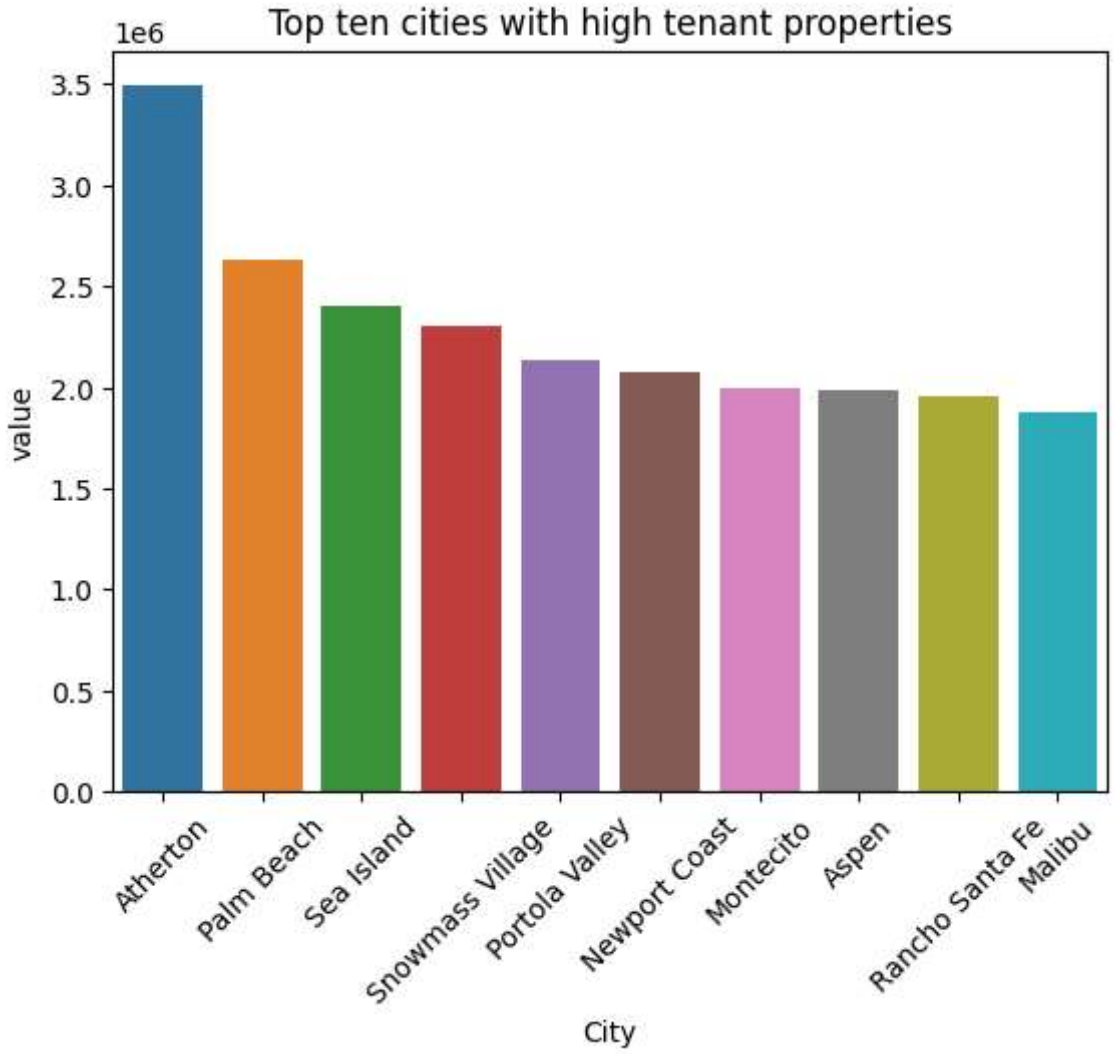
In [24]:
```
#Market crash effects analysis.
plt.figure(figsize=(10,5))
group_crisis2 = pd.DataFrame(filtered_df.groupby(['RegionName','Year'])
['value'].agg('mean').sort_values(ascending=True)).head(12)
sns.barplot(data=group_crisis2.reset_index(), x='RegionName', y='value',
hue='Year')
plt.title("Average Price distribution of low-end zipcodes during the
financial crisis")
plt.show()
```



Average City price distribution

We aggregate the average price of property in every city.

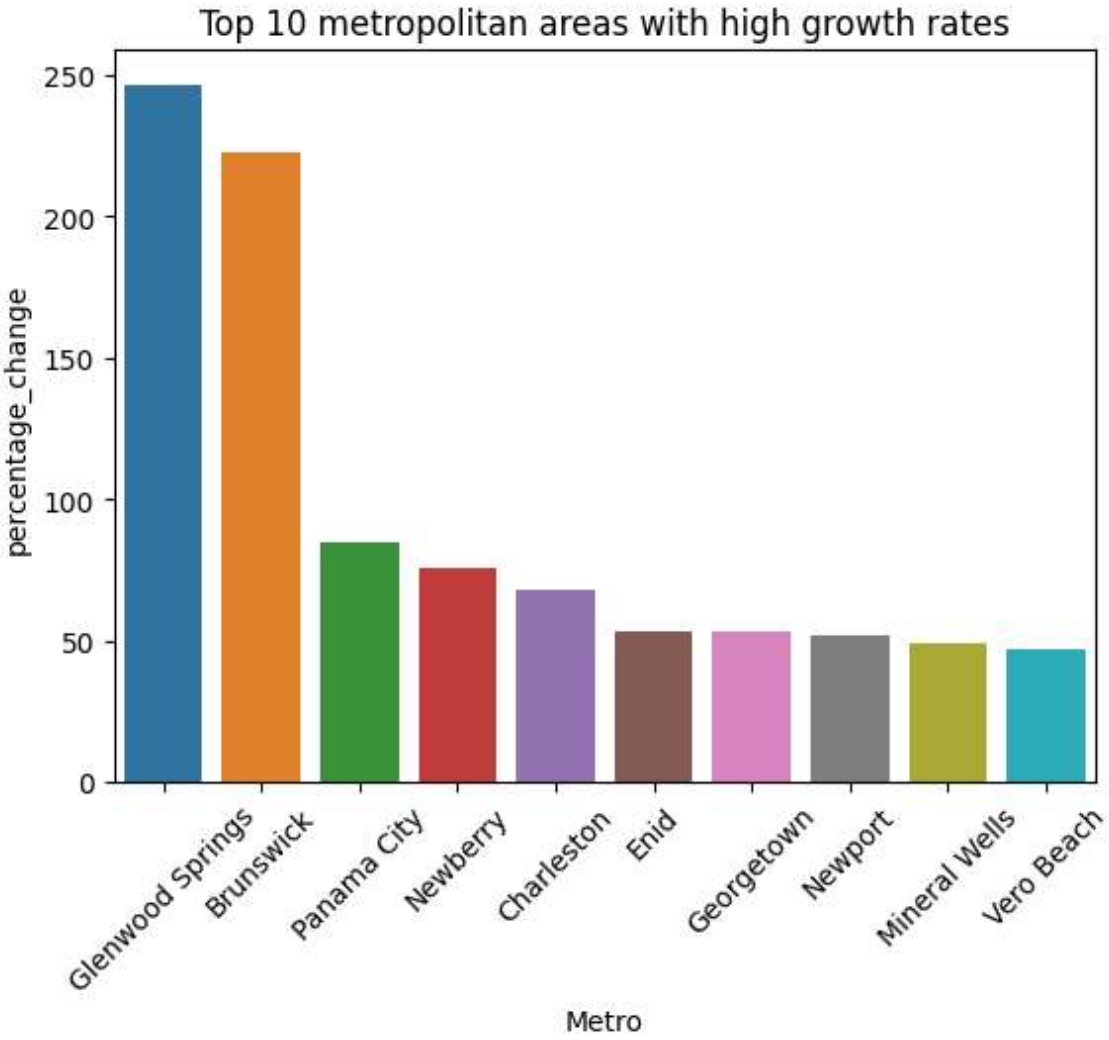
In [25]:
```
# Analyze distribution of tenant properties per city.
group_cities = pd.DataFrame(df_eda.groupby('City')
['value'].agg('mean').sort_values(ascending=False)).head(10)

sns.barplot(data=group_cities.reset_index(), x='City', y='value')
plt.title("Top ten cities with high tenant properties")
plt.xticks(rotation=45)
plt.show()
```



Metropolitan growth rates

Grouping data based on the paercentage difference of past house values using the **pct_change** from pandas.

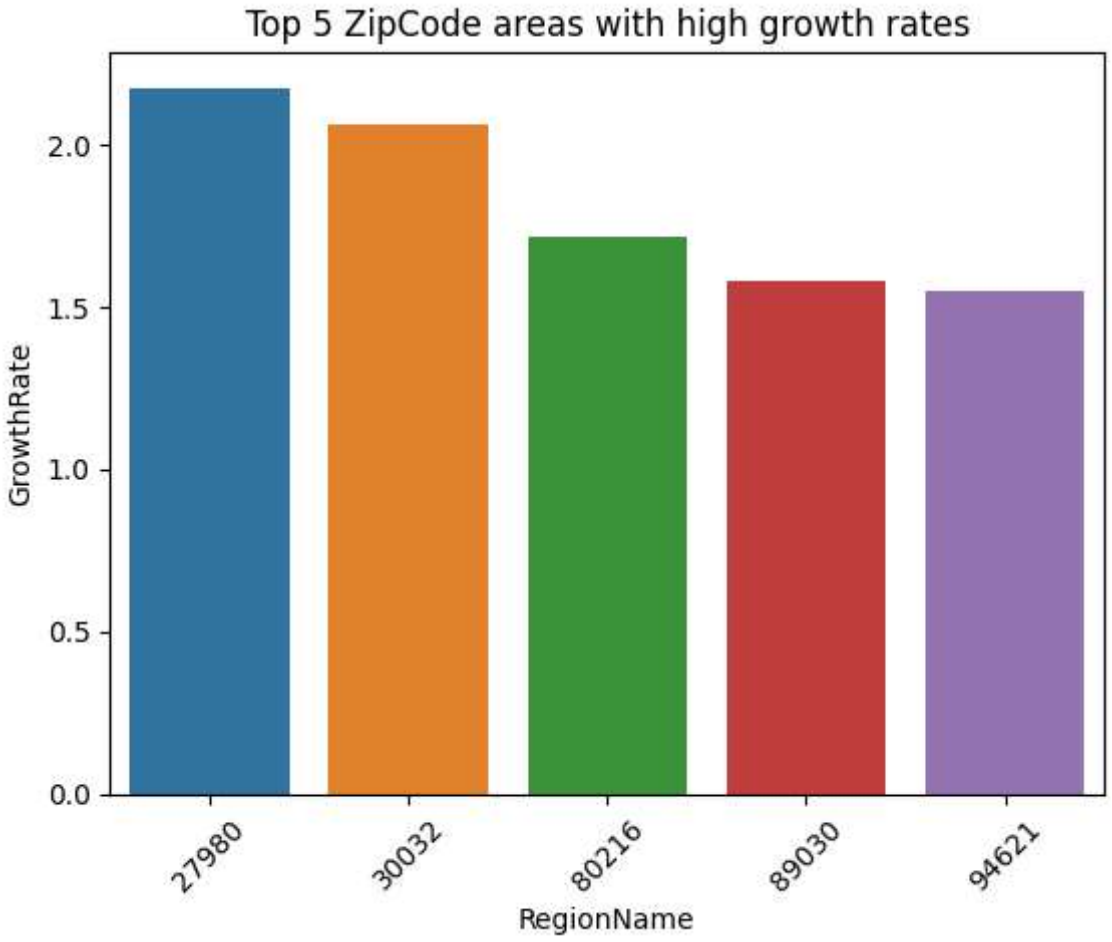
In [26]:
```
# Growth rates of metroplolitan areas.
df_eda['percentage_change'] = df_eda.groupby('Metro')
['value'].pct_change() * 100

average_percentage_change = pd.DataFrame(df_eda.groupby('Metro')
['percentage_change'].mean().sort_values(ascending=False)).head(10)
sns.barplot(data=average_percentage_change.reset_index(), x='Metro',
y='percentage_change')
plt.title("Top 10 metropolitan areas with high growth rates")
plt.xticks(rotation=45)
plt.show()
```



ZipCodes growth rates

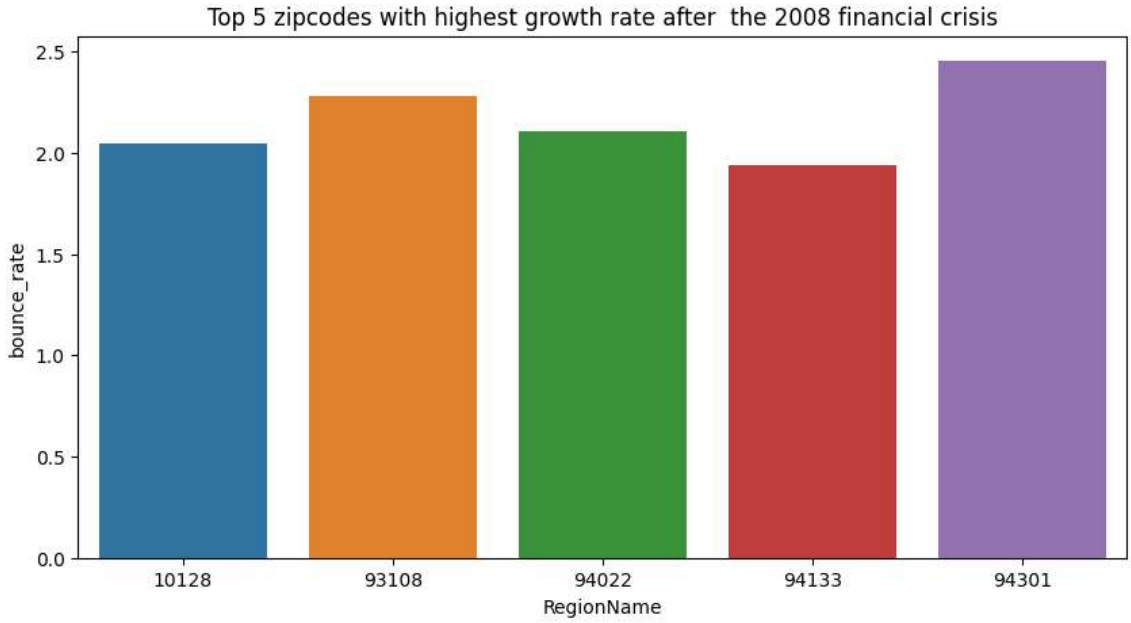
Getting the rates of different zipcodes specifically the top 5.

In [27]:
```
# Growth rates for various Zipcodes(Regions)
df_eda['GrowthRate'] = df_eda.groupby('RegionName')['value'].pct_change()
* 100

average_pct_change = pd.DataFrame(df_eda.groupby('RegionName')
['GrowthRate'].mean().sort_values(ascending=False)).head(5)
sns.barplot(data=average_pct_change.reset_index(), x='RegionName',
y='GrowthRate')
plt.title("Top 5 ZipCode areas with high growth rates")
plt.xticks(rotation=45)
plt.show()
```



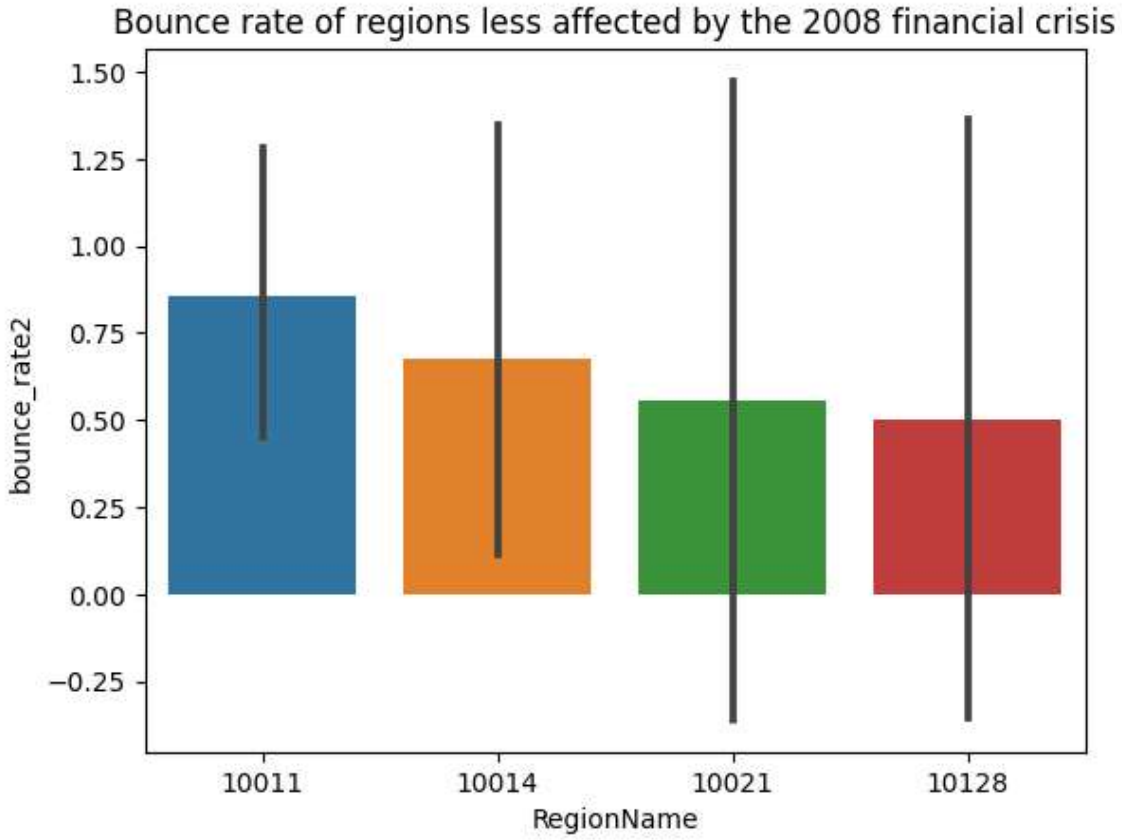
Growth rates after crisis

In [28]:
```
# Investigate among the high end estates, which regions specifically zip
codes that had a good growth rate after the crisis.
years2 = [2011, 2012, 2013, 2014, 2015 ]

plt.figure(figsize=(10,5))
filtered_df2 = df_eda[(df_eda['Year'].isin(years2)) &
(df_eda['RegionName'].isin(regions))]
filtered_df2['bounce_rate'] = df_eda.groupby('RegionName')
['value'].pct_change() * 100
group_crisis = pd.DataFrame(filtered_df2.groupby(['RegionName','Year'])
['bounce_rate'].agg('mean').sort_values(ascending=False)).head(5)
sns.barplot(data=group_crisis.reset_index(), x='RegionName',
y='bounce_rate')
plt.title("Top 5 zipcodes with highest growth rate after  the 2008
financial crisis")
plt.show()
```



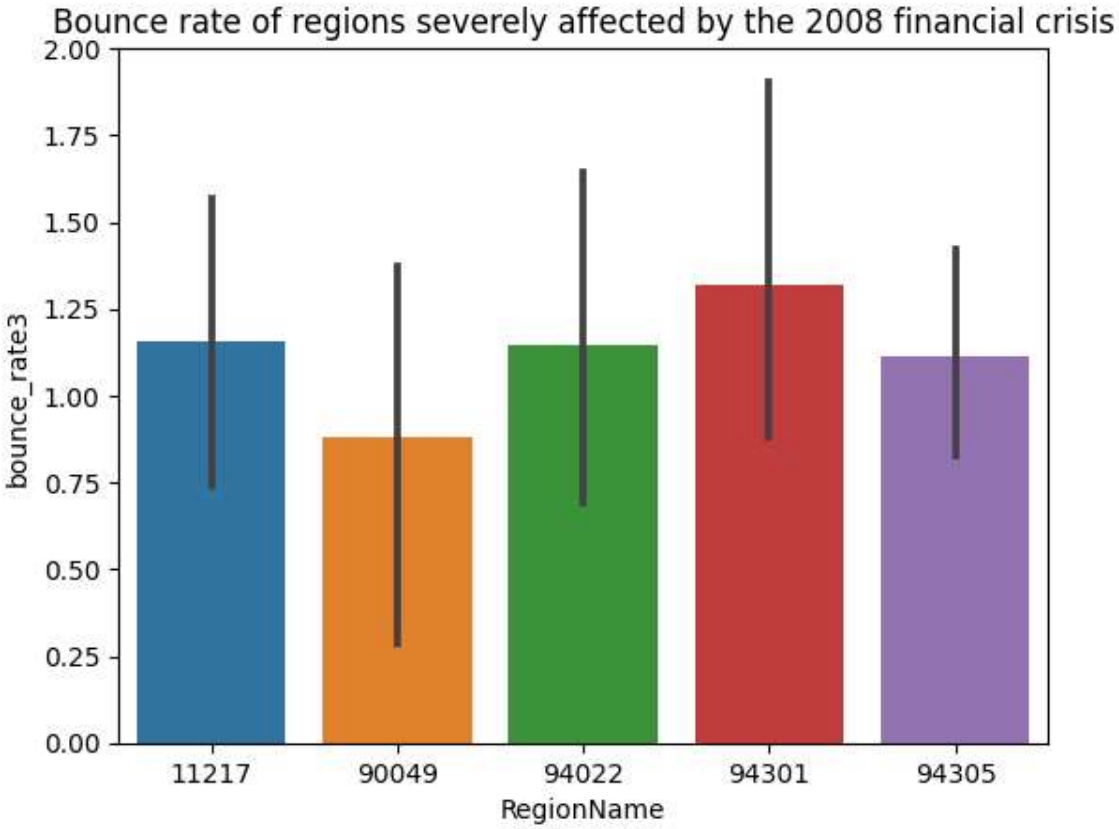
Bounce back rate after crisis

In [29]:
```
# Analyze how regions bounced back after the 2008 financial crisis.
regions2 = [10011, 10014, 10021, 10128]

filtered_df2 = df_eda[(df_eda['Year'].isin(years2)) &
(df_eda['RegionName'].isin(regions2))]
filtered_df2['bounce_rate2'] = df_eda.groupby('RegionName')
['value'].pct_change() * 100
group_crisis = pd.DataFrame(filtered_df2.groupby(['RegionName','Year'])
['bounce_rate2'].agg('mean').sort_values(ascending=False))
sns.barplot(data=group_crisis.reset_index(), x='RegionName',
y='bounce_rate2')
plt.title("Bounce rate of regions less affected by the 2008 financial
crisis")
plt.show()
```



Bounce back rate of regions severly affected by 2008 crisis

In [30]:
```
# Analyze bounce back rate of regions severly affected by the 2008
financial crisis.
regions3 = [11217, 90049, 94022, 94301, 94305]

filtered_df3 = df_eda[(df_eda['Year'].isin(years2)) &
(df_eda['RegionName'].isin(regions3))]
filtered_df3['bounce_rate3'] = df_eda.groupby('RegionName')
['value'].pct_change() * 100
group_crisis3 = pd.DataFrame(filtered_df3.groupby(['RegionName','Year'])
['bounce_rate3'].agg('mean').sort_values(ascending=False))
sns.barplot(data=group_crisis3.reset_index(), x='RegionName',
y='bounce_rate3')
plt.title("Bounce rate of regions severely affected by the 2008 financial
crisis")
plt.show()
```



EDA Conclusion

- Wasington DC has the highest priced homes.
- ZipCodes **10021, 10011, 10014** and **10128** were among the top priced regions less affected by the 2008 financial crisis.
- Glenwood Springs, Brunswick are the metropolitan areas with high growth rates of 250% and 225%.
- Zipcode **10011** had the highest bounce back rate of 0.85% after the market crush.
- ZipCodes **94301** and **94022** which were severely affected by the crisis had the highest rates at 1.3% and 1.1% respectively.
- Regions most hit by the crisis include **1121, 90049, 94022** all above 1.5%.

- ZipCodes 27980, 30032 and 80216 are the region swith high growth rates generally at 2.5%, 2% and 1.75%.

3. Modelling

Modelling in time series involves the use of Auto Regressive models which uses lagged features to predict the future values. Since most time series data always has the component of trend and sesonality, the ARIMA models have the idea of differencing to ensure stationarity. This is also one of the assumptions of ARMA models.

With that said, the data we have has to be transformed from the wide formart to the long formart to ensure uniformity and an easy resampling and modelling process.

In this section, we seek out to build a forecasting model to predict the house prices for the **ZipCodes** which are high priced and have good prospects for a good **Return on Investment**. This will finally help us have a solid recommendation to give to our investors regarding the best zipCodes to invest in.

The contents of this section include:

- Transforming data
- Data Resampling
- Baseline model
- Auto Arima model
- Arima Models
- Facebook Prophet(ETL pipeline)

In [32]:
```
# Create a modelling variable using the melt function.
df_model = melt_data(df)
df_model.set_index('time', inplace=True)
```

Setting the time as the index column allows easier resampling as well as slicing of data. The from the melting function has many rows and as such training would only need a fraction or small sample of the data to represent the whole population. It is computationally expensive to run models on very huge sets of data especially ARMA models. They are not efficient to handle big data.

In [33]:
```
df_model.head()
```

Out[33]:

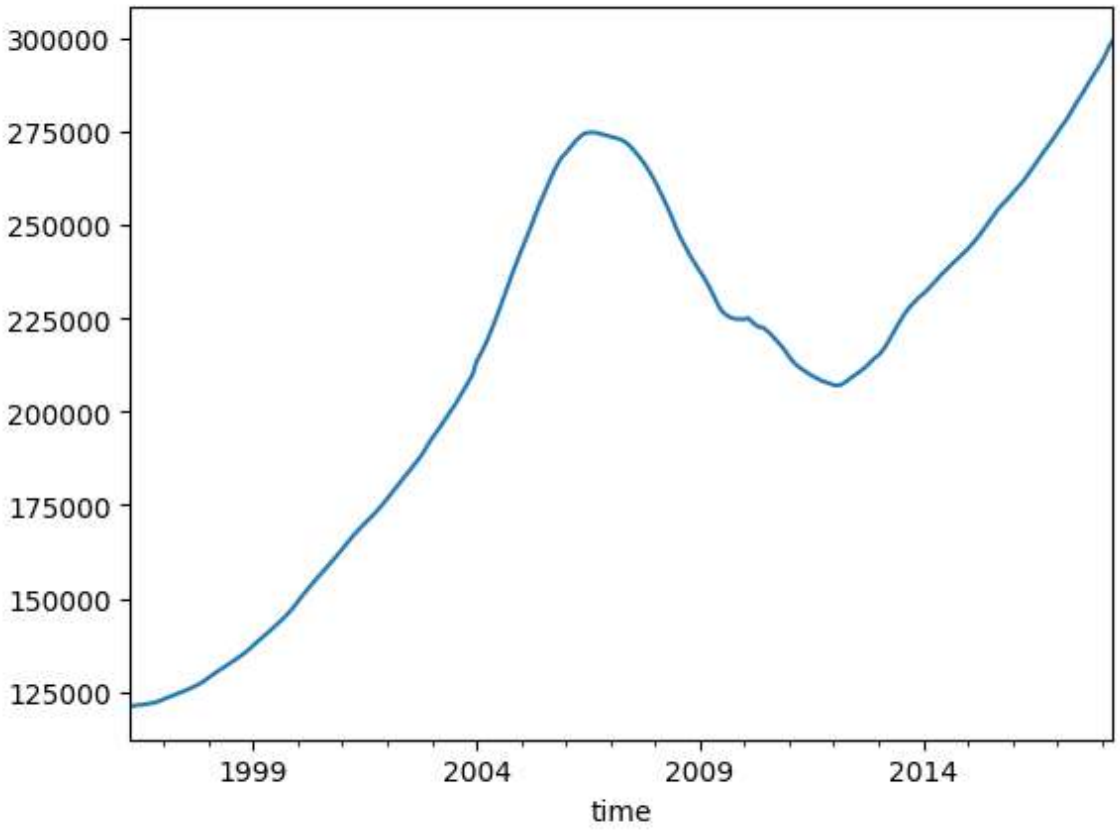
time	RegionName	RegionID	SizeRank	City	State	Metro	CountyName	value
1996-04-01	60657	84654	1	Chicago	IL	Chicago	Cook	334200.0
1996-04-01	75070	90668	2	McKinney	TX	Dallas-Fort Worth	Collin	235700.0
1996-04-01	77494	91982	3	Katy	TX	Houston	Harris	210400.0
1996-04-01	60614	84616	4	Chicago	IL	Chicago	Cook	498100.0
1996-04-01	79936	93144	5	El Paso	TX	El Paso	El Paso	77300.0

Data Resampling

Data resampling in Time series data is very important because it aggregates data on a different frequency than what it was collected and changes frequency of a time series data. Resampling serves to **remove noise** or rather **random fluctuations**. The various ways to resample data are: **A - Year End, M - Month end, W - Weekly, D - Daily and MS - Month start**. The data at our disposal seems to be on a month start basis and that is the best resampling technique for this problem.

In [34]:
```
# Resample data on a monthly basis.
df_model2 = df_model['value'].resample('MS').mean()
df_model2.plot()
```

Out[34]: <AxesSubplot: xlabel='time'>



The plot from the monthly resampled data shows a steady growth from 1999 upto 2007. This was greatly attribute to the subprime housing mortgages given to low income earners. As from 2008, there is a increasingly staedy drop in the value of houses and this was due to the housing market crash in 2008 This is the period when the **US** had a housing Crisis majorly caused by subprime mortgages, huge debts and low regulation in the financial sector. The value of homes dropped and many home owners and investors could not keep up with their mortgages.

As outlined in [https://www.investopedia.com/articles/economics/09/subprime-market-2008.asp] (https://www.investopedia.com/articles/economics/09/subprime-market-2008.asp%5D), the US was faced with a financial crisis when the housing syatem collapsed and value of house prices plummuted.

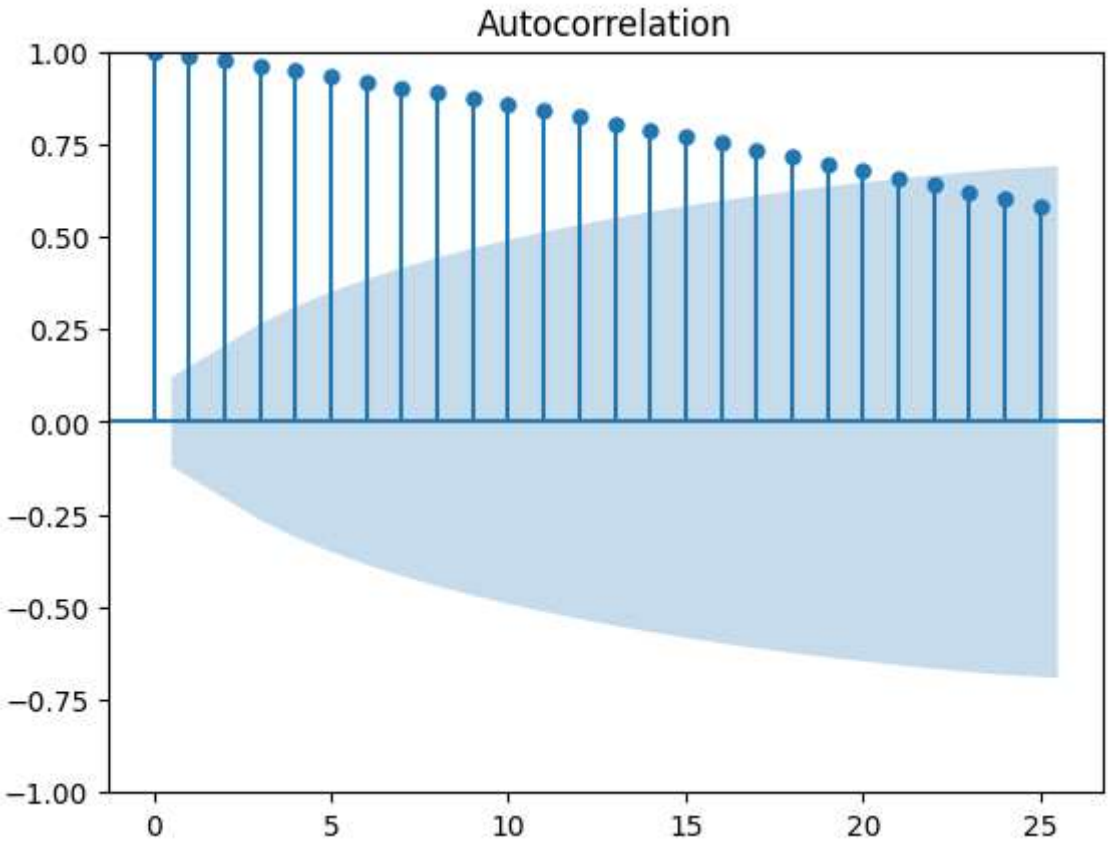
This housing project by the government was however cut short by the ever increasing default rates and debts caused by the high risk mortgages given by the US government. Many of the low income earners would pay up low initial payments and thereafter huge interest rates. This build up of debts caused a major reccession to the point of the government bailing out home owners because most ended up bankrupt with low value homes.

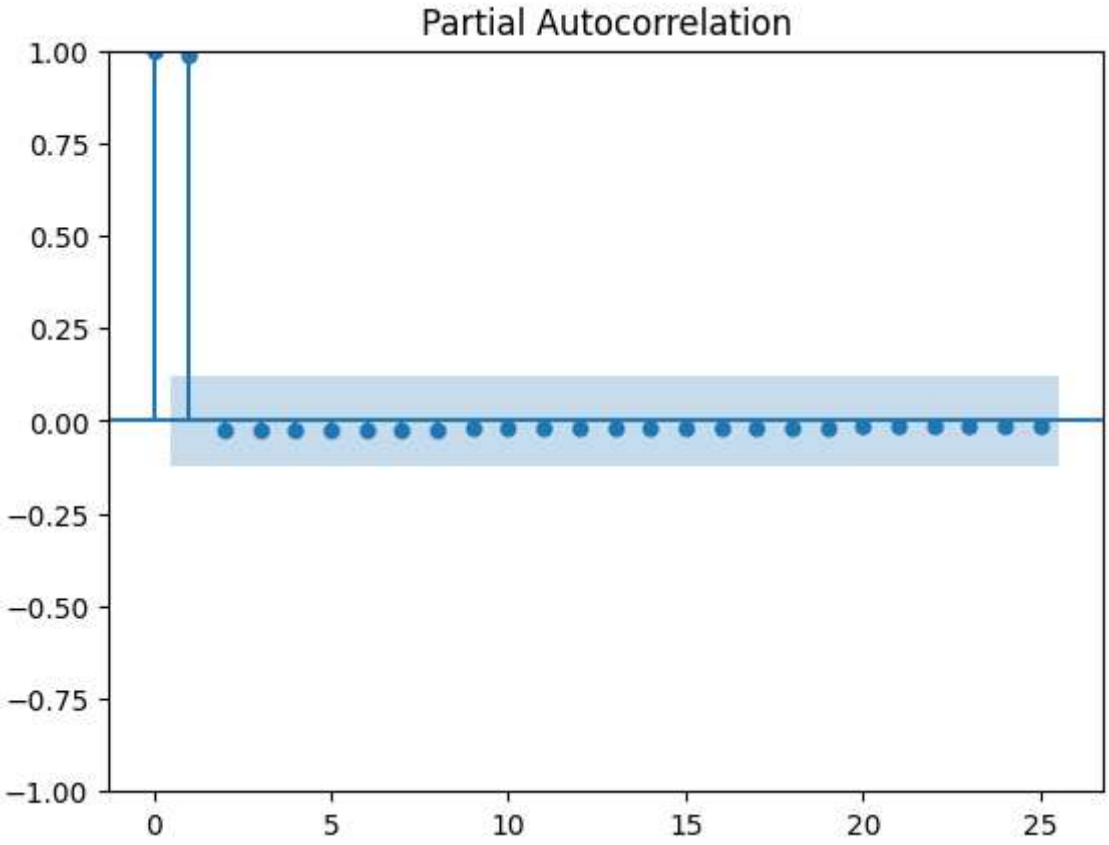
In [35]:
```
# Function for the autocorrellation plots
def corr_plots(data):
    acf_diff = plot_acf(data)
    pacf_plot = plot_pacf(data)
    adfttest = adfuller(data)
    print(f"The p value is {adfttest[1]}")

    return acf_diff, pacf_plot
```

In [36]:
```
corr_plots(df_model2)
```

```
The p value is 0.27562767729226234
```

Out[36]:
```
(<Figure size 640x480 with 1 Axes>, <Figure size 640x480 with 1 Axes>)
```





From the auto-correlation plots, it is very evident that the house value drop between 2008 and 2012 introduced very high seasonality.

Baseline model

Building a forecasting model on the whole data seems infeasible. We seek to narrow down the forecast to only the afore mentioned **top 30 zipcodes** with the highest prices.

The approach taken was looping through the zipcodes column, train a baseline model, forecast using lagged values and finally saving the forecasted values together with their respective percentage change. The **TimseSeriesSplit** is used to split the data into train and test splits on the index column. The baseline model uses the **pdq values** of order **1,0,1** as obtained from the correlation plots.

In [51]:
```
# Baseline model for forecast of specific zip codes.
unique_counties = [10021, 10011, 10014, 10128, 94027, 81611, 90210,
33480, 94123, 31561,
                   94115, 94109, 81615, 90402, 92067, 90020, 94028,
94301, 7620, 92657,
                   94022, 94957, 93108, 11217, 94305, 2116, 90265, 94133,
11976, 90049]

tsc = TimeSeriesSplit(n_splits=5)

all_train_dates, all_test_dates = [], []
all_train_values, all_test_values = [], []
all_forecasts = []

results_list1 = []

for county in unique_counties:
    county_data = df_model[df_model['RegionName'] == county]

    # Split the data
    dates = county_data.index
    values = county_data['value'].values

    # accumulate values for each fold
    zip_train_dates, zip_test_dates = [], []
    zip_train_values, zip_test_values = [], []

    for train_index, test_index in tsc.split(dates):
        train_dates, test_dates = dates[train_index], dates[test_index]
        train_values, test_values = values[train_index],
values[test_index]

        zip_train_dates.extend(train_dates)
        zip_test_dates.extend(test_dates)
        zip_train_values.extend(train_values)
        zip_test_values.extend(test_values)


        baseline_model = ARIMA(train_values, order=(1, 0, 1)).fit()
        forecast =
pd.Series(baseline_model.predict(start=len(train_values),
end=len(train_values) + 11, typ='levels'))

        print(baseline_model.summary())

        all_forecasts.extend(forecast)


        forecasted_value = forecast.iloc[-1]
        real_value = test_values[-1]
        percentage_change = ((forecasted_value - real_value) /
real_value) * 100
        # Append the forecasted values to the list
        results_list1.append({'County': county, 'Forecast':
forecasted_value, 'Real': real_value, 'Percentage Change':
percentage_change})
```

```
results_df = pd.DataFrame(results_list1)
```

```
                               SARIMAX Results
==========================================================================
=======
Dep. Variable:                        y   No. Observations:
32
Model:                 ARIMA(1, 0, 1)   Log Likelihood                -
441.951
Date:                Thu, 18 Jan 2024   AIC
891.901
Time:                        20:24:26   BIC
897.764
Sample:                             0   HQIC
893.844
                                  - 32
Covariance Type:                  opg
==========================================================================
=======
                 coef    std err          z      P>|z|      [0.025
0.975]
```

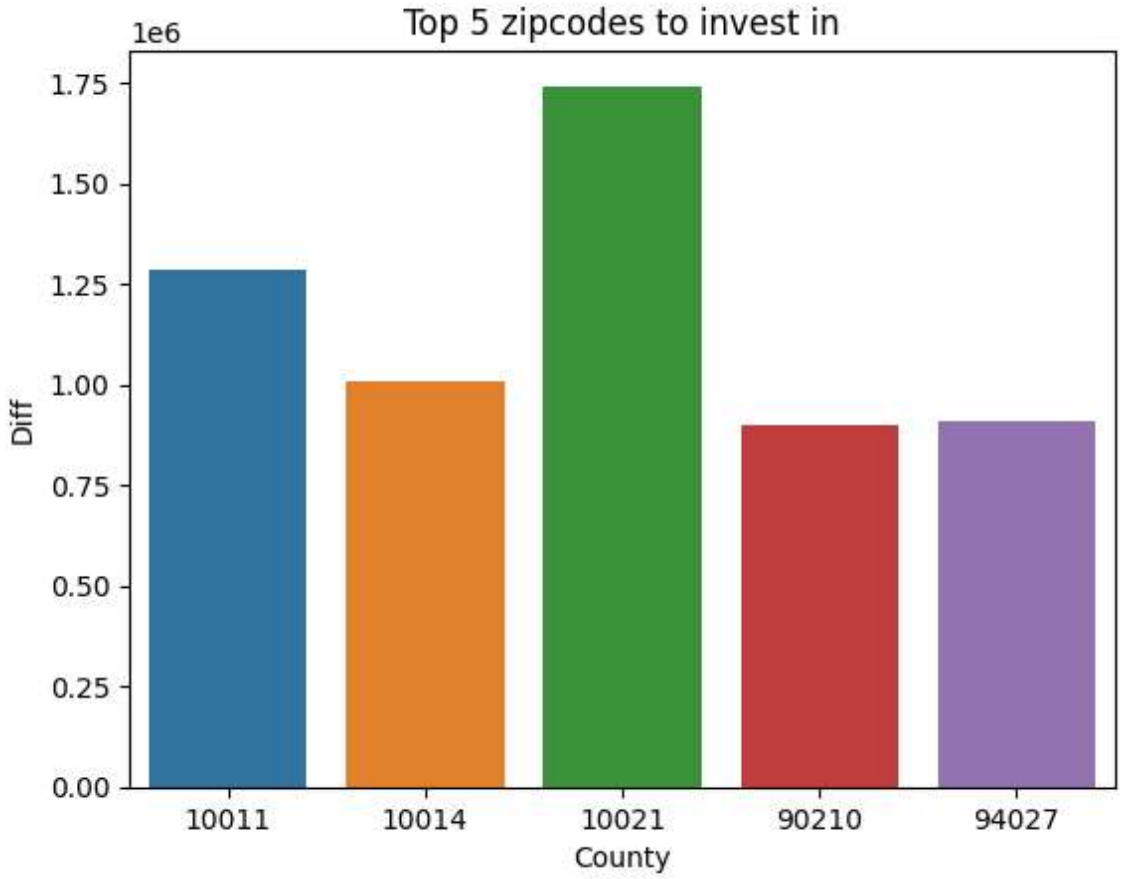
In [50]:
```
results_df.head()
```

Out[50]:

	County	Forecast	Real	Percentage Change
0	10021	9.830246e+06	10883800.0	-9.680016
1	10021	1.077231e+07	13043800.0	-17.414338
2	10021	1.275458e+07	11630200.0	9.667799
3	10021	1.158979e+07	18341500.0	-36.811086
4	10021	1.814077e+07	17894900.0	1.373952

The county column seems to have duplicated values and this is because the **TimeSeriesSplit** acts as a cross validation technique and splits data into folds.

In [38]:
```
#Plotting the zipcodes with high forecasted values
avg_results = results_df.groupby('County')[['Forecast',
'Real']].mean().reset_index()
avg_results['Diff'] =avg_results['Real'] - avg_results['Forecast']
top_5 = avg_results.sort_values(by='Diff', ascending=False).head(5)
sns.barplot(data=top_5, x='County', y='Diff')
plt.title("Top 5 zipcodes to invest in")
plt.show()
```



The **ZipCodes 10021(1.70%), 10011(1.25%), 10014(0.95%), 90210(0.8%) and 94027(0.8%)** are the regions with greater prospects of having value price increase. In consideration of future prices, these are the best zipcodes to invest in.

Auto arima model

After doing forecasting on the selected zipcodes using a baseline model, we go ahead and try to use the auto arima to get the best pdq combinations to minimze the AIC score.

The idea of looping and using the **TrainTestSplit** comes in handy as various combinations are tried for every fold to get the lowest AIC scores.

In [57]:
```
# Use the auto arima model to get the best pdq values.
for county in unique_counties:
    county_data = df_model[df_model['RegionName'] == county]

    # Split the data
    dates = county_data.index
    values = county_data['value'].values

    # accumulate results for each fold
    zip_train_dates, zip_test_dates = [], []
    zip_train_values, zip_test_values = [], []

    for train_index, test_index in tsc.split(dates):
        train_dates, test_dates = dates[train_index], dates[test_index]
        train_values, test_values = values[train_index],
values[test_index]

        zip_train_dates.extend(train_dates)
        zip_test_dates.extend(test_dates)
        zip_train_values.extend(train_values)
        zip_test_values.extend(test_values)


        auto_model = auto_arima(train_values, trace=True,
suppress_warnings=True)
        auto_model.summary()
```

```
Performing stepwise search to minimize aic
 ARIMA(2,0,2)(0,0,0)[0] intercept   : AIC=798.472, Time=0.20 sec
 ARIMA(0,0,0)(0,0,0)[0] intercept   : AIC=991.727, Time=0.01 sec
 ARIMA(1,0,0)(0,0,0)[0] intercept   : AIC=983.334, Time=0.04 sec
 ARIMA(0,0,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.10 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=1124.743, Time=0.01 sec
 ARIMA(1,0,2)(0,0,0)[0] intercept   : AIC=858.266, Time=0.16 sec
 ARIMA(2,0,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.26 sec
 ARIMA(3,0,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.39 sec
 ARIMA(2,0,3)(0,0,0)[0] intercept   : AIC=783.861, Time=0.28 sec
 ARIMA(1,0,3)(0,0,0)[0] intercept   : AIC=833.611, Time=0.18 sec
 ARIMA(3,0,3)(0,0,0)[0] intercept   : AIC=inf, Time=0.45 sec
 ARIMA(2,0,4)(0,0,0)[0] intercept   : AIC=770.890, Time=0.24 sec
 ARIMA(1,0,4)(0,0,0)[0] intercept   : AIC=823.210, Time=0.16 sec
 ARIMA(3,0,4)(0,0,0)[0] intercept   : AIC=inf, Time=0.50 sec
 ARIMA(2,0,5)(0,0,0)[0] intercept   : AIC=779.522, Time=0.46 sec
 ARIMA(1,0,5)(0,0,0)[0] intercept   : AIC=848.143, Time=0.18 sec
 ARIMA(3,0,5)(0,0,0)[0] intercept   : AIC=inf, Time=0.64 sec
 ARIMA(2,0,4)(0,0,0)[0]             : AIC=777.923, Time=0.18 sec
```

The best pdq values are printed for each fold. This is so because the **TimeSeriesSplit** splits the training and testing data into various folds for easier analysis. With the **Cross-validation** technique, we are sure whatver the pdq combinations to be obtained give the best scores for analysis.

- **Challenge:** The results may be alot and requires going through eac fold to get the combination that is frequent in several folds with lower **AIC scores**.

ARIMA(0,2,0)

From the various pdq combinations, the one with the lowest score accross several folds is **0,2,0** and **1,2,1**. These are the combinations to test out on the **ARMA** models and analyze their AIC scores. This is because most time series models are prone to overfitting and as such there is need to test out more than one comnination.

In [40]:

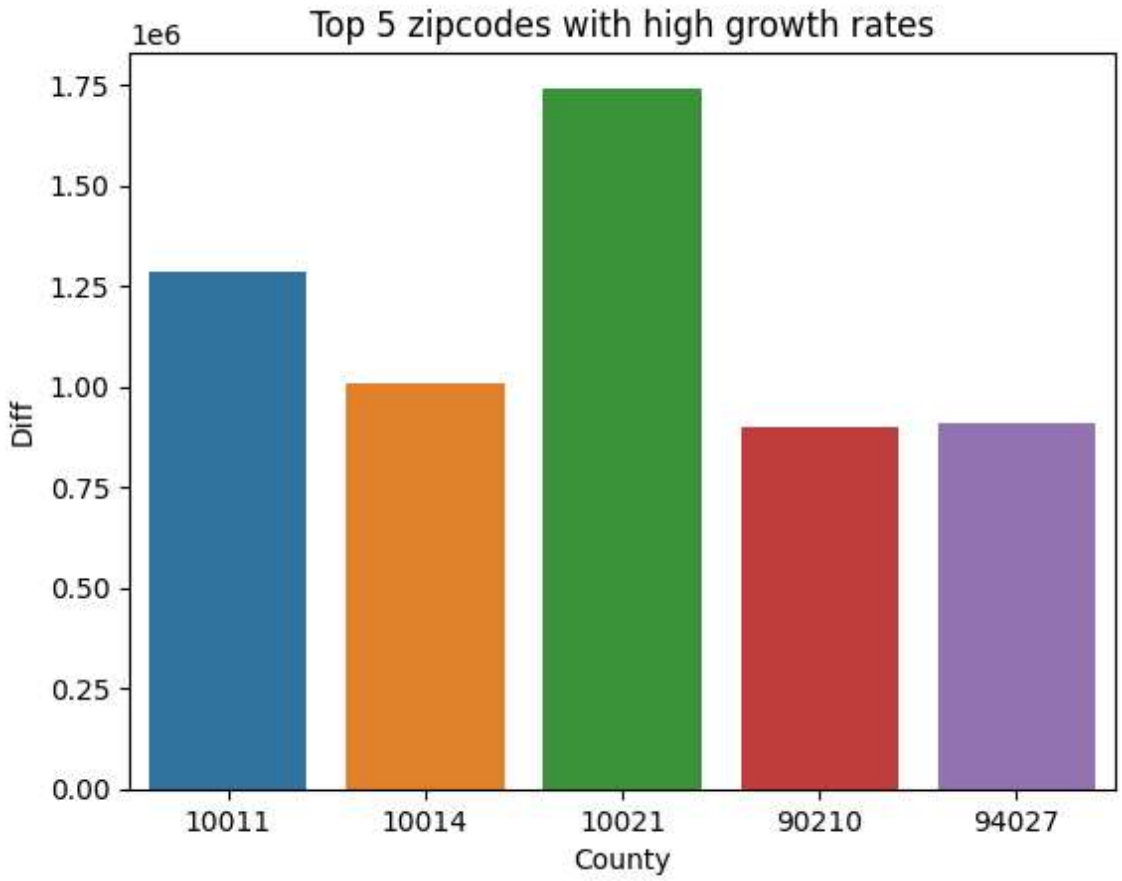
```
def model_func(order):
    """
    Function to take in various pdq values and do forecasting on the
selected values.
    """

    results_list2 = []
    for county in unique_counties:
        county_data = df_model[df_model['RegionName'] == county]

        # Split the data
        dates = county_data.index
        values = county_data['value'].values

        # accumulate values for each fold
        zip_train_dates, zip_test_dates = [], []
        zip_train_values, zip_test_values = [], []

        for train_index, test_index in tsc.split(dates):
            train_dates, test_dates = dates[train_index],
dates[test_index]
            train_values, test_values = values[train_index],
values[test_index]

            zip_train_dates.extend(train_dates)
            zip_test_dates.extend(test_dates)
            zip_train_values.extend(train_values)
            zip_test_values.extend(test_values)


            model = ARIMA(train_values, order=order).fit()
            forecast = pd.Series(model.predict(start=len(train_values),
end=len(train_values) + 11, typ='levels'))

            print(model.summary())

            all_forecasts.extend(forecast)


            forecasted_value = forecast.iloc[-1]
            real_value = test_values[-1]
            percentage_change = ((forecasted_value - real_value) /
real_value) * 100

            results_df = pd.DataFrame(results_list2.append({'County':
county, 'Forecast': forecasted_value, 'Real': real_value, 'Percentage
Change': percentage_change}))

    return results_df
```

In [41]:
```
#Apply function with the pdq values of (0,2,0)
model_func(order=(0,2,0))
```

```
                               SARIMAX Results
==========================================================================
=======
Dep. Variable:                        y   No. Observations:
32
Model:                 ARIMA(0, 2, 0)   Log Likelihood                -
387.275
Date:                Thu, 18 Jan 2024   AIC
776.550
Time:                        19:54:00   BIC
777.951
Sample:                             0   HQIC
776.998
                                  - 32
Covariance Type:                  opg
==========================================================================
=======
                 coef    std err          z      P>|z|      [0.025
0.975]
```

This model generates an AIC score of 776 which is pretty much lower than the baseline's model score of 891.

In [42]:
```
# Analyze the forecasted house values to get their respective growth
rates.
avg_results2 = results_df.groupby('County')[['Forecast',
'Real']].mean().reset_index()
avg_results2['Diff'] =avg_results['Real'] - avg_results2['Forecast']
top_5 = avg_results2.sort_values(by='Diff', ascending=False).head(5)
sns.barplot(data=top_5, x='County', y='Diff')
plt.title("Top 5 zipcodes with high growth rates")
plt.show()
```



ARIMA(1,2,1)

We set out to try another combination of pdq values to analyze the **AIC scores** as compaired to the other **ARIMA** model.

In [44]:
```
#Apply function using the order (1,2,1).
model_func(order=(1,2,1))
```

```
                               SARIMAX Results
==============================================================================
=======
Dep. Variable:                          y   No. Observations:
32
Model:                 ARIMA(1, 2, 1)   Log Likelihood                -
387.712
Date:                Thu, 18 Jan 2024   AIC
781.424
Time:                        19:59:10   BIC
785.627
Sample:                             0   HQIC
782.768
                                  - 32
Covariance Type:                  opg
==============================================================================
=======
                 coef    std err          z      P>|z|      [0.025
0.975]
```

This model generates a higher AIC than the previous model. We aim to reduce the AIC scores. This model is not the perfect model to get the zipcodes which have a better prospects of growth.

FaceBook model

After testing out the various ARIMA models, we seek to try out the **FaceBook** prophet model which handles the concept of seasonalty pretty well. The prophet model however requires the data columns to be in **ds** for the date column and **y** for the target column.

To have a simple modelling process, we reset the index for the modelling data to make the time column extraction easy using the loop and later save the results to a CSV file. This data will be processed through an **Extract-Transform-Load** pipeline having the Prophet model.

The concept of automation is very paramount because it helps in easier code manageability and debugging. It also offers better scalability especially when dealing with huge projects. this is manageable by using the traditional **OOP** approach where everything is automated from the ingestion of data, renaming of columns, dropping null values, training, evaluating a model and later saving the model for future deployment purposes.

In [45]:
```
df_model.reset_index(inplace=True)
```

In [46]:
```
# Create a csv file for easier prediction of top performing zipcodes.
for county in unique_counties:
    county_data = df_model[df_model['RegionName'] == county][['time',
'value']].copy()
    county_data.to_csv('Zipcode_predict.csv')
```

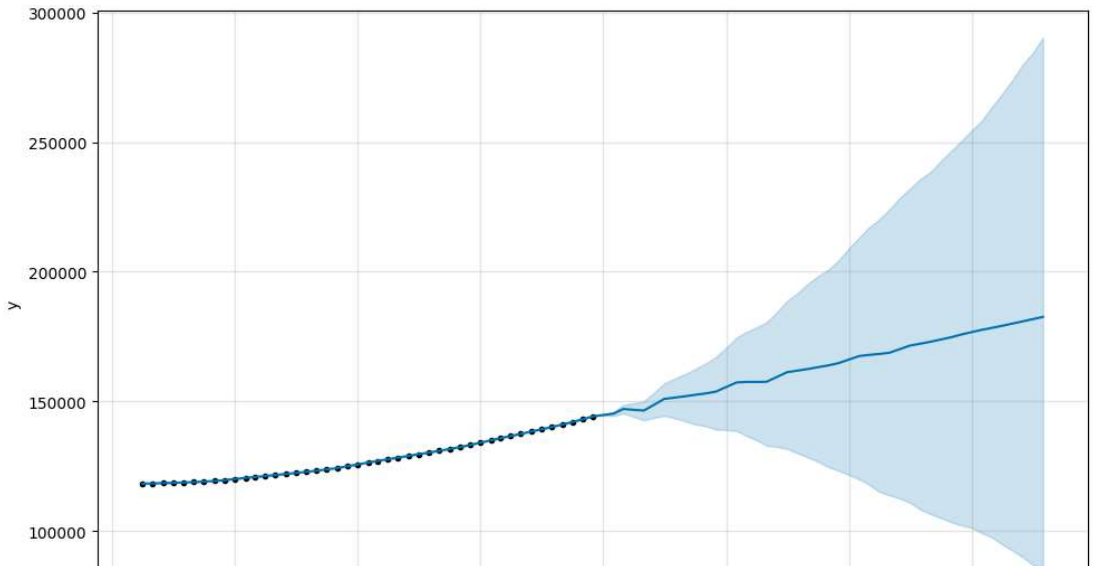
Average House prediction

The data to be used in the forecasting of the average house prices for every month comes from the melt function. The data is aggregated on a monthly basis together with their respective average values.

In [56]:
```
# ETL-like class pipeline to apply the Extract-Transform-Load
methodology.
class TimeSeriesPipeline:
    def __init__(self, data_path, n_splits=5):
        self.data_path = data_path
        self.data = None
        self.model = None
        self.forecast = None
        self.n_splits = n_splits
    # Load data from Excel file.
    def load_data(self):
        self.data = pd.read_excel(self.data_path)
    # Preprocess data by converting date column to datetime format and
removing duplicates.
    def preprocess_data(self, target_column='y', ds_column='ds'):
        self.data[ds_column] = pd.to_datetime(self.data[ds_column])
        self.data = self.data.rename(columns={ds_column: 'ds',
target_column: 'y'})

        if self.data.duplicated('ds').any():
            self.data = self.data.drop_duplicates(subset='ds',
keep='first')
    # Train Prophet model on training data.
    def train_model(self, train_data):
        self.model = Prophet(seasonality_mode='multiplicative')
        self.model.fit(train_data)
    # Make predictions on test data using trained Prophet model.
    def make_predictions(self, test_data):
        future = self.model.make_future_dataframe(periods=len(test_data),
freq='MS')
        self.forecast = self.model.predict(future)
    # Plot the forecasted values to analyze performance visually.
    def plot_forecast(self):
        fig = self.model.plot(self.forecast)
        fig2 = self.model.plot_components(self.forecast)
        plt.show()
    # Evaluate the performance of the trained model on test data.
    def evaluate_model(self, test_data, y_true='y', y_pred='yhat'):
        y_true = test_data[y_true].values
        y_pred = self.forecast[-len(test_data):][y_pred].values
        mse = mean_squared_error(y_true, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_true, y_pred)

        return r2, mse, rmse
    # Run the entire pipeline to train, make predictions, and evaluate
the model.
    def run_pipeline(self, target_column='y', ds_column='ds'):
        self.load_data()
        self.preprocess_data(target_column=target_column,
ds_column=ds_column)

        tscv = TimeSeriesSplit(n_splits=self.n_splits)
        evaluation_scores = []
        for train_index, test_index in tscv.split(self.data):
            train_data, test_data = self.data.iloc[train_index],
self.data.iloc[test_index]

            self.train_model(train_data)
            self.make_predictions(test_data)
            self.plot_forecast()
```

```
            r2, mse, rmse = self.evaluate_model(test_data, y_true='y',
y_pred='yhat')
            evaluation_scores.append((r2, mse, rmse))

        return evaluation_scores
# Save the trained model to a file.
class SaveModel(TimeSeriesPipeline):
    def save_model(self, model_path='house_value_predictor.pkl'):
        joblib.dump(self.model, model_path)


data_path = "output_2.xlsx"
target_column = 'y'
pipeline = SaveModel(data_path)
evaluation_scores = pipeline.run_pipeline(target_column='value',
ds_column='time')
pipeline.save_model()

for i, scores in enumerate(evaluation_scores, 1):
    print(f"R2 = {scores[0]}, MSE = {scores[1]}, RMSE = {scores[2]}")
```

```
20:57:15 - cmdstanpy - INFO - Chain [1] start processing
20:57:45 - cmdstanpy - INFO - Chain [1] done processing
```



Working with the ETL pipeline produces the best results because the training and test sets are split into folds. This provides a cross validation technique. From the plots, the model learns the pattern pretty well even during the 2008 crisis.

Doing a cross validation of every split to get the test scores for the **evaluate model** is however not feasible because of larget training times it takes. It is for this reason that we opt to depend on the visualization of forecasted and real values.

The prophet model learns the seasonality in the zillow housing data as shown by the plots of every fold. The forecast shows a steady rise of home value for the specified zipcodes.

There is an yearly season where the house values normally peak December.

ZipCodes house prediction

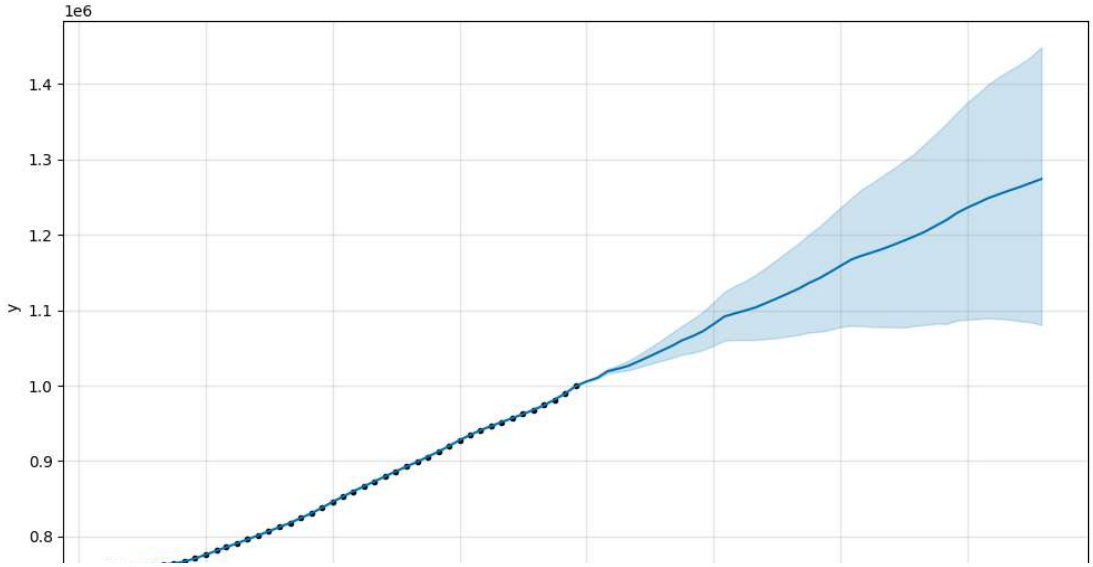
Using the data saved in the csv file **Zipcode_predict.csv**, we set out to investigate how the facebook prophet handles the predictions for the top 30 zipcodes that had houses with higher values.

Just like before, obtaining the scores from the model seems pretty time consuming and may not be feasible at this time.

In [62]:
```
# Pipeline to predict the prices of the specified zip codes.
class TimeSeriesPipeline2:
    def __init__(self, data_path, n_splits=5):
        self.data_path = data_path
        self.data = None
        self.model = None
        self.forecast = None
        self.n_splits = n_splits
    # Function to read in the data from the specified file path.
    def load_data(self):
        self.data = pd.read_csv(self.data_path)
    # Function to preprocess the data by converting the date column to
datetime format,
    def preprocess_data(self, target_column='y', ds_column='ds'):
        self.data[ds_column] = pd.to_datetime(self.data[ds_column])
        self.data = self.data.rename(columns={ds_column: 'ds',
target_column: 'y'})

        if self.data.duplicated('ds').any():
            self.data = self.data.drop_duplicates(subset='ds',
keep='first')
    # Function to train the model on the training data.
    def train_model(self, train_data):
        self.model = Prophet(seasonality_mode='multiplicative')
        self.model.fit(train_data)
    # Function to make predictions on the test data.
    def make_predictions(self, test_data):
        future = self.model.make_future_dataframe(periods=len(test_data),
freq='MS')
        self.forecast = self.model.predict(future)
    # Function to plot the forecast.
    def plot_forecast(self):
        fig = self.model.plot(self.forecast)
        fig2 = self.model.plot_components(self.forecast)
        plt.show()
    # Function to evaluate the model on the test data.
    def evaluate_model(self, test_data, y_true='y', y_pred='yhat'):
        y_true = test_data[y_true].values
        y_pred = self.forecast[-len(test_data):][y_pred].values
        mse = mean_squared_error(y_true, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_true, y_pred)

        return r2, mse, rmse
    # Function to run the entire pipeline.
    def run_pipeline(self, target_column='y', ds_column='ds'):
        self.load_data()
        self.preprocess_data(target_column=target_column,
ds_column=ds_column)

        tscv = TimeSeriesSplit(n_splits=self.n_splits)
        evaluation_scores = []
        for train_index, test_index in tscv.split(self.data):
            train_data, test_data = self.data.iloc[train_index],
self.data.iloc[test_index]

            self.train_model(train_data)
            self.make_predictions(test_data)
            self.plot_forecast()
            r2, mse, rmse = self.evaluate_model(test_data, y_true='y',
y_pred='yhat')
```

```
            evaluation_scores.append((r2, mse, rmse))

        return evaluation_scores

# Pipeline to save the trained model.
class SaveModel(TimeSeriesPipeline2):
    def save_model(self, model_path='zipCode_value_predictor.pkl'):
        joblib.dump(self.model, model_path)


data_path = "Zipcode_predict.csv"
target_column = 'y'
pipeline = SaveModel(data_path)
evaluation_scores = pipeline.run_pipeline(target_column='value',
ds_column='time')
pipeline.save_model()

# Print the scores for every fold.
for i, scores in enumerate(evaluation_scores, 1):
    print(f"R2 = {scores[0]}, MSE = {scores[1]}, RMSE = {scores[2]}")
```

```
22:02:03 - cmdstanpy - INFO - Chain [1] start processing
22:02:03 - cmdstanpy - INFO - Chain [1] done processing
```



From the plots, what can be deduced is that the model predicts an upward trend of house prices in the specific ZipCodes and in the event of any future crisis, the model has demonstrated its capability of learning the patterns and trends exhibited by the data.

Modelling Conclusion

Three models were used specifically the baseline ARIMA model, the auto arima model to find the best pdq combinations, ARIMA models with the best combination sto minimize the AIC scores and finally the prophet model using an ETL pipeline.

The ARIMA model with the lowest AIC scores of **716** was the one of order **0,2,0.** This model gave us the best possible forecasted values to answer our business objective. To find the forecasted value for each zipcode, the forecasted value was averaged.

The use of **TimeSeriesSplit** as a cross validation technique came in handy. This is because the data in the training and test sets is split into folds so as to get a variety of several forecasted value for each zipCode. These forecasted values are then averaged to get the real forecast value for every zipCode.

The **Facebook** model also came in handy because it handles seasonality well especially with the financial crisis in 2008 when house prices tanked. From the plots, the model showed an upward trend of house prices in the zipcodes but each with a different percentage as observed from the forecasted results best ARIMA model. This model captured the seasonality and from the plots, it is clear houses fetch for higher prices between December.

The **facebook model** has an **R2** score of 74% and 73.66%.

In the event of any future financial crisis, the facebook prophet model has parameters which can be adjusted to capture the seasonality and learn the pattern in data. The parameters include:

- The add_seasonality function
- Adjusting the fourier order to capture complex seasonality(High fourier orders capture complex patterns)
- Adding other regressors which may impact on the way the model learns the data.

With all this said, the **Prophet** model has very useful parameters that can be adjusted in the event of any crisis.

The models to forecast both the average monthly house values and the zip codes values have been saved to pickle files and may be used during the production and deployment

4. Conclusion

1.The top 5 zipcodes to invest in that guarantee a good **ROI** include:

- 10021, 10011, 10014, 90210 and 94027.

2. The market crisis had a severe effect on zipcodes 11217, 90049, 94022, and 94301.
3. The forecasting model had a R2 score of 74% for forecasting average house values per month. The second model had an accuracy score of 73.66% on the forecasting of house prices for the Regions(ZipCodes) with high house prices.
4. The metropolitan areas showing higher growth rates are Gleenwood Springs at 200% followed by Brunswick and Panama at 225% and 75% respectively. The ZipCodes(Regions) with high growth rates are 27980(2.5%), 30032(2%), 80216(1.75%) and 89030(1.5%).

Type *Markdown* and LaTeX: α^2

Type *Markdown* and LaTeX: α^2

In [ ]:

In [ ]: