1 Scouting New Development Locations with Time Series Forecasting

2 Overview

In this project, I use data on the monthly average of real estate prices across different zip codes to build a time series model. This time series model is used to make forecasts on the most profitable retail development location in the Tucson, AZ metro area.

Data from Zillow (https://www.zillow.com/research/data/) & unitedstateszipcodes.org (https://www.unitedstateszipcodes.org/zip-code-database/).

3 Business Problem

Recent census data (https://www.census.gov/newsroom/press-releases/2019/popest-nation.html) suggests that Arizona is one of the 5 fastest growing states in the nation, in terms of both raw numbers & percent growth.

Currently, Simon Property Group owns 3 retail spaces (https://en.wikipedia.org/wiki/List_of_Simon_Property_Group_properties#Arizona) in the state. They'd like to develop another, given that other states with similar population counts (6.5 - 8.5 million compared to Arizona's 7.15 million) have anywhere from 4-14 properties owned by the group. Since 2 of their Arizona properties are located in the Phoenix metro area, their primary focus is on the state's next biggest metro: Tucson.

As part of a team hired by Simon Property, my job is to **use time series analysis of data from Zillow to forecast which zip codes out of the most populous in Tucson will have the best Return on Investment.**

Finally, even though the Zillow data covers housing (and not commercial) real estate, a positive ROI value for residential real estate will be a useful indicator of:

- A projected increase in retail spending & economic growth.

- A proof of concept for future work with commercial data.

4 Importing Data, Necessary Libraries

In [1]:
```
import warnings
warnings.filterwarnings('ignore')

import pandas as pd
import numpy as np
import itertools
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.dates as mpdates
%matplotlib inline

from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.arima.model import ARIMA
```

In [2]:
```
# Main dataset
df = pd.read_csv('Data/zillow_data.csv')

# Supplementary dataset w/ population estimates
zip_info = pd.read_csv('Data/zip_code_database.csv')
```

In [3]:
```
df.head()
```

Out[3]:

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

5 rows × 272 columns

5 Data Preprocessing, Reformatting & Filtering for Locations of Interest

Looking at the DataFrame head above, a couple of things stick out:

- The DataFrame is in *wide* format, where the average transaction price for every month is grouped into the same row (by zip code, in this instance). This is different from the *long* format data I'm used to working with, where the index for each row is a datetime measure (in this case, a month marker for each distinct zip code). The *wide* format has its benefits -- namely, that it drastically reduces the number of rows needed. For this time series project, though, it makes more sense to switch over to the *long* format before I begin exploring the data.
- From previous experience, even monthly datetime objects in Pandas have had a '-01' (day) marker tacked onto the end of each month. In other words, the date column names (which will eventually become the DataFrame row indices) appear to be strings and not datetime objects, which will cause problems down the line. Once this is confirmed, a conversion of these column names to datetime objects is necessary.

5.1 Preprocessing: Date Column Names as Datetime Objects

In [4]:
```
df.columns.values[:10]
```

Out[4]:
```
array(['RegionID', 'RegionName', 'City', 'State', 'Metro', 'CountyName',
       'SizeRank', '1996-04', '1996-05', '1996-06'], dtype=object)
```

As seen above, the date columns names are currently numeric strings. Since the date columns are the only numeric ones in this dataset, I write a function to convert all numeric column names to datetime objects.

In [5]:
```
# Creating a function that changes numeric
# columns to datetime object columns

def get_datetimes(df):
    """Separates DataFrame columns into numeric
    and non-numeric categories before converting
    numeric columns to datetime objects."""
    non_dt_cols = []
    dt_cols = []

    for c in df.columns:
        if c[0].isnumeric():
            dt_cols.append(c)
        else:
            non_dt_cols.append(c)

    dt_cols = list(pd.to_datetime(dt_cols,
                                  format='%Y-%m'))

    df.columns = non_dt_cols + dt_cols

    return df
```

In [6]:
```
df = get_datetimes(df)
df.head()
```

Out[6]:

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04-01 00:00:00
0	84654	60657	Chicago	IL	Chicago	Cook	1	334200.(
1	90668	75070	McKinney	TX	Dallas-Fort Worth	Collin	2	235700.(
2	91982	77494	Katy	TX	Houston	Harris	3	210400.(
3	84616	60614	Chicago	IL	Chicago	Cook	4	498100.(
4	93144	79936	El Paso	TX	El Paso	El Paso	5	77300.(

5 rows × 272 columns

In [7]:
```
df.columns.values[:10]
```

Out[7]:
```
array(['RegionID', 'RegionName', 'City', 'State', 'Metro', 'CountyName',
       'SizeRank', Timestamp('1996-04-01 00:00:00'),
       Timestamp('1996-05-01 00:00:00'), Timestamp('1996-06-01 00:00:0
0')],
      dtype=object)
```

As seen in both the new format of the column names, as well as the *Timestamp* at the front of each column name in the array above, the conversion has been successful.

5.2 Reformatting: Switch From Wide to Long Format & Set Datetime Index

Next, I switch from wide to long format using the Pandas *.melt()* method. This method creates rows for each month's average transaction value per zipcode. The new column of months is labeled 'Time' & and the new column of monthly average transaction price is labeled 'Price'.

In [8]:
```
# Creating a function that takes in a DataFrame
# & switches it from wide to long format

def melt_data(df):
    """Switches DataFrame from wide
    format to long format using Pandas
    .melt() method."""
    melted = pd.melt(df,
                     id_vars=['RegionID', 'RegionName',
                              'City', 'State', 'Metro',
                              'CountyName', 'SizeRank'],
                     var_name='Time',
                     value_name='Price')

    melted = melted.dropna(subset=['Price'])

    return melted
```

In [9]:
```
# Reformatting the main DataFrame

df = melt_data(df)
df
```

Out[9]:

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	Ti
0	84654	60657	Chicago	IL	Chicago	Cook	1	
1	90668	75070	McKinney	TX	Dallas-Fort Worth	Collin	2	
2	91982	77494	Katy	TX	Houston	Harris	3	
3	84616	60614	Chicago	IL	Chicago	Cook	4	
4	93144	79936	El Paso	TX	El Paso	El Paso	5	
...	...	...	...	...	...	...	...	
3901590	58333	1338	Ashfield	MA	Greenfield Town	Franklin	14719	
3901591	59107	3293	Woodstock	NH	Claremont	Grafton	14720	
3901592	75672	40404	Berea	KY	Richmond	Madison	14721	
3901593	93733	81225	Mount Crested Butte	CO	NaN	Gunnison	14722	
3901594	95851	89155	Mesquite	NV	Las Vegas	Clark	14723	

3744704 rows × 9 columns

Now that all this reformatting has been applied, it would be safe practice to check for any lingering null values.

In [10]:
```
df.isna().sum()
```

Out[10]:
```
RegionID           0
RegionName         0
City               0
State              0
Metro         236023
CountyName         0
SizeRank           0
Time               0
Price              0
dtype: int64
```

Fortunately, the only nulls left are in the 'Metro' column, which I drop along with any other regional signifier that isn't zip code.

I am also subsetting the DataFrame so that, going forward, I'm only looking at zip codes in the state of Arizona.

Finally, in order to make time series plotting & modeling easier, I set the index of the data to the 'Time' column of months.

In [11]:
```
# Filtering & re-indexing the main DataFrame

df_az = df[df['State'] == 'AZ']

to_drop = ['RegionID', 'City', 'State', 'Metro', 'CountyName', 'SizeRank']

df_az.drop(columns=to_drop, axis=1, inplace=True)
df_az.set_index('Time', inplace=True)
df_az.head()
```

Out[11]:

	RegionName	Price
Time		
1996-04-01	85032	95400.0
1996-04-01	85710	94600.0
1996-04-01	85225	101200.0
1996-04-01	85308	124800.0
1996-04-01	85281	81200.0

5.3 Selecting Zipcode Candidates & Filtering Data

The original DataFrame has now been properly reformatted. I've also narrowed the scope of the dataset to just the state of Arizona.

As the initial business problem states, though, Simon Property is looking to develop in the Tuscon metro specifically. Therefore, further specification is necessary.

Though the data provided by Zillow doesn't tell me anything about population by zip code, the data I imported from unitedstateszipcodes does have recent estimates on the populations for each US zip code.

In the first part of this section, I inspect **zip_info** to look for the 15 most populous zip codes in Tucson that shouldn't compete with the existing Tucson Premium Outlets. In the second part, I filter the main **df_az** DataFrame based on the selections in part one.

5.3.1 Selecting Tucson Zipcodes Based on Population, Proximity to Tucson Premium Outlets

In [12]:
```
# Using the supplementary dataset to find
# population estimates for Tucson zip codes

zips_tucson = zip_info[(zip_info['state'] == 'AZ') &
                        (zip_info['primary_city'] == 'Tucson')]
```

In [13]:
```
print(zips_tucson.shape)
```

```
(55, 15)
```

So, based on the printout directly above, the initial pool I'm working with is 55 zip codes in the Tucson area. Additionally, the DataFrame head below shows a column 'irs_estimated_population_2015' that provides recent population estimates for each zip code in the metro.

In [14]:
```
zips_tucson.head()
```

Out[14]:

	zip	type	decommissioned	primary_city	acceptable_cities	unacceptable_cities
37155	85701	STANDARD	0	Tucson	NaN	N
37156	85702	PO BOX	0	Tucson	NaN	N
37157	85703	PO BOX	0	Tucson	NaN	N
37158	85704	STANDARD	0	Tucson	Oro Valley	N
37159	85705	STANDARD	0	Tucson	NaN	N

In [15]:
```
# Cleaning the DataFrame so that I'm just left with
# zip codes, regional info & population estimates

to_drop = ['type', 'decommissioned', 'acceptable_cities',
           'unacceptable_cities', 'timezone', 'area_codes',
           'world_region', 'country', 'latitude', 'longitude']

zips_tucson.drop(columns=to_drop, axis=1, inplace=True)
```

In [16]:
```
zips_tucson.head()
```

Out[16]:

	zip	primary_city	state	county	irs_estimated_population_2015
37155	85701	Tucson	AZ	Pima County	3900
37156	85702	Tucson	AZ	Pima County	1060
37157	85703	Tucson	AZ	Pima County	805
37158	85704	Tucson	AZ	Pima County	27140
37159	85705	Tucson	AZ	Pima County	41250

In [17]:
```
# Further DataFrame refinement, this time returning
# list of most populous zip codes in descending order

zips_tucson = zips_tucson.groupby('zip').aggregate({'irs_estimated_populati

zips_tucson.rename(columns={'irs_estimated_population_2015': 'population_es
                   inplace=True)
zips_tucson.sort_values(by=['population_est'],
                        ascending=False, inplace=True)

zips_tucson.head(20)
```

Out[17]:

	population_est
zip	
85706	48760
85710	45660
85705	41250
85713	38340
85746	37800
85711	32720
85730	32640
85745	29860
85741	29680
85756	27950
85704	27140
85743	25940
85742	24380
85718	23930
85747	23620
85712	23610
85716	23520
85719	22540
85750	21390
85737	20430

With a list of Tucson zipcode populations in descending order, I can make a selection of 15 zip codes to analyze.

Before doing that, I must make some business problem-based decisions on which candidates to drop preemptively. Since Tucson Premium Outlets is already located in 85742, I drop this entry before choosing. Furthermore, it makes sense to drop any zip codes in 85742's immediate vicinity, so based on this map provided by the City of Tucson (https://www.tucsonaz.gov/files/pdsd/wardzip.pdf), the areas of 85743, 85741, 85704, 85745, 85705, & 85718 will be dropped as well.

In [18]:
```
# Dropping zip codes near existing retail
# property, choosing 15 most populous afterward

to_drop = [85742, 85743, 85741, 85704, 85745, 85705, 85718]

zips_tucson.drop(labels=to_drop, axis=0, inplace=True)
zips_tucson.head(15)
```

Out[18]:

	population_est
zip	
85706	48760
85710	45660
85713	38340
85746	37800
85711	32720
85730	32640
85756	27950
85747	23620
85712	23610
85716	23520
85719	22540
85750	21390
85737	20430
85757	17980
85748	17210

5.3.2 Filtering Main DataFrame Based on Selections

Now that all of my selections are ordered in the **zips_tucson** DataFrame, I can use it to check the size of each zip code subset. I also use **zips_tucson** to create a filtered version of the **df_az** dataset, **df_zips_15**, that only covers the pool of candidates I'll end up choosing from.

In [19]:
```
# Creating callable list of most populous
# zip codes in descending order

zips_tucson.reset_index(inplace=True)
zipcodes = list(zips_tucson['zip'])[:15]
```

In [20]:
```python
# Ensuring each zip code has the same amount
# of data entries

for z in zipcodes:
    print(f'Shape/size of data subset for zipcode {z}:')
    print(df_az[df_az['RegionName'] == z].shape)
    print('\n ---------------------- \n')
```

```
Shape/size of data subset for zipcode 85706:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85710:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85713:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85746:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85711:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85730:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85756:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85747:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85712:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85716:
(265, 2)

 ----------------------
```

```
Shape/size of data subset for zipcode 85719:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85750:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85737:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85757:
(265, 2)

 ----------------------

Shape/size of data subset for zipcode 85748:
(265, 2)

 ----------------------
```

In [21]:
```
# Making DataFrame of 15 zip codes from new list

df_zips_15 = df_az[df_az['RegionName'].isin(zipcodes)]
```

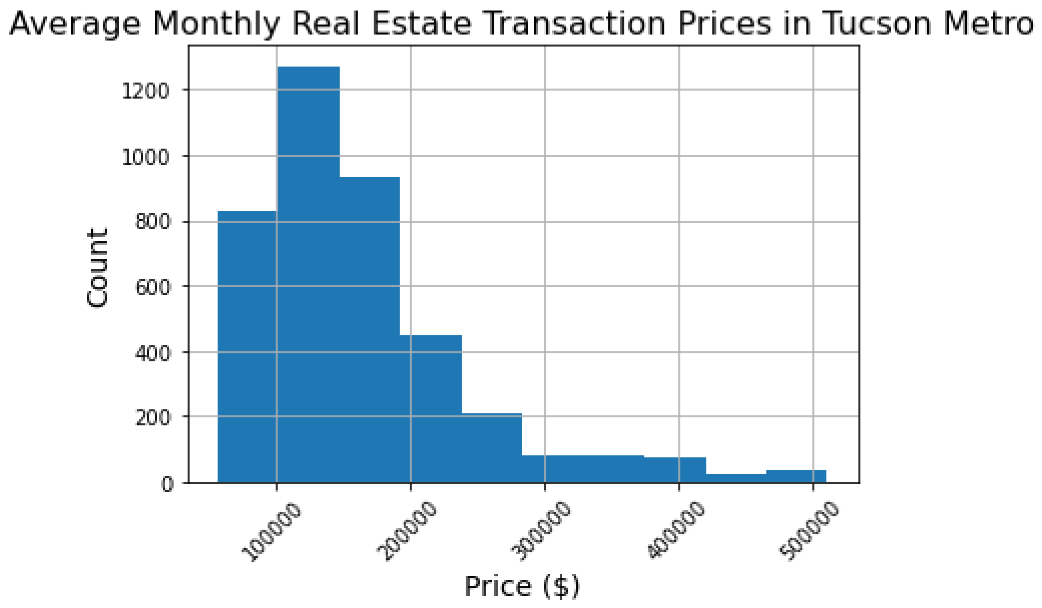
6 EDA & Visualization

6.1 Statistical Description of State & Metro Monthly Prices

In [22]:
```
# Statistical description of metro data

df_zips_15['Price'].describe()
```

Out[22]:
```
count      3975.000000
mean     162654.842767
std       77981.883267
min       57400.000000
25%      108700.000000
50%      143800.000000
75%      190000.000000
max      510900.000000
Name: Price, dtype: float64
```

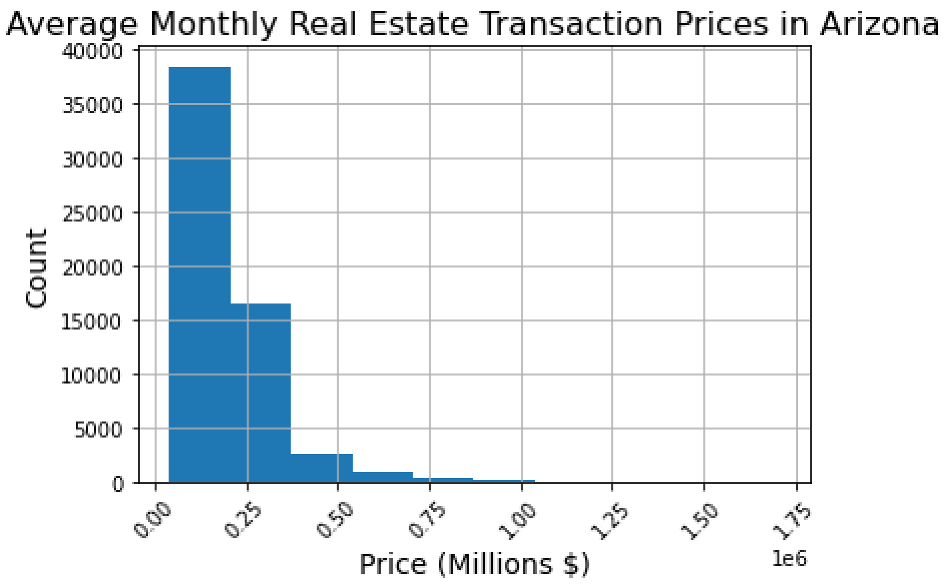
In [23]:
```
# Histogram of metro data

df_zips_15['Price'].hist()
plt.xticks(rotation=45)
plt.xlabel('Price ($)', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.title('Average Monthly Real Estate Transaction Prices in Tucson Metro',
          fontsize=16)
# plt.savefig('TucsonPrice.png', bbox_inches='tight')
plt.show()
```



In [24]:
```
# Statistical description of state data

df_az['Price'].describe()
```

Out[24]:
```
count    5.877500e+04
mean     2.012443e+05
std      1.332284e+05
min      4.070000e+04
25%      1.228000e+05
50%      1.702000e+05
75%      2.394000e+05
max      1.706300e+06
Name: Price, dtype: float64
```

In [25]:
```python
# Histogram of state data

df_az['Price'].hist()
plt.xticks(rotation=45)
plt.xlabel('Price (Millions $)', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.title('Average Monthly Real Estate Transaction Prices in Arizona',
          fontsize=16)
# plt.savefig('AZPrice.png', bbox_inches='tight')
plt.show()
```



Although both histograms are skewed to the right, the Tucson histogram has a shorter tailthan the state histogram.

From the two *.describe()* methods, we can compare the center and variability of prices at both the state and regional level.

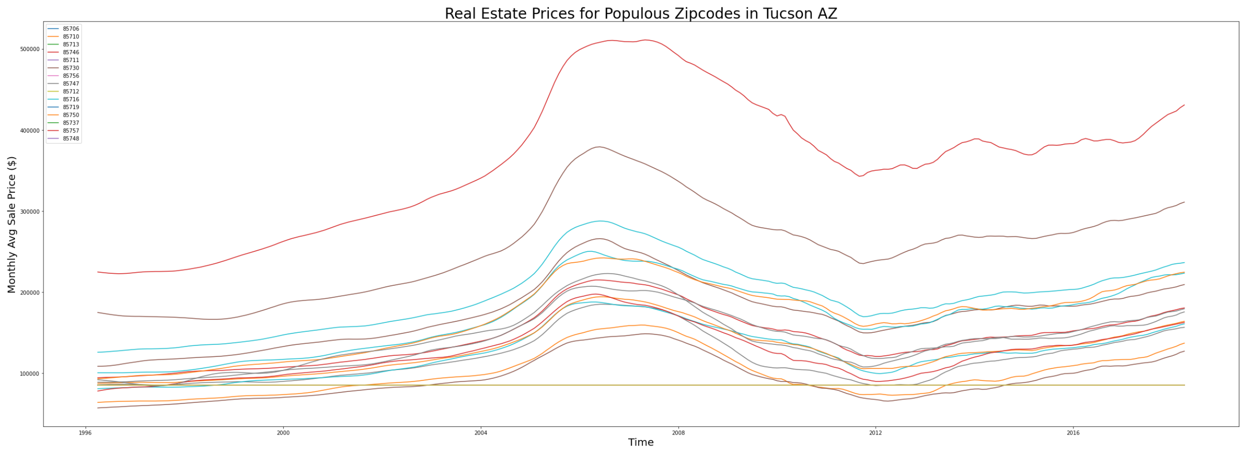
- Mean Price per Real Estate Sale: ~201,000 USD (Arizona) vs. ~163,000 USD (Tucson)
- Standard Deviation of Price per Real State Sale: ~133,0000 USD (Arizona) vs. ~78,000 USD (Tucson)

6.2 Line Plots

6.2.1 Line Plots Across Multiple Years

I start by taking a look at the plot of all zip codes across the entire time frame of the dataset.

In [26]:
```python
# Line plots of real estate price from 1996 - 2018
# for 15 zip codes of interest

fig, ax = plt.subplots(figsize=(33,12))
for z in zipcodes:
    y = df_az[df_az['RegionName'] == z]
    ax.plot(y, label = z)
ax.set_xlabel('Time', fontsize=20)
ax.set_ylabel('Monthly Avg Sale Price ($)', fontsize=20)
ax.set_title('Real Estate Prices for Populous Zipcodes in Tucson AZ',
             fontsize=28)
ax.legend(zipcodes, loc='upper left')
fig.tight_layout()
# fig.savefig('PricePlotLong.png', bbox_inches='tight')
```

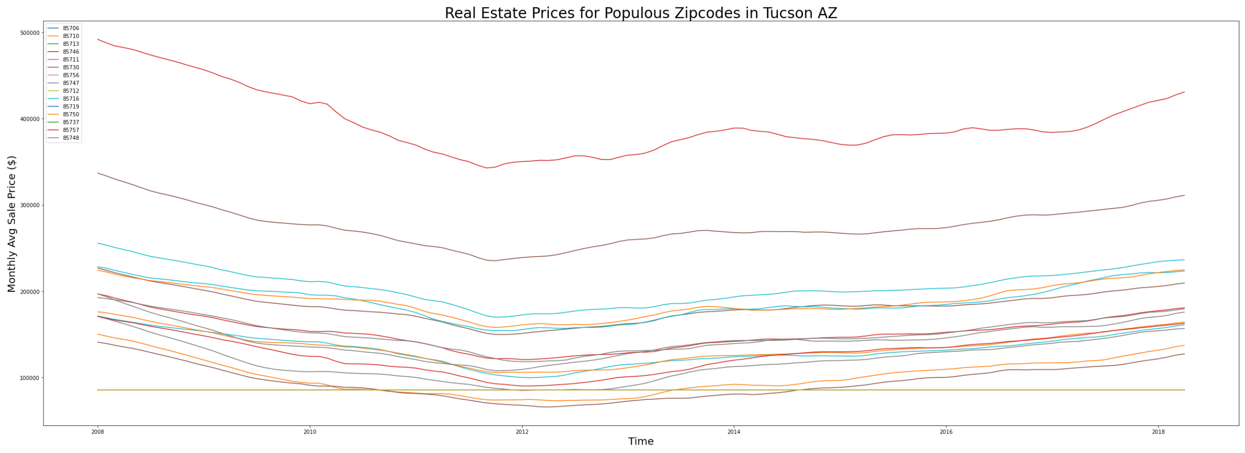


With a market as volatile as real estate, it might not make much sense to take this entire 20+ year span into consideration -- though this plot is useful in showing an overall upward trend from left end to right end.

Furthermore, the bubble, crash & subsequent recovery surrounding 2008 is a unique part of the line plot and any models trained on this catastrophe may not perform as they should.

Let's move on to narrowing the scope to 2008 onward.

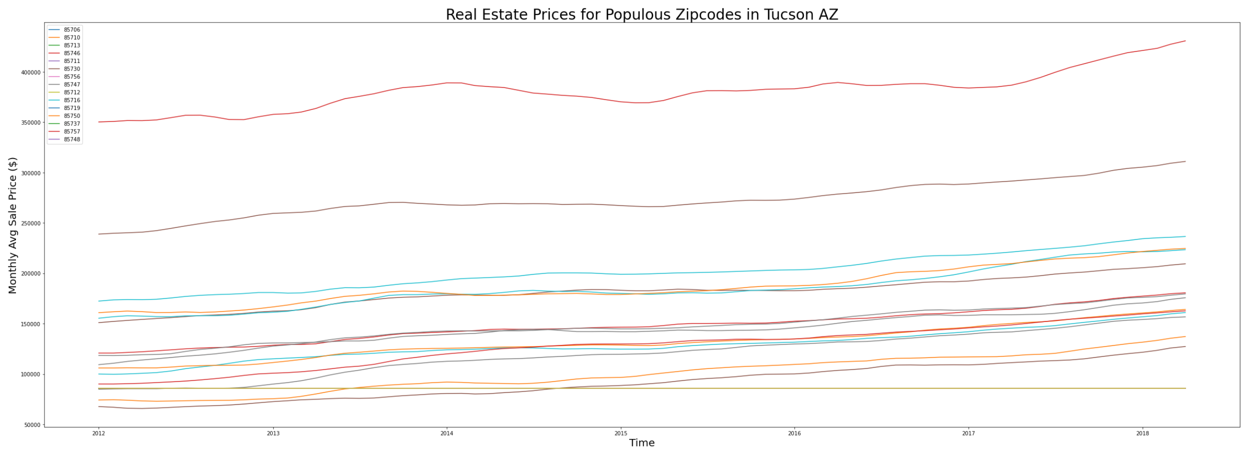
```
In [27]: # Line plots of real estate price from 2008 - 2018
         # for 15 zip codes of interest

         fig, ax = plt.subplots(figsize=(33,12))
         for z in zipcodes:
             y = df_az[df_az['RegionName'] == z]
             ax.plot(y['2008-01-01':], label = z)
         ax.set_xlabel('Time', fontsize=20)
         ax.set_ylabel('Monthly Avg Sale Price ($)', fontsize=20)
         ax.set_title('Real Estate Prices for Populous Zipcodes in Tucson AZ', fonts
         ax.legend(zipcodes, loc='upper left')
         fig.tight_layout()
```



This is much less volatile than the previous plot, but it looks like proper recovery from 2008 hasn't really begun to take place until 2012.

Let's narrow the scope once more to the year 2012 onward.

In [28]:
```
# Line plots of real estate price from 2012 - 2018
# for 15 zip codes of interest

fig, ax = plt.subplots(figsize=(33,12))
for z in zipcodes:
    y = df_az[df_az['RegionName'] == z]
    ax.plot(y['2012-01-01':], label = z)
ax.set_xlabel('Time', fontsize=20)
ax.set_ylabel('Monthly Avg Sale Price ($)', fontsize=20)
ax.set_title('Real Estate Prices for Populous Zipcodes in Tucson AZ', fonts
ax.legend(zipcodes, loc='upper left')
fig.tight_layout()
# fig.savefig('PricePlot.png', bbox_inches='tight')
```



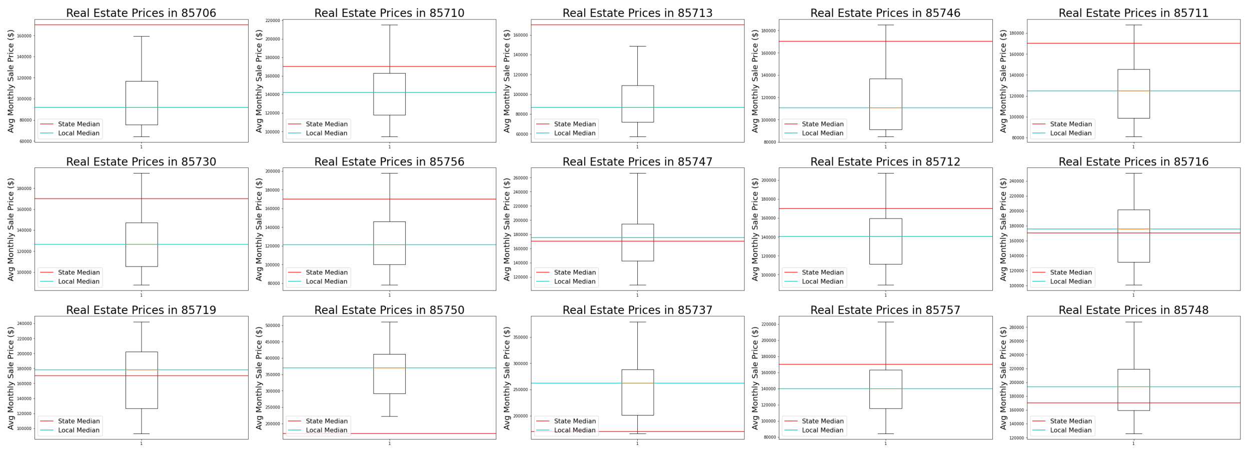
In this more recent plot, there is a noticeable upward trend across the years 2012 - 2017, and it exhibits much more stability than the previous plot. This section suggests to me that I should train my eventual model on data from the year 2012 onward, so that it doesn't anticipate another 2008-sized crash in the limited forecasts that it makes.

6.3 Box Plots

Lastly, before moving onto the modeling stage, I look at boxplots of each of the 15 zip codes' median monthly real estate price & compare them to the state median.

With this, I can get a better idea of which zip codes are relatively similar to the state in terms of wealth, and which are not.

In [29]:
```python
# Visual comparison of each zip code's
# median price with the Arizona median

fig, ax_lst = plt.subplots(nrows=3, ncols=5, figsize=(44, 16))
ax_lst = ax_lst.flatten()
for ax, zipcode in zip(ax_lst, zipcodes):
    location = df_az[df_az['RegionName'] == zipcode]
    ax.boxplot(location['Price'])
    ax.set_ylabel('Avg Monthly Sale Price ($)',
                  fontsize=20)
    ax.set_title(f'Real Estate Prices in {zipcode}',
                 fontsize=28)
    ax.axhline(y=df_az['Price'].median(),
               c='r', label='State Median')
    ax.axhline(y=location['Price'].median(),
               c='c', label='Local Median')
    ax.legend(loc='lower left', prop={'size': 16})
fig.tight_layout(pad=2)
# fig.savefig('MeanPrices.png', bbox_inches='tight')
```



Most of the 15 selected Tucson zip codes seem, based on real estate prices, relatively poor for the state of Arizona. It is not an overwhelming majority, though.

Here are the 6 zip codes with a higher average monthly real estate price than Arizona:

- 85747
- 85716
- 85719
- 85750

- 85737
- 85748

7 ARIMA Modeling

7.1 Stationarity Check with Dickey-Fuller Test

For ARIMA models to function properly, the data they're trained on needs to exhibit *stationarity*. In other words, there cannot be any notable seasonality, trends, etc.

The Dickey-Fuller test, to put it simply, starts with the null hypothesis that a given dataset is *not* stationary, then tests against the null. This null can be rejected if the test-statistic has a large enough absolute value, or the associated p-value is smaller than the rejection threshold.

Here, my p-value threshold for null-rejection (and therefore assuming sufficient stationarity) is alpha=0.05.

In [30]:
```
# Creating a function that performs a Dickey-Fuller stationarity
# test on a given DataFrame

def stationarity_check(TS):
    """Returns test statistic, p-value,
    number of lags & number of observations
    associated with a Dickey-Fuller
    stationarity test of DataFrame passed in."""

    dftest = adfuller(TS['Price'])

    dfoutput = pd.Series(dftest[0:4],
                         index=['Test Statistic', 'p-value',
                                '#Lags Used',
                                'Number of Observations Used'])
    return dfoutput
```

In [31]:
```
# Creating a list of each zip code's DataFrame subset to iterate
# through for stationarity check & modeling

zip_dfs = []

for z in zipcodes:
    zip_data = df_zips_15[df_zips_15['RegionName'] == z]
    copy_df = zip_data['2012-01-01':][['Price']].copy()
    zip_dfs.append(pd.DataFrame(copy_df))
```

In [32]:
```
# Iterating through above list and performing Dickey-Fuller test
# on each zip code candidate. Storing test statistic p-value in p_val

p_val = []

for x in zip_dfs:
    p_val.append(stationarity_check(x)[1])
```

In [33]:
```
# Printing test statistic p-value for each zip code's test

zips_p = pd.DataFrame()
zips_p['p_val'] = p_val
zips_p['zip'] = zipcodes
print('Dickey-Fuller p-values for each Zip Code: \n')
print(zips_p)
```

```
Dickey-Fuller p-values for each Zip Code: 

       p_val    zip
0   0.992353  85706
1   0.991580  85710
2   0.997098  85713
3   0.606485  85746
4   0.995512  85711
5   0.980688  85730
6   0.733693  85756
7   0.929463  85747
8   0.991424  85712
9   0.979057  85716
10  0.950019  85719
11  0.941872  85750
12  0.913318  85737
13  0.869780  85757
14  0.986253  85748
```

7.1.1 Improving Stationarity: Subtract Rolling Mean

In [34]:
```
# Updated list of DataFrames that have their rolling mean subtracted

rolling_dfs = []

for x in zip_dfs:
    roll_mean = x.rolling(window=4).mean()
    x_minus_roll = x - roll_mean
    x_minus_roll.dropna(inplace=True)
    rolling_dfs.append(x_minus_roll)
```

In [35]:
```
p_val = []

for x in rolling_dfs:
    p_val.append(stationarity_check(x)[1])
```

In [36]:
```
zips_p = pd.DataFrame()
zips_p['p_val'] = p_val
zips_p['zip'] = zipcodes
print('Dickey-Fuller p-values (minus rolling mean): \n')
print(zips_p)
```

```
Dickey-Fuller p-values (minus rolling mean): 

       p_val    zip
0   0.000057  85706
1   0.374204  85710
2   0.002823  85713
3   0.096910  85746
4   0.374095  85711
5   0.091329  85730
6   0.630944  85756
7   0.070954  85747
8   0.016667  85712
9   0.320964  85716
10  0.046144  85719
11  0.021829  85750
12  0.125711  85737
13  0.308179  85757
14  0.022449  85748
```

7.1.2 Improving Stationarirty: Differencing

In [37]:
```
# Updated list of DataFrames that are also differenced

diff_zip_dfs = []

for x in rolling_dfs:
    x_diff = x.diff(periods=12)
    x_diff.dropna(inplace=True)
    diff_zip_dfs.append(x_diff)
```

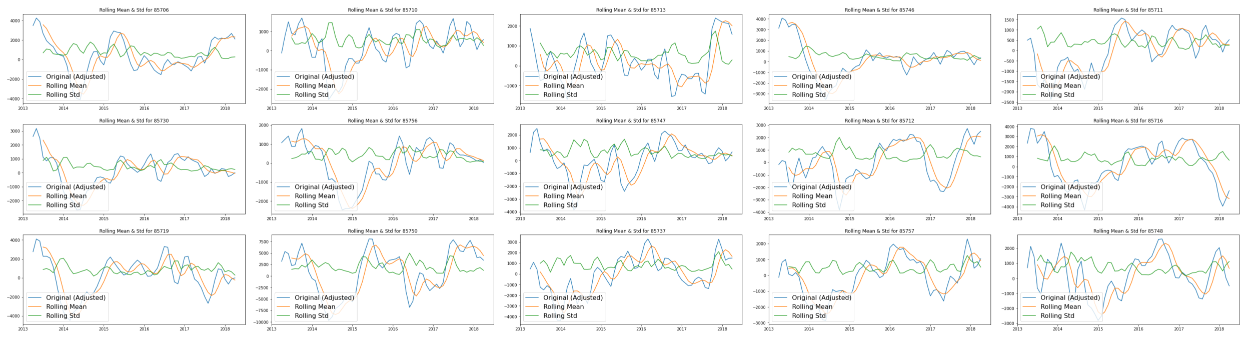
In [38]:
```
p_val = []

for x in diff_zip_dfs:
    p_val.append(stationarity_check(x)[1])
```

```
In [39]: zips_p = pd.DataFrame()
         zips_p['p_val'] = p_val
         zips_p['zip'] = zipcodes
         print('Dickey-Fuller p-values (minus rolling mean & differenced): \n')
         print(zips_p)
```

```
Dickey-Fuller p-values (minus rolling mean & differenced):

       p_val    zip
0   0.059258  85706
1   0.251714  85710
2   0.108648  85713
3   0.011938  85746
4   0.612666  85711
5   0.018238  85730
6   0.026788  85756
7   0.670991  85747
8   0.032316  85712
9   0.122079  85716
10  0.030019  85719
11  0.000043  85750
12  0.020170  85737
13  0.236581  85757
14  0.356526  85748
```

7.1.3 Visualizing Transformed Time Series with Rolling Mean, Standard Deviation

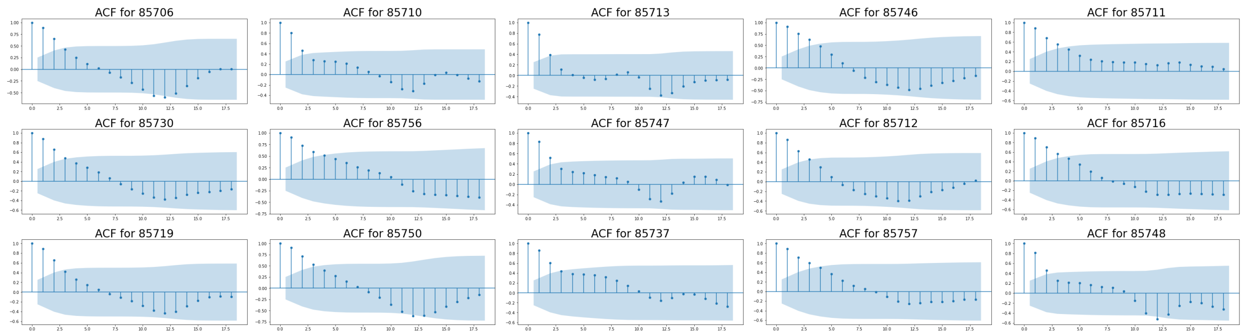
In [40]:
```
# Visualizing stationarity of transformed data for each zip code

fig, ax_lst = plt.subplots(nrows=3, ncols=5, figsize=(44, 12))

ax_lst = ax_lst.flatten()

for ax, dfz, zipcode in zip(ax_lst, diff_zip_dfs, zipcodes):
    roll_mean = dfz.rolling(window=4).mean()
    roll_std = dfz.rolling(window=4).std()

    ax.plot(dfz, label = 'Original (Adjusted)')
    ax.plot(roll_mean, label='Rolling Mean')
    ax.plot(roll_std, label='Rolling Std')
    ax.set_title(f'Rolling Mean & Std for {zipcode}')
    ax.legend(loc='lower left', prop={'size': 16})

fig.tight_layout(pad=2)
```



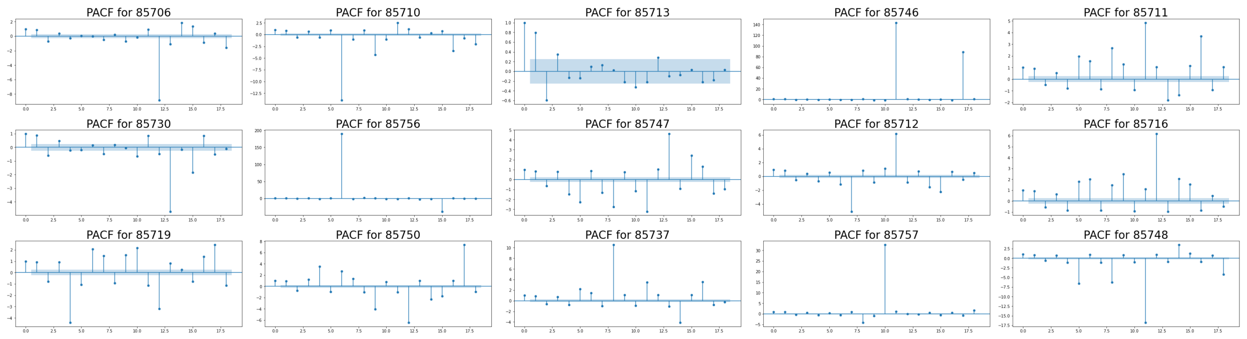
Thought the time series for the transformed data is quite volatile for each zip code, there is no evident seasonality or long term trend to the motion. Additionally, the rolling standard deviation appears to be relatively stable in each plot. After a quick look at the autocorrelation & partial autocorrelation for each zip code, I'm ready to move on to final modeling.

7.2 Autocorrelation (ACF) & Partial Autocorrelation (PACF)

In [41]:
```
# Autocorrelation of transformed data for each zip code

fig, ax_lst = plt.subplots(nrows=3, ncols=5, figsize=(44, 12))

ax_lst = ax_lst.flatten()

for ax, dfz, z in zip(ax_lst, diff_zip_dfs, zipcodes):
    plot_acf(dfz, ax=ax)
    ax.set_title(f'ACF for {z}', fontsize=28)

fig.tight_layout(pad=2)
```



In [42]:
```
# Partial autocorrelation of transformed data for each zip code

fig, ax_lst = plt.subplots(nrows=3, ncols=5, figsize=(44, 12))

ax_lst = ax_lst.flatten()

for ax, dfz, z in zip(ax_lst, diff_zip_dfs, zipcodes):
    plot_pacf(dfz, ax=ax)
    ax.set_title(f'PACF for {z}', fontsize=28)

fig.tight_layout(pad=2)
```



7.3 Grid Search for Optimal ARIMA Parameters

Here, I iterate through different combinations of p, d & q paramters to find the best ARIMA settings for each zip code subset. Optimization here is based on lowest AIC score.

ARIMA models do not require the data fed into them to exhibit stationarity, and making understandable interpretations of ROI on transformed price figures will be difficult, if not impossible. So, from this point on, I go back to the non-transformed data for each of the 15 zip codes.

In [43]:
```
# List of all possible combinations of 0,1 & 2 for p, d & q
# ARIMA parameter values

p = d = q = range(0,3)
pdq = list(itertools.product(p,d,q))
```

In [44]:
```
# Making multi-dimensional list containing optimized parameters,
# associated (lowest) AIC score for each zip code

warnings.filterwarnings('ignore')

ans = []

for dfz, zipcode in zip(zip_dfs, zipcodes):

    best_aic = np.inf
    best_order = None

    for order in pdq:
        model = ARIMA(dfz, order=order,
                      enforce_invertibility=False)
        output = model.fit()
        if output.aic < best_aic:
            best_aic = output.aic
            best_order = order

    ans.append([zipcode, best_order, best_aic])
```

In [45]:
```
# Turning list into DataFrame with column names

best_params = pd.DataFrame(ans, columns = ['Zip','pdq','AIC'])
best_params
```

Out[45]:

	Zip	pdq	AIC
0	85706	(2, 2, 2)	10.000000
1	85710	(0, 2, 0)	1118.145015
2	85713	(1, 2, 2)	1103.184995
3	85746	(2, 2, 2)	1104.745208
4	85711	(2, 2, 2)	1079.469681
5	85730	(0, 2, 0)	1077.033355
6	85756	(0, 2, 0)	1082.601103
7	85747	(0, 2, 0)	1126.759381
8	85712	(2, 2, 2)	1147.679096
9	85716	(0, 2, 0)	1166.420086
10	85719	(2, 2, 2)	1145.205301
11	85750	(1, 0, 2)	1271.854162
12	85737	(0, 2, 0)	1168.664550
13	85757	(0, 2, 0)	1109.266611
14	85748	(2, 2, 2)	1117.796390

7.4 Forecasting Existing Dates to Test Model Accuracy

Having obtained optimized parameters for each zip code's ARIMA model, I'm now ready to test their individual predictive abilities. I'm testing the models against existing data from June 2017 onward, and the metric used for judgement here is Root Mean Square Error.

In [46]:
```
# Using iteration to test the predictive power of each optimized
# ARIMA model on existing data from June 2017 onward. Results,
# including RMSE, are added to a new DataFrame

summary_table = pd.DataFrame()

Zipcode = []
RMSE = []
models = []

for zipcode, pdq, dfz in zip(best_params['Zip'], best_params['pdq'],
                             zip_dfs):

    model = ARIMA(dfz, order=pdq, enforce_invertibility=False)
    output = model.fit()
    models.append(output)

    pred = output.get_prediction(start=pd.to_datetime('2017-06-01'),
                                 dynamic=True)
    y_hat = pred.predicted_mean
    y = dfz['2017-06-01':]['Price']

    sqrt_mse = np.sqrt(((y_hat - y)**2).mean())

    Zipcode.append(zipcode)
    RMSE.append(sqrt_mse)

summary_table['Zipcode'] = Zipcode
summary_table['RMSE'] = RMSE
```

In [47]:
```
# Showing DataFrame of RMSE for each zip code's optimized model

summary_table
```

Out[47]:

	Zipcode	RMSE
0	85706	128316.926539
1	85710	2651.757565
2	85713	4251.732775
3	85746	1799.389877
4	85711	4249.332897
5	85730	1222.144166
6	85756	1469.693846
7	85747	2750.371876
8	85712	8265.550979
9	85716	9974.786396
10	85719	2834.696373
11	85750	23372.555997
12	85737	3444.758865
13	85757	5174.236878
14	85748	849.469912

In [48]:
```
# Gauging the average performance across all zip codes

summary_table['RMSE'].mean()
```

Out[48]:
```
13375.16032933652
```

7.5 Making Future Predictions: Predicted Mean

In [49]:
```
# Making 1month, 6month, 12month & 3year forecasts of average
# real estate price for each zip code. Also calculating ROI
# projection for each prediction point

forecast_table = pd.DataFrame()
current = []
forecasts_1mo = []
interval_1mo = []
forecasts_6mo = []
forecasts_1yr = []
forecasts_3yr = []

for zipcode, output, dfz in zip(Zipcode, models, zip_dfs):

    pred_1mo = output.get_forecast(steps=1)
    pred_conf_1mo = pred_1mo.conf_int()
    forecast_1mo = pred_1mo.predicted_mean.to_numpy()[-1]
    forecasts_1mo.append(forecast_1mo)

    pred_6mo = output.get_forecast(steps=6)
    pred_conf_6mo = pred_6mo.conf_int()
    forecast_6mo = pred_6mo.predicted_mean.to_numpy()[-1]
    forecasts_6mo.append(forecast_6mo)

    pred_1yr = output.get_forecast(steps=12)
    pred_conf_1yr = pred_1yr.conf_int()
    forecast_1yr = pred_1yr.predicted_mean.to_numpy()[-1]
    forecasts_1yr.append(forecast_1yr)

    pred_3yr = output.get_forecast(steps=36)
    pred_conf_3yr = pred_3yr.conf_int()
    forecast_3yr = pred_3yr.predicted_mean.to_numpy()[-1]
    forecasts_3yr.append(forecast_3yr)

    current.append(dfz['2018-04']['Price'][0])

forecast_table['Zipcode'] = Zipcode
forecast_table['Current Value'] = current
forecast_table['1 Month Value'] = forecasts_1mo
forecast_table['6 Months Value'] = forecasts_6mo
forecast_table['1 Year Value'] = forecasts_1yr
forecast_table['3 Years Value'] = forecasts_3yr

forecast_table['1mo-ROI']=(forecast_table['1 Month Value'] - forecast_table
forecast_table['6mo-ROI']=(forecast_table['6 Months Value'] - forecast_tabl
forecast_table['1yr-ROI']=(forecast_table['1 Year Value'] - forecast_table[
forecast_table['3yr-ROI']=(forecast_table['3 Years Value'] - forecast_table
```

In [50]:
```
# Displaying final results

forecast_table
```

Out[50]:

	Zipcode	Current Value	1 Month Value	6 Months Value	1 Year Value	3 Years Value	1mo-ROI	
0	85706	137200.0	0.000000	0.000000	0.000000	0.000000	-1.000000	-1
1	85710	180600.0	181500.000000	186000.000000	191400.000000	213000.000000	0.004983	0
2	85713	127300.0	128657.177690	135986.399671	144935.610505	180749.574411	0.010661	0
3	85746	156700.0	157184.616177	159815.410805	162973.858628	175706.548399	0.003093	0
4	85711	161000.0	162300.208428	169266.119611	177421.348035	210248.659629	0.008076	0
5	85730	164100.0	165200.000000	170700.000000	177300.000000	203700.000000	0.006703	0
6	85756	162700.0	163700.000000	168700.000000	174700.000000	198700.000000	0.006146	0
7	85747	209400.0	210700.000000	217200.000000	225000.000000	256200.000000	0.006208	0
8	85712	175700.0	177266.648261	185430.867472	195108.135525	233976.941596	0.008917	0
9	85716	223400.0	224400.000000	229400.000000	235400.000000	259400.000000	0.004476	0
10	85719	224600.0	225322.461483	228804.690994	232997.016924	249761.892840	0.003217	0
11	85750	430700.0	430910.848011	429347.487547	428227.047191	423997.649141	0.000490	-0
12	85737	311000.0	312800.000000	321800.000000	332600.000000	375800.000000	0.005788	0
13	85757	179500.0	180800.000000	187300.000000	195100.000000	226300.000000	0.007242	0
14	85748	236500.0	237389.759716	241917.038043	247400.331438	268992.442966	0.003762	0

In [51]:
```
# Condensed table that shows 5 zip codes with the best ROI
# projections at 1month, 6months & 12months

to_drop = ['Current Value', '1 Month Value', '6 Months Value',
           '1 Year Value', '3 Years Value', '3yr-ROI']

table_final = forecast_table.drop(columns=to_drop, axis=1)
table_final = table_final[table_final['1yr-ROI'] > 0.08]
table_final.sort_values(by=['1yr-ROI'], ascending=False, inplace=True)
table_final.reset_index(inplace=True)
table_final.drop(columns=['index'], inplace=True)
table_final
```

Out[51]:

	Zipcode	1mo-ROI	6mo-ROI	1yr-ROI
0	85713	0.010661	0.068236	0.138536
1	85712	0.008917	0.055383	0.110462
2	85711	0.008076	0.051342	0.101996
3	85757	0.007242	0.043454	0.086908
4	85730	0.006703	0.040219	0.080439

8 Results

As seen in the final table above, the top five Tucson zip codes to invest in (based on 1, 6 or 12-month ROI forecasts) are, in descending order of profitability:

- 85713 (13.85% 12mo)
- 85712 (11.04% 12mo)
- 85711 (10.2% 12mo)
- 85757 (8.69% 12mo)
- 85730 (8.04% 12mo)

9 Recommendations

Based on these results, my recommendation for Simon Property Group is to **begin developing commercially in 85713.**

ROI Projections for 85713:

```
* 1-month: +1.06% (next best zip +0.89%)
* 6-month: +6.82% (next best zip +5.53%)
* 12-month: +13.85% (next best zip +11.04%)
```

10 Future Work

Given more time with the job, I would:

- Acquire commercial data to corroborate the zip code choices made.
- Explore more of the 55 zip codes within the Tucson metro.
- Inspect other metros within the state of Arizona, such as Prescott Valley & Lake Havasu City.