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 allows me 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

Simon Property Group, the US's top retail-oriented real estate trust, has set their sights on expansion in the state of Arizona. 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 properties (https://en.wikipedia.org/wiki/List_of_Simon_Property_Group_properties#Arizona) in the state. 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 trust, Simon Property would like to begin development of at least one new property in Arizona.

Of the 3 existing properties, 2 are already located in the Phoenix metropolitan area, leading the trust to prefer focusing their next development in Arizona's 2nd biggest metro area: Tucson. Though Simon Property already owns the Tucson Premium Outlets (https://www.premiumoutlets.com/outlet/tucson), they believe that the area is still ripe for a new retail space, especially as the growth of the state continues to soar.

As part of a data science team brought in, my assignment is to analyze data provided by [Zillow (https://www.zillow.com/research/data/)](https://www.zillow.com/research/data/) that tracks real estate prices across the US. In this project, I use time series analysis of the Zillow data to forecast which 5 zipcodes out of a group of the most populous in Tucson will exhibit the most economic growth (and therefore potential for retail spending). For a given zipcode, economic growth is indicated by higher real estate prices.

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]:
```
df = pd.read_csv('Data/zillow_data.csv')

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]:
```
def get_datetimes(df):
    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]:
```
def melt_data(df):
    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]:
```
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]:
```
df_az = df[df['State'] == 'AZ']
```

In [12]:
```
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[12]:

	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 [13]:
```
zips_tucson = zip_info[zip_info['state'] == 'AZ'][zip_info['primary_city']
```

In [14]:
```
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 population estimates for each zip code in the metro.

In [15]:
```
zips_tucson.head()
```

Out[15]:

	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 [16]:
```
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 [17]:
```
zips_tucson.head()
```

Out[17]:

	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 [18]:
```
zips_tucson = zips_tucson.groupby('zip').aggregate({'irs_estimated_populati
zips_tucson.rename(columns={'irs_estimated_population_2015': 'population_es
                   inplace=True)
zips_tucson.head()
```

Out[18]:

	population_est
zip	
85701	3900
85702	1060
85703	805
85704	27140
85705	41250

In [19]:
```
zips_tucson.sort_values(by=['population_est'],
                        ascending=False, inplace=True)
zips_tucson.head(20)
```

Out[19]:

zip	population_est
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)](https://www.tucsonaz.gov/files/pdsd/wardzip.pdf), the areas of 85743, 85741, 85704, 85745, 85705, & 85718 will be dropped as well.

In [20]:
```
to_drop = [85742, 85743, 85741, 85704, 85745, 85705, 85718]

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

Out[20]:

	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 [21]:
```
zips_tucson.reset_index(inplace=True)
zipcodes = list(zips_tucson['zip'])[:15]
```

In [22]:
```
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 [23]:
```
df_zips_15 = df_az[df_az['RegionName'].isin(zipcodes)]
```

6 EDA & Visualization

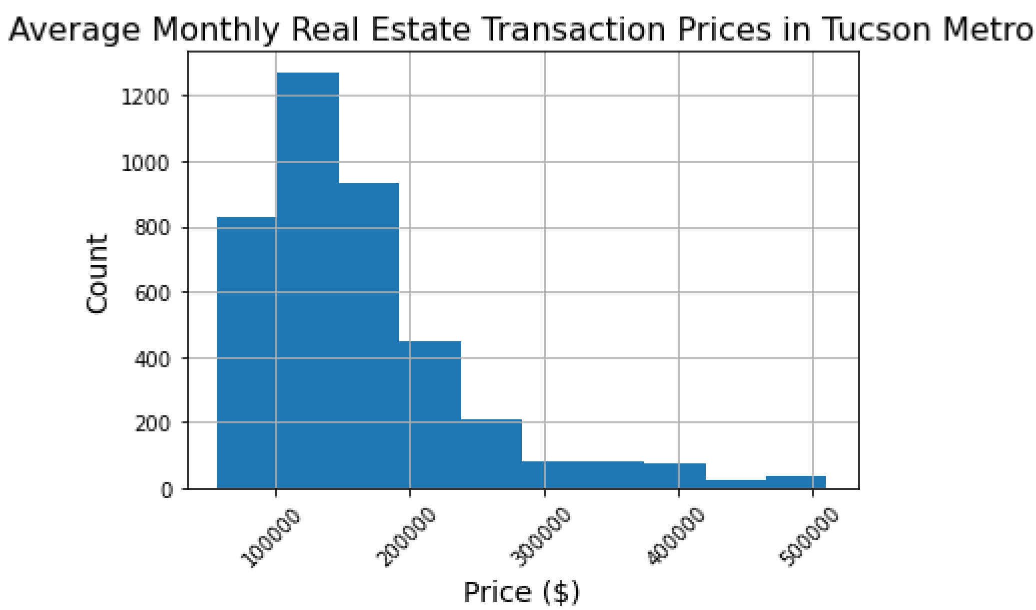
In [24]:
```
# font = {'family' : 'normal',
#         'weight' : 'bold',
#         'size'   : 22}

# matplotlib.rc('font', **font)

# plt.gcf().autofmt_xdate()

# NOTE: if you visualizations are too cluttered to read, try calling 'plt.g
```

6.1 Statistical Description of State & Metro Monthly Prices

In [25]:
```
df_zips_15['Price'].describe()
```

Out[25]:
```
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
```

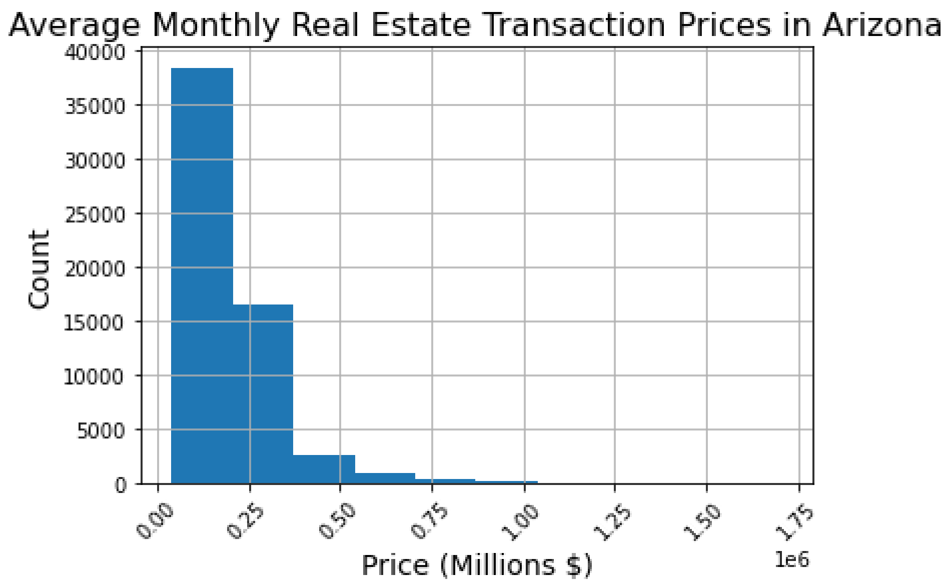
In [26]:
```
df_zips_15['Price'].hist()
plt.xticks(rotation=45)
plt.xlabel('Price ($)', fontsize=14)
plt.ylabel('Count', fontsize=14)
plt.title('Average Monthly Real Estate Transaction Prices in Tucson Metro',
plt.show()
```



In [27]:
```
df_az['Price'].describe()
```

Out[27]:
```
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 [28]:
```
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', font
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.

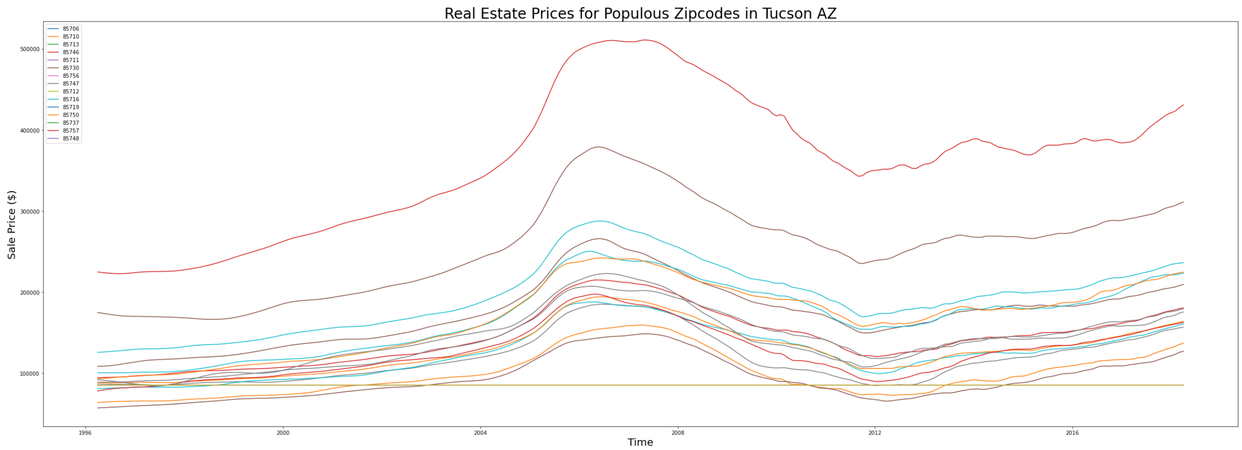
- Mean Price per Real Estate Sale: ~201,000 USD (Arizona) vs. ~163,000 USD (Tucson)
- Standard Deviation of Price per Real State Sale: ~133,0000 USD (Arizona) vs. ~78,000 USD (Tucson)

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 [29]:
```
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('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()
```



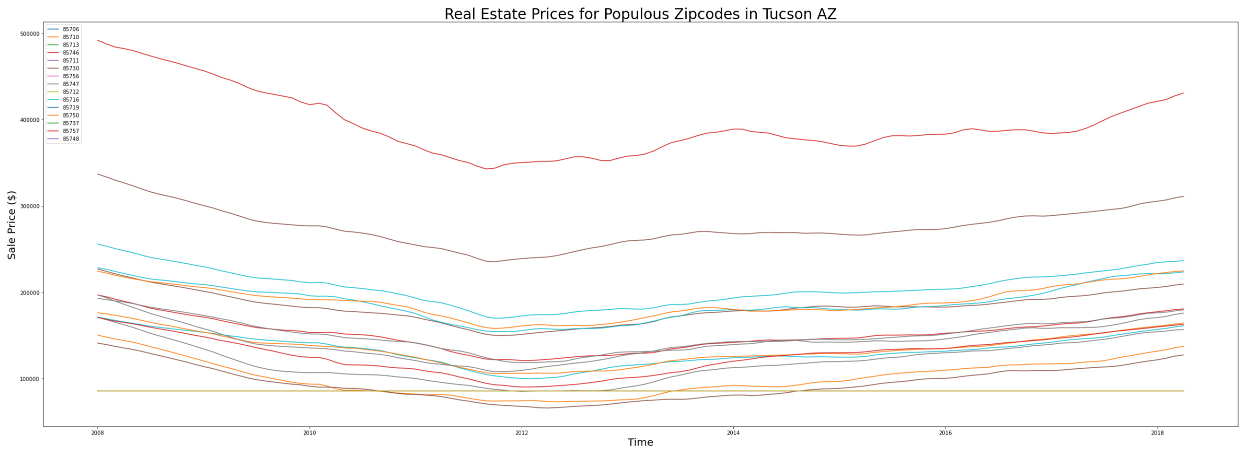
With a market as volatile as real estate, it might not make much sense to take this entire 20+ year span into consideration -- though this plot is useful in showing an overall upward trend from left end to right end.

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.

2008 up to 2018:

In [30]:
```
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('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()
```

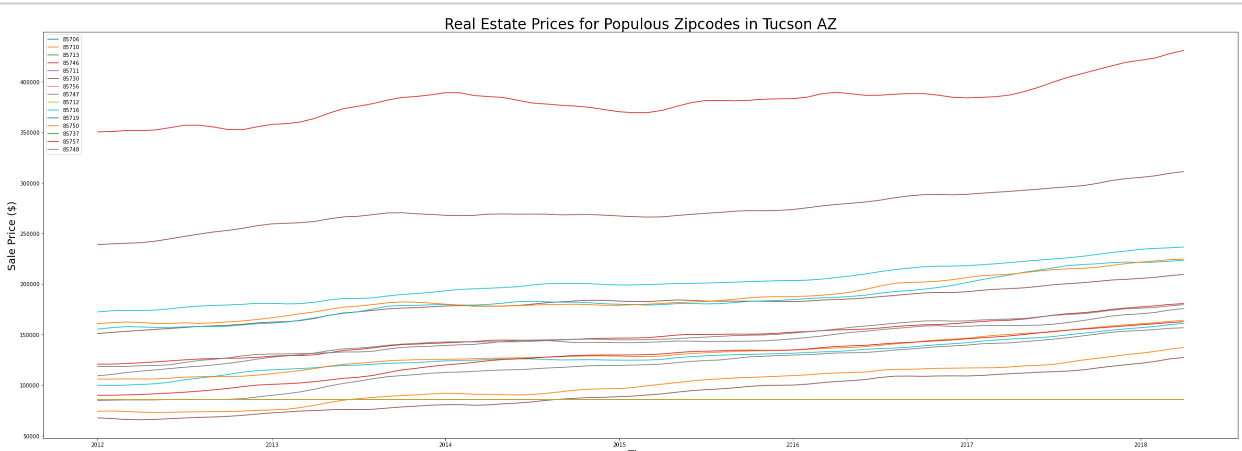


This is much less volatile than the previous plot, but it looks like proper recovery from 2008 hasn't really begun to take place until 2012.

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

2012 up to 2018:

In [31]:
```
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('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()
```



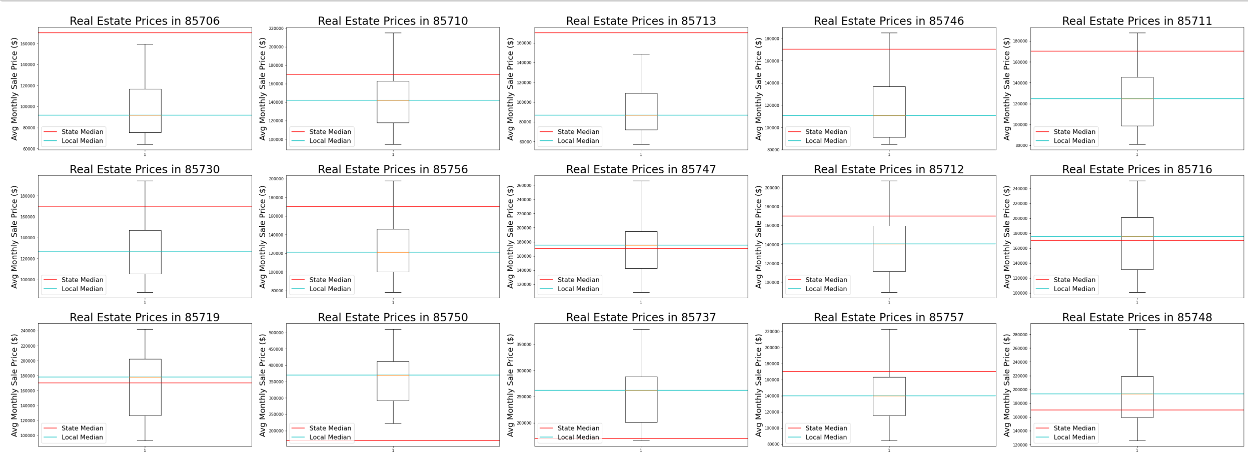
In this more recent plot, there is a noticeable upward trend across the years 2012 - 2017, and it exhibits much more stability than the previous plot. This section suggests to me that I should train my eventual model on data from the year 2012 onward, so that it doesn't anticipate another 2008-sized 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' mean monthly real estate price & compare it to that of the state as a whole.

As seen at the beginning of the EDA section, the metro of Tucson is, on average, poorer than the state of Arizona. But 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 [32]:
```
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)
```



As expected, most of the 15 selected Tucson zip codes are, based on real estate prices, poor for the state of Arizona. It is not an overwhelming majority, though, which I find somewhat surprising.

Here are the 6 zip codes with a higher avg monthly real estate price than AZ:

- 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 [33]:
```
def stationarity_check(TS):

    dftest = adfuller(TS['Price'])

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

In [34]:
```
zip_dfs = []

for z in zipcodes:
    zip_dfs.append(pd.DataFrame(df_zips_15[df_zips_15['RegionName'] == z]['
```

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

for x in zip_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 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: Log Transform

In [37]:
```
log_dfs = []

for x in zip_dfs:
    log_data = np.log(x)
    log_data.dropna(inplace=True)
    log_dfs.append(log_data)
```

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

for x in log_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 (log-transformed): \n')
print(zips_p)
```

```
Dickey-Fuller p-values (log-transformed):

       p_val    zip
0   0.778212  85706
1   0.970530  85710
2   0.936641  85713
3   0.065209  85746
4   0.985157  85711
5   0.899595  85730
6   0.377990  85756
7   0.860545  85747
8   0.975437  85712
9   0.951964  85716
10  0.889777  85719
11  0.925250  85750
12  0.831889  85737
13  0.735967  85757
14  0.947114  85748
```

7.1.2 Improving Stationarity: Subtract Rolling Mean

In [40]:
```
rolling_dfs = []

for x in log_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 [41]:
```
p_val = []

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

In [42]:
```
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   0.000005  85706
1   0.298576  85710
2   0.000205  85713
3   0.242745  85746
4   0.076012  85711
5   0.053239  85730
6   0.645490  85756
7   0.064862  85747
8   0.011000  85712
9   0.244042  85716
10  0.030092  85719
11  0.019815  85750
12  0.100635  85737
13  0.342645  85757
14  0.013423  85748
```

7.1.3 Improving Stationarirty: Differencing

In [43]:
```
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 [44]:
```
p_val = []

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

In [45]:
```
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   0.022600  85706
1   0.202300  85710
2   0.163884  85713
3   0.019672  85746
4   0.575814  85711
5   0.012567  85730
6   0.029161  85756
7   0.670405  85747
8   0.487454  85712
9   0.123844  85716
10  0.019889  85719
11  0.000038  85750
12  0.013567  85737
13  0.277893  85757
14  0.309153  85748
```

7.1.4 Removing Non-Stationary Zip Codes

After multiple rounds of transformation on the data, most zip codes have Dickey-Fuller p-values that fall below the alpha=0.05 threshold. There are a few, though, that do not. In response to this, I choose to drop any zip codes that still do not meet the stationarity requirements, leaving me with a pool of 7 remaining zip codes to choose from.

In [46]:
```
for p in p_val:
    if p > 0.05:
        zips_p.drop(zips_p.loc[zips_p['p_val']==p].index, inplace=True)
    else:
        continue

zipcodes = list(zips_p['zip'])
```

In [47]:
```
print(zipcodes)
```

```
[85706, 85746, 85730, 85756, 85719, 85750, 85737]
```

In [48]:
```
zip_dfs = []

for z in zipcodes:
    zip_dfs.append(pd.DataFrame(df_zips_15[df_zips_15['RegionName'] == z]['
```

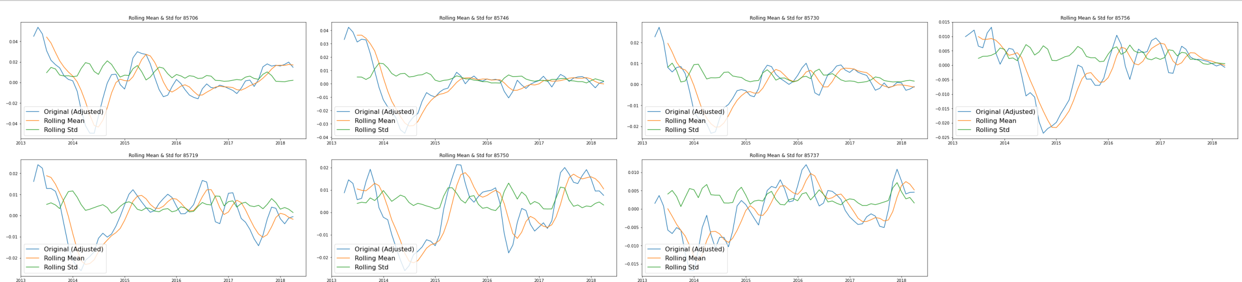
In [49]:
```
log_dfs = []

for x in zip_dfs:
    log_data = np.log(x)
    log_data.dropna(inplace=True)
    log_dfs.append(log_data)
```

In [50]:
```
rolling_dfs = []

for x in log_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 [51]:
```
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)
```

7.1.5 Visualizing Adjusted Time Series

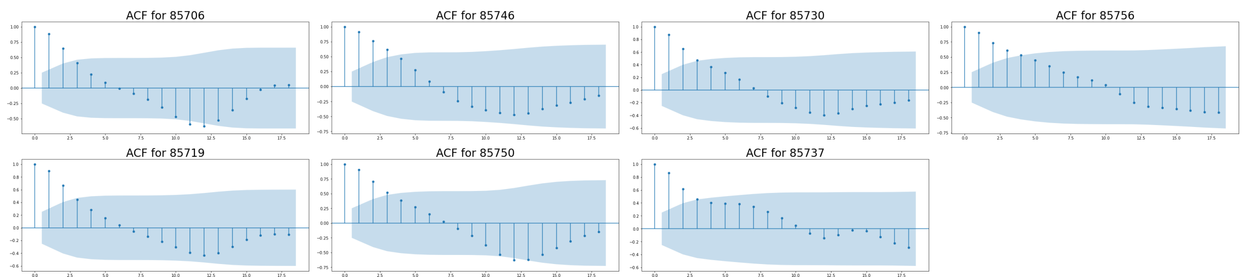
In [52]:
```
fig, ax_lst = plt.subplots(nrows=2, ncols=4, figsize=(44, 10))
fig.delaxes(ax_lst[1,3])
ax_lst = ax_lst.flatten()

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)
```



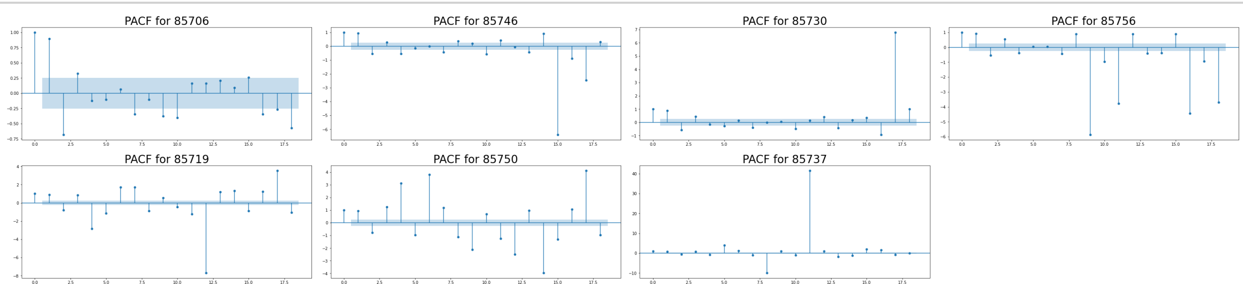
Thought the time series for the transformed data is quite volatile for each zip code, there is no evident seasonality or long term trend to the motion. Additionally, the rolling standard deviation appears to be relatively stable in each plot. After a quick look at the autocorrelation & partial autocorrelation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

7.2 Autocorrelation (ACF) & Partial Autocorrelation (PACF)

In [53]:
```
fig, ax_lst = plt.subplots(nrows=2, ncols=4, figsize=(44, 10))
fig.delaxes(ax_lst[1,3])
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 [54]:
```
fig, ax_lst = plt.subplots(nrows=2, ncols=4, figsize=(44, 10))
fig.delaxes(ax_lst[1,3])
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.

In [55]:
```
p = d = q = range(0,3)
pdq = list(itertools.product(p,d,q))
```

In [56]:
```
warnings.filterwarnings('ignore')

ans = []

for dfz, zipcode in zip(diff_zip_dfs, zipcodes):
    for param in pdq:
        model = ARIMA(dfz, order=param, enforce_invertibility=False)
        output = model.fit()
        ans.append([zipcode, param, output.aic])
```

In [57]:
```
result = pd.DataFrame(ans, columns = ['Zip','pdq','AIC'])
best_params = result.loc[result.groupby('Zip')['AIC'].idxmin()]
```

In [58]:
```
best_params.sort_index(inplace=True)
best_params
```

Out[58]:

	Zip	pdq	AIC
11	85706	(1, 0, 2)	-468.847994
38	85746	(1, 0, 2)	-496.196456
65	85730	(1, 0, 2)	-539.927734
92	85756	(1, 0, 2)	-535.062327
119	85719	(1, 0, 2)	-524.434270
146	85750	(1, 0, 2)	-539.290547
173	85737	(1, 0, 2)	-563.509520

7.4 Forecasting Existing Dates to Test Model Accuracy

In [59]:
```
Using the optimi
```

```
  File "<ipython-input-59-235d3d816ccc>", line 1
    Using the optimi
            ^
SyntaxError: invalid syntax
```

In [ ]:
```
summary_table = pd.DataFrame()

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

for zipcode, pdq, dfz in zip(best_params['Zip'], best_params['pdq'], diff_z

    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'), dynami
    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 [ ]:
```
summary_table
```

7.5 Making Future Predictions: Upper Confidence Interval

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

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

    pred_1mo = output.get_forecast(steps=1)
    pred_conf_1mo = pred_1mo.conf_int()
    forecast_1mo = pred_conf_1mo['upper Price'].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_conf_6mo['upper Price'].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_conf_1yr['upper Price'].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_conf_3yr['upper Price'].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 [ ]:
```
forecast_table
```

7.6 Making Future Predictions: Predicted Mean

In [ ]:
```
t_table = pd.DataFrame()
 = []
ts_1mo = []
l_1mo = []
ts_6mo = []
ts_1yr = []
ts_3yr = []

code, output, dfz in zip(Zipcode, models, diff_zip_dfs):

d_1mo = output.get_forecast(steps=1)
d_conf_1mo = pred_1mo.conf_int()
ecast_1mo = pred_1mo.predicted_mean.to_numpy()[-1]
ecasts_1mo.append(forecast_1mo)

d_6mo = output.get_forecast(steps=6)
d_conf_6mo = pred_6mo.conf_int()
ecast_6mo = pred_6mo.predicted_mean.to_numpy()[-1]
ecasts_6mo.append(forecast_6mo)

d_1yr = output.get_forecast(steps=12)
d_conf_1yr = pred_1yr.conf_int()
ecast_1yr = pred_1yr.predicted_mean.to_numpy()[-1]
ecasts_1yr.append(forecast_1yr)

d_3yr = output.get_forecast(steps=36)
d_conf_3yr = pred_3yr.conf_int()
ecast_3yr = pred_3yr.predicted_mean.to_numpy()[-1]
ecasts_3yr.append(forecast_3yr)

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

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

t_table['1mo-ROI']=(forecast_table['1 Month Value'] - forecast_table['Curre
t_table['6mo-ROI']=(forecast_table['6 Months Value'] - forecast_table['Curr
t_table['1yr-ROI']=(forecast_table['1 Year Value'] - forecast_table['Curren
t_table['3yr-ROI']=(forecast_table['3 Years Value'] - forecast_table['Curre
```

In [ ]:
```
forecast_table
```

8 Conclusions

To keep conclusions & recommendations as sure as possible, I'll focus only on the shorter term 1-month and 6-month ROI values for each zip code. Longer term predictions, while interesting to speculate about, are not nearly as dependable.

Only 2 zip codes exhibited a positive 1-month ROI (though both were very significantly positive): 85719 & 85756.

Only 1 zip code exhibited a positive 6-month ROI: 85756.

Based on 6-month ROI, the zip codes that perform the next best are: 85706, 85750 & 84746.

A final point to consider is which of these zip codes falls in the list of zip codes that seem wealthier, on average, compared to the state of Arizona.

Of the 5 zip codes selected here, 2 were in the original list of relatively wealthy zip codes: 85719 & 85750.

9 Recommendations

Though the original aim of this project was to provide a list of 5 zip codes to choose from, my recommendation is that Simon Property develop in the specific zip code of 85756, as it's the only populous area of Tucson to exhibit positive 1-month (278.3%) & 6-month (84.7%) ROI values based on the my model's forecast.

If the firm needs other candidates to look at within the Tucson, metro, these 4 were the next best performers (though none exhibited positive 6-month ROI's): 85719, 85706, 85750, 84746.

10 Future Work

Given more time, I would expand the scope of the model to include more areas within the state of Arizona. Primarily, I would observe:

- More of the 55 possible zip codes in the Tucson metro
- Populous zip codes within other state metros, like Prescott Valley & Lake Havasu City