Exploratory Data Analysis

Business Case

As Data Scientists working for a home renovation company, we were given the task to investigate and predict the average housing prices in the next two years using various zip codes in Springfield, MO. Using data from 1996 to 2018, we will use various time series models to determine which zip code would be the best investment to buy houses in to renovate.

Loading and Filtering

Analyzing 15,000 different zip codes isn't very feasible, so after looking at the data we have decided to focus on the six zip codes in one city: Springfield, Missouri. Since we are only interested in analyzing the trends over time, we will remove the unecessary columns from our dataset.

In [1]:
```
import  pandas as pd
import numpy as np
from IPython.display import display
pd.set_option('display.max_columns', None)
```

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

Out[2]:

	RegionID	RegionName	City	State	Metro	CountyName	SizeRank	1996-04	
0	84654	60657	Chicago	IL	Chicago	Cook	1	334200.0	3
1	90668	75070	McKinney	TX	Dallas-Fort Worth	Collin	2	235700.0	2
2	91982	77494	Katy	TX	Houston	Harris	3	210400.0	2
3	84616	60614	Chicago	IL	Chicago	Cook	4	498100.0	5
4	93144	79936	El Paso	TX	El Paso	El Paso	5	77300.0	
...	...	...	...	...	...	...	...	...	
14718	58333	1338	Ashfield	MA	Greenfield Town	Franklin	14719	94600.0	
14719	59107	3293	Woodstock	NH	Claremont	Grafton	14720	92700.0	
14720	75672	40404	Berea	KY	Richmond	Madison	14721	57100.0	

In [3]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14723 entries, 0 to 14722
Columns: 272 entries, RegionID to 2018-04
dtypes: float64(219), int64(49), object(4)
memory usage: 30.6+ MB
```

In [4]:
```
data.describe()
```

Out[4]:

	RegionID	RegionName	SizeRank	1996-04	1996-05	1996-06	
count	14723.000000	14723.000000	14723.000000	1.368400e+04	1.368400e+04	1.368400e+04	1.36
mean	81075.010052	48222.348706	7362.000000	1.182991e+05	1.184190e+05	1.185374e+05	1.18
std	31934.118525	29359.325439	4250.308342	8.600251e+04	8.615567e+04	8.630923e+04	8.64
min	58196.000000	1001.000000	1.000000	1.130000e+04	1.150000e+04	1.160000e+04	1.18
25%	67174.500000	22101.500000	3681.500000	6.880000e+04	6.890000e+04	6.910000e+04	6.92
50%	78007.000000	46106.000000	7362.000000	9.950000e+04	9.950000e+04	9.970000e+04	9.97
75%	90920.500000	75205.500000	11042.500000	1.432000e+05	1.433000e+05	1.432250e+05	1.43
max	753844.000000	99901.000000	14723.000000	3.676700e+06	3.704200e+06	3.729600e+06	3.75

In [5]:
```
data_springfield = data.loc[(data['City']=='Springfield') & (data['State']=='MO')
data_springfield = data_springfield.drop(['RegionID', 'City', 'State', 'Metro', '
data_springfield
```

Out[5]:

ZipCode	1996-04	1996-05	1996-06	1996-07	1996-08	1996-09	1996-10	1996-11	1996-12
65807	80800.0	80800.0	80900.0	81100.0	81400.0	81900.0	82400.0	83100.0	83700.0
65802	64800.0	64100.0	63500.0	63000.0	62600.0	62400.0	62400.0	62600.0	62900.0
65804	83200.0	83200.0	83300.0	83500.0	83700.0	84200.0	84800.0	85500.0	86400.0
65810	117900.0	116800.0	115900.0	115200.0	114800.0	114700.0	115000.0	115500.0	116200.0
65806	38800.0	38500.0	38200.0	38000.0	37600.0	37300.0	37000.0	36800.0	36700.0
65809	158200.0	158000.0	158000.0	158100.0	158300.0	158500.0	158400.0	158000.0	157800.0

In [6]:
```
data_springfield.isna().sum().sum()
```

Out[6]: 0

Preprocessing

In order to run models on this data, we need to convert the string dates into datetime objects. We also need to convert our dataframe from wide format to long format.

In [7]:
```
dates = pd.to_datetime(data_springfield.columns.values, format='%Y-%m')
type(dates), dates
```

Out[7]:
```
(pandas.core.indexes.datetimes.DatetimeIndex,
 DatetimeIndex(['1996-04-01', '1996-05-01', '1996-06-01', '1996-07-01',
                '1996-08-01', '1996-09-01', '1996-10-01', '1996-11-01',
                '1996-12-01', '1997-01-01',
                ...
                '2017-07-01', '2017-08-01', '2017-09-01', '2017-10-01',
                '2017-11-01', '2017-12-01', '2018-01-01', '2018-02-01',
                '2018-03-01', '2018-04-01'],
               dtype='datetime64[ns]', length=265, freq=None))
```

In [8]:
```
springfield = data_springfield.T.set_index(dates)
springfield
```

Out[8]:

ZipCode	65807	65802	65804	65810	65806	65809
1996-04-01	80800.0	64800.0	83200.0	117900.0	38800.0	158200.0
1996-05-01	80800.0	64100.0	83200.0	116800.0	38500.0	158000.0
1996-06-01	80900.0	63500.0	83300.0	115900.0	38200.0	158000.0
1996-07-01	81100.0	63000.0	83500.0	115200.0	38000.0	158100.0
1996-08-01	81400.0	62600.0	83700.0	114800.0	37600.0	158300.0
...	...	...	...	...	...	...
2017-12-01	119900.0	94800.0	141800.0	192000.0	61000.0	257400.0
2018-01-01	120500.0	95400.0	141800.0	192900.0	61600.0	257500.0
2018-02-01	121400.0	95900.0	142800.0	195000.0	61700.0	260400.0
2018-03-01	122800.0	96600.0	145100.0	198000.0	62000.0	266200.0
2018-04-01	123800.0	97200.0	146900.0	200200.0	62200.0	270400.0

265 rows × 6 columns

In [9]:
```
type(springfield.index)
```

Out[9]:
```
pandas.core.indexes.datetimes.DatetimeIndex
```

Statistical Testing and Visualizations

Here we use matplotlib to visualize our data and get a general idea of it. We will also perform various statistical tests to check for stationarity, seasonality, correlation, and autocorrelation.

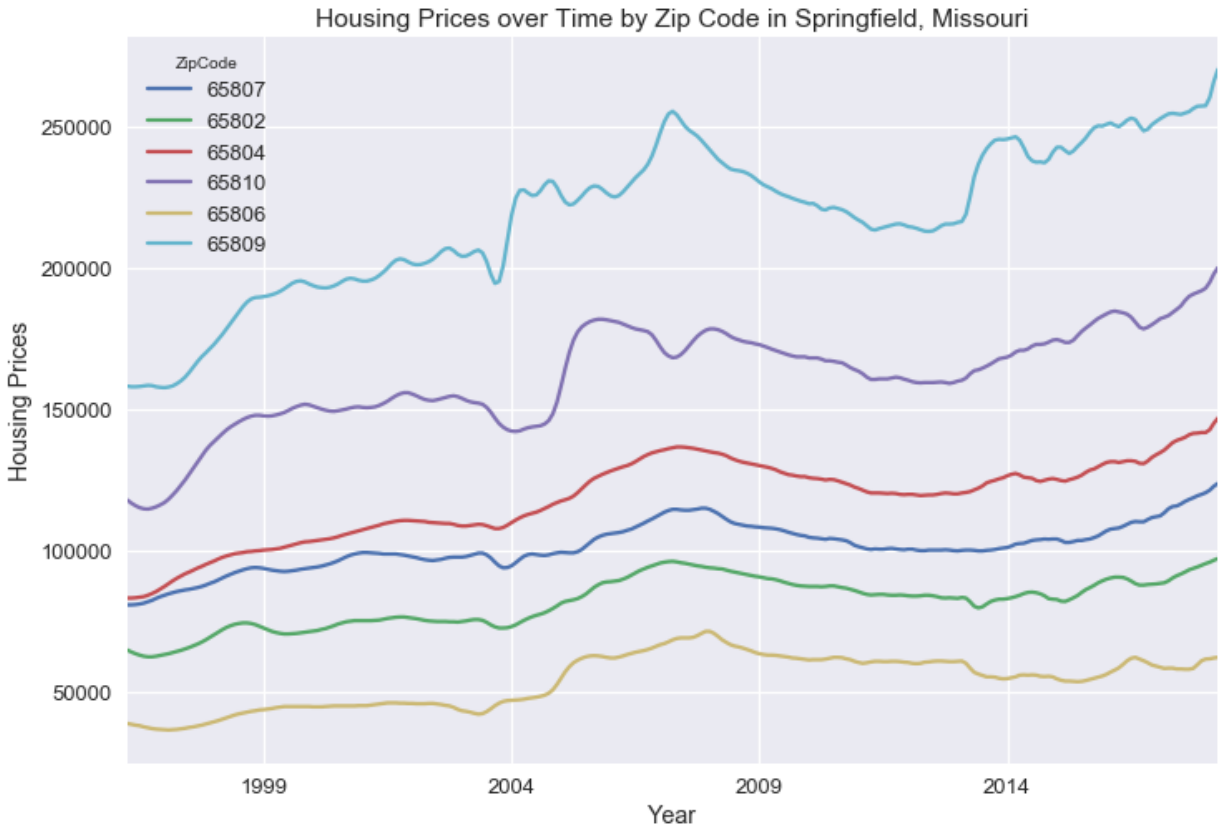
In [10]:
```
import matplotlib.pyplot as plt
plt.style.use('seaborn')
plt.style.use('seaborn-talk')
```

Stationarity

In [11]:
```
from statsmodels.tsa.stattools import adfuller
```

In [12]:
```
def stationarity_check(data, title='Housing Prices over Time by Zip Code in Sprin
    data.plot(title=title)
    plt.xlabel('Year')
    plt.ylabel('Housing Prices')
    plt.tight_layout()
    plt.savefig(f'visualizations/{filename}.png')
    plt.show()

    for col in data.columns:
        print (f'Dickey-Fuller Test for zip code {col}: \n')
        dftest = adfuller(data[col])
        print(pd.Series(dftest[0:4], index=['Test Statistic', 'p-value', '#Lags U
```

In [13]:
```
stationarity_check(springfield, 'Housing Prices over Time by Zip Code in Springfi
```

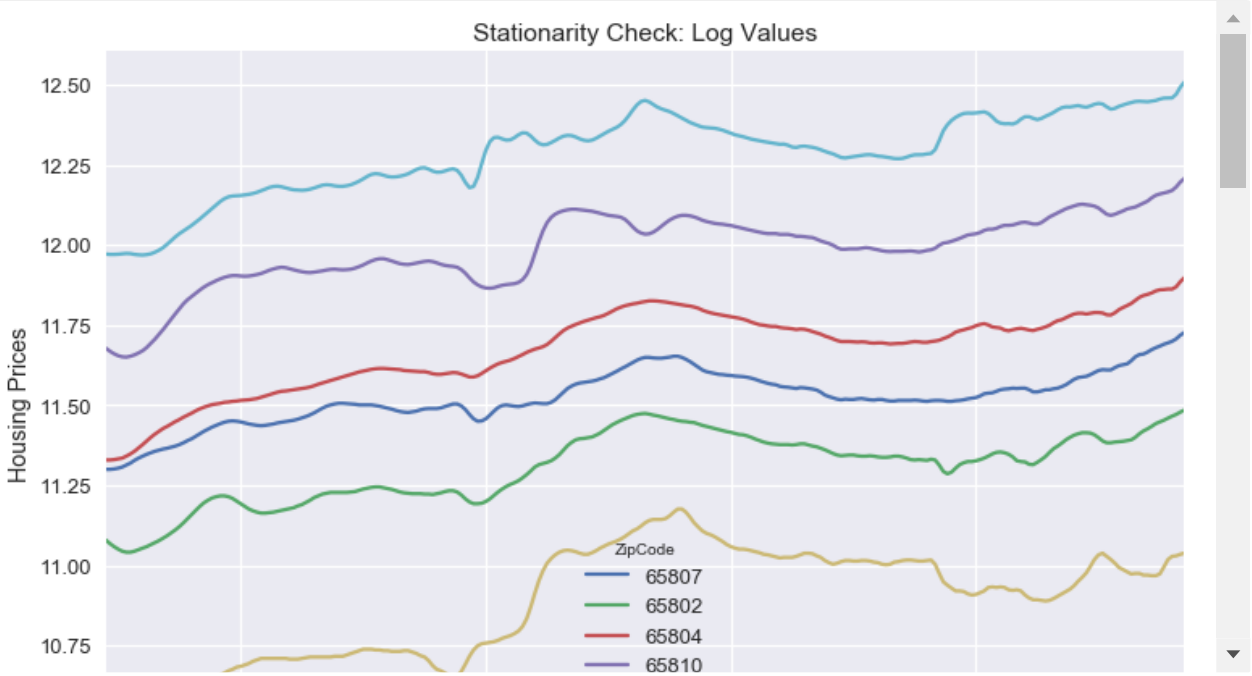



```
Dickey-Fuller Test for zip code 65807:

Test Statistic                  -0.692119
p-value                          0.848780
#Lags Used                      13.000000
Number of Observations Used    251.000000
dtype: float64

Dickey-Fuller Test for zip code 65802:

Test Statistic                  -1.601534
p-value                          0.482823
#Lags Used                       7.000000
Number of Observations Used    257.000000
dtype: float64

Dickey-Fuller Test for zip code 65804:

Test Statistic                  -0.313942
p-value                          0.923506
#Lags Used                      13.000000
Number of Observations Used    251.000000
dtype: float64

Dickey-Fuller Test for zip code 65810:

Test Statistic                  -2.094196
p-value                          0.246855
#Lags Used                       5.000000
```

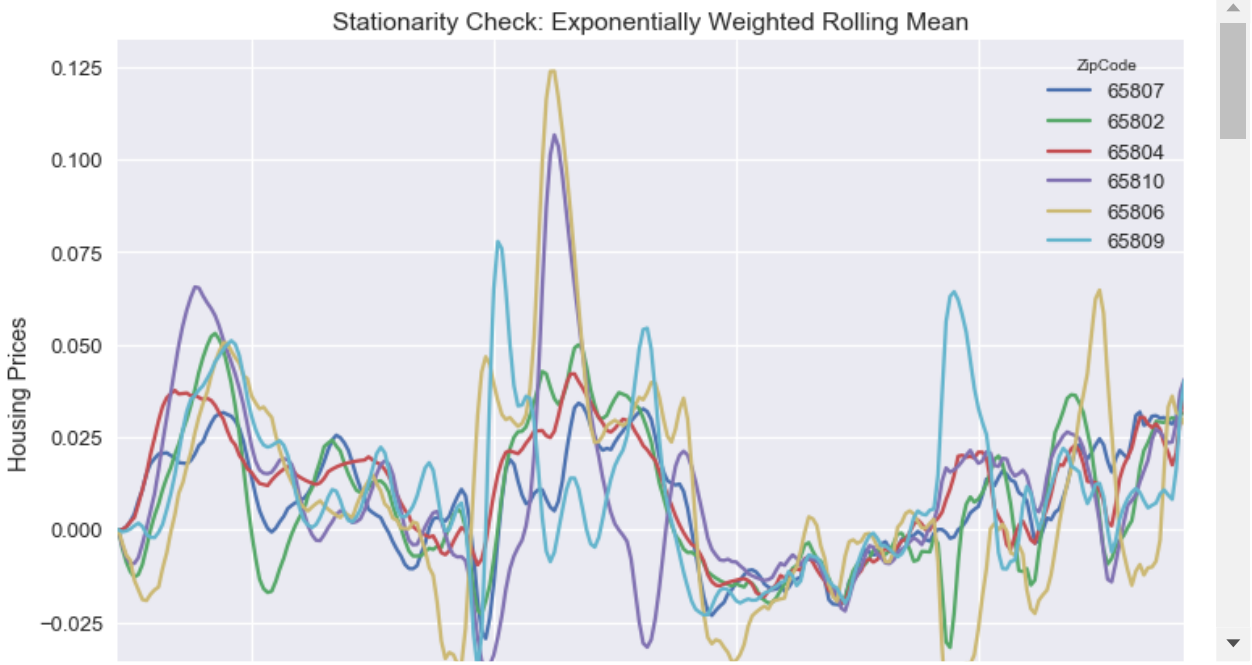
```
Number of Observations Used    259.000000
dtype: float64

Dickey-Fuller Test for zip code 65806:

Test Statistic                  -2.145813
p-value                          0.226496
#Lags Used                      16.000000
Number of Observations Used    248.000000
dtype: float64

Dickey-Fuller Test for zip code 65809:

Test Statistic                  -1.826638
p-value                          0.367319
#Lags Used                       9.000000
Number of Observations Used    255.000000
dtype: float64
```

We can see that the data is not stationary. The Dickey-Fuller test tells us when our data is not stationary enough - we want a p-value below 0.05 on all zip codes before we can run our baseline models. The p-value for all of our zip codes are high, so we need to force stationarity on each of them.

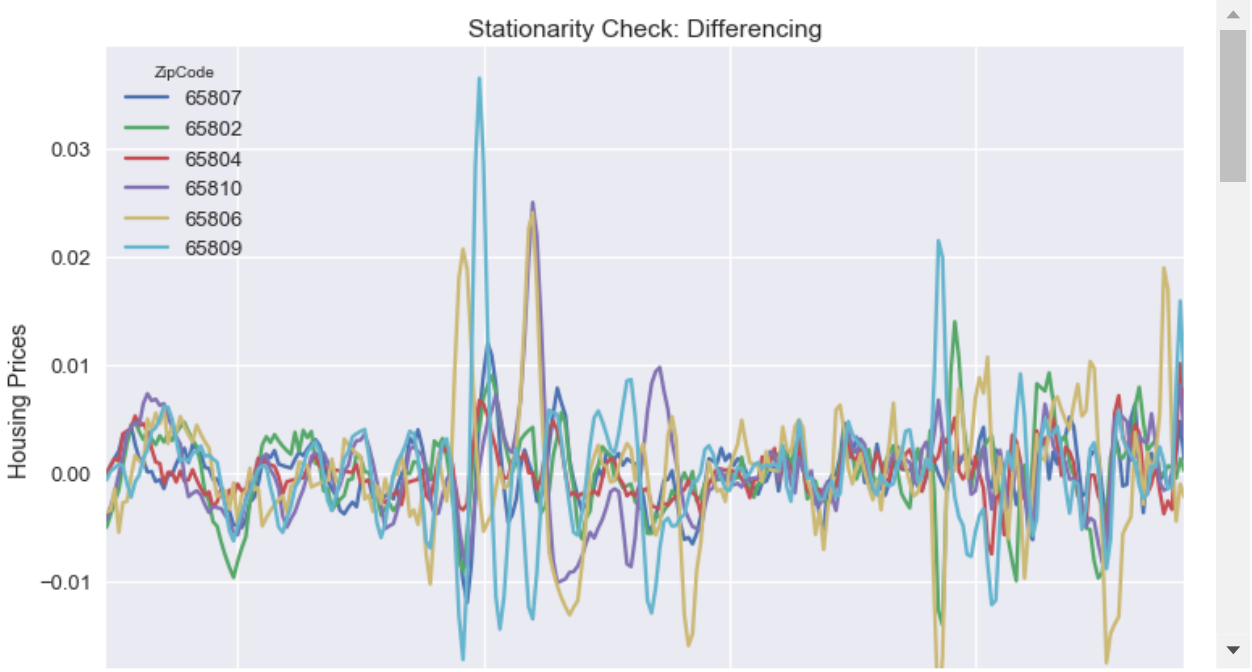
In [14]:
```
springfield_log = np.log(springfield)
stationarity_check(springfield_log, 'Stationarity Check: Log Values', 'stationari
```



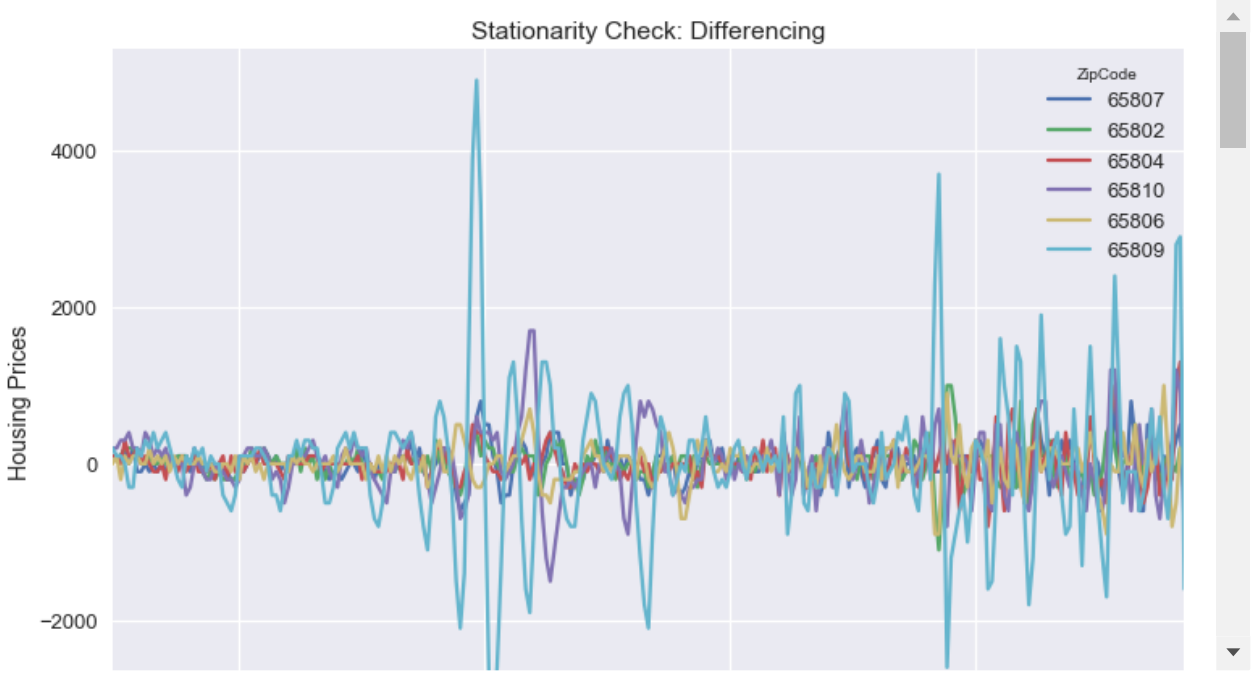
In [15]:
```
springfield_ewrm = springfield_log.ewm(halflife=4).mean()
stationarity_check((springfield_log-springfield_ewrm).dropna(), 'Stationarity Che
```



In [16]:
```
springfield_diff = (springfield_log-springfield_ewrm).diff(periods=1).dropna()
stationarity_check(springfield_diff, 'Stationarity Check: Differencing', 'station
```



In [17]:
```
springfield_diff_2 = springfield.diff(periods=1).diff(periods=1).dropna()

stationarity_check(springfield_diff_2, 'Stationarity Check: Differencing', 'stati
```

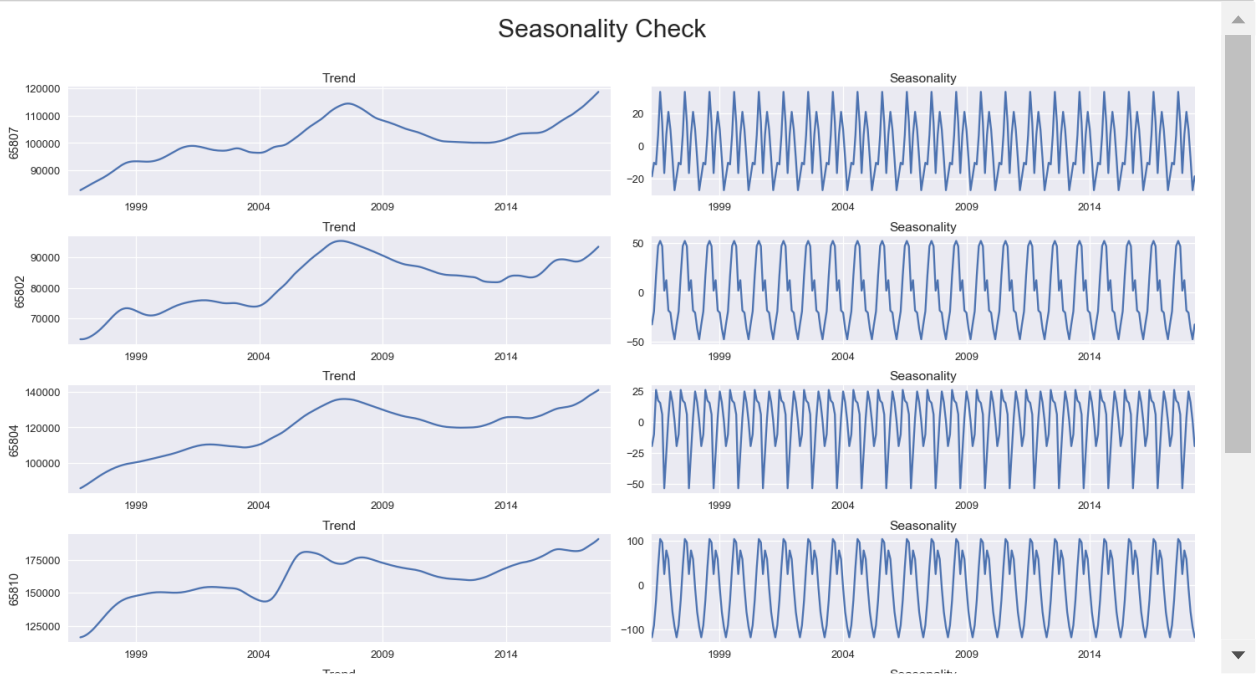


Although our data originally has some stationary concerns, we can minimize them with a combination of taking the log values, subtracting the exponentially weighted rolling mean, and then differencing the data. This differenced data is when will be used on our baseline ARMA model. The more advanced ARIMA model will perform the differencing automatically.

Seasonality

In [18]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
```

In [19]:
```
n = len(springfield.columns)
fig, axs = plt.subplots(n, 2, figsize=(20, 15))
fig.suptitle('Seasonality Check', y=1.05, fontsize=30)

for i, col in enumerate(springfield.columns):
    decomposition = seasonal_decompose(springfield[col])
    trend = decomposition.trend
    seasonal = decomposition.seasonal

    trend.plot(title='Trend', ax=axs[i, 0])
    axs[i, 0].set_ylabel(col)
    seasonal.plot(title='Seasonality', ax=axs[i, 1])

plt.tight_layout()
plt.savefig('visualizations/seasonality.png')
plt.show()
```



The scale of our seasonality is ±$100. Considering we are talking about housing prices in the hundreds of thousands, this accounts for ~0.1% of our values. We can safely disregard all seasonality concerns.

Correlation

In [20]:
```
springfield.corr()
```

Out[20]:

ZipCode	65807	65802	65804	65810	65806	65809
ZipCode						
65807	1.000000	0.950895	0.964716	0.919735	0.843225	0.908372
65802	0.950895	1.000000	0.974361	0.916246	0.945833	0.892115
65804	0.964716	0.974361	1.000000	0.946921	0.905693	0.956201
65810	0.919735	0.916246	0.946921	1.000000	0.822982	0.909652
65806	0.843225	0.945833	0.905693	0.822982	1.000000	0.802841
65809	0.908372	0.892115	0.956201	0.909652	0.802841	1.000000

The different zip codes seem to be highly correlated, but this is to be expected. They are, after all, all located in the same city. We are trying to pick one out of these six, so their correlation isn't really a concern for us.

The gently downwards-sloping autocorrelation plots indicate a high autocorrelation with a difference of one. The partial autocorrelation plots confirm this. We also do not see any autocorrelation with twelve, verifying our lack of seasonality we found earlier.

In [21]:
```
springfield.to_csv('data/springfield.csv', index_label='date')
springfield_diff_2.to_csv('data/springfield_diff_2.csv', index_label='date')
```

Finally, we save our base and differenced datasets to a csv, so they can be used in our modeling process.

Business Case

As Data Scientists working for a home renovation company, we were given the task to investigate and predict the average housing prices in the next two years using various zip codes in Springfield, MO. Using data from 1996 to 2018, we will use various time series models to determine which zip code would be the best investment to buy houses in to renovate.

In [1]:
```
import pandas as pd
```

In [2]:
```
springfield = pd.read_csv('data/springfield.csv', index_col='date')
springfield.index = pd.to_datetime(springfield.index)
springfield
```

Out[2]:

	65807	65802	65804	65810	65806	65809
date						
1996-04-01	80800.0	64800.0	83200.0	117900.0	38800.0	158200.0
1996-05-01	80800.0	64100.0	83200.0	116800.0	38500.0	158000.0
1996-06-01	80900.0	63500.0	83300.0	115900.0	38200.0	158000.0
1996-07-01	81100.0	63000.0	83500.0	115200.0	38000.0	158100.0
1996-08-01	81400.0	62600.0	83700.0	114800.0	37600.0	158300.0
...	...	...	...	...	...	...
2017-12-01	119900.0	94800.0	141800.0	192000.0	61000.0	257400.0
2018-01-01	120500.0	95400.0	141800.0	192900.0	61600.0	257500.0
2018-02-01	121400.0	95900.0	142800.0	195000.0	61700.0	260400.0
2018-03-01	122800.0	96600.0	145100.0	198000.0	62000.0	266200.0
2018-04-01	123800.0	97200.0	146900.0	200200.0	62200.0	270400.0

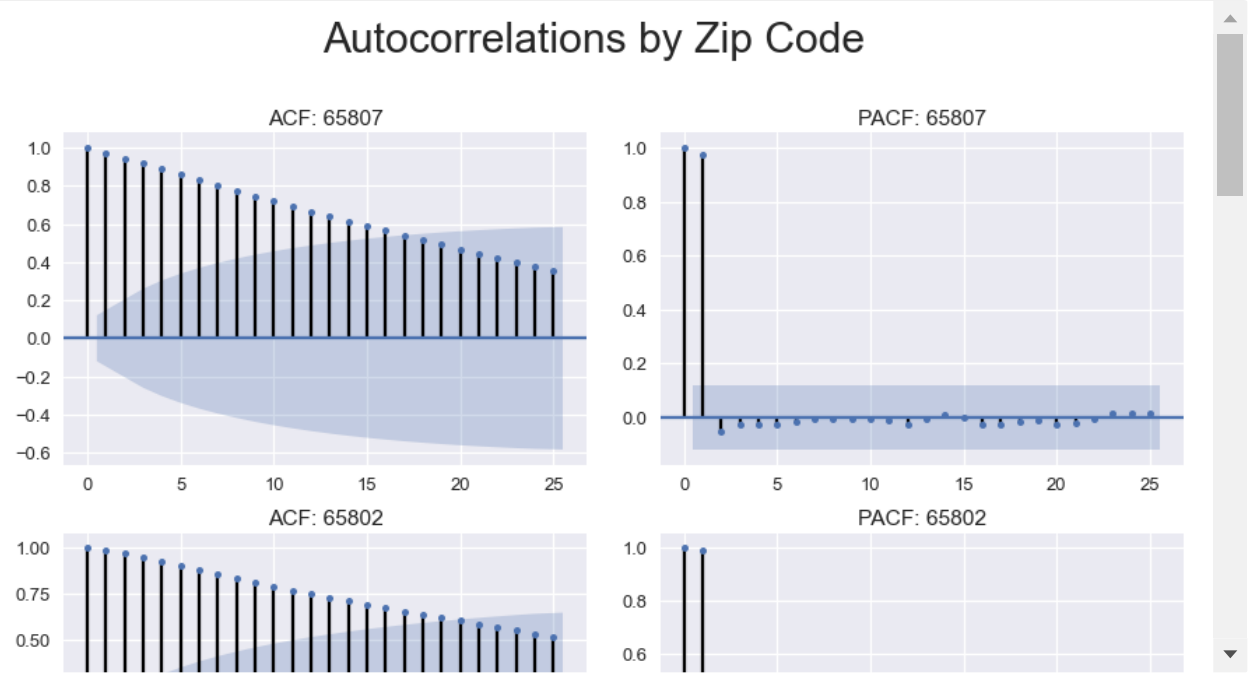
Autocorrelation and Partial Autocorrelation

Autocorrelation (ACF) helps us study how each time series observations is related to the past. Partial Autocorrelation (PACF) gives us a correlation of a time series with its own lagged values, controlling forthe values of the time series at all shorter lags.

These charts will help us determine the starting orders in a ARMA model.

In [3]:
```
from scripts import acf_pacf_charts
```

In [4]:
```
acf_pacf_charts(springfield, filename='autocorrelations')
```

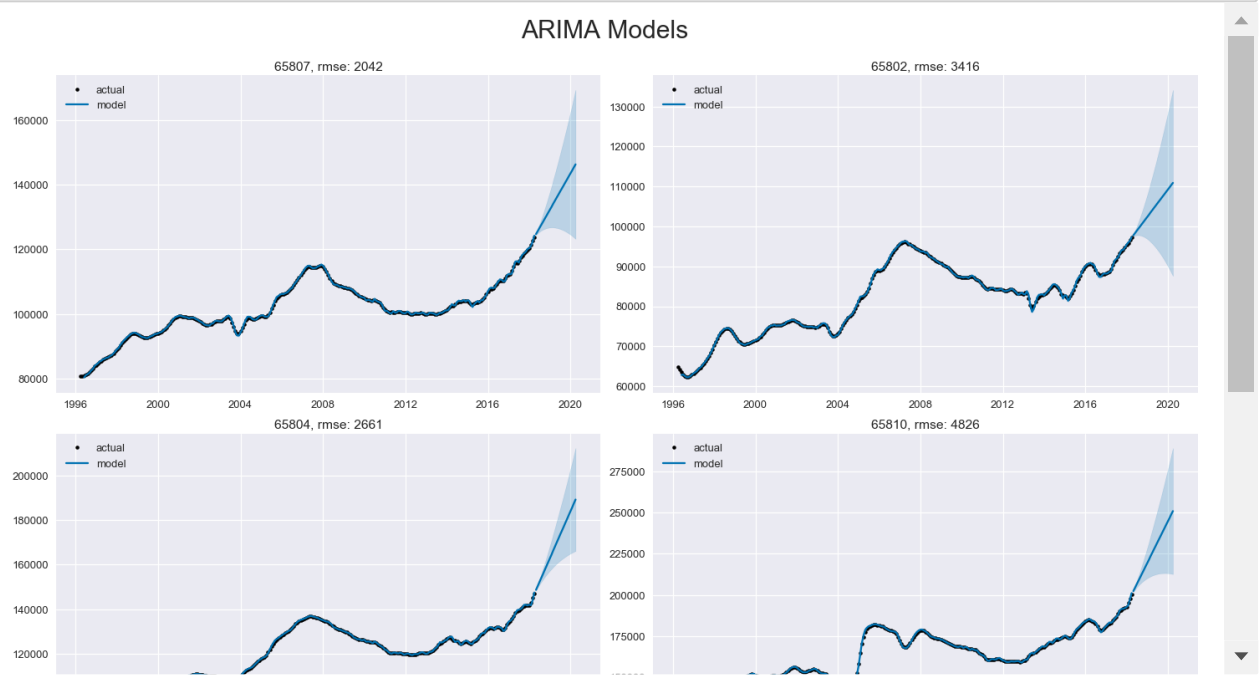


The gently downwards-sloping autocorrelation plots indicate a high autocorrelation with a difference of one. The partial autocorrelation plots confirm this. We also do not see any autocorrelation with twelve, verifying our lack of seasonality we found in our EDA. We will use this data to chose our order for the baseline ARIMA model.

Baseline ARIMA

In [3]:
```
from scripts import arima_analyze, arima_cross_validation
```

In [4]:
```
# order (1, 2, 0) chosen based on eda
baseline_arima_forecasts = arima_analyze(springfield, (1,2,0), filename='arima-ba
```



In [5]:
```
baseline_arima_forecasts.tail()
```

Out[5]:

	65807	65802	65804	65810	65806	65809
2019-12-01	142608.066618	108620.293529	182252.894643	242481.876901	65424.108209	353233.647886
2020-01-01	143548.019237	109190.875495	184020.434006	244595.504484	65584.530419	357375.223612
2020-02-01	144487.971857	109761.457461	185787.973369	246709.132067	65744.952629	361516.799337
2020-03-01	145427.924477	110332.039427	187555.512732	248822.759650	65905.374838	365658.375063
2020-04-01	146367.877097	110902.621393	189323.052095	250936.387233	66065.797048	369799.950789

This is the baseline models using ARIMA and the (1, 2, 0) order as determined in our EDA. The models have cross-validated RMSE values of 2042, 3416, 2661, 4826, 3368, and 11553. Based on a cursory look at the charts, the 65804 zip code seems to have the sharpest inclune in its forecast, indicating a good investment opportunity.

Auto-ARIMA

In [8]:
```
# !pip install pmdarima
```

In [9]:
```
from pmdarima.arima import auto_arima
```

In [10]:
```
order_dict = {}

for col in springfield.columns:
    auto = auto_arima(springfield[col], max_order=None, start_p=1, start_q=1,
                      max_p=10, max_q=10, d=2, max_d=5, information_criterion='ai
                      seasonal=False)

    order_dict[col] = auto.get_params()['order']

    print('Results:', col)
    print('order:', auto.get_params()['order'])
    print('aic:', auto.aic(), '\n\n\n')

order_dict
```

```
Results: 65807
order: (2, 2, 1)
aic: 3594.326805105192



Results: 65802
order: (0, 2, 1)
aic: 3557.1182912090926



Results: 65804
order: (1, 2, 2)
aic: 3627.4542630736646



Results: 65810
```

In [6]:
```
# temporary
# order_dict ={'65807': (2, 2, 1),
#              '65802': (0, 2, 1),
#              '65804': (1, 2, 2),
#              '65810': (2, 2, 2),
#              '65806': (0, 2, 1),
#              '65809': (1, 2, 0)}
```

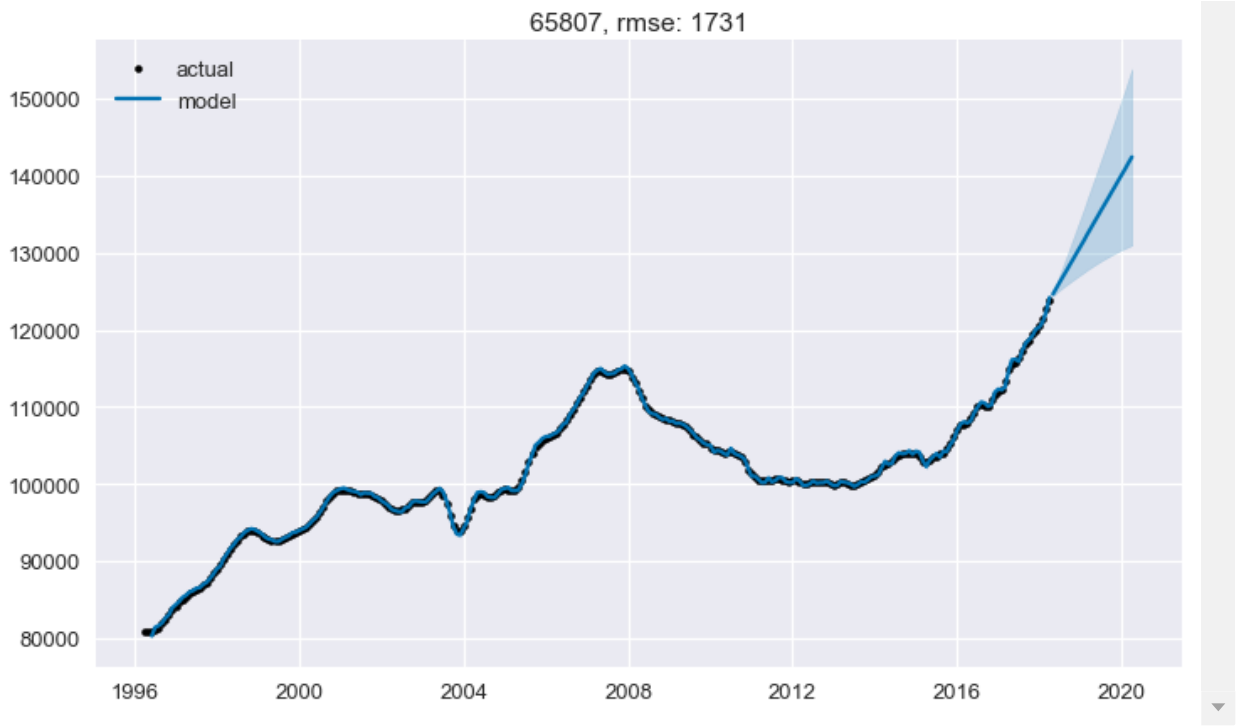
By running this auto-ARIMA function, we have determined the optimal order for each of our six models. We can use this to create more accurate ARIMA models.

Final ARIMA

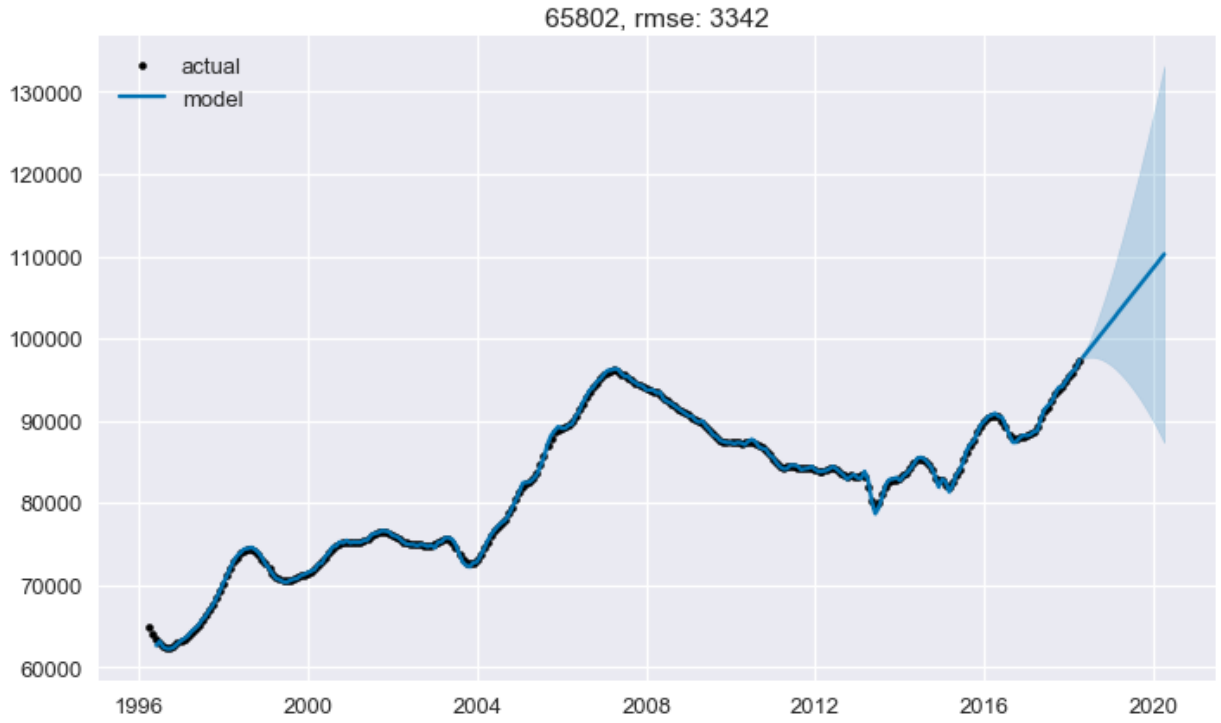
In [7]:
```
import numpy as np
from statsmodels.tsa.arima_model import ARIMA
```

In [8]:
```
forecasts = {}
rmse_dict = {}

for col in springfield.columns:
    order = order_dict[col]
    model = ARIMA(springfield[col], order=order, freq='MS').fit()
    pvalues = model.pvalues[1:]

    print('Zip Code:', col)
    print('Starting Order:', order)

    # looping while model still has high p-values
    while np.any(pvalues>0.05):
        # loops over every p-value
        for i, p in pvalues.iteritems():
            if p > .05:
                # ar causing high p-value
                if i.startswith('ar'):
                    print('AR p-value too high, lowered order from', order)
                    order = tuple(np.subtract(order, (1,0,0)))
                # ma causing high p-value
                elif i.startswith('ma'):
                    print('MA p-value too high, lowered order from', order)
                    order = tuple(np.subtract(order, (0,0,1)))
                else:
                    print('Unable to lower order - may need to force stop to prev
                    continue
                break

        model = ARIMA(springfield[col], order=order, freq='MS').fit()
        pvalues = model.pvalues[1:]


    print('Final Order:', order)
    forecasts[col] = (arima_analyze(springfield[[col]], order, filename=f'arima-f

    # collecting rmse into dict for future analysis
    rmses = arima_cross_validation(springfield[col], order)
    rmse = sum(rmses)/len(rmses)
    rmse_dict[col] = int(rmse)
```

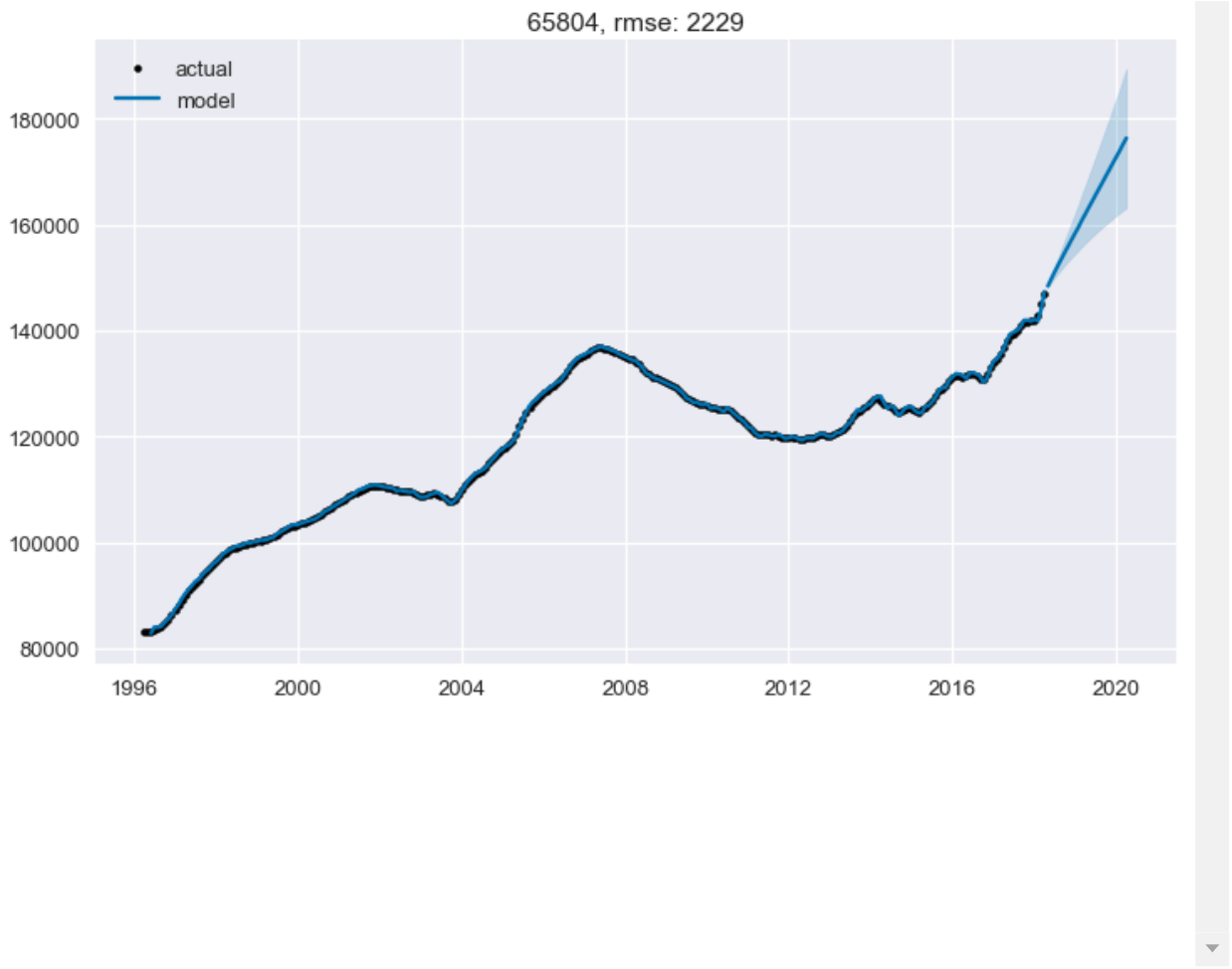
```
Zip Code: 65807
Starting Order: (2, 2, 1)
Final Order: (2, 2, 1)
```



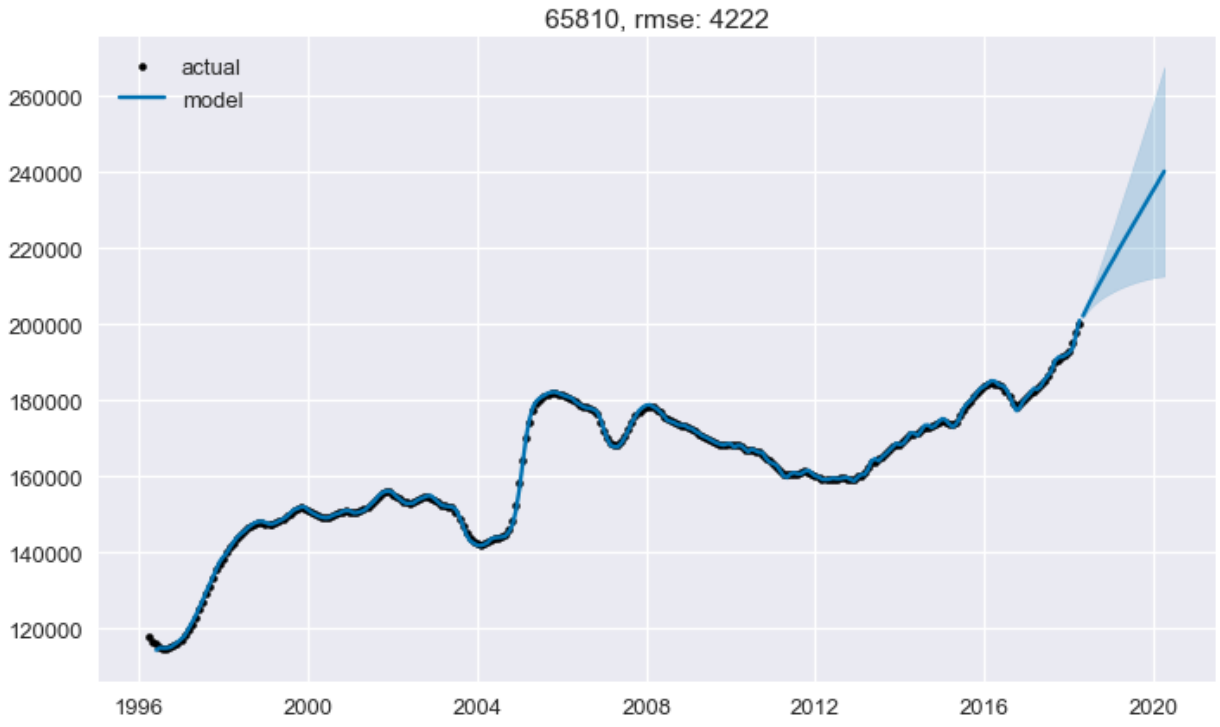
Zip Code: 65802
Starting Order: (0, 2, 1)
Final Order: (0, 2, 1)



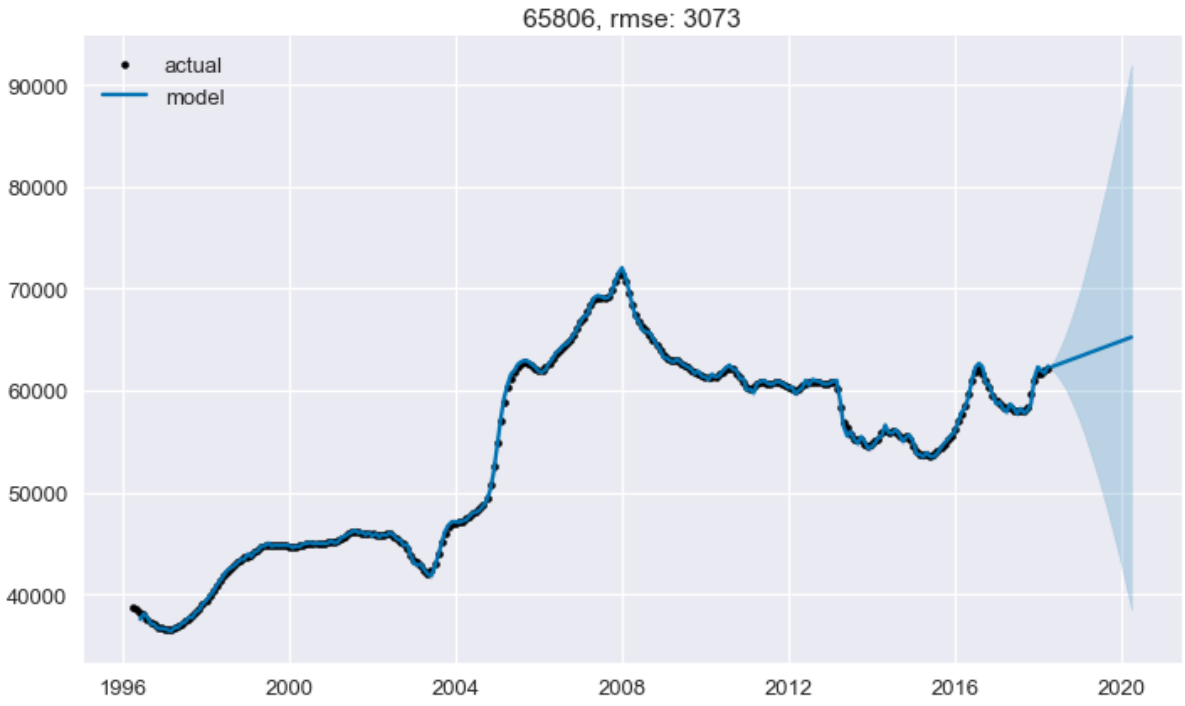
Zip Code: 65804
Starting Order: (1, 2, 2)
Final Order: (1, 2, 2)



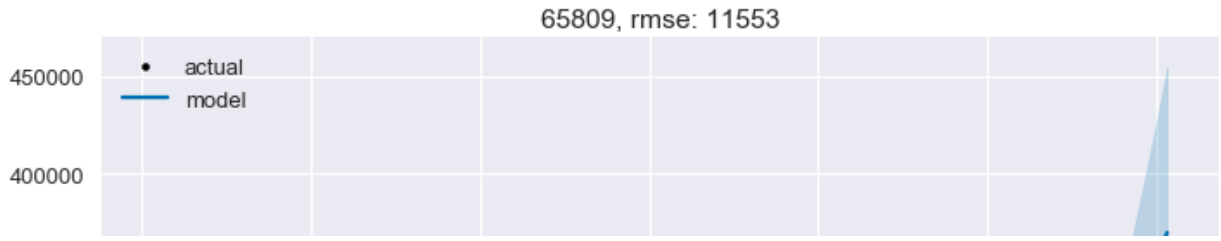

```
Zip Code: 65810
Starting Order: (2, 2, 2)
Final Order: (2, 2, 2)
```



```
Zip Code: 65806
Starting Order: (0, 2, 1)
Final Order: (0, 2, 1)
```




```
Zip Code: 65809
Starting Order: (1, 2, 0)
Final Order: (1, 2, 0)
```



Our final models all have different orders. Their rmse are 1731, 3342, 2229, 4222, 3073, and 11553, all lower than their baselines.

Despite the low rmse values, these charts show how closely fit to the data the predictions are, which may indicate overfitting.

Based on the charts, zip codes 65807 and 65804 have the steepest slopes, and might be good investment opportunities.

In [9]:
```
arima_forecasts = {i:k.values.flatten() for i, k in forecasts.items()}
arima_forecasts['date'] = list(forecasts.values())[0].index
forecasts_df = pd.DataFrame(arima_forecasts)
forecasts_df.set_index('date', inplace=True)
forecasts_df.tail()
```

Out[9]:

date	65807	65802	65804	65810	65806	65809
2019-12-01	139339.981663	108079.052464	171541.281566	233859.235450	64732.990786	353233.647886
2020-01-01	140112.341416	108623.005087	172732.054216	235450.906312	64859.640326	357375.223612
2020-02-01	140884.701514	109166.957710	173922.813285	237042.395971	64986.289865	361516.799337
2020-03-01	141657.061809	109710.910334	175113.563685	238633.760300	65112.939404	365658.375063
2020-04-01	142429.422217	110254.862957	176304.308555	240225.038176	65239.588944	369799.950789

In [10]:
```
forecasts_df.to_csv('data/arima_predictions.csv')
```

In [11]:
```
arima_rmses = pd.DataFrame.from_dict(rmse_dict.items())
arima_rmses.columns=['zip_code','arima_rmse']
arima_rmses
```

Out[11]:

	zip_code	arima_rmse
0	65807	1731
1	65802	3342
2	65804	2229
3	65810	4222
4	65806	3073
5	65809	11553

In [12]:
```
arima_rmses.to_csv('data/arima_rmses.csv')
```

Facebook Prophet

In [1]:
```
import pandas as pd
```

In [2]:
```
springfield = pd.read_csv('data/springfield.csv')
springfield['date'] = pd.to_datetime(springfield['date'])
springfield.rename(columns={'date':'ds'}, inplace=True)
springfield
```

Out[2]:

	ds	65807	65802	65804	65810	65806	65809
0	1996-04-01	80800.0	64800.0	83200.0	117900.0	38800.0	158200.0
1	1996-05-01	80800.0	64100.0	83200.0	116800.0	38500.0	158000.0
2	1996-06-01	80900.0	63500.0	83300.0	115900.0	38200.0	158000.0
3	1996-07-01	81100.0	63000.0	83500.0	115200.0	38000.0	158100.0
4	1996-08-01	81400.0	62600.0	83700.0	114800.0	37600.0	158300.0
...	...	...	...	...	...	...	...
260	2017-12-01	119900.0	94800.0	141800.0	192000.0	61000.0	257400.0
261	2018-01-01	120500.0	95400.0	141800.0	192900.0	61600.0	257500.0
262	2018-02-01	121400.0	95900.0	142800.0	195000.0	61700.0	260400.0
263	2018-03-01	122800.0	96600.0	145100.0	198000.0	62000.0	266200.0
264	2018-04-01	123800.0	97200.0	146900.0	200200.0	62200.0	270400.0

265 rows × 7 columns

Baseline

In [3]:
```
from fbprophet import Prophet
from fbprophet.diagnostics import cross_validation, performance_metrics
import matplotlib.pyplot as plt
plt.style.use('seaborn')
plt.style.use('seaborn-talk')
import warnings
warnings.filterwarnings('ignore')
```

In [4]:
```
forecasts = {}
fig = plt.figure(figsize=(20, 18))
axs = fig.subplots(3, 2).flatten()

for i, col in enumerate(springfield.columns[1:]):
    ts = springfield[['ds', col]].rename(columns={col:'y'})
    ts_model = Prophet()
    ts_model.fit(ts)
    forecast = ts_model.predict(ts_model.make_future_dataframe(periods=24, freq='
    df_cv = cross_validation(ts_model, initial='15 y', period='180 days', horizon
    df_p = performance_metrics(df_cv, rolling_window=1)
    rmse = df_p['rmse'].values[0]

    forecasts[col] = forecast[['ds', 'yhat']].rename(columns={'ds':'date', 'yhat'

    ax = axs[i]
    ax.plot('ds', col, 'k.', data=springfield)
    ax.plot(forecast['ds'], forecast['yhat'], ls='-', c='#0072B2')
    ax.fill_between(forecast['ds'], forecast['yhat_lower'], forecast['yhat_upper'
    ax.set_title(f'{col}, rmse: {int(rmse)}')
    ax.legend(labels=('actual', 'prediction'))

plt.suptitle('Facebook Prophet Models', y=1.03, fontsize=30)
plt.tight_layout()
plt.savefig('visualizations/prophet-baselines.png')
plt.show()
```

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonal
ity=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonalit
y=True to override this.
INFO:fbprophet:Making 13 forecasts with cutoffs between 2011-05-02 18:10:48 a
nd 2017-03-31 18:10:48
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonal
ity=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonalit
y=True to override this.

INFO:fbprophet:Making 13 forecasts with cutoffs between 2011-05-02 18:10:48 a
nd 2017-03-31 18:10:48
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonal
ity=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonalit
y=True to override this.
INFO:fbprophet:Making 13 forecasts with cutoffs between 2011-05-02 18:10:48 a
nd 2017-03-31 18:10:48
```

In [5]:
```
prophet_forecasts = {i:k.values.flatten() for i, k in forecasts.items()}
prophet_forecasts['date'] = list(forecasts.values())[0].index
prophet_baseline_df = pd.DataFrame(prophet_forecasts)
prophet_baseline_df.set_index('date', inplace=True)
prophet_baseline_df
```

Out[5]:

date	65807	65802	65804	65810	65806	65809
1996-04-01	80524.182390	61600.421777	82295.596158	112526.038084	36536.083108	153471.728985
1996-05-01	80869.557307	61796.811998	82806.004462	113278.649378	36560.487068	153704.686167
1996-06-01	81256.484504	62094.571223	83419.383034	114262.118611	36713.681273	154265.431965
1996-07-01	81665.619932	62418.672773	84012.254980	115335.600844	36987.452743	155278.762442
1996-08-01	82068.008507	62792.135191	84671.330337	116510.038826	37210.034227	156729.019048
...	...	...	...	...	...	...
2019-12-01	126644.870344	98674.301891	148020.254925	202551.754958	61124.717393	271687.142359

Hyperparameter Tuning

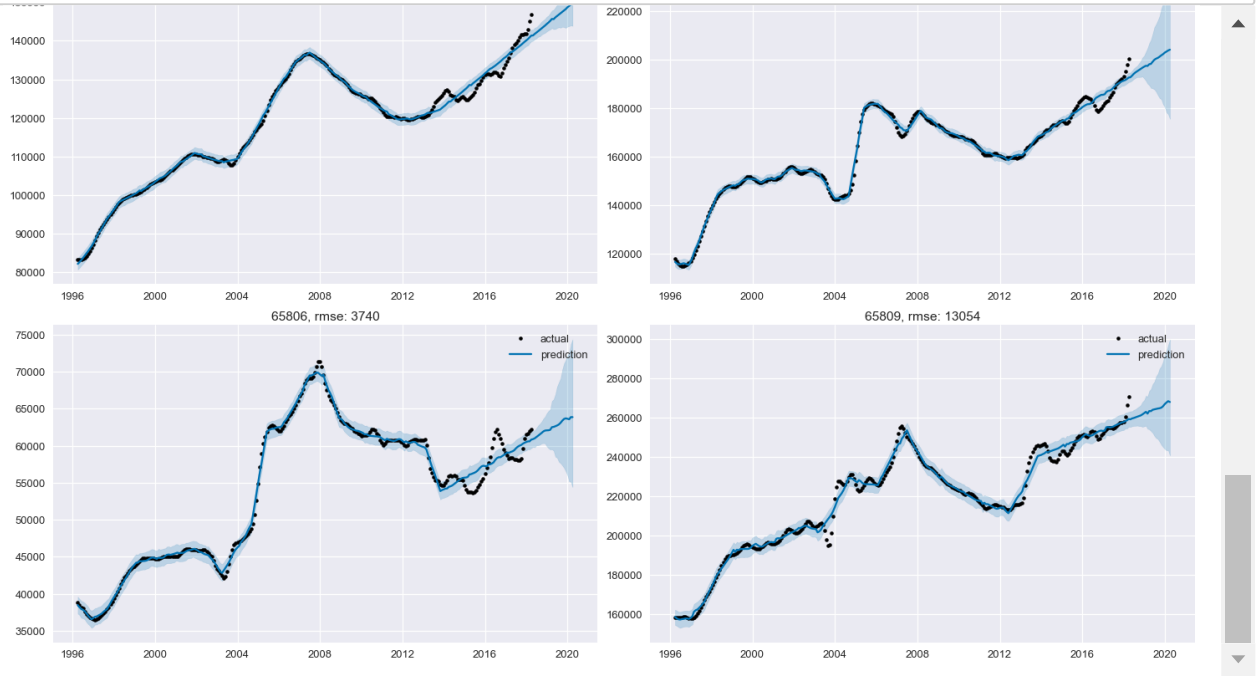
In [6]:
```
import itertools
```

In [7]:
```
param_grid = {
    'changepoint_prior_scale': [0.005, 0.05, 0.5], # default 0.05
    'seasonality_prior_scale': [1.0, 10.0], # default 10
    'holidays_prior_scale': [1.0, 10.0], # default 10
    'seasonality_mode': ['additive', 'multiplicative'] # default 'additive'
}
all_params = [dict(zip(param_grid.keys(), v)) for v in itertools.product(*param_g
best_params_all = []

for col in springfield.columns[1:]:
    ts = springfield[['ds', col]].rename(columns={col:'y'})

    best_params = dict.fromkeys(param_grid.keys())
    best_params['zipcode'] = col

    rmses = []  # Store the RMSEs for each params here

    # Use cross validation to evaluate all parameters
    for params in all_params:
        m = Prophet(**params).fit(ts)  # Fit model with given params
        df_cv = cross_validation(m, initial='15 y', period='180 days', horizon =
        df_p = performance_metrics(df_cv, rolling_window=1)
        rmse = df_p['rmse'].values[0]

        if 'rmse' not in best_params.keys() or rmse<best_params['rmse']:
            best_params.update(params)
            best_params['rmse'] = rmse

    best_params_all.append(best_params)
    print(col, 'model best parameters:')
    print(best_params)

tuning_results = pd.DataFrame(best_params_all)
tuning_results.set_index('zipcode', inplace=True)
tuning_results
```

```
y=True to override this.
INFO:fbprophet:Making 13 forecasts with cutoffs between 2011-05-02 18:10:48 a
nd 2017-03-31 18:10:48
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonal
ity=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonalit
y=True to override this.
INFO:fbprophet:Making 13 forecasts with cutoffs between 2011-05-02 18:10:48 a
nd 2017-03-31 18:10:48

INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonal
ity=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonalit
y=True to override this.
INFO:fbprophet:Making 13 forecasts with cutoffs between 2011-05-02 18:10:48 a
nd 2017-03-31 18:10:48
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonal
ity=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonalit
y=True to override this.
```

Final Model

In [18]:
```
forecasts = {}
rmse_dict = {}
fig = plt.figure(figsize=(20, 18))
axs = fig.subplots(3, 2).flatten()

for i, col in enumerate(tuning_results.index):
    ts = springfield[['ds', col]].rename(columns={col:'y'})
    ts_model = Prophet(**tuning_results.drop(columns='rmse').iloc[i].to_dict())
    ts_model.fit(ts)
    forecast = ts_model.predict(ts_model.make_future_dataframe(periods=24, freq='
    df_cv = cross_validation(ts_model, initial='15 y', period='180 days', horizon
    df_p = performance_metrics(df_cv, rolling_window=1)
    rmse = df_p['rmse'].values[0]
    rmse_dict[col] = int(rmse)

    forecasts[col] = forecast[['ds', 'yhat']].rename(columns={'ds':'date', 'yhat'

    ax = axs[i]
    ax.plot('ds', col, 'k.', data=springfield)
    ax.plot(forecast['ds'], forecast['yhat'], ls='-', c='#0072B2')
    ax.fill_between(forecast['ds'], forecast['yhat_lower'], forecast['yhat_upper'
    ax.set_title(f'{col}, rmse: {int(rmse)}')
    ax.legend(labels=('actual', 'prediction'))

plt.suptitle('Facebook Prophet Models', y=1.03, fontsize=30)
plt.tight_layout()
plt.savefig('visualizations/prophet-models.png')
plt.show()
```



In [9]:
```
prophet_forecasts = {i:k.values.flatten()[-24:] for i, k in forecasts.items()}
prophet_forecasts['date'] = list(forecasts.values())[0].index[-24:]
prophet_final_df = pd.DataFrame(prophet_forecasts)
prophet_final_df.set_index('date', inplace=True)
prophet_final_df.tail()
```

Out[9]:

date	65807	65802	65804	65810	65806	65809
2019-12-01	127070.233087	99174.442951	148194.536870	202212.222251	63707.912363	265616.752851
2020-01-01	127517.386168	99590.750794	148669.931055	202847.907779	63653.907614	266730.304083
2020-02-01	127809.488868	99904.390392	149031.206487	203300.825200	63584.750557	267542.519975
2020-03-01	128064.701521	100012.391682	149444.405223	203721.232062	63874.360932	268200.753897
2020-04-01	128651.011816	100255.124506	149841.237435	204064.850838	63873.122266	267778.466056

In [10]:
```
prophet_final_df.to_csv('data/prophet_predictions.csv')
```

In [19]:
```
fb_rmses = pd.DataFrame.from_dict(rmse_dict.items())
fb_rmses.columns=['zip_code','fb_rmse']
fb_rmses
```

Out[19]:

	zip_code	fb_rmse
0	65807	3829
1	65802	3074
2	65804	4339
3	65810	6539
4	65806	3740
5	65809	13054

In [20]:
```
fb_rmses.to_csv('data/fb_rmses.csv')
```

Analysis

In [1]:
```
import pandas as pd
```

In [2]:
```
arima_predictions = pd.read_csv('data/arima_predictions', index_col='date')
arima_predictions.index = pd.to_datetime(arima_predictions.index)

prophet_predictions = pd.read_csv('data/prophet_predictions.csv', index_col='date
prophet_predictions.index = pd.to_datetime(arima_predictions.index)
```

In [4]:
```
arima_predictions.tail()
```

Out[4]:

date	65807	65802	65804	65810	65806	65809
2019-12-01	137481.612848	100931.114780	168107.429120	208566.975000	64758.461240	314575.448793
2020-01-01	138193.703838	101074.440374	169160.295731	208933.789680	64901.576122	316724.333984
2020-02-01	138907.295866	101217.932291	170215.629355	209301.077634	65046.137845	318882.407084
2020-03-01	139622.377104	101361.578804	171273.430893	209668.783798	65192.146407	321049.672687
2020-04-01	140338.942900	101505.374645	172333.700868	210036.867794	65339.601810	323226.133471

In [5]:
```
prophet_predictions.tail()
```

Out[5]:

date	65807	65802	65804	65810	65806	65809
2019-12-01	127070.233087	99174.442951	148194.536870	202212.222251	63707.912363	265616.752851
2020-01-01	127517.386168	99590.750794	148669.931055	202847.907779	63653.907614	266730.304083
2020-02-01	127809.488868	99904.390392	149031.206487	203300.825200	63584.750557	267542.519975
2020-03-01	128064.701521	100012.391682	149444.405223	203721.232062	63874.360932	268200.753897
2020-04-01	128651.011816	100255.124506	149841.237435	204064.850838	63873.122266	267778.466056

RMSE comparison

In [41]:
```
arima_rmses = pd.read_csv('data/arima_rmses.csv', index_col=0)
arima_rmses.set_index('zip_code', inplace=True)

fb_rmses = pd.read_csv('data/fb_rmses.csv', index_col=0)
fb_rmses.set_index('zip_code', inplace=True)

df_rmse = pd.concat([arima_rmses, fb_rmses], axis=1)
```

In [42]:
```
def best_model(row):
    if row['arima_rmse'] < row['fb_rmse']:
        val = 'arima'
    else:
        val = 'fb'
    return val
```

In [43]:
```
df_rmse['best_model'] = df_rmse.apply(best_model, axis=1)
df_rmse
```

Out[43]:

zip_code	arima_rmse	fb_rmse	best_model
65807	1731	3829	arima
65802	3342	3074	fb
65804	2229	4339	arima
65810	4222	6539	arima
65806	3073	3740	arima
65809	11553	13054	arima

The RMSE is lower on all ARIMA models except for zip code 65802.

Zip Code Selection

In [39]:
```
from scipy import stats
import datetime as dt
```

In [44]:
```
arima_predictions
```

Out[44]:

date	65807	65802	65804	65810	65806	65809
2018-05-01	124546.038914	97684.464161	148184.832451	201396.968517	62314.178082	273671.988982
2018-06-01	125169.802562	98064.249991	149353.760900	202105.993874	62429.803003	276407.338543
2018-07-01	125760.381789	98360.364776	150456.588094	202585.449081	62546.874765	278833.986536
2018-08-01	126364.401331	98595.126886	151522.163437	202962.986466	62665.393367	281084.685736
2018-09-01	126997.888221	98788.004939	152567.198616	203300.201554	62785.358809	283236.752389
2018-10-01	127659.710730	98954.192982	153601.374266	203625.686574	62906.771091	285335.216077

In [45]:
```
arima_predictions['date_ordinal'] = arima_predictions.index.map(dt.datetime.toord
prophet_predictions['date_ordinal'] = prophet_predictions.index.map(dt.datetime.t

print('Arima model slopes')
for col in arima_predictions.columns[:-1]:
    slope = stats.linregress(arima_predictions['date_ordinal'], arima_predictions
    print(col, 'slope:', round(slope,2))

print('\nProphet model slopes')
for col in arima_predictions.columns[:-1]:
    slope = stats.linregress(prophet_predictions['date_ordinal'], prophet_predict
    print(col, 'slope:', round(slope,2))
```

```
Arima model slopes
65807 slope: 22.76
65802 slope: 4.99
65804 slope: 34.19
65810 slope: 11.8
65806 slope: 4.32
65809 slope: 69.41

Prophet model slopes
65807 slope: 12.9
65802 slope: 8.21
65804 slope: 11.88
65810 slope: 15.84
65806 slope: 4.36
65809 slope: 12.58
```

Choosing the correct model for each zip code, we get the following results:

- 65809: 69.41
- 65804: 34.19
- 65807: 22.76

- 65810: 11.8
- 65802: 8.21
- 65806: 4.32

A steep slope indicates a quick rise in value in the future, indicating a good investment. Based on this, zip code 65809 would be a good investment. However, the RMSE values for that zip code's model are much higher than the others, possibly indicating a poorer accuracy forecast or more volatility. Instead, the second steepest slope, zip code 65804, should be chosen as the top investment opportunity.

Conclusion

Based on the models, we have concluded that zip code 65804 would be the best zip code for the home renovation company to invest their money in. If focusing in the 65804 area, the company should see the average housing prices continuing to increase over the next two years.

Future Work

It is important to note that these values are already old. Even our 2-year prediction is already 9 months out of date. As well, the chosen ARIMA models had apparent overfitting issues based on their prediction charts, and larger confidence intervals as well. More work would need to be done before making actual investments based on these models.

More recent data would be needed in order to make this model useful for actual investments. If obtained, it could also be used as a sort of holdout dataset, verifying our current conclusions. We have also chosen a 2-year forecast range arbitrarily. In order to improve our model, we would need to perform an analysis on what range these models are most accurate over, and what time frame is useful in the field of housing investments.

```
import matplotlib.pyplot as plt
plt.style.use('seaborn')
plt.style.use('seaborn-talk')
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
import warnings
warnings.filterwarnings('ignore')

def arima_cross_validation(data, order, initial=12*15, horizon=12, period=6, verbose=False):
    k = (len(data)-initial-horizon)//period
    if verbose: print('Cross validating over', str(k), 'folds.')

    rmses = []
    for i in range(1, k+1):
        n = len(data)-horizon-((k-i)*period)
        model = ARIMA(data[:n], order=order, freq='MS').fit()
        y_hat = model.get_forecast(steps=horizon).predicted_mean.to_numpy()
        y = data[n:n+horizon].to_numpy()
        rmse = np.sqrt(mean_squared_error(y, y_hat))
        if verbose: print(f'fold {i}: train[0:{n}], test[{n}:{n+horizon}] of {len(data)}, rmse={rmse}')
        rmses.append(rmse)

    return rmses

def arima_analyze(data, order, initial=12*15, horizon=12, period=6, forecast_length=24, filename=None):

    forecast_index = pd.date_range(data.index[-1], periods=forecast_length+1, freq='MS')[1:]
    forecast_df = pd.DataFrame(index=forecast_index)

    n = len(data.columns)
    rows, cols = -(-n//2), (1+(n>1))
    fig = plt.figure(figsize=(10*cols, 6*rows))
    axs = fig.subplots(rows, cols, squeeze=False).flatten()

    for i, col in enumerate(data.columns):
        #cross-validation to get rmse
        rmses = arima_cross_validation(data=data[col], order=order, initial=initial, horizon=horizon, period=period)
        rmse = sum(rmses)/len(rmses)

        #run model to get 1-year forecast
        model = ARIMA(data[col], order=order, freq='MS').fit()
        prediction_results = model.get_prediction(start=2, typ='levels')
        prediction = prediction_results.predicted_mean
        prediction_conf_int = prediction_results.conf_int(alpha=0.2)
        prediction_lower = prediction_conf_int[f'lower {col}'].tolist()
        prediction_upper = prediction_conf_int[f'upper {col}'].tolist()

        forecast_results = model.get_forecast(steps=forecast_length)
```

```
52          forecast = forecast results.predicted mean
```