notebook

December 14, 2020

1 Problem Statement

For this project, we imagine that an individual is moving the city of Baltimore, Maryland. They plan to buy a residential property somewhere within city limits and live there for two years. After two years have passed they plan to sell their property and move to a new city. They want to restrict their search for a property to five zip codes that have the highest expected return on investment for buying, holding for two years, and then selling a residential property.

We will use Seasonal AutoRegressive Moving Average (SARIMA) models to forecast median sale price for residential properties by zip code. We train our models on Zillow's Home Value Index for single family homes reporting values up to 2020-10-31.

Since we want to identify the five best zip codes for our client, we need to forecast an expected median sale price for single family homes, for each zip code in the city of Baltimore. The historical data contains data for twenty zip codes in the city of Baltimore. That means that we will need to fit twenty SARIMA models and evaluate return on investment for based on their forecasts.

To keep out analysis organized and maximize re-usability of our code, we have decided to build a class that wraps a nested dictionary which will contain all of the objects needed for our analysis. Essentially, we build a collection of helper functions that execute all of the steps of our analysis for a single zip code, store the more computationally expensive outputs of our analysis in a dictionary, then collect all of our individual zip code level analyses into an outer dictionary which provides all of the information needed for our city-wide analysis. The precise data structure is not important, since we have provided methods that access all of required information for our analysis.

2 Step 0: Defining the ZipCodeROIModel Class

In the cell below, we have imported all required modules and defined our ZipCodeROIModel class. All class methods have docstrings that describe their expected behavior.

[1]:
```
# Basics
import pandas as pd
import numpy as np

#Plotting
from matplotlib import pyplot as plt
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.graphics.tsaplots import plot_acf
plt.style.use('ggplot')
```

```
# Warnings
import warnings
warnings.filterwarnings('ignore')

#Auto ARIMA
try:
    from pmdarima.arima import auto_arima
except:
    ! pip install pmdarima
    from pmdarima.arima import auto_arima

# Import statsmodels api
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Tools for saving the model
import pickle
from bz2 import BZ2File

# Mapping
import folium

class ZipCodeROIModel(object):
    """Uses SARIMA models to forecast median sale price of residential real␣
↪estate by zip code."""
    def __init__(self, test_size=36, forecast_size=24, start_date='2012-01-01'):
        """Initialize a ZipCodeROIModel.

        Keyword Arguments:
        test_size -- Number of months to hold out as a test set for fitting␣
↪SARIMA models. (default 36)
        forecast_size -- Number of months to forcast into the future. (default␣
↪24)
        start_date -- Train models using provided timeseries starting with this␣
↪date. (default '2012-01-01')
        """
        self.model_dictionary = {}
        self.test_size = test_size
        self.forecast_size = forecast_size
        self.start_date = start_date

    # Load data from data frame
    def load_data(self, df):
        """Load data from provided data fram into `model_dictionary`.
        This function
        """
        self._make_datetime_index(df)
```

```
        for ind in df.index.values:
            row = df.loc[ind]
            zip_code = self._get_zip_code(row)
            row_dict = self._make_row_dict(row)
            self.model_dictionary[zip_code] = row_dict

    def _make_datetime_index(self, df):
        """Convert column names from the provided data frame to a datetime␣
↪index for timeseries.

        Keyword Arguments:
        df -- dataframe loaded using the `load_data` method.
        """
        string_index = df.columns.values[8:]
        self.datetime_index = pd.to_datetime(string_index)

    def _get_zip_code(self, row):
        """Given a row from the dataframe loded by `load_data` return the zip␣
↪code."""
        return row['RegionName']

    def _make_row_dict(self, row):
        """Given a row from the dataframe loaded by `load_data`,
        construct a dictionary populated with zipcode level data.
        """
        row_dict = row.iloc[1:8].to_dict()
        time_series = row.iloc[8:]
        df = self._make_time_series_df(time_series)
        df = df.fillna(df.bfill())
        row_dict['TimeSeries'] = df
        return row_dict

    def _make_time_series_df(self, time_series):
        """Given the time series portion of a row from the datafarme loaded by␣
↪`laod_data`,
        return a time time series dataframe with a datetime index.
        """
        time_series.index = self.datetime_index
        df = pd.DataFrame(time_series, dtype=float)
        df.columns = ['MedianSales']
        return df

    # Build the model from archive if availabel, otherwise build from data.
    def build_model(self):
        """This function will loop over all loaded data and exicute the␣
↪following steps for each zip code:
         - Select optimal hyperparameters for a SARIMA model.
```

```
        - Fit the optimal SARIMA model to the data.
        - Predict median sale price for the date range from the training data␣
↪for model validation.
        - Forecast the median saleprice for `forecast_size` months into the␣
↪future.
        - Compute expected return on investment.
        - Save an archive of the model_dictionary to a compressed pickle file.

        If an archive file is present, this function will load the model from␣
↪the archive.
        To force a refit, rename or remove modelArchive.bz2 from the working␣
↪directory.
        """
        try:
            with BZ2File('modelArchive.bz2', 'rb') as file:
                self.model_dictionary = pickle.load(file)
        except:
            print("File not found. Building model.")
            zip_codes = self.model_dictionary.keys()
            N = len(zip_codes)
            for ind, zip_code in enumerate(zip_codes):
                self.fit(zip_code, trace=False)
                self.predict(zip_code)
                self.forecast(zip_code)
                self.compute_roi(zip_code)
                print(f'Finished processing {ind+1} out of {N} zip codes.')
            with BZ2File('modelArchive.bz2', 'wb') as file:
                pickle.dump(self.model_dictionary, file)

    def get_time_series(self, zip_code):
        """Given a zip code, return the corrisponding time series dataframe.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        return self.model_dictionary[zip_code]['TimeSeries'][self.start_date:]

    def get_city_name(self, zip_code):
        """Given a zip code, return the corrisponding city name.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        return self.model_dictionary[zip_code]['City']

    def get_state_abbreviation(self, zip_code):
        """Given a zip code, return the corrisponding state abberiviation.
```

```
        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        return self.model_dictionary[zip_code]['State']

    # Train Test Split
    def get_train_test_split(self, zip_code):
        """Given a zip code, split the corrisponding time series dataframe into
        a training time series and a test time series consisting of `self.
↪test_size`
        months of data.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        time_series_df = self.get_time_series(zip_code)
        train = time_series_df[:-self.test_size]
        test = time_series_df[-self.test_size:]
        return train, test

    # Plotting
    def time_series_plot(self, zip_code, show_prediction=True,␣
↪show_forecast=True):
        """Given a zip code, plot the train and test time series.
        Optionally, show the SARIMA model prediction over the train and test␣
↪data
        or the forcast `self.forcast_size` months into the future.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        show_prediction -- the prediction is shown. (default True)
        show_forcast -- the forcast is shown. (default True)
        """
        fig, ax = plt.subplots(figsize=(8, 5));
        self._time_series_plot_train_test_ax(zip_code, ax)
        legend_labels = ['Train', 'Test']
        if show_prediction:
            self._time_series_plot_prediction_ax(zip_code, ax)
            legend_labels.append('Prediction')
        if show_forecast:
            self._time_series_plot_forcast_ax(zip_code, ax)
            legend_labels.append('Forecast')
        self._time_series_plot_annotate(zip_code, ax, legend_labels)
        return fig

    def _time_series_plot_train_test_ax(self, zip_code, ax):
```

```python
        """Add the plot of train and test data to the time series plot."""
        train, test = self.get_train_test_split(zip_code)
        ax.plot(train);
        ax.plot(test);

    def _time_series_plot_prediction_ax(self, zip_code, ax):
        """Add the prediction to the time series plot."""
        prediction = self.predict(zip_code)
        ax.plot(prediction['mean'])
        ax.fill_between(
            prediction.index,
            prediction['mean_ci_lower'],
            prediction['mean_ci_upper'],
            color = 'g',
            alpha = 0.5
        )

    def _time_series_plot_forcast_ax(self, zip_code, ax):
        """Add the forcast to the timeseries plot."""
        forecast = self.forecast(zip_code)
        ax.plot(forecast['mean'])
        ax.fill_between(
            forecast.index,
            forecast['mean_ci_lower'],
            forecast['mean_ci_upper'],
            color = 'g',
            alpha = 0.5
        )

    def _time_series_plot_annotate(self, zip_code, ax, legend_labels):
        """Annotate the time serties plot."""
        city_name = self.get_city_name(zip_code)
        state_abbreviation = self.get_state_abbreviation(zip_code)
        ax.set_title(f'Median Sale Price {city_name}, {state_abbreviation}␣
↪{zip_code}')
        ax.set_xlabel('Date')
        ax.set_ylabel('Price')
        ax.legend(legend_labels);

    def acf_plot(self, zip_code):
        """Given a zip code, plot the autocorrelation function for the training␣
↪data.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        city_name = self.get_city_name(zip_code)
```

```
        state_abbreviation = self.get_state_abbreviation(zip_code)
        train, test = self.get_train_test_split(zip_code)
        fig, ax = plt.subplots(figsize=(8,5));
        plot_acf(
            x=train,
            ax=ax,
            lags=24,
            title=f'Autocorrelation for {city_name}, {state_abbreviation}␣
↪{zip_code}'
        );
        return fig

    def pacf_plot(self, zip_code):
        """Given a zip code, plot the partial autocorrelation function for the␣
↪training data.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        city_name = self.get_city_name(zip_code)
        state_abbreviation = self.get_state_abbreviation(zip_code)
        train, test = self.get_train_test_split(zip_code)
        fig, ax = plt.subplots(figsize=(8,5));
        plot_pacf(
            x=train,
            ax=ax,
            lags=24,
            title=f'Partial Autocorrelation for {city_name},␣
↪{state_abbreviation} {zip_code}'
        );
        return fig

    # Model Selection
    def fit(self, zip_code, trace=True):
        """Given a zip code, automatically select optimal hyperparameters for a
        Seasonal AutoRegessive Integrated Moving Average model
        fit to the train data and evaluated in terms of Mean Squared
        Error on the test data. Return a fitted SARIMA model.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        try:
            best_model = self.model_dictionary[zip_code]['BestModel']
        except:
            y = self.get_time_series(zip_code)
            model = auto_arima(
```

```
            y = y,
            X=None,
            start_p=0,
            d=1,
            start_q=0,
            max_p=2,
            max_d=2,
            max_q=2,
            start_P=0,
            D=1,
            start_Q=0,
            max_P=2,
            max_D=2,
            max_Q=2,
            max_order=None,
            m=12,
            seasonal=True,
            stationary=False,
            information_criterion='oob',
            alpha=0.05,
            test='kpss',
            seasonal_test='OCSB',
            stepwise=True,
            suppress_warnings=True,
            error_action='warn',
            trace=trace,
            out_of_sample_size= self.test_size,
            scoring='mse'
        )
        order = model.order
        seasonal_order = model.seasonal_order
        trend = model.trend
        best_model = SARIMAX(
            endog=y,
            order=order,
            seasonal_order=seasonal_order,
            trend='c'
        ).fit()
        self.model_dictionary[zip_code]['BestModel'] = best_model
    return best_model

# Model Validation
def predict(self, zip_code):
    """Given a zip code, predict the median sale price for the time peroid
    spanned by the train and test data.

    Keyword Arguments:
```

```
        zip_code -- five digit zipcode as an integer.
        """
        try:
            prediction = self.model_dictionary[zip_code]['Prediction']
        except:
            model = self.fit(zip_code)
            prediction = model.get_prediction().summary_frame()
            self.model_dictionary[zip_code]['Prediction'] = prediction
        return prediction

    def plot_diagnostics(self, zip_code):
        """Given a zip code, return diagnostic plots for the
        corresponding fitted SARIMA model

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        best_model = self.fit(zip_code)
        city = self.get_city_name(zip_code)
        state = self.get_state_abbreviation(zip_code)
        fig = best_model.plot_diagnostics(figsize=(16, 10));
        fig.suptitle(f'Diagnostics for {city}, {state} {zip_code}', fontsize=16)
        return fig

    # Predict future prices
    def forecast(self, zip_code):
        """Given a zip code, forecast median sale price `self.forecast_size`
        months into the future.

        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        try:
            forecast = self.model_dictionary[zip_code]['Forecast']
        except:
            model = self.fit(zip_code)
            forecast = model.get_forecast(steps=self.forecast_size).
↪summary_frame()
            self.model_dictionary[zip_code]['Forecast'] = forecast
        return forecast

    # Compute ROI
    def compute_roi(self, zip_code):
        """Given a zip code, compute the expected retun on investment
        for a property puchased durring the last month of the provided
        timeseries and sold durring the last month of the forecast.
```

```
        Keyword Arguments:
        zip_code -- five digit zipcode as an integer.
        """
        try:
            roi = self.model_dictionary[zip_code]['ROI']
        except:
            initial_price = self.get_time_series(zip_code)['MedianSales'][-1]
            final_price = self.forecast(zip_code)['mean'][-1]
            roi = (final_price-initial_price)/initial_price
            self.model_dictionary[zip_code]['ROI'] = roi
        return roi

    # Return on investment dataframe
    def roi_df(self):
        """Return a dataframe containing the expected return on investment
        for every zip code in the provided data.
        """
        roi_dict = {
            'zip_code': [],
            'ROI': []
        }
        for zip_code in self.model_dictionary.keys():
            roi_dict['zip_code'].append(zip_code)
            roi_dict['ROI'].append(self.model_dictionary[zip_code]['ROI'])
        roi_df = pd.DataFrame(roi_dict)
        roi_df['zip_code'] = roi_df['zip_code'].astype(str)
        return roi_df

    def zip_code_map(self):
        """Return an interactive map of zip codes colorized to reflect
        expected return on investment.
        """
        geojason_url = 'https://raw.githubusercontent.com/OpenDataDE/
↪State-zip-code-GeoJSON/master/md_maryland_zip_codes_geo.min.json'
        zip_code_map = folium.Map(location=[39.29, -76.61], width=800,␣
↪height=600, zoom_start=12)
        folium.Choropleth(
            geo_data=geojason_url,
            name='choropleth',
            data=self.roi_df(),
            columns=['zip_code','ROI'],
            key_on='feature.properties.ZCTA5CE10',
            fill_color='RdPu',
            fill_opacity=0.7,
            nan_fill_opacity=0
        ).add_to(zip_code_map)
        return zip_code_map
```

3 Step 1: Import the Data

Our main data set is stored in the ZHVI.csv.gz spreadsheet, which was downloaded from Zillow's Zillow Home Value Index for single family homes data source. The data in this file was downloaded on 2020-12-10 and contains data up to 2020-10-31. Assuming that the file structure has not changed substantially, it should be possible to download an updated file and run this analysis with up-to-date data.

[2]:
```
df = pd.read_csv('../data/ZHVI.csv.gz', index_col='RegionID',␣
    ↪compression='gzip')
print(df.shape)
df.head()
```

```
(30205, 306)
```

[2]:
```
          SizeRank  RegionName RegionType StateName State      City  \
RegionID
61639            0       10025        Zip        NY    NY  New York
84654            1       60657        Zip        IL    IL   Chicago
61637            2       10023        Zip        NY    NY  New York
91982            3       77494        Zip        TX    TX      Katy
84616            4       60614        Zip        IL    IL   Chicago

                                   Metro       CountyName  1996-01-31  \
RegionID
61639        New York-Newark-Jersey City  New York County         NaN
84654           Chicago-Naperville-Elgin      Cook County    296113.0
61637        New York-Newark-Jersey City  New York County         NaN
91982     Houston-The Woodlands-Sugar Land    Harris County    203140.0
84616           Chicago-Naperville-Elgin      Cook County    462086.0

          1996-02-29  …  2020-01-31  2020-02-29  2020-03-31  2020-04-30  \
RegionID              …
61639            NaN  …    930560.0    932099.0    933253.0    930160.0
84654       295520.0  …    786707.0    787854.0    789482.0    790451.0
61637            NaN  …   1290836.0   1291613.0   1288723.0   1283261.0
91982       203391.0  …    340112.0    340320.0    340828.0    341998.0
84616       461720.0  …   1010879.0   1012589.0   1014209.0   1015467.0

          2020-05-31  2020-06-30  2020-07-31  2020-08-31  2020-09-30  \
RegionID
61639       926279.0    920531.0    919481.0    920766.0    927266.0
84654       790939.0    791300.0    793322.0    796143.0    801148.0
61637      1278518.0   1279537.0   1279105.0   1280177.0   1282240.0
91982       343077.0    343858.0    344397.0    345495.0    346575.0
84616      1015662.0   1017251.0   1020360.0   1023859.0   1029882.0

          2020-10-31
```

```
RegionID
61639       932302.0
84654       806603.0
61637      1289935.0
91982       348416.0
84616      1036427.0

[5 rows x 306 columns]
```

We restrict our attention to the city of Baltimore, MD. By changing the query below it should be possible to reproduce this analysis for any locality. Please note that fitting SARIMA models is fairly computationally expensive, so including a large set of zip codes may cause the `build_model` to run for quite a while.

[3]:
```python
query = "City == 'Baltimore' and State == 'MD'"
df = df.query(query)
print(df.shape)
df
```

```
(20, 306)
```

[3]:
```
          SizeRank  RegionName RegionType StateName State       City  \
RegionID
66825          368       21215        Zip        MD    MD  Baltimore
66834          484       21224        Zip        MD    MD  Baltimore
66828          744       21218        Zip        MD    MD  Baltimore
66816          783       21206        Zip        MD    MD  Baltimore
66839         1088       21229        Zip        MD    MD  Baltimore
66840         1422       21230        Zip        MD    MD  Baltimore
66827         2536       21217        Zip        MD    MD  Baltimore
66847         3029       21239        Zip        MD    MD  Baltimore
66822         3124       21212        Zip        MD    MD  Baltimore
66811         3631       21201        Zip        MD    MD  Baltimore
66823         4039       21213        Zip        MD    MD  Baltimore
66826         4062       21216        Zip        MD    MD  Baltimore
66812         4682       21202        Zip        MD    MD  Baltimore
66833         5322       21223        Zip        MD    MD  Baltimore
66821         5541       21211        Zip        MD    MD  Baltimore
66841         5664       21231        Zip        MD    MD  Baltimore
66824         6466       21214        Zip        MD    MD  Baltimore
66820         8018       21210        Zip        MD    MD  Baltimore
66815         8056       21205        Zip        MD    MD  Baltimore
66836        12112       21226        Zip        MD    MD  Baltimore

                              Metro      CountyName  1996-01-31  1996-02-29  \
RegionID
66825     Baltimore-Columbia-Towson  Baltimore City     80880.0     80666.0
```

```
66834     Baltimore-Columbia-Towson  Baltimore City     93965.0     93858.0
66828     Baltimore-Columbia-Towson  Baltimore City     71417.0     71853.0
66816     Baltimore-Columbia-Towson  Baltimore City     82607.0     82660.0
66839     Baltimore-Columbia-Towson  Baltimore City     82654.0     82794.0
66840     Baltimore-Columbia-Towson  Baltimore City     90409.0     90431.0
66827     Baltimore-Columbia-Towson  Baltimore City         NaN         NaN
66847     Baltimore-Columbia-Towson  Baltimore City     97281.0     97401.0
66822     Baltimore-Columbia-Towson  Baltimore City    114617.0    114726.0
66811     Baltimore-Columbia-Towson  Baltimore City    101226.0     99667.0
66823     Baltimore-Columbia-Towson  Baltimore City     51157.0     51227.0
66826     Baltimore-Columbia-Towson  Baltimore City     62429.0     62559.0
66812     Baltimore-Columbia-Towson  Baltimore City     81884.0     81205.0
66833     Baltimore-Columbia-Towson  Baltimore City     26187.0     26267.0
66821     Baltimore-Columbia-Towson  Baltimore City     65650.0     65671.0
66841     Baltimore-Columbia-Towson  Baltimore City     66776.0     67251.0
66824     Baltimore-Columbia-Towson  Baltimore City     81822.0     81702.0
66820     Baltimore-Columbia-Towson  Baltimore City    197608.0    197314.0
66815     Baltimore-Columbia-Towson  Baltimore City     35585.0     35559.0
66836     Baltimore-Columbia-Towson  Baltimore City    104467.0    104657.0

          …  2020-01-31  2020-02-29  2020-03-31  2020-04-30  2020-05-31  \
RegionID  …
66825     …    141847.0    142485.0    142719.0    143417.0    143771.0
66834     …    181401.0    181205.0    181259.0    181296.0    181779.0
66828     …    158316.0    158647.0    159052.0    159753.0    160528.0
66816     …    157113.0    156408.0    155997.0    155919.0    156004.0
66839     …    120521.0    121307.0    121899.0    122483.0    123296.0
66840     …    211650.0    211316.0    211311.0    211631.0    212497.0
66827     …    126041.0    126464.0    127252.0    128442.0    128978.0
66847     …    149645.0    149836.0    149860.0    150389.0    151544.0
66822     …    252601.0    252341.0    252421.0    253093.0    254204.0
66811     …    176576.0    176841.0    177293.0    177390.0    177386.0
66823     …     84482.0     85402.0     86312.0     87766.0     88895.0
66826     …     86731.0     87367.0     88091.0     88256.0     88652.0
66812     …    211922.0    211137.0    210184.0    209885.0    209624.0
66833     …     47111.0     46740.0     46548.0     46394.0     46521.0
66821     …    215312.0    215862.0    216822.0    217553.0    218417.0
66841     …    244017.0    242966.0    242407.0    242423.0    242648.0
66824     …    180667.0    180145.0    180318.0    180385.0    181606.0
66820     …    475208.0    471185.0    469858.0    469581.0    469449.0
66815     …     56937.0     55863.0     55658.0     55474.0     55885.0
66836     …    228578.0    229409.0    230206.0    230948.0    232205.0

          2020-06-30  2020-07-31  2020-08-31  2020-09-30  2020-10-31
RegionID
66825       143966.0    144283.0    145614.0    147702.0    150608.0
66834       182020.0    182601.0    183819.0    186429.0    189625.0
```

```
66828        160478.0   160820.0   161670.0   164124.0   166767.0
66816        155849.0   156458.0   158033.0   160752.0   163549.0
66839        124109.0   125143.0   126557.0   129136.0   131301.0
66840        213129.0   214564.0   216322.0   218907.0   221575.0
66827        129375.0   130015.0   131026.0   132595.0   133613.0
66847        152196.0   153590.0   155444.0   158207.0   161381.0
66822        255393.0   257217.0   259662.0   263268.0   267119.0
66811        177041.0   177095.0   177748.0   179115.0   181027.0
66823         90445.0    92077.0    93839.0    96604.0    99420.0
66826         89215.0    90572.0    92263.0    94713.0    97222.0
66812        209614.0   209878.0   210795.0   212054.0   213491.0
66833         46641.0    46773.0    47030.0    47755.0    48734.0
66821        218393.0   219505.0   221470.0   224780.0   228346.0
66841        242927.0   243823.0   245201.0   247382.0   249962.0
66824        181245.0   181556.0   183058.0   186746.0   191524.0
66820        468944.0   469592.0   473241.0   480013.0   487889.0
66815         56059.0    56236.0    56527.0    57176.0    58365.0
66836        233810.0   236026.0   238968.0   242245.0   245953.0

[20 rows x 306 columns]
```

4 Step 2: Instantiate the Model and Load in the Data

Below we instantiate a Model object and load our selected data into the object.

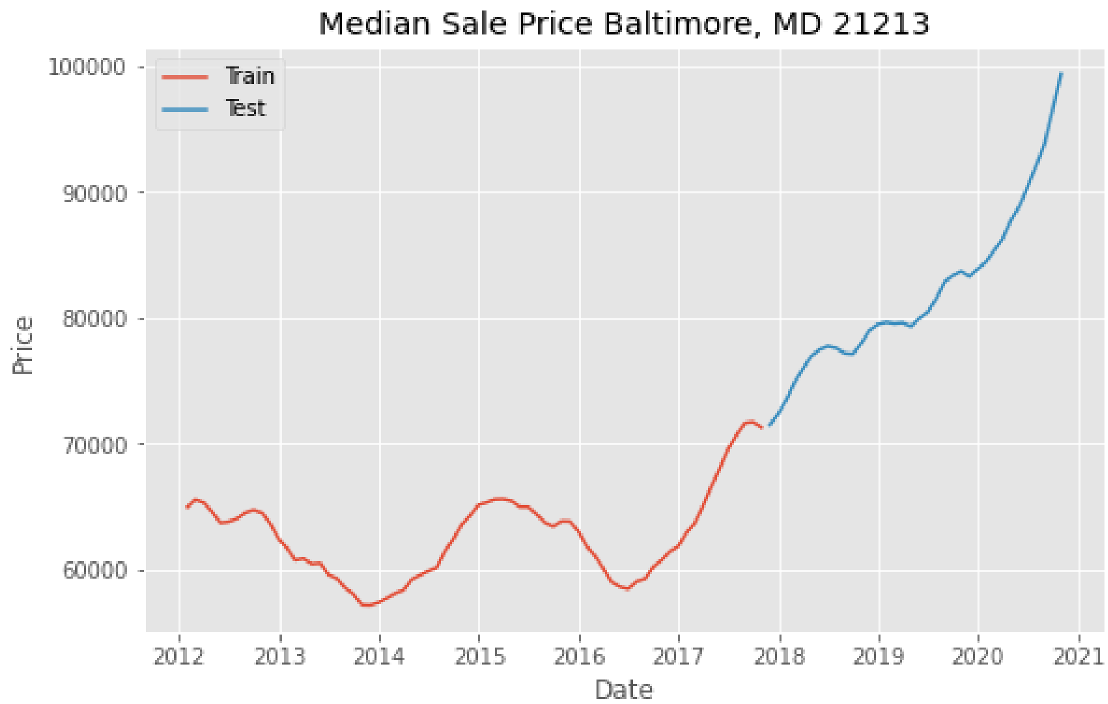
[4]:
```
model = ZipCodeROIModel()
model.load_data(df)
```

5 Step 3: EDA and Visualization

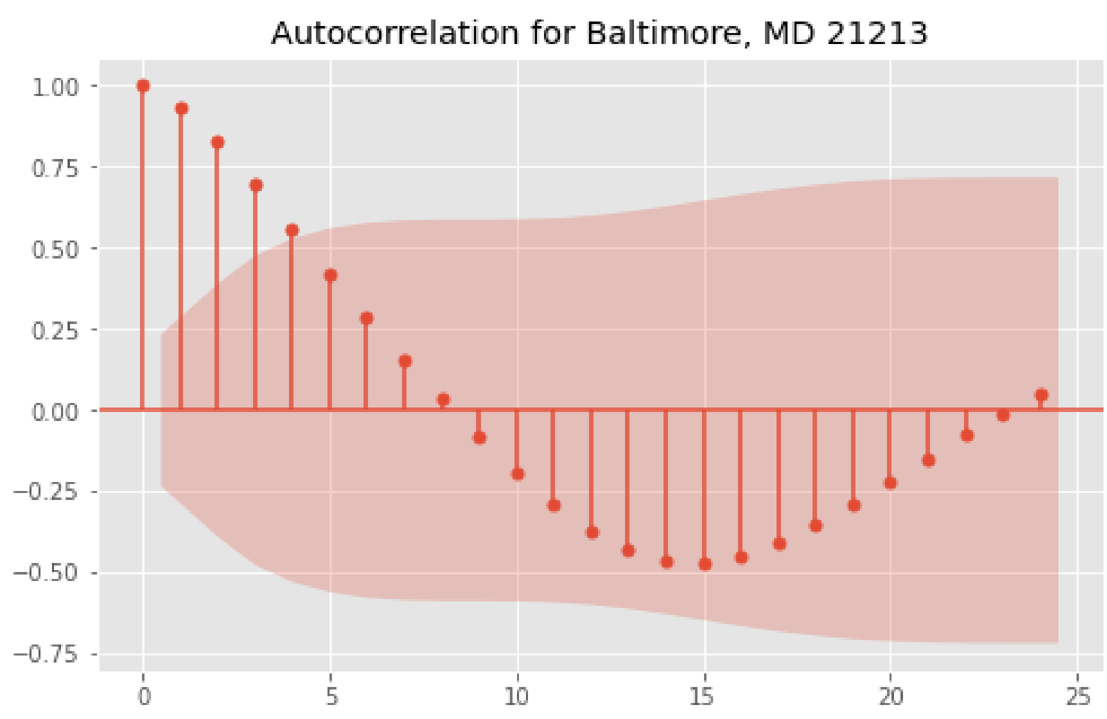
Below we display a time series plot, ACF plot, and PACF plot for the zip code 21213. The class methods used below will work for any zip code in our model.

[5]:
```
zip_code = 21213
```

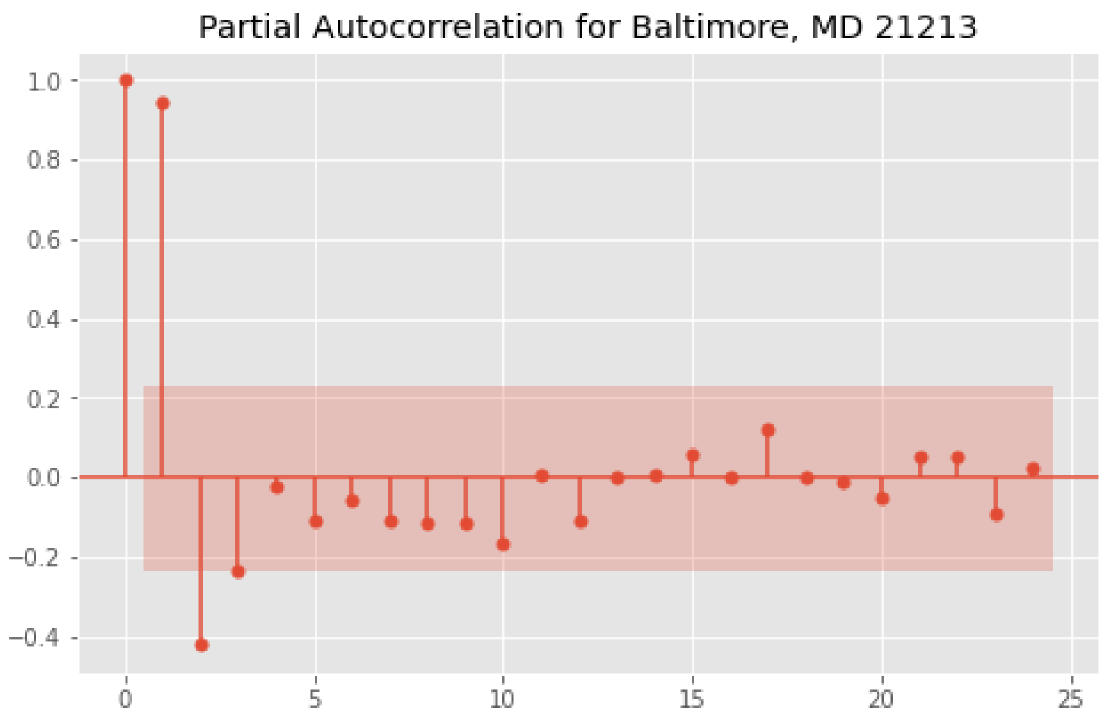
[6]:
```
model.time_series_plot(zip_code, show_prediction=False, show_forecast=False);
```



[7]:
```
model.acf_plot(zip_code);
```



[8]:
```
model.pacf_plot(zip_code);
```



6 Step 4: ARIMA Modeling

Below we select the best ARIMA model for our example zip code.

[9]:
```
best_model = model.fit(zip_code)
```

```
Performing stepwise search to minimize oob
 ARIMA(0,1,0)(0,1,0)[12]             : OOB=52772650.667, Time=0.07 sec
 ARIMA(1,1,0)(1,1,0)[12]             : OOB=29457988.554, Time=0.38 sec
 ARIMA(0,1,1)(0,1,1)[12]             : OOB=30594675.816, Time=0.18 sec
 ARIMA(1,1,0)(0,1,0)[12]             : OOB=48805054.475, Time=0.13 sec
 ARIMA(1,1,0)(2,1,0)[12]             : OOB=26204365.978, Time=0.82 sec
 ARIMA(1,1,0)(2,1,1)[12]             : OOB=inf, Time=1.51 sec
 ARIMA(1,1,0)(1,1,1)[12]             : OOB=27726141.200, Time=0.54 sec
 ARIMA(0,1,0)(2,1,0)[12]             : OOB=26743843.811, Time=0.55 sec
 ARIMA(2,1,0)(2,1,0)[12]             : OOB=25117471.555, Time=1.00 sec
 ARIMA(2,1,0)(1,1,0)[12]             : OOB=28101869.754, Time=0.45 sec
 ARIMA(2,1,0)(2,1,1)[12]             : OOB=inf, Time=1.83 sec
 ARIMA(2,1,0)(1,1,1)[12]             : OOB=28823342.167, Time=0.88 sec
 ARIMA(2,1,1)(2,1,0)[12]             : OOB=28515052.628, Time=2.12 sec
 ARIMA(1,1,1)(2,1,0)[12]             : OOB=24838444.006, Time=1.41 sec
 ARIMA(1,1,1)(1,1,0)[12]             : OOB=27755905.886, Time=0.57 sec
```

```
 ARIMA(1,1,1)(2,1,1)[12]             : OOB=inf, Time=2.17 sec
 ARIMA(1,1,1)(1,1,1)[12]             : OOB=25140406.146, Time=0.83 sec
 ARIMA(0,1,1)(2,1,0)[12]             : OOB=28352248.163, Time=0.51 sec
 ARIMA(1,1,2)(2,1,0)[12]             : OOB=28615767.254, Time=1.68 sec
 ARIMA(0,1,2)(2,1,0)[12]             : OOB=24219514.537, Time=0.91 sec
 ARIMA(0,1,2)(1,1,0)[12]             : OOB=27243828.001, Time=0.34 sec
 ARIMA(0,1,2)(2,1,1)[12]             : OOB=inf, Time=2.38 sec
 ARIMA(0,1,2)(1,1,1)[12]             : OOB=25650411.792, Time=0.74 sec
 ARIMA(0,1,2)(2,1,0)[12] intercept   : OOB=139125062.221, Time=0.96 sec

Best model:  ARIMA(0,1,2)(2,1,0)[12]
Total fit time: 22.996 seconds
```

7 Step 5: Model Validation

Below we make an in sample prediction using our best model, display the prediction with the actual data, and display diagnostic plots for the fit.

[10]:
```python
model.predict(zip_code)
```

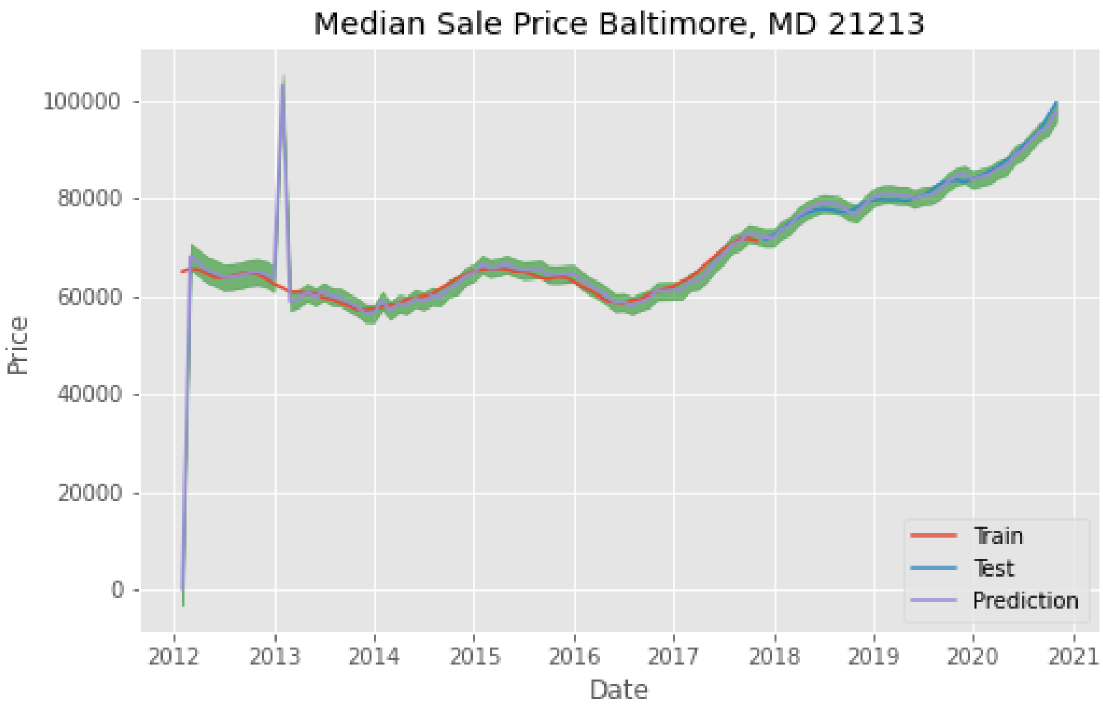
[10]:

MedianSales	mean	mean_se	mean_ci_lower	mean_ci_upper
2012-01-31	172.854134	1695.071486	-3149.424930	3495.133199
2012-02-29	68028.672487	1366.593043	65350.199342	70707.145632
2012-03-31	66707.522922	1365.264195	64031.654271	69383.391574
2012-04-30	65343.087801	1365.142998	62667.456691	68018.718911
2012-05-31	64684.116979	1365.140335	62008.491089	67359.742870
…	…	…	…	…
2020-06-30	89695.867256	918.968071	87894.722934	91497.011578
2020-07-31	91710.022754	918.968071	89908.878433	93511.167076
2020-08-31	93613.174860	918.968071	91812.030539	95414.319182
2020-09-30	94531.959974	918.968071	92730.815653	96333.104296
2020-10-31	97474.195227	918.968071	95673.050905	99275.339548

[106 rows x 4 columns]

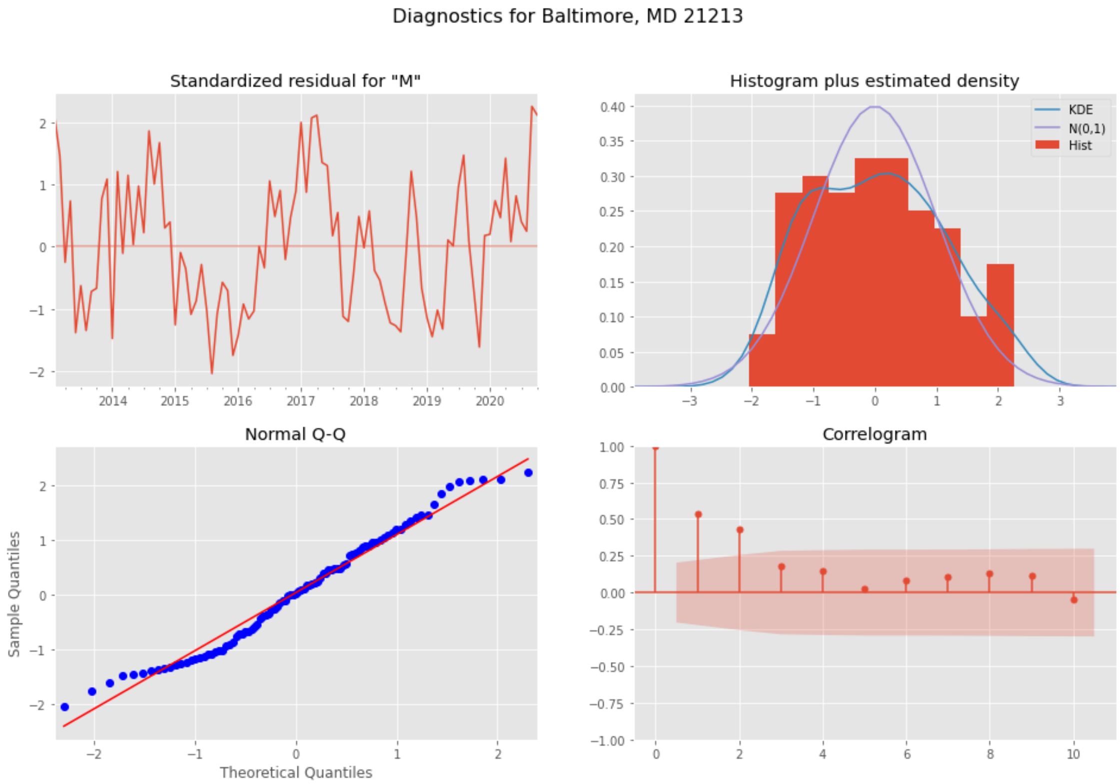
We note that both the initial predicted value and the value predicted approximately one year after the time series begins are clearly incorrect. This is not particularly concerning since these errors occur early in the training period and are most likely an artifact of the seasonal component converging.

[11]:
```python
fig = model.time_series_plot(zip_code, show_forecast=False);
```



The diagnostics below indicate that residuals are reasonably normal, but there may be some undescribed autocorrelation structure in the data. Our model might be slightly under-fit and might benefit from including higher order autoregressive or moving average components. We have restricted to second order terms in both cases due to computational limitations.

[12]:
```
model.plot_diagnostics(zip_code);
```



8 Step 6: Making a Forecast

Below we forecast into the future, make a final time series plot which includes our forecast, and compute our expected return on investment for the example zip code.

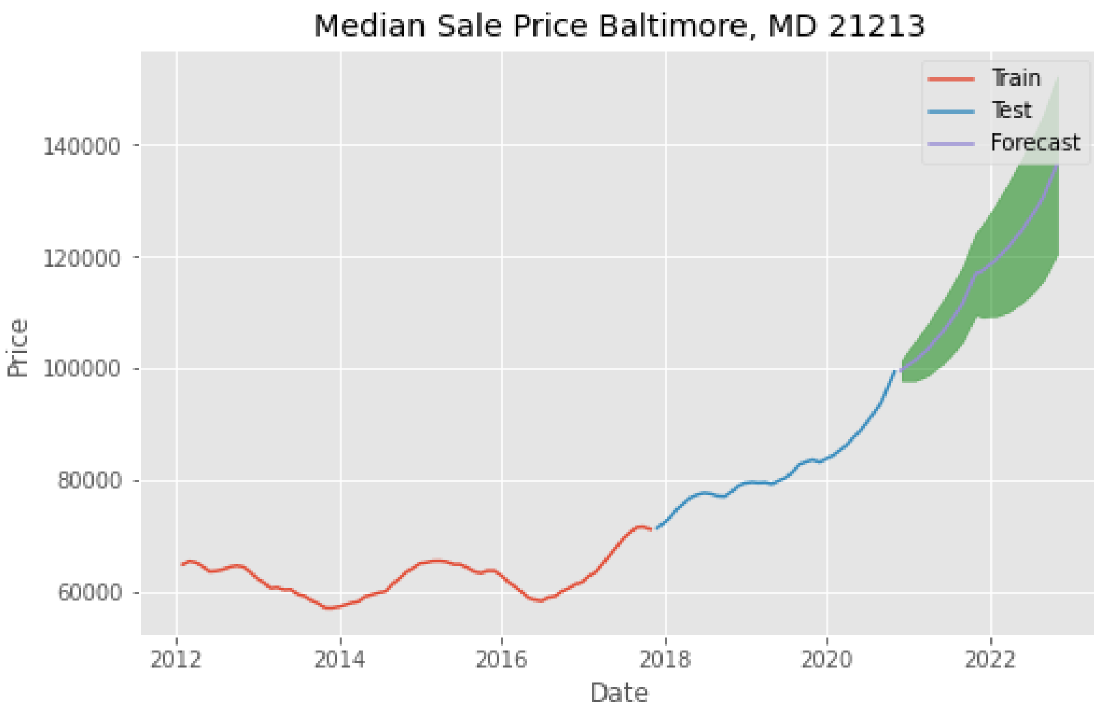
[13]:
```
model.forecast(zip_code)
```

[13]:

MedianSales	mean	mean_se	mean_ci_lower	mean_ci_upper
2020-11-30	99701.279969	918.968071	97900.135647	101502.424291
2020-12-31	100629.677124	1394.656229	97896.201145	103363.153103
2021-01-31	101383.191417	1778.012523	97898.350907	104868.031927
2021-02-28	102445.222234	2092.269359	98344.449644	106545.994823
2021-03-31	103502.674913	2365.132894	98867.099622	108138.250205
2021-04-30	105022.431038	2609.619924	99907.669975	110137.192102
2021-05-31	106294.611182	2833.086423	100741.863827	111847.358537
2021-06-30	107929.253101	3040.171248	101970.626948	113887.879255
2021-07-31	109658.074655	3234.022844	103319.506355	115996.642955
2021-08-31	111517.759486	3416.894247	104820.769823	118214.749148
2021-09-30	114235.485666	3590.463596	107198.306330	121272.665001
2021-10-31	117021.677582	3756.020683	109660.012319	124383.342846
2021-11-30	117477.136762	4223.341844	109199.538853	125754.734670
2021-12-31	118566.335552	4693.841940	109366.574401	127766.096703

2022-01-31	119483.220829	5140.869200	109407.302348	129559.139310
2022-02-28	120692.531823	5552.019462	109810.773636	131574.290009
2022-03-31	121902.920389	5934.753919	110271.016450	133534.824327
2022-04-30	123553.495279	6294.258340	111216.975623	135890.014935
2022-05-31	124990.512209	6634.310215	111987.503126	137993.521292
2022-06-30	126777.299548	6957.762284	113140.336058	140414.263038
2022-07-31	128672.257147	7266.831495	114429.529136	142914.985159
2022-08-31	130705.020648	7563.281295	115881.261705	145528.779592
2022-09-30	133547.126617	7848.541770	118164.267416	148929.985819
2022-10-31	136449.848886	8123.791719	120527.509698	152372.188073

[14]:
```
model.time_series_plot(zip_code, show_prediction=False);
```



The plot above shows our forecast for median sale price two years into the future. We also include a 95% confidence interval for our prediction.

[15]:
```
model.compute_roi(zip_code)
```

[15]: 0.37245874960678477

The expected return on investment for our example zip code is 30%.

9 Step 7: Building the Model

Having tested all of the steps of our zip code level analysis for the example zip code above, we build the full city-wide model by repeating all of the steps above for each zip code in the city. Iterating over all of the zip codes is handled by the `build_model` method below. The try except block below simply avoids issues that arise during a demonstration of this notebook when cells are run out of order.

[16]:
```
try:
    model.build_model()
except:
    model = ZipCodeROIModel()
    model.build_model()
```

10 Step 8: Picking Top Five Zip Codes

Having forecast prices for every zip code in the city, we extract a dataframe from the model containing expected ROI for each zip code.

[17]:
```
roi_df = model.roi_df()
```

Below we list the top five zip codes ranked by expected ROI.

[18]:
```
top_five = roi_df.sort_values(by='ROI', ascending=False).iloc[:5]
top_five
```

[18]:

	zip_code	ROI
10	21213	0.302163
4	21229	0.195755
11	21216	0.190483
7	21239	0.159679
19	21226	0.135404

11 Step 9: Map Zip Codes

The map below shows the zip codes that we included in our analysis colorized to indicate ROI. Note that the centering of this map is hard coded and will not automatically adapt to new localities.

[19]:
```
model.zip_code_map()
```

[19]:
```
<folium.folium.Map at 0x7f2ff2a592b0>
```

12 Conclusion

- We have successfully identified 21213, 21229, 21216, 21239, and 21226 as the best five zip codes for our client's needs.

- By developing our analysis at the zip code level and wrapping our city-wide model in a class with high level methods, we have made the complex structure of our model accessible and easy to work with, keeping most of the complexity encapsulated within the class.
- Our approach is portable and scalable. The model can be applied directly to any locality where Zillow data is available, and the only limiting factor for the number of zip codes included is computational resources.

13 Future Work

- Optimize and parallelize code for a nation-wide search.
- Try Long Short-Term Memory neural networks in place of SARIMA models.
- Incorporate exogenous variables.
- Model a more realistic investment strategy.
  - House flipping
  - Rental property investment
- Build project dashboard.