Bitcoin Time Series Forecasting

In [4]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from plotly import tools
import plotly.plotly as py
from plotly.offline import init_notebook_mode, iplot
init_notebook_mode(connected=True)
import plotly.graph_objs as go
import gc

import matplotlib.pyplot as plt
import seaborn as sns
```

Data Exploration

In this section, we will explore the data, specifically the historic Bitcoin prices, and attempt to uncover some insights. We will be using the Coinbase dataset, as it is one of the most widely used Bitcoin exchange/wallet platforms in the world.

In [7]:
```
import datetime, pytz
#define a conversion function for the native timestamps in the csv file
def dateparse (time_in_secs):
    return pytz.utc.localize(datetime.datetime.fromtimestamp(float(time_in_secs)))


data = pd.read_csv('../input/bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv', parse_dates=[0], date_parser=dateparse)
```

In [8]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4857377 entries, 0 to 4857376
Data columns (total 8 columns):
 #   Column             Dtype
---  ------             -----
 0   Timestamp          datetime64[ns, UTC]
 1   Open               float64
 2   High               float64
 3   Low                float64
 4   Close              float64
 5   Volume_(BTC)       float64
 6   Volume_(Currency)  float64
 7   Weighted_Price     float64
dtypes: datetime64[ns, UTC](1), float64(7)
memory usage: 296.5 MB
```

In [9]:
```
data.head()
```

Out[9]:

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
0	2011-12-31 20:52:00+00:00	4.39	4.39	4.39	4.39	0.455581	2.0	4.39
1	2011-12-31 20:53:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	2011-12-31 20:54:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	2011-12-31 20:55:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	2011-12-31 20:56:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Little preprocessing required, replacing the NaN values with zeroes and previous data.

In [10]:
```
# First thing is to fix the data for bars/candles where there are no trades.
# Volume/trades are a single event so fill na's with zeroes for relevant fields.
data['Volume_(BTC)'].fillna(value=0, inplace=True)
data['Volume_(Currency)'].fillna(value=0, inplace=True)
data['Weighted_Price'].fillna(value=0, inplace=True)

# next we need to fix the OHLC (open high low close) data which is a continuous timeseries so
# lets fill forwards those values...
data['Open'].fillna(method='ffill', inplace=True)
data['High'].fillna(method='ffill', inplace=True)
data['Low'].fillna(method='ffill', inplace=True)
data['Close'].fillna(method='ffill', inplace=True)

data.head()
```

Out[10]:

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
0	2011-12-31 20:52:00+00:00	4.39	4.39	4.39	4.39	0.455581	2.0	4.39
1	2011-12-31 20:53:00+00:00	4.39	4.39	4.39	4.39	0.000000	0.0	0.00
2	2011-12-31 20:54:00+00:00	4.39	4.39	4.39	4.39	0.000000	0.0	0.00
3	2011-12-31 20:55:00+00:00	4.39	4.39	4.39	4.39	0.000000	0.0	0.00
4	2011-12-31 20:56:00+00:00	4.39	4.39	4.39	4.39	0.000000	0.0	0.00

Creating Weekly Rows for the Data Visualization

In [11]:
```
# create valid date range
start = datetime.datetime(2015, 1, 1, 0, 0, 0, 0, pytz.UTC)
end = datetime.datetime(2018, 11, 11, 0, 0, 0, 0, pytz.UTC)

# find rows between start and end time and find the first row (00:00 monday morning)
weekly_rows = data[(data['Timestamp'] >= start) & (data['Timestamp'] <= end)].groupby([pd.Grouper(key='Timestamp', freq='W-MON')]).first().reset_index()
weekly_rows.head()
```

Out[11]:

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
0	2015-01-05 00:00:00+00:00	314.22	314.22	313.91	313.91	1.910619	600.006381	314.037624
1	2015-01-12 00:00:00+00:00	272.45	305.00	272.45	291.10	175.837078	50502.066285	287.209426
2	2015-01-19 00:00:00+00:00	266.64	266.89	266.62	266.89	8.220587	2193.240935	266.798567
3	2015-01-26 00:00:00+00:00	210.19	211.15	210.19	211.15	5.074056	1066.806139	210.247200
4	2015-02-02 00:00:00+00:00	302.74	302.74	302.40	302.69	13.645647	4129.409860	302.617365

Lets visualize Historical Bitcoin Prices (2015-2018)

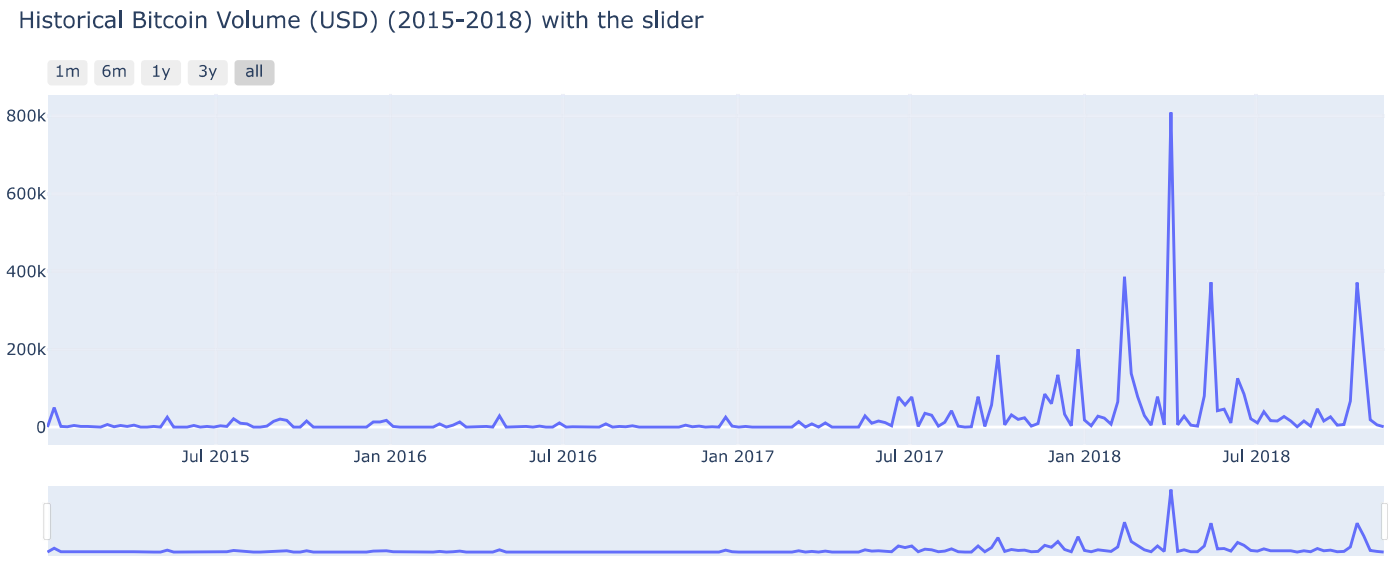
In [13]:
```
# We use Plotly to create the plots
trace1 = go.Scatter(
    x = weekly_rows['Timestamp'],
    y = weekly_rows['Open'].astype(float),
    mode = 'lines',
    name = 'Open'
)

trace2 = go.Scatter(
    x = weekly_rows['Timestamp'],
    y = weekly_rows['Close'].astype(float),
    mode = 'lines',
    name = 'Close'
```

```
)
trace3 = go.Scatter(
    x = weekly_rows['Timestamp'],
    y = weekly_rows['Weighted_Price'].astype(float),
    mode = 'lines',
    name = 'Weighted Avg'
)

layout = dict(
    title='Historical Bitcoin Prices (2015-2018) with the Slider ',
    xaxis=dict(
        rangeselector=dict(
            buttons=list([
                #change the count to desired amount of months.
                dict(count=1,
                     label='1m',
                     step='month',
                     stepmode='backward'),
                dict(count=6,
                     label='6m',
                     step='month',
                     stepmode='backward'),
                dict(count=12,
                     label='1y',
                     step='month',
                     stepmode='backward'),
                dict(count=36,
                     label='3y',
                     step='month',
                     stepmode='backward'),
                dict(step='all')
            ])
        ),
        rangeslider=dict(
            visible = True
        ),
        type='date'
    )
)

data = [trace1,trace2, trace3]
fig = dict(data=data, layout=layout)
iplot(fig, filename = "Time Series with Rangeslider")
```

Lets visualize Historical Bitcoin Market Volume (2015-2018)

In [14]:
```
trace1 = go.Scatter(
    x = weekly_rows['Timestamp'],
    y = weekly_rows['Volume_(Currency)'].astype(float),
    mode = 'lines',
    name = 'Bitcoin Price (Open)'
)

layout = dict(
    title='Historical Bitcoin Volume (USD) (2015-2018) with the slider',
    xaxis=dict(
        rangeselector=dict(
            buttons=list([
                dict(count=1,
                     label='1m',
                     step='month',
                     stepmode='backward'),
                dict(count=6,
                     label='6m',
                     step='month',
                     stepmode='backward'),
                dict(count=12,
                     label='1y',
                     step='month',
                     stepmode='backward'),
                dict(count=36,
                     label='3y',
                     step='month',
                     stepmode='backward'),
                dict(step='all')
            ])
        ),
        rangeslider=dict(
```

```
            visible = True
        ),
        type='date'
    )
)

data = [trace1]
fig = dict(data=data, layout=layout)
iplot(fig, filename = "Time Series with Rangeslider")
```



In [15]:

```
#BTC Volume vs USD visualization
trace = go.Scattergl(
    y = weekly_rows['Volume_(BTC)'].astype(float),
    x = weekly_rows['Weighted_Price'].astype(float),
    mode = 'markers',
    marker = dict(
        color = '#FFBAD2',
        line = dict(width = 1)
    )
)
layout = go.Layout(
    title='BTC Volume v/s USD',
    xaxis=dict(
        title='Weighted Price',
        titlefont=dict(
            family='Courier New, monospace',
            size=18,
            color='#7f7f7f'
        )
    ),
    yaxis=dict(
        title='Volume BTC',
        titlefont=dict(
            family='Courier New, monospace',
```

```
            size=18,
            color='#7f7f7f'
        )))
data = [trace]
fig = go.Figure(data=data, layout=layout)
iplot(fig, filename='compare_webgl')
```

Time Series Forecasting

Time series data is experimental data observed at different points in time, usually evenly spaced, such as daily, hourly, or even minute intervals. For instance, the daily data of airline ticket sales constitutes a time series. However, having a time element in a series of events does not automatically classify it as a time series. For example, the dates of major airline disasters are randomly spaced and do not fall into the time series category. Such random processes are referred to as point processes.

Time series data typically exhibits several key features, including trends, seasonality, and noise. Forecasting involves making predictions about the future based on past and current data.

In this kernel, we endeavor to perform time series analysis on historical Bitcoin price data. As demonstrated in the Data Exploration section, Bitcoin prices have shown significant volatility and inconsistency over the years. Analyzing time series data with such characteristics can be challenging. Nevertheless, we aim to explore various time series forecasting models.

- Time Series forecasting with LSTM
- Time Series forecasting with XGBoost
- Time Series forecasting with Facebook Prophet
- Time Series forecasting with ARIMA

Predicting using LSTM

In the first section, we utilize LSTM (Long Short-Term Memory) units. LSTM units are integral components of a recurrent neural network (RNN). An RNN constructed with LSTM units is commonly referred to as an LSTM network or simply LSTM. A standard LSTM unit consists of a cell, an input gate, an output gate, and a forget gate. The cell has the ability to remember values over arbitrary time intervals, while the three gates control the information flow into and out of the cell. You can find more information about LSTMs in this article.

Although LSTM may not be the ideal choice for forecasting in a turbulent market like Bitcoin, we are taking a chance here.

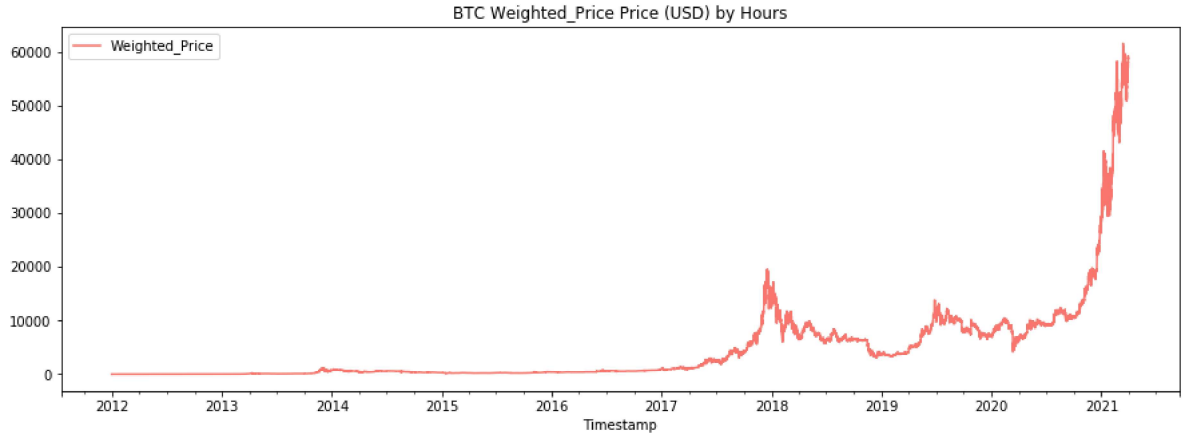
In [13]:
```
#Load the dataset
data = pd.read_csv('../input/bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv',parse_dates=[0], date_parser=dateparse)
data['Timestamp'] = data['Timestamp'].dt.tz_localize(None)
data = data.groupby([pd.Grouper(key='Timestamp', freq='H')]).first().reset_index()
data = data.set_index('Timestamp')
data = data[['Weighted_Price']]
data['Weighted_Price'].fillna(method='ffill', inplace=True)
```

We used '25-Jun-2018' as the split date to divide the data into training and test sets. Notably, during the June-July 2018 period, Bitcoin prices experienced a significant dip. The historical price data shows that the market began to recover from this date after reaching its lowest point, even though the price dropped to as low as 5,972 on June 29th, 2018. Following the historic peak of 20,000 on December 18th, there were multiple corrections in the market, leading to several price dips.

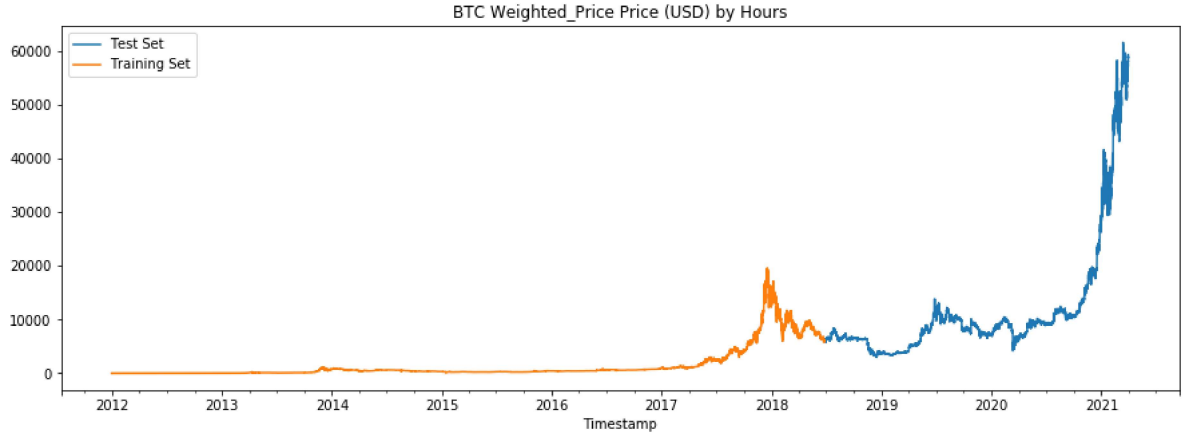
In [14]:
```
# split data
split_date = '25-Jun-2018'
data_train = data.loc[data.index <= split_date].copy()
data_test = data.loc[data.index > split_date].copy()
```

In [15]:
```
# Data preprocess
training_set = data_train.values
training_set = np.reshape(training_set, (len(training_set), 1))
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler()
training_set = sc.fit_transform(training_set)
X_train = training_set[0:len(training_set)-1]
y_train = training_set[1:len(training_set)]
X_train = np.reshape(X_train, (len(X_train), 1, 1))
```

In [16]:
```
color_pal = ["#F8766D", "#D39200", "#93AA00", "#00BA38", "#00C19F", "#00B9E3", "#619CFF", "#DB72FB"]
_ = data.plot(style='', figsize=(15,5), color=color_pal[0], title='BTC Weighted_Price Price (USD) by Hours')
```



In [17]:
```
_ = data_test \
    .rename(columns={'Weighted_Price': 'Test Set'}) \
    .join(data_train.rename(columns={'Weighted_Price': 'Training Set'}), how='outer') \
    .plot(figsize=(15,5), title='BTC Weighted_Price Price (USD) by Hours', style='')
```



We will use a Vanilla LSTM here for forecasting. The model is trained on pre 25-Jun-2018 data.

In [18]:
```
# Importing the Keras libraries and packages
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers import Activation


model = Sequential()
model.add(LSTM(128,activation="sigmoid",input_shape=(1,1)))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(X_train, y_train, epochs=100, batch_size=50, verbose=2)
```

```
Using TensorFlow backend.
```

```
Epoch 1/100
 - 8s - loss: 0.0289
Epoch 2/100
 - 4s - loss: 0.0069
Epoch 3/100
 - 4s - loss: 0.0035
Epoch 4/100
 - 4s - loss: 0.0023
Epoch 5/100
 - 4s - loss: 0.0017
Epoch 6/100
 - 4s - loss: 0.0013
Epoch 7/100
 - 4s - loss: 0.0010
Epoch 8/100
 - 4s - loss: 7.9753e-04
Epoch 9/100
 - 4s - loss: 6.5614e-04
Epoch 10/100
 - 4s - loss: 5.4542e-04
Epoch 11/100
 - 4s - loss: 4.6057e-04
Epoch 12/100
 - 4s - loss: 4.0367e-04
Epoch 13/100
 - 4s - loss: 3.5480e-04
Epoch 14/100
 - 4s - loss: 3.0707e-04
Epoch 15/100
 - 4s - loss: 2.7136e-04
Epoch 16/100
 - 4s - loss: 2.5064e-04
Epoch 17/100
 - 4s - loss: 2.1998e-04
Epoch 18/100
 - 4s - loss: 2.0819e-04
Epoch 19/100
 - 4s - loss: 1.8796e-04
Epoch 20/100
 - 4s - loss: 1.7629e-04
Epoch 21/100
 - 4s - loss: 1.6541e-04
Epoch 22/100
 - 4s - loss: 1.5671e-04
Epoch 23/100
 - 4s - loss: 1.4732e-04
Epoch 24/100
 - 4s - loss: 1.3940e-04
Epoch 25/100
 - 4s - loss: 1.4129e-04
Epoch 26/100
 - 4s - loss: 1.3101e-04
Epoch 27/100
 - 4s - loss: 1.3210e-04
Epoch 28/100
 - 4s - loss: 1.3066e-04
Epoch 29/100
 - 4s - loss: 1.2625e-04
Epoch 30/100
 - 4s - loss: 1.2926e-04
Epoch 31/100
 - 4s - loss: 1.2711e-04
Epoch 32/100
 - 4s - loss: 1.2534e-04
Epoch 33/100
 - 4s - loss: 1.2559e-04
Epoch 34/100
 - 4s - loss: 1.3219e-04
```

```
Epoch 35/100
 - 4s - loss: 1.2666e-04
Epoch 36/100
 - 4s - loss: 1.2864e-04
Epoch 37/100
 - 4s - loss: 1.2525e-04
Epoch 38/100
 - 4s - loss: 1.2289e-04
Epoch 39/100
 - 4s - loss: 1.2865e-04
Epoch 40/100
 - 4s - loss: 1.2183e-04
Epoch 41/100
 - 4s - loss: 1.2689e-04
Epoch 42/100
 - 4s - loss: 1.2119e-04
Epoch 43/100
 - 4s - loss: 1.2566e-04
Epoch 44/100
 - 4s - loss: 1.2313e-04
Epoch 45/100
 - 4s - loss: 1.2398e-04
Epoch 46/100
 - 4s - loss: 1.1904e-04
Epoch 47/100
 - 4s - loss: 1.2255e-04
Epoch 48/100
 - 4s - loss: 1.1879e-04
Epoch 49/100
 - 4s - loss: 1.2567e-04
Epoch 50/100
 - 4s - loss: 1.2059e-04
Epoch 51/100
 - 4s - loss: 1.2014e-04
Epoch 52/100
 - 4s - loss: 1.2141e-04
Epoch 53/100
 - 4s - loss: 1.2044e-04
Epoch 54/100
 - 4s - loss: 1.2011e-04
Epoch 55/100
 - 4s - loss: 1.1597e-04
Epoch 56/100
 - 4s - loss: 1.1775e-04
Epoch 57/100
 - 4s - loss: 1.2235e-04
Epoch 58/100
 - 4s - loss: 1.2709e-04
Epoch 59/100
 - 4s - loss: 1.2522e-04
Epoch 60/100
 - 4s - loss: 1.1926e-04
Epoch 61/100
 - 4s - loss: 1.1470e-04
Epoch 62/100
 - 4s - loss: 1.1514e-04
Epoch 63/100
 - 4s - loss: 1.1885e-04
Epoch 64/100
 - 4s - loss: 1.1579e-04
Epoch 65/100
 - 4s - loss: 1.1945e-04
Epoch 66/100
 - 4s - loss: 1.2136e-04
Epoch 67/100
 - 4s - loss: 1.1993e-04
Epoch 68/100
 - 4s - loss: 1.1828e-04
```

```
Epoch 69/100
 - 4s - loss: 1.1587e-04
Epoch 70/100
 - 4s - loss: 1.1528e-04
Epoch 71/100
 - 4s - loss: 1.1923e-04
Epoch 72/100
 - 4s - loss: 1.2082e-04
Epoch 73/100
 - 4s - loss: 1.2069e-04
Epoch 74/100
 - 4s - loss: 1.1871e-04
Epoch 75/100
 - 4s - loss: 1.1979e-04
Epoch 76/100
 - 4s - loss: 1.1856e-04
Epoch 77/100
 - 4s - loss: 1.1640e-04
Epoch 78/100
 - 4s - loss: 1.1871e-04
Epoch 79/100
 - 4s - loss: 1.1567e-04
Epoch 80/100
 - 4s - loss: 1.1802e-04
Epoch 81/100
 - 4s - loss: 1.1654e-04
Epoch 82/100
 - 4s - loss: 1.1906e-04
Epoch 83/100
 - 4s - loss: 1.1917e-04
Epoch 84/100
 - 4s - loss: 1.2136e-04
Epoch 85/100
 - 4s - loss: 1.1452e-04
Epoch 86/100
 - 4s - loss: 1.1894e-04
Epoch 87/100
 - 4s - loss: 1.2347e-04
Epoch 88/100
 - 4s - loss: 1.2176e-04
Epoch 89/100
 - 4s - loss: 1.1720e-04
Epoch 90/100
 - 4s - loss: 1.1812e-04
Epoch 91/100
 - 4s - loss: 1.1679e-04
Epoch 92/100
 - 4s - loss: 1.2010e-04
Epoch 93/100
 - 4s - loss: 1.1646e-04
Epoch 94/100
 - 4s - loss: 1.1656e-04
Epoch 95/100
 - 4s - loss: 1.1292e-04
Epoch 96/100
 - 4s - loss: 1.1662e-04
Epoch 97/100
 - 4s - loss: 1.1826e-04
Epoch 98/100
 - 4s - loss: 1.1364e-04
Epoch 99/100
 - 4s - loss: 1.1487e-04
Epoch 100/100
 - 4s - loss: 1.1589e-04
```

Out[18]:
```
<keras.callbacks.History at 0x788cbe493d68>
```

In [19]:
```
model.summary()
```

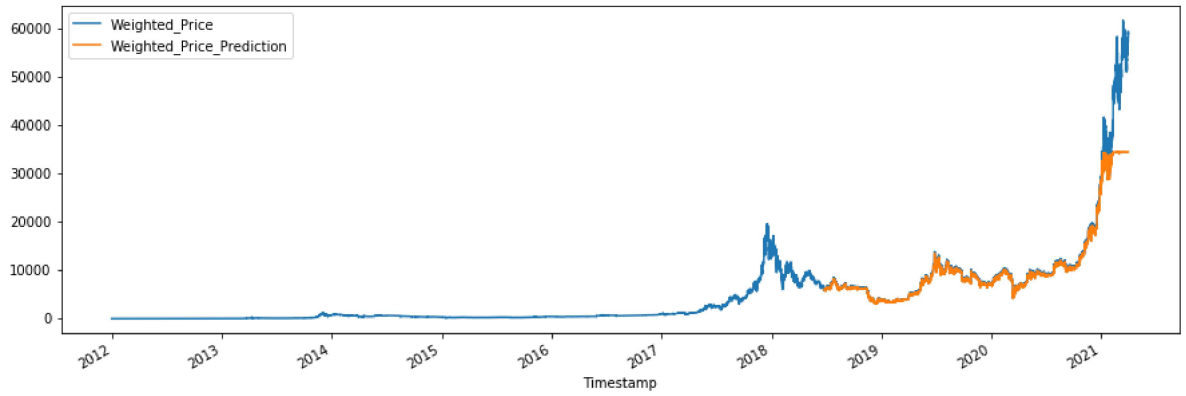
```
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 128)               66560
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 66,689
Trainable params: 66,689
Non-trainable params: 0
_________________________________________________________________
```

In [20]:
```
# Making the predictions
test_set = data_test.values
inputs = np.reshape(test_set, (len(test_set), 1))
inputs = sc.transform(inputs)
inputs = np.reshape(inputs, (len(inputs), 1, 1))
predicted_BTC_price = model.predict(inputs)
predicted_BTC_price = sc.inverse_transform(predicted_BTC_price)
```

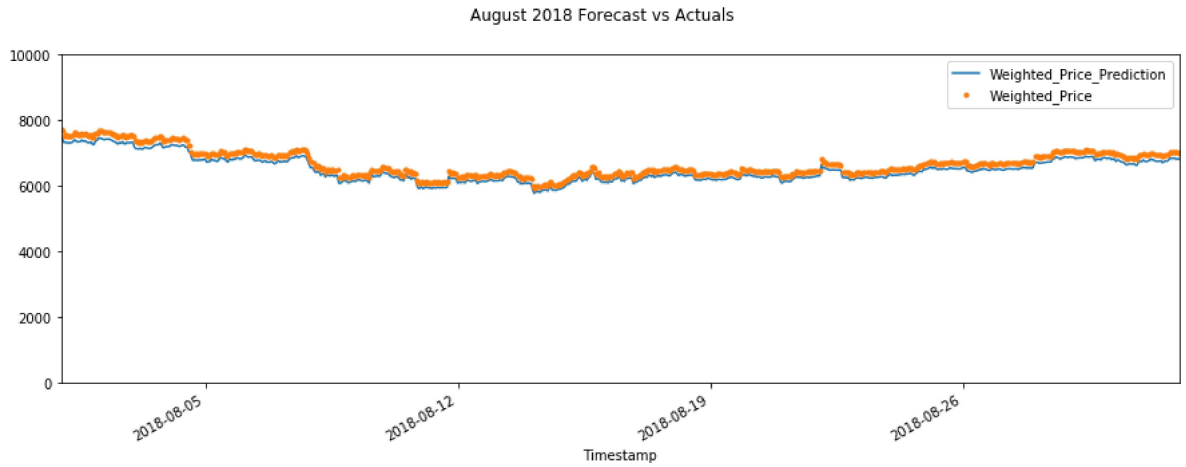
In [21]:
```
data_test['Weighted_Price_Prediction'] = predicted_BTC_price
data_all = pd.concat([data_test, data_train], sort=False)
```

In [22]:
```
#saving the predicted values in a common data frame for future comparision
final_data = data_all
final_data = final_data.reset_index()
final_data = final_data.rename(columns={'Weighted_Price_Prediction': 'lstm'})
final_data = final_data[['Timestamp','Weighted_Price','lstm']]
```

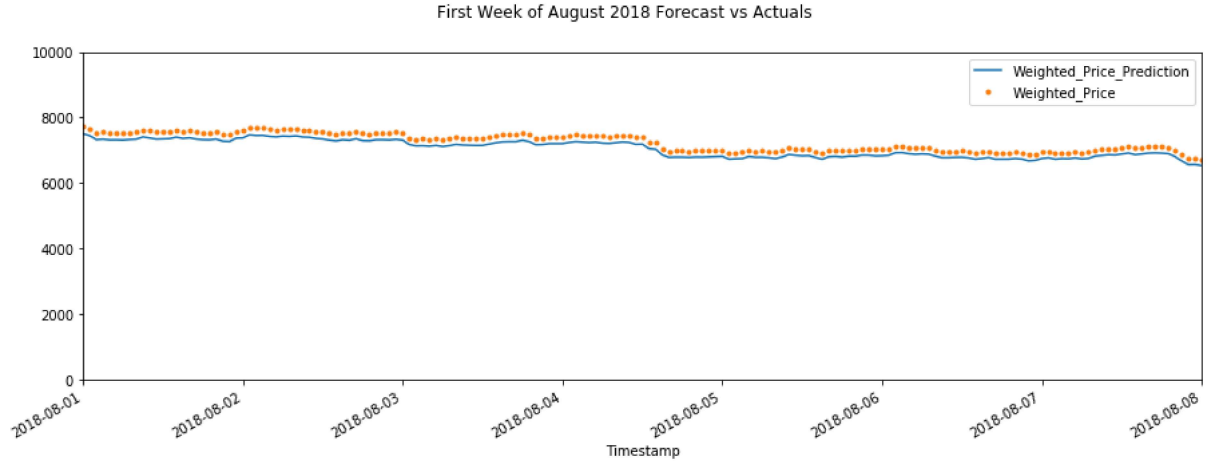
In [23]:
```
_ = data_all[['Weighted_Price','Weighted_Price_Prediction']].plot(figsize=(15, 5))
```



In [24]:
```
# Plot the forecast with the actuals
f, ax = plt.subplots(1)
f.set_figheight(5)
f.set_figwidth(15)
_ = data_all[['Weighted_Price_Prediction','Weighted_Price']].plot(ax=ax,
                                                  style=['-','.'])
ax.set_xbound(lower='08-01-2018', upper='09-01-2018')
ax.set_ylim(0, 10000)
plot = plt.suptitle('August 2018 Forecast vs Actuals')
```



In [25]:
```
# Plot the forecast with the actuals
f, ax = plt.subplots(1)
f.set_figheight(5)
f.set_figwidth(15)
_ = data_all[['Weighted_Price_Prediction','Weighted_Price']].plot(ax=ax,
                                          style=['-','.'])
ax.set_xbound(lower='08-01-2018', upper='08-08-2018')
ax.set_ylim(0, 10000)
plot = plt.suptitle('First Week of August 2018 Forecast vs Actuals')
```



In [26]:
```
#calculate MSE and MAE
from sklearn.metrics import mean_squared_error, mean_absolute_error
mean_squared_error(y_true=data_test['Weighted_Price'],
                   y_pred=data_test['Weighted_Price_Prediction'])
```

Out[26]:
```
17754215.834390406
```

In [27]:
```
mean_absolute_error(y_true=data_test['Weighted_Price'],
                   y_pred=data_test['Weighted_Price_Prediction'])
```

Out[27]:
```
1221.4940110684397
```

We can clearly see the entire model is over-fitted.

Time Series forecasting with XGBoost

XGBoost is an implementation of gradient boosted decision trees known for its speed and high performance. It's a powerful and versatile tool. Now, let's assess how effectively XGBoost can predict future values in a time-series dataset, such as Bitcoin prices.

In [28]:
```
import seaborn as sns
import matplotlib.pyplot as plt
from fbprophet import Prophet
from sklearn.metrics import mean_squared_error, mean_absolute_error
plt.style.use('fivethirtyeight')
```

In [30]:
```
data = pd.read_csv('../input/bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv',parse_dates=[0], date_parser=dateparse)
data['Timestamp'] = data['Timestamp'].dt.tz_localize(None)
data = data.groupby([pd.Grouper(key='Timestamp', freq='H')]).first().reset_index()
data = data.set_index('Timestamp')
data = data[['Weighted_Price']]
data['Weighted_Price'].fillna(method='ffill', inplace=True)
```

In [31]:
```
color_pal = ["#F8766D", "#D39200", "#93AA00", "#00BA38", "#00C19F", "#00B9E3", "#619CFF", "#DB72FB"]
_ = data.plot(style='', figsize=(15,5), color=color_pal[0], title='BTC Weighted_Price Price (USD) by Hours')
```

In [32]:
```
split_date = '25-Jun-2018'
data_train = data.loc[data.index <= split_date].copy()
data_test = data.loc[data.index > split_date].copy()
```

In [33]:
```
_ = data_test \
    .rename(columns={'Weighted_Price': 'Test Set'}) \
    .join(data_train.rename(columns={'Weighted_Price': 'Training Set'}), how='outer') \
    .plot(figsize=(15,5), title='BTC Weighted_Price Price (USD) by Hours', style='')
```

In [34]:
```python
def create_features(df, label=None):
    """
    Creates time series features from datetime index
    """
    df['date'] = df.index
    df['hour'] = df['date'].dt.hour
    df['dayofweek'] = df['date'].dt.dayofweek
    df['quarter'] = df['date'].dt.quarter
    df['month'] = df['date'].dt.month
    df['year'] = df['date'].dt.year
    df['dayofyear'] = df['date'].dt.dayofyear
    df['dayofmonth'] = df['date'].dt.day
    df['weekofyear'] = df['date'].dt.weekofyear

    X = df[['hour','dayofweek','quarter','month','year',
           'dayofyear','dayofmonth','weekofyear']]
    if label:
        y = df[label]
        return X, y
    return X
```

In [35]:
```python
X_train, y_train = create_features(data_train, label='Weighted_Price')
X_test, y_test = create_features(data_test, label='Weighted_Price')
```

Here we use a basic XGBRegressor model,

In [36]:
```python
import xgboost as xgb
from xgboost import plot_importance, plot_tree
model =  xgb.XGBRegressor(objective ='reg:linear',min_child_weight=10, booster='gbtree', colsample_bytree = 0.3, learning_rate = 0.1,
                max_depth = 5, alpha = 10, n_estimators = 100)
model.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_test, y_test)],
        early_stopping_rounds=50,
       verbose=False) # Change verbose to True if you want to see it train
```

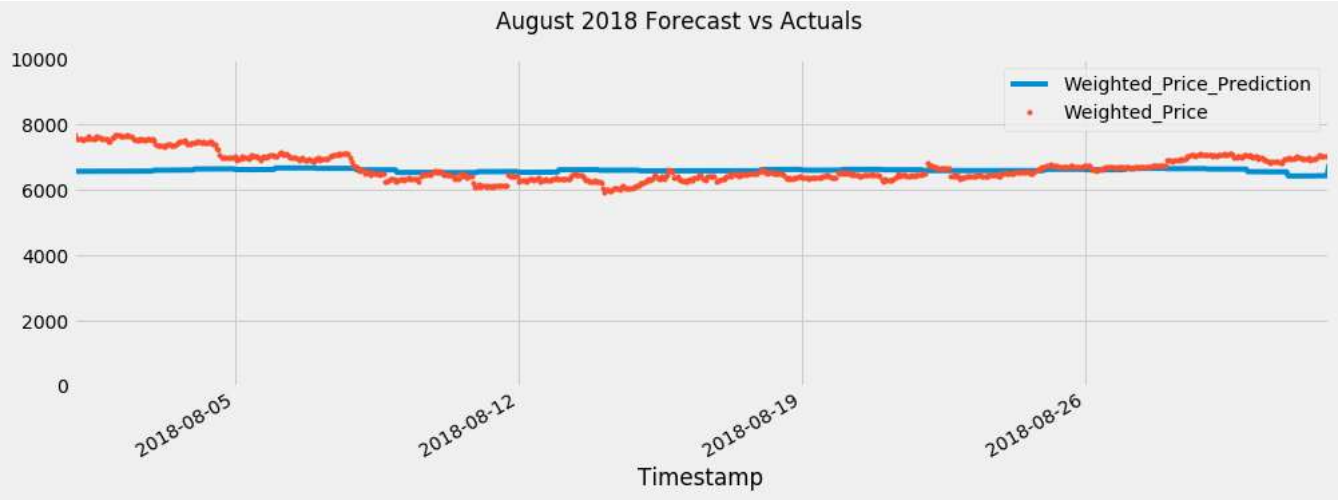
Out[36]:
```
XGBRegressor(alpha=10, base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=0.3, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=5, min_child_weight=10, missing=None, n_estimators=100,
       n_jobs=1, nthread=None, objective='reg:linear', random_state=0,
       reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
       silent=True, subsample=1)
```

In [37]:
```
data_test['Weighted_Price_Prediction'] = model.predict(X_test)
data_all = pd.concat([data_test, data_train], sort=False)
```

In [38]:
```
#adding to final data for comparision
final_data = pd.merge(final_data, data_all, sort=False)
final_data = final_data.rename(columns={'Weighted_Price_Prediction': 'xgboost'})
final_data = final_data[['Timestamp','Weighted_Price','lstm','xgboost']]
```

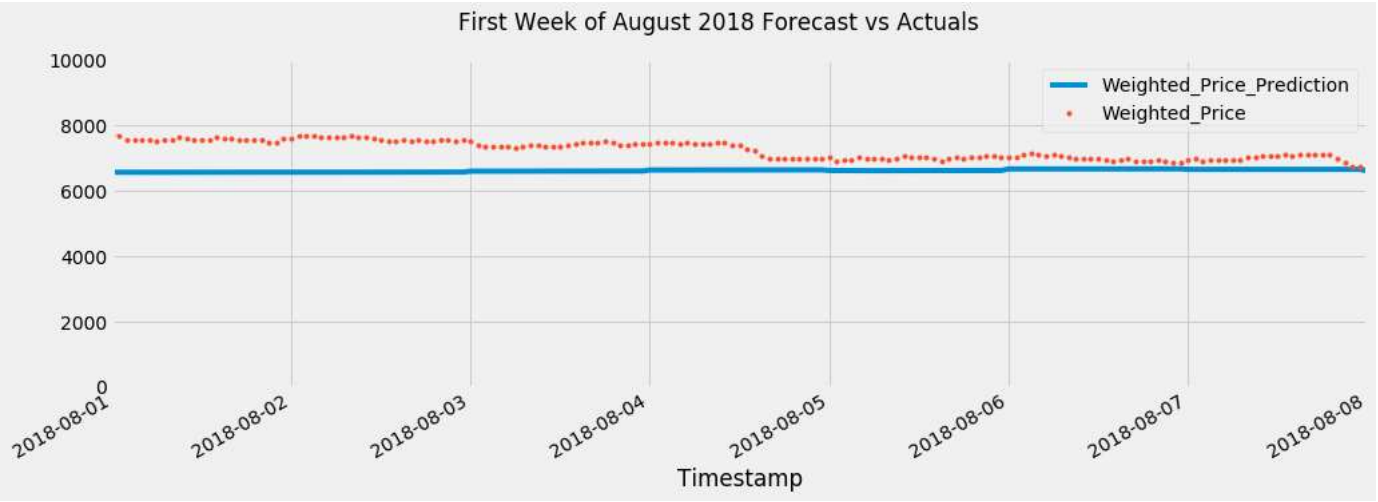
In [39]:
```
_ = data_all[['Weighted_Price','Weighted_Price_Prediction']].plot(figsize=(15, 5))
```

In [40]:
```
# Plot the forecast with the actuals
f, ax = plt.subplots(1)
f.set_figheight(5)
f.set_figwidth(15)
_ = data_all[['Weighted_Price_Prediction','Weighted_Price']].plot(ax=ax,
                                                  style=['-','.'])
ax.set_xbound(lower='08-01-2018', upper='09-01-2018')
ax.set_ylim(0, 10000)
plot = plt.suptitle('August 2018 Forecast vs Actuals')
```



In [41]:
```
# Plot the forecast with the actuals
f, ax = plt.subplots(1)
```

```
f.set_figheight(5)
f.set_figwidth(15)
_ = data_all[['Weighted_Price_Prediction','Weighted_Price']].plot(ax=ax,
                                                  style=['-','.'])
ax.set_xbound(lower='08-01-2018', upper='08-08-2018')
ax.set_ylim(0, 10000)
plot = plt.suptitle('First Week of August 2018 Forecast vs Actuals')
```



In [42]:
```
mean_squared_error(y_true=data_test['Weighted_Price'],
                   y_pred=data_test['Weighted_Price_Prediction'])
```

Out[42]:
```
135860677.50074437
```

In [43]:
```
mean_absolute_error(y_true=data_test['Weighted_Price'],
                   y_pred=data_test['Weighted_Price_Prediction'])
```

Out[43]:
```
6203.501684425248
```

Time Series forecasting with Prophet

Prophet is a forecasting procedure designed for time series data. It employs an additive model to capture non-linear trends, incorporating yearly, weekly, and daily seasonality, as well as holiday effects. Prophet performs exceptionally well with time series data that exhibit strong seasonal patterns and have extensive historical data. It demonstrates robustness in handling missing data, trend shifts, and outliers.

One of the notable features of the Prophet package is its user-friendly and intuitive parameters, making it accessible even to individuals with minimal expertise in forecasting models. This tool can be effectively utilized to generate meaningful predictions for a variety of business scenarios.

In [45]:
```
data = pd.read_csv('../input/bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv',parse_dates=[0], date_parser=dateparse)
data['Timestamp'] = data['Timestamp'].dt.tz_localize(None)
data = data.groupby([pd.Grouper(key='Timestamp', freq='H')]).first().reset_index()
data = data.set_index('Timestamp')
data = data[['Weighted_Price']]
data['Weighted_Price'].fillna(method='ffill', inplace=True)
```

In [46]:
```
color_pal = ["#F8766D", "#D39200", "#93AA00", "#00BA38", "#00C19F", "#00B9E3", "#619CFF", "#DB72FB"]
_ = data.plot(style='', figsize=(15,5), color=color_pal[0], title='BTC Weighted_Price Price (USD) by Hours')
```

In [47]:
```
split_date = '25-Jun-2018'
data_train = data.loc[data.index <= split_date].copy()
data_test = data.loc[data.index > split_date].copy()
```

In [48]:
```
_ = data_test \
    .rename(columns={'Weighted_Price': 'Test Set'}) \
    .join(data_train.rename(columns={'Weighted_Price': 'Training Set'}), how='outer') \
    .plot(figsize=(15,5), title='BTC Weighted_Price Price (USD) by Hours', style='')
```

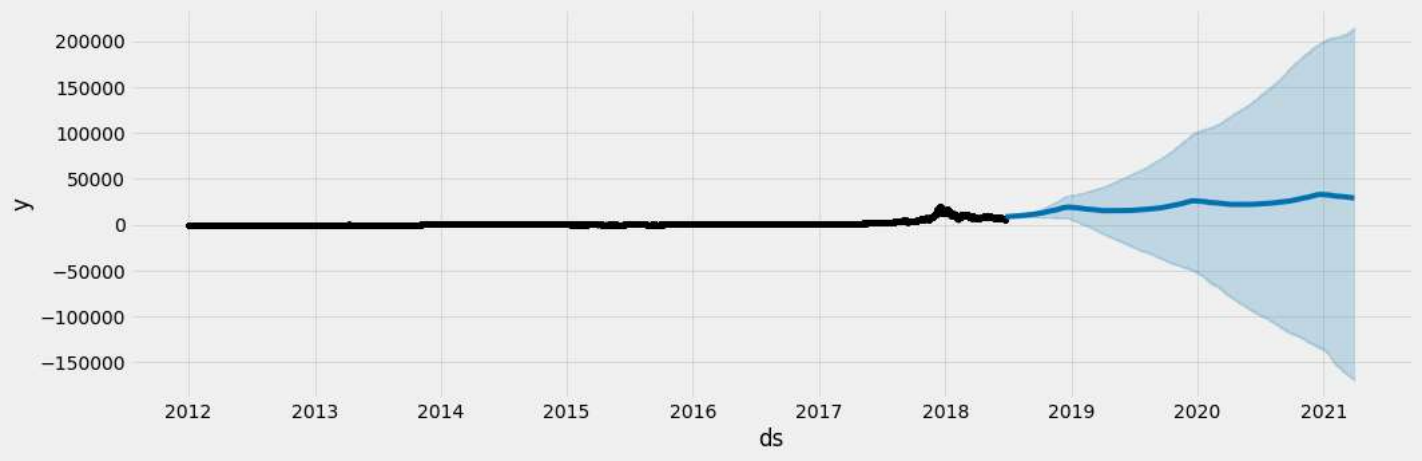
In [49]:
```
data_train = data_train.reset_index().rename(columns={'Timestamp':'ds', 'Weighted_Price':'y'})
```

In [50]:
```
# Setup and train model
model = Prophet()
model.fit(data_train)
```

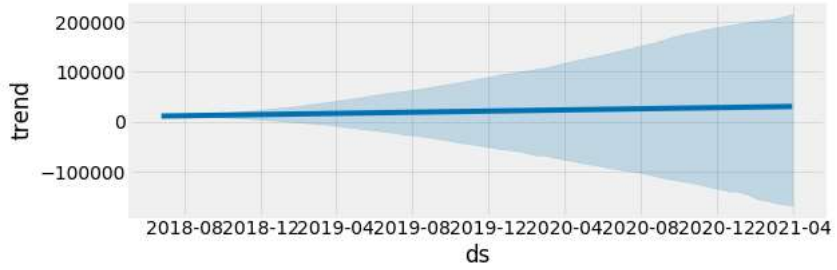
Out[50]:
```
<fbprophet.forecaster.Prophet at 0x788cb1775a58>
```

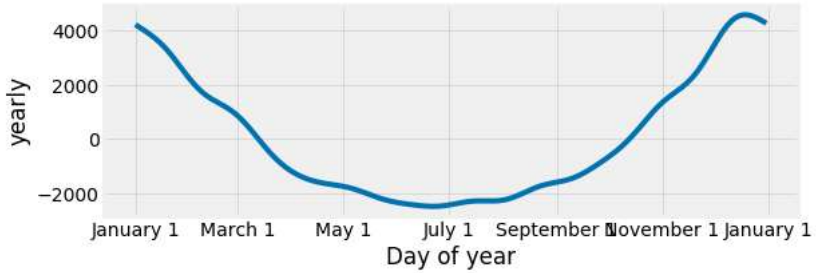
In [51]:
```
# Predict on training set with model
data_test_fcst = model.predict(df=data_test.reset_index().rename(columns={'Timestamp':'ds'}))
```

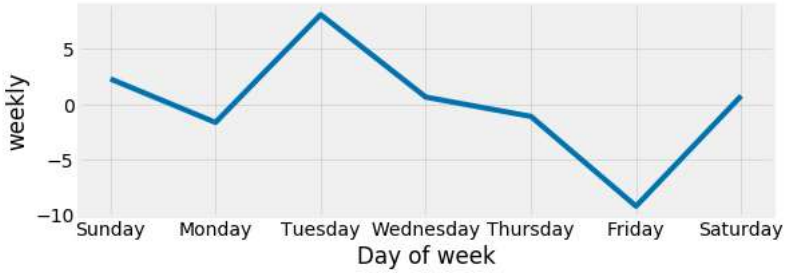
In [52]:
```
# Plot the forecast
f, ax = plt.subplots(1)
f.set_figheight(5)
f.set_figwidth(15)
fig = model.plot(data_test_fcst, ax=ax)
```

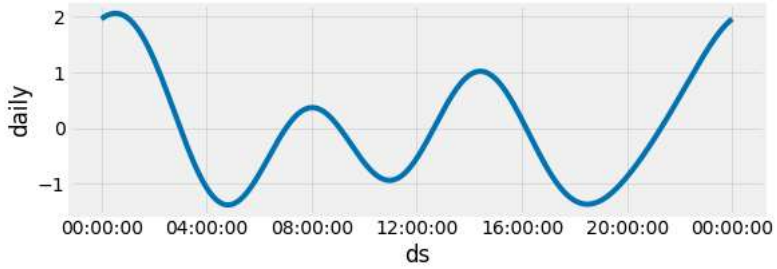


In [53]:
```
# Plot the components
fig = model.plot_components(data_test_fcst)
```

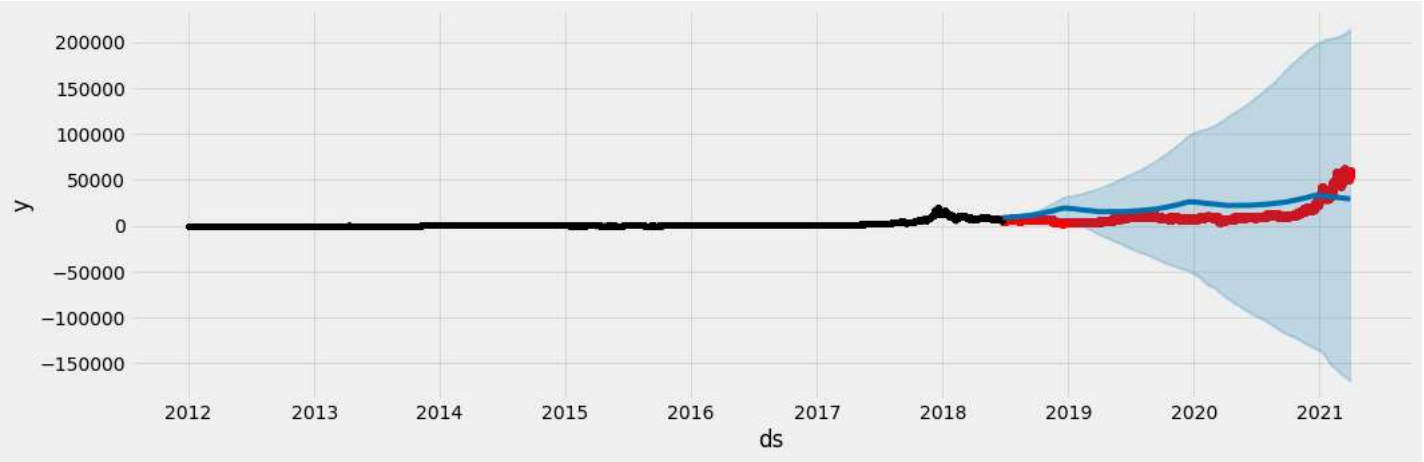






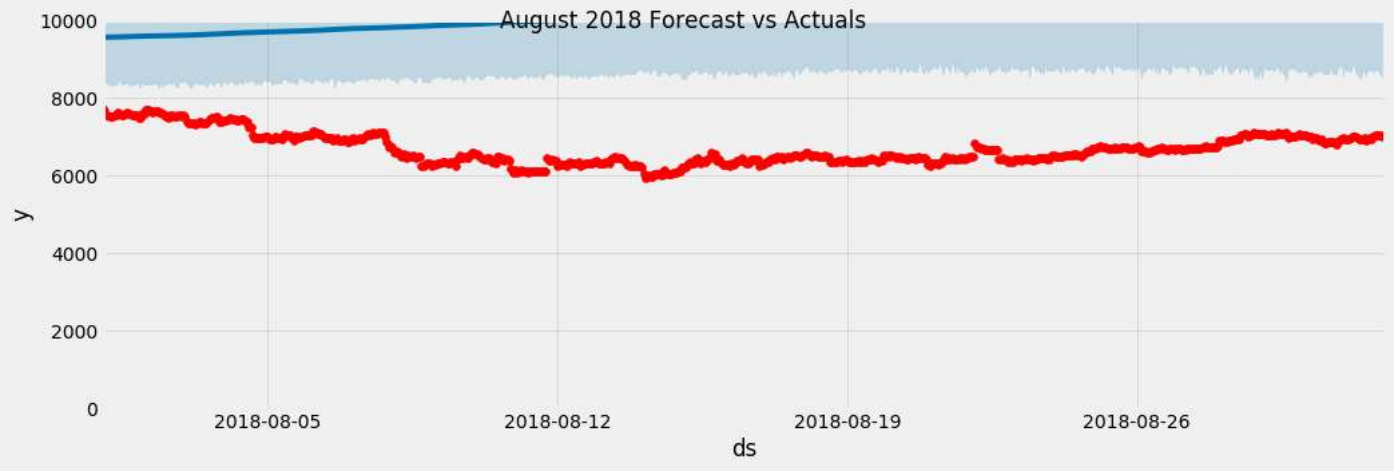


In [54]:
```
# Plot the forecast with the actuals
f, ax = plt.subplots(1)
f.set_figheight(5)
f.set_figwidth(15)
ax.scatter(data_test.index, data_test['Weighted_Price'], color='r')
fig = model.plot(data_test_fcst, ax=ax)
```



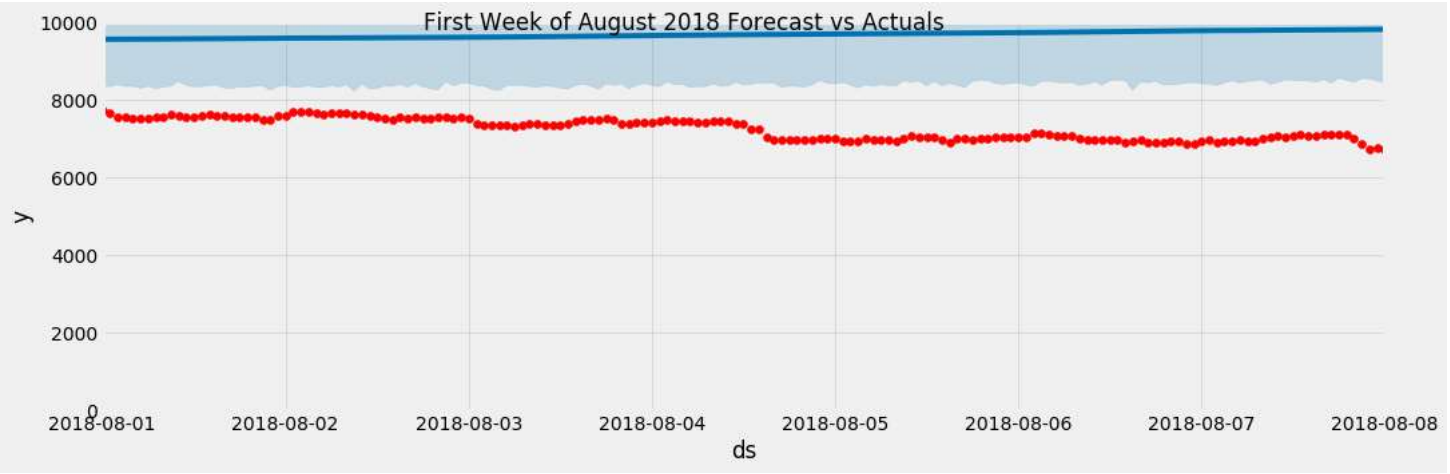
In [55]:
```
#for comparision of predictions
data_fcst = data_test_fcst
data_fcst = data_fcst.rename(columns={'ds': 'Timestamp'})
data_all = pd.concat([data_fcst, data_train], sort=False)
final_data = pd.merge(final_data, data_all, sort=False)
final_data = final_data.rename(columns={'yhat': 'prophet'})
final_data = final_data[['Timestamp','Weighted_Price','lstm','xgboost','prophet']]
```

In [56]:
```
# Plot the forecast with the actuals
f, ax = plt.subplots(1)
f.set_figheight(5)
f.set_figwidth(15)
ax.scatter(data_test.index, data_test['Weighted_Price'], color='r')
fig = model.plot(data_test_fcst, ax=ax)
ax.set_xbound(lower='08-01-2018', upper='09-01-2018')
ax.set_ylim(0, 10000)
plot = plt.suptitle('August 2018 Forecast vs Actuals')
```



In [57]:
```
# Plot the forecast with the actuals
f, ax = plt.subplots(1)
f.set_figheight(5)
f.set_figwidth(15)
```

```
ax.scatter(data_test.index, data_test['Weighted_Price'], color='r')
fig = model.plot(data_test_fcst, ax=ax)
ax.set_xbound(lower='08-01-2018', upper='08-08-2018')
ax.set_ylim(0, 10000)
plot = plt.suptitle('First Week of August 2018 Forecast vs Actuals')
```



In [58]:
```
mean_squared_error(y_true=data_test['Weighted_Price'],
                   y_pred=data_test_fcst['yhat'])
```

Out[58]:
```
160825797.2098115
```

In [59]:
```
mean_absolute_error(y_true=data_test['Weighted_Price'],
                   y_pred=data_test_fcst['yhat'])
```

Out[59]:
```
11568.292079622006
```

Time Series forecasting using ARIMA

ARIMA stands for AutoRegressive Integrated Moving Average, which is a class of models designed to capture various standard temporal structures in time series data. This acronym effectively describes the fundamental components of the model:

AR (Autoregression): This part of the model leverages the dependent relationship between an observation and a certain number of lagged observations. I (Integrated): It involves differencing raw observations (e.g., subtracting an observation from the previous time step) to make the time series stationary. MA (Moving Average): This component of the model utilizes the dependency between an observation and a residual error from a moving average model applied to lagged observations.

ARIMA is among the most widely used techniques for time series analysis. In Python, you can create ARIMA-based forecasting models using either AutoARIMA (Pyramid ARIMA) or StatsModel.

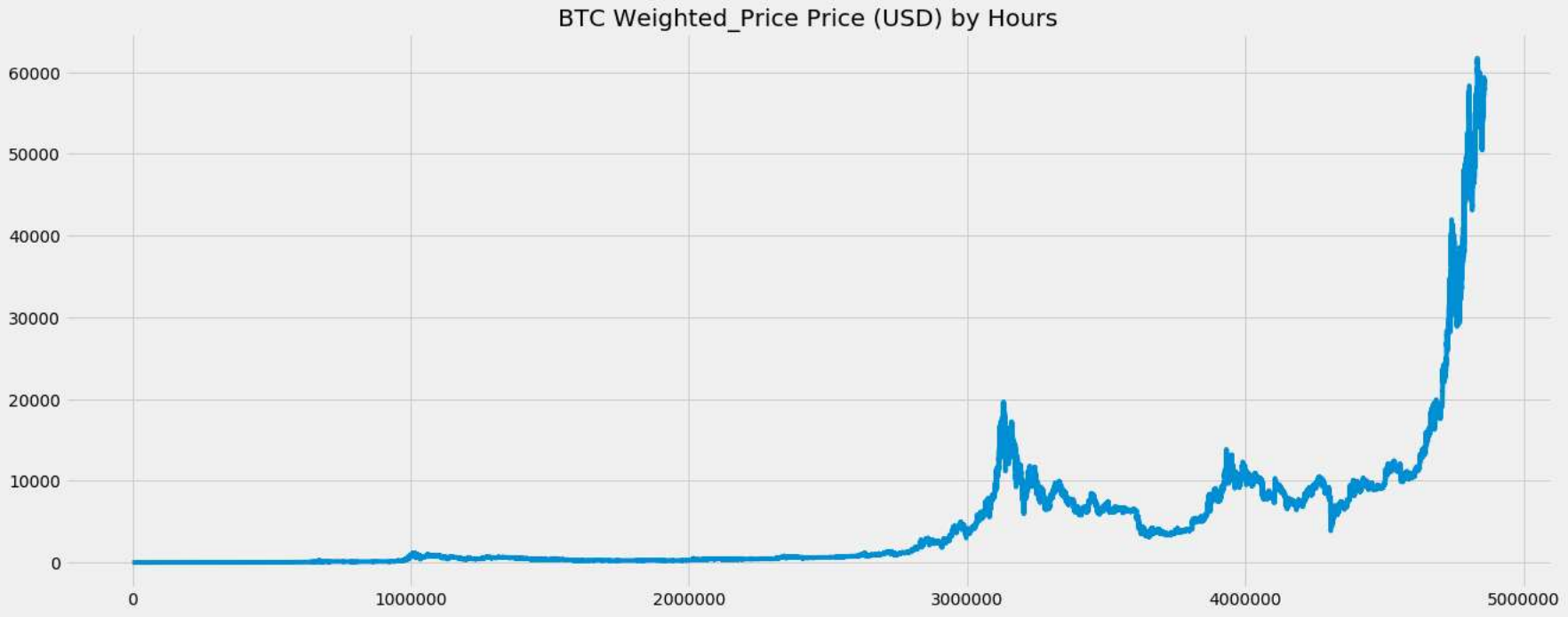
In [61]:
```
from scipy import stats
import statsmodels.api as sm
import warnings
from itertools import product
```

In [62]:
```
data = pd.read_csv('../input/bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv',parse_dates=[0], date_parser=dateparse)
```

In [63]:
```
data['Open'].fillna(method='ffill', inplace=True)
data['High'].fillna(method='ffill', inplace=True)
data['Low'].fillna(method='ffill', inplace=True)
data['Close'].fillna(method='ffill', inplace=True)
data['Weighted_Price'].fillna(method='ffill', inplace=True)
data['Volume_(BTC)'].fillna(method='ffill', inplace=True)
data['Volume_(Currency)'].fillna(method='ffill', inplace=True)
```

In [64]:
```
plt.figure(figsize=[20,8])
plt.title('BTC Weighted_Price Price (USD) by Hours')
plt.plot(data.Weighted_Price, '-', label='By Hours')
```

Out[64]:
```
[<matplotlib.lines.Line2D at 0x788cb22e36d8>]
```



In previous sections of LSTM,XGBoost and Prophet, we used hourly data to train the model. But here we will use the monthly data (for Seasonality).

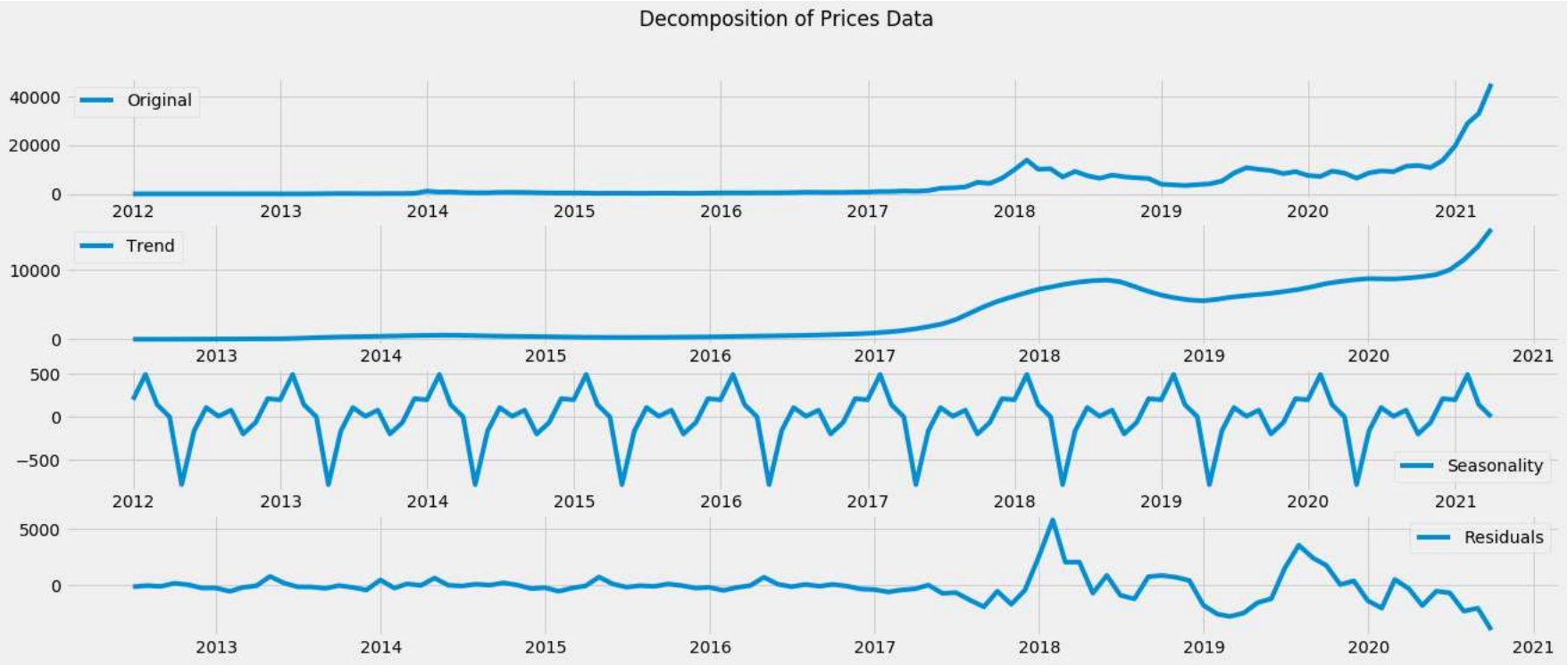
In [65]:
```
data['Timestamp'] = data['Timestamp'].dt.tz_localize(None)
data = data.groupby([pd.Grouper(key='Timestamp', freq='M')]).first().reset_index()
data = data.set_index('Timestamp')
data['Weighted_Price'].fillna(method='ffill', inplace=True)
```

In [66]:
```
plt.figure(figsize=[20,8])
plt.title('BTC Weighted_Price Price (USD) by Months')
plt.plot(data.Weighted_Price, '-', label='By Months')
```

Out[66]:
```
[<matplotlib.lines.Line2D at 0x788cb0d232e8>]
```

Decomposition

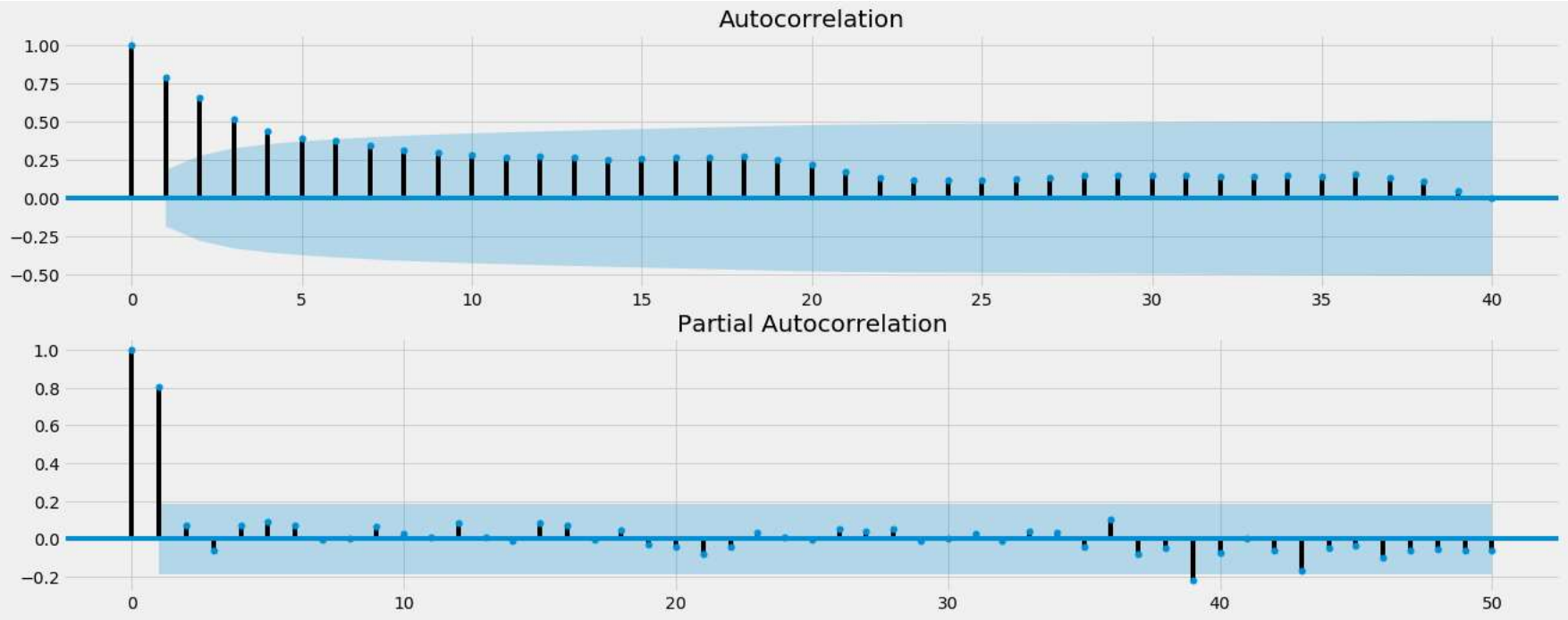
In [67]:
```
decomposition = sm.tsa.seasonal_decompose(data.Weighted_Price)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

fig = plt.figure(figsize=(20,8))

plt.subplot(411)
plt.plot(data.Weighted_Price, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')

fig.suptitle('Decomposition of Prices Data')
plt.show()
```



In [68]:
```
print("Dickey-Fuller test: p=%f" % sm.tsa.stattools.adfuller(data.Weighted_Price)[1])
```

```
Dickey-Fuller test: p=0.999084
```

In [69]:
```
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf
from matplotlib import pyplot
pyplot.figure(figsize=(20,8))
pyplot.subplot(211)
plot_acf(data.Weighted_Price, ax=pyplot.gca(),lags=40)
pyplot.subplot(212)
plot_pacf(data.Weighted_Price, ax=pyplot.gca(), lags=50)
pyplot.show()
```



Here's the Statespace ARIMA Model, the best model is selected using different parameters.

In [70]:
```
# Initial approximation of parameters
Qs = range(0, 2)
qs = range(0, 3)
Ps = range(0, 3)
ps = range(0, 3)
D=1
d=1
parameters = product(ps, qs, Ps, Qs)
parameters_list = list(parameters)
len(parameters_list)

# Model Selection
results = []
best_aic = float("inf")
warnings.filterwarnings('ignore')
for param in parameters_list:
    try:
        model=sm.tsa.statespace.SARIMAX(data.Weighted_Price, order=(param[0], d, param[1]),
                                        seasonal_order=(param[2], D, param[3], 12),enforce_stationarity=False,
                                            enforce_invertibility=False).fit(disp=-1)
    except ValueError:
        #print('wrong parameters:', param)
        continue
    aic = model.aic
    if aic < best_aic:
        best_model = model
        best_aic = aic
        best_param = param
    results.append([param, model.aic])
```

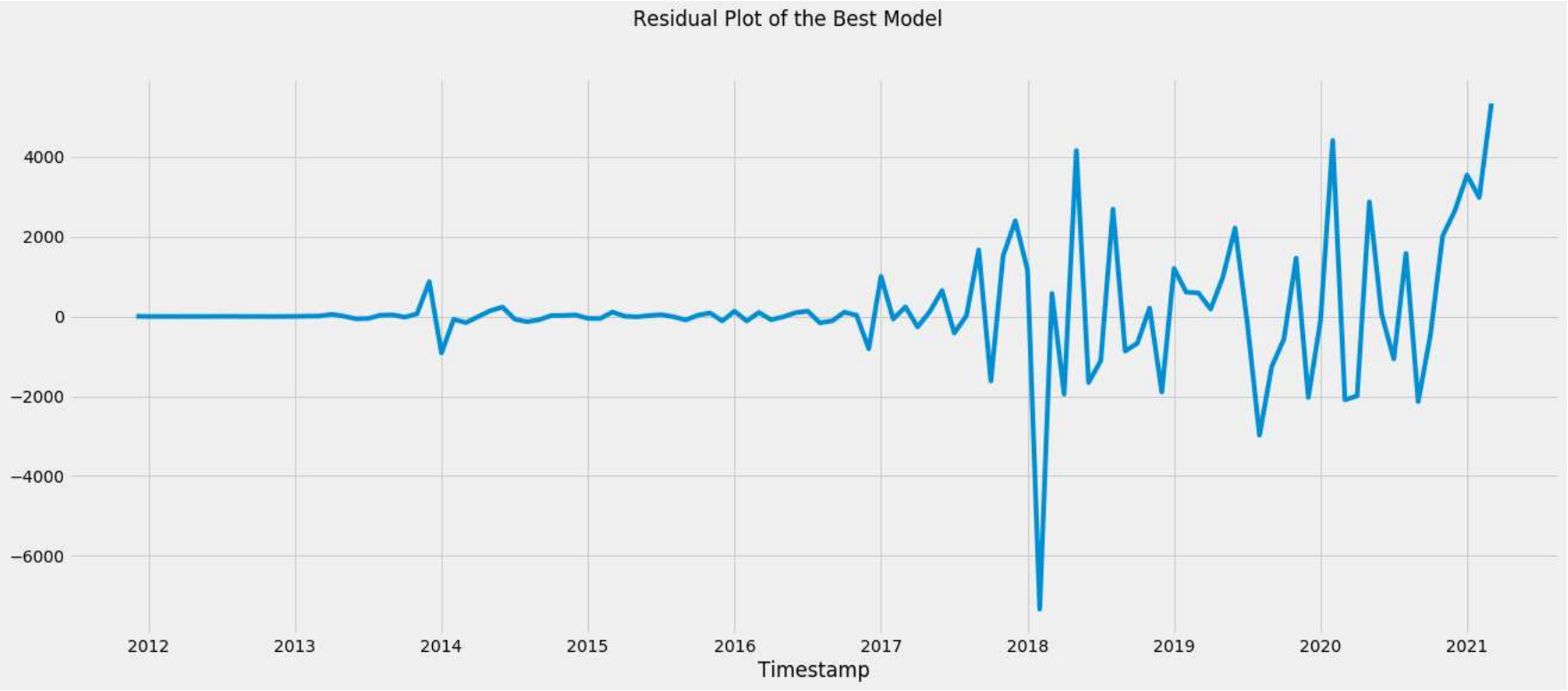
In [71]:
```
# Best Models
result_table = pd.DataFrame(results)
```

```
result_table.columns = ['parameters', 'aic']
print(result_table.sort_values(by = 'aic', ascending=True).head())
print(best_model.summary())
```

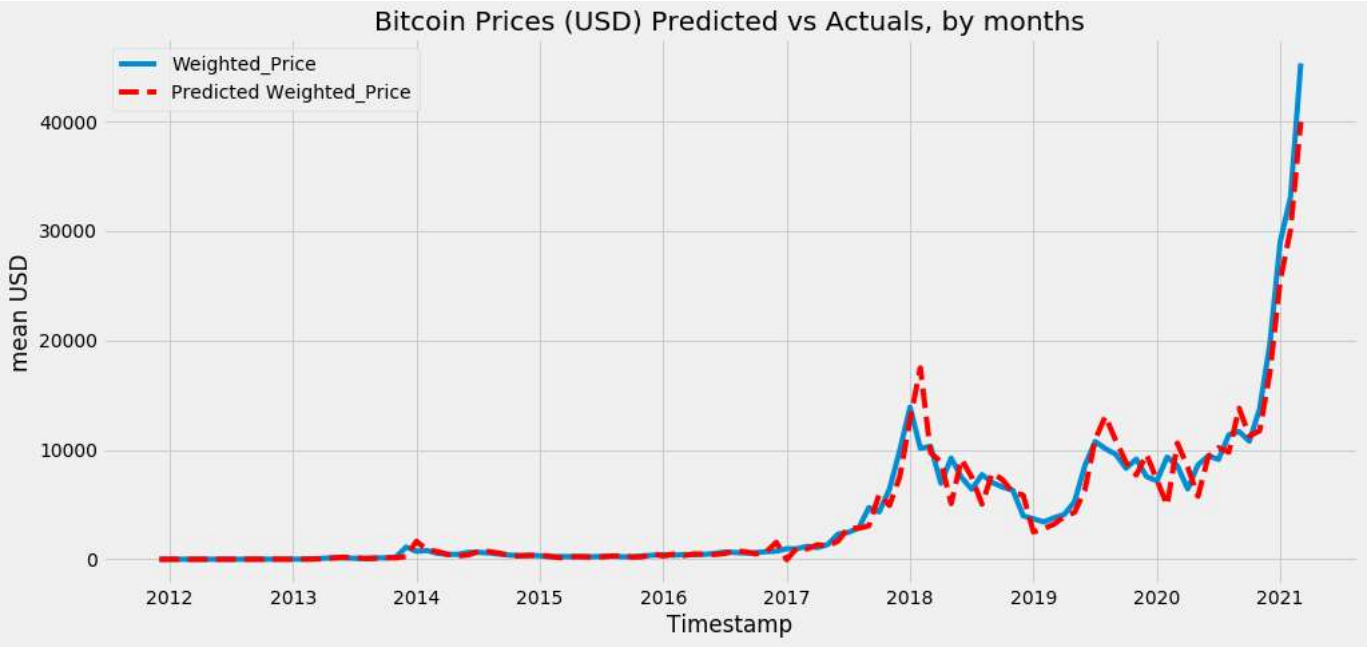
```
      parameters          aic
40  (2, 0, 2, 0)  1312.410259
41  (2, 0, 2, 1)  1313.765417
46  (2, 1, 2, 0)  1314.063597
52  (2, 2, 2, 0)  1315.306534
47  (2, 1, 2, 1)  1315.384819
                                Statespace Model Results
==========================================================================================
Dep. Variable:                    Weighted_Price   No. Observations:                  112
Model:             SARIMAX(2, 1, 0)x(2, 1, 0, 12)   Log Likelihood                -651.205
Date:                            Tue, 17 Oct 2023   AIC                           1312.410
Time:                                    20:50:56   BIC                           1323.863
Sample:                                12-31-2011   HQIC                          1316.974
                                     - 03-31-2021
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.5665      0.110      5.152      0.000       0.351       0.782
ar.L2          0.4075      0.143      2.845      0.004       0.127       0.688
ar.S.L12      -1.0465      0.095    -11.073      0.000      -1.232      -0.861
ar.S.L24      -0.8876      0.094     -9.473      0.000      -1.071      -0.704
sigma2      3.28e+06   3.37e+05      9.720      0.000    2.62e+06    3.94e+06
===================================================================================
Ljung-Box (Q):                       26.08   Jarque-Bera (JB):                44.44
Prob(Q):                              0.96   Prob(JB):                         0.00
Heteroskedasticity (H):              69.99   Skew:                            -0.34
Prob(H) (two-sided):                  0.00   Kurtosis:                         6.76
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

In [72]:
```
fig = plt.figure(figsize=(20,8))
best_model.resid.plot()
fig.suptitle('Residual Plot of the Best Model')
print("Dickey-Fuller test:: p=%f" % sm.tsa.stattools.adfuller(best_model.resid)[1])
```

```
Dickey-Fuller test:: p=0.000000
```



In [73]:
```
df_month2 = data[['Weighted_Price']]
future = pd.DataFrame()
df_month2 = pd.concat([df_month2, future])
df_month2['forecast'] = best_model.predict(start=0, end=200)
plt.figure(figsize=(15,7))
df_month2.Weighted_Price.plot()
df_month2.forecast.plot(color='r', ls='--', label='Predicted Weighted_Price')
plt.legend()
plt.title('Bitcoin Prices (USD) Predicted vs Actuals, by months')
plt.ylabel('mean USD')
plt.show()
```



In [74]:
```
from scipy import stats
import statsmodels.api as sm
import warnings
from itertools import product
```

In [75]:
```
data = pd.read_csv('../input/bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv',parse_dates=[0], date_parser=dateparse)
data.head()
```

Out[75]:

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
0	2011-12-31 07:52:00+00:00	4.39	4.39	4.39	4.39	0.455581	2.0	4.39
1	2011-12-31 07:53:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	2011-12-31 07:54:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	2011-12-31 07:55:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	2011-12-31 07:56:00+00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN

In [76]:
```
data['Timestamp'] = data['Timestamp'].dt.tz_localize(None)
data = data.groupby([pd.Grouper(key='Timestamp', freq='M')]).first().reset_index()
data = data.set_index('Timestamp')
data['Weighted_Price'].fillna(method='ffill', inplace=True)
```

In [77]:
```
plt.figure(figsize=[20,8])
plt.title('BTC Weighted_Price Price (USD) by Months')
plt.plot(data.Weighted_Price, '-', label='By Months')
```

Out[77]:
```
[<matplotlib.lines.Line2D at 0x788cc430a128>]
```

We can assess the performance of the aforementioned models with historical Bitcoin price data. However, the results are often less than satisfactory. Bitcoin prices are extremely volatile and unpredictable, often influenced by external factors such as cryptocurrency regulations, investments, or even rumors circulating on social media. To enhance the accuracy and effectiveness of these models, it's essential to incorporate additional data from news sources and social media, which can provide valuable insights into the cryptocurrency market's behavior.