Cryptocurrency Predictions with ARIMA and Prophet

Contents:

In [1]:
```
import pandas as pd
from pandas import DataFrame
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (15,7)
from datetime import datetime, timedelta

from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.seasonal import seasonal_decompose

from scipy import stats
import statsmodels.api as sm
from itertools import product

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

In [2]:
```
dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m-%d')
df = pd.read_csv('ETH-USD.csv', parse_dates=['Date'], index_col='Date', date_parser=date
df.head()
```

Out[2]:

Date	Open	High	Low	Close	Adj Close	Volume
2017-11-09	308.644989	329.451996	307.056000	320.884003	320.884003	893249984
2017-11-10	320.670990	324.717987	294.541992	299.252991	299.252991	885985984
2017-11-11	298.585999	319.453003	298.191986	314.681000	314.681000	842300992

2017-11-12	314.690002	319.153015	298.513000	307.907990	307.907990	1613479936
2017-11-13	307.024994	328.415009	307.024994	316.716003	316.716003	1041889984

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

Out[3]:

Date	Open	High	Low	Close	Adj Close	Volume
2023-04-12	1891.949707	1929.881226	1860.036865	1920.682129	1920.682129	11010714187
2023-04-13	1917.698364	2022.150146	1901.860352	2012.634644	2012.634644	12546950499
2023-04-14	2013.930664	2126.316650	2011.503296	2101.635498	2101.635498	16298099411
2023-04-15	2101.616455	2111.075439	2076.510742	2092.466797	2092.466797	8036468153
2023-04-16	2091.595215	2102.360840	2080.340576	2087.440186	2087.440186	6878934528

In [4]:
```
# Extract the Ethereum data only
eth=df
# Drop some columns
#eth=eth.drop('Volume',axis=1,inplace=True)
eth
```

Out[4]:

Date	Open	High	Low	Close	Adj Close	Volume
2017-11-09	308.644989	329.451996	307.056000	320.884003	320.884003	893249984
2017-11-10	320.670990	324.717987	294.541992	299.252991	299.252991	885985984
2017-11-11	298.585999	319.453003	298.191986	314.681000	314.681000	842300992
2017-11-12	314.690002	319.153015	298.513000	307.907990	307.907990	1613479936
2017-11-13	307.024994	328.415009	307.024994	316.716003	316.716003	1041889984
...	...	...	...	...	...	...
2023-04-12	1891.949707	1929.881226	1860.036865	1920.682129	1920.682129	11010714187
2023-04-13	1917.698364	2022.150146	1901.860352	2012.634644	2012.634644	12546950499
2023-04-14	2013.930664	2126.316650	2011.503296	2101.635498	2101.635498	16298099411
2023-04-15	2101.616455	2111.075439	2076.510742	2092.466797	2092.466797	8036468153
2023-04-16	2091.595215	2102.360840	2080.340576	2087.440186	2087.440186	6878934528

1985 rows × 6 columns

In [5]:
```
eth.head()
```

Out[5]:

Date	Open	High	Low	Close	Adj Close	Volume
2017-11-09	308.644989	329.451996	307.056000	320.884003	320.884003	893249984
2017-11-10	320.670990	324.717987	294.541992	299.252991	299.252991	885985984
2017-11-11	298.585999	319.453003	298.191986	314.681000	314.681000	842300992
2017-11-12	314.690002	319.153015	298.513000	307.907990	307.907990	1613479936

2017-11-13	307.024994	328.415009	307.024994	316.716003	316.716003	1041889984

Monthly Forecasting

In [6]:
```
# Resampling to monthly frequency
eth_month = eth.resample('M').mean()
eth_month
```

Out[6]:

Date	Open	High	Low	Close	Adj Close	Volume
2017-11-30	373.696317	393.111908	363.283634	379.732093	379.732093	1.225341e+09
2017-12-31	630.583997	667.252580	596.163133	640.209291	640.209291	2.576202e+09
2018-01-31	1093.099893	1163.799714	1024.934606	1103.646004	1103.646004	5.277749e+09
2018-02-28	882.527006	917.850394	825.723679	873.116318	873.116318	2.978337e+09
2018-03-31	640.787129	653.875259	606.506935	625.761325	625.761325	1.732780e+09
...	...	...	...	...	...	...
2022-12-31	1240.294394	1255.038866	1221.342147	1237.105890	1237.105890	5.197763e+09
2023-01-31	1454.913763	1486.601736	1434.767791	1466.950026	1466.950026	7.256547e+09
2023-02-28	1623.919765	1654.276873	1595.175406	1624.605630	1624.605630	7.991048e+09
2023-03-31	1668.488836	1712.653257	1635.392877	1675.357851	1675.357851	9.757034e+09
2023-04-30	1900.191414	1938.581299	1879.125748	1916.605385	1916.605385	8.883953e+09

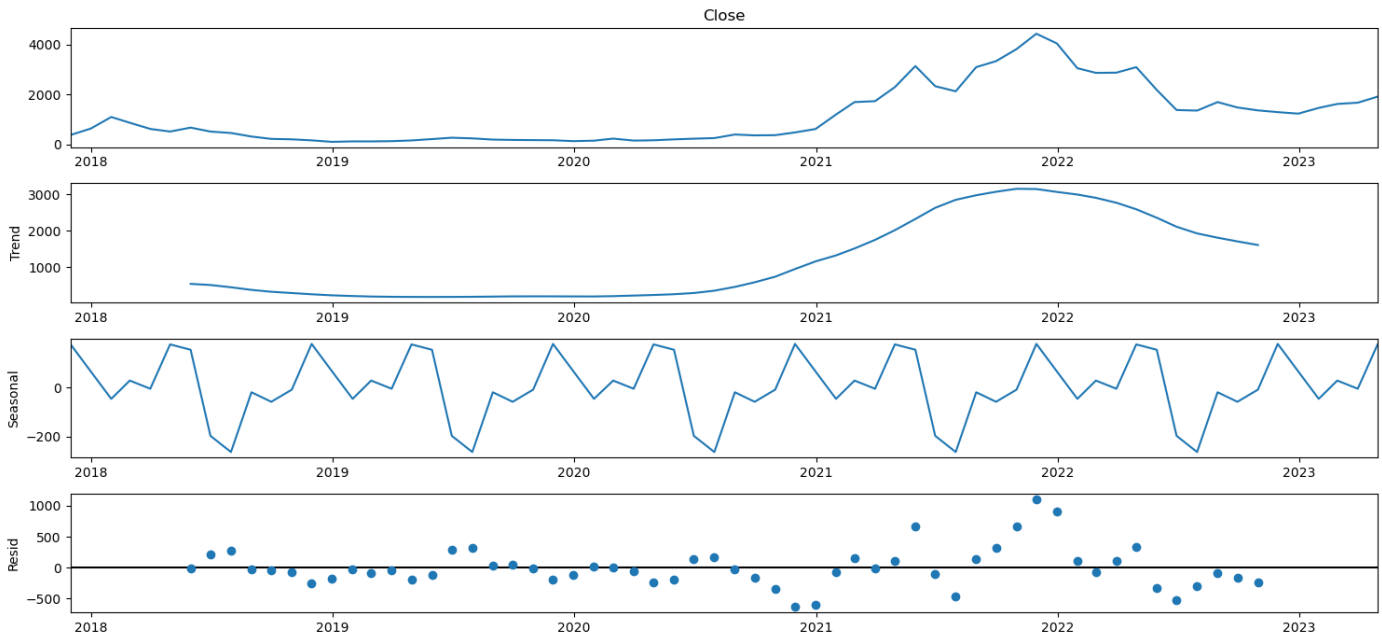
66 rows × 6 columns

Stationarity check and Seasonal decomposition

If a time series is stationary, it implies the lack of broad trends (changes in mean and variance over time) in the data. This is important as a consideration in time series forecasting.

In [7]:
```
#seasonal_decompose(eth_month.close, freq=12).plot()
seasonal_decompose(eth_month.Close).plot()
print("Dickey-Fuller test: p=%f" % adfuller(eth_month.Close)[1])
plt.show()
```

```
Dickey-Fuller test: p=0.340309
```



The p-value indicates that series is not stationary with 99.88% confidence.

Box-Cox Transformation

Lets use the Box-Cox transformation to suppress some of the variance.

The Box-Cox transformation is a family of power transformations indexed by a parameter lambda. Whenever you use it the parameter needs to be estimated from the data. In time series the process could have a non-constant variance. if the variance changes with time the process is nonstationary. It is often desirable to transform a time series to make it stationary. Sometimes after applying Box-Cox with a particular value of lambda the process may look stationary. It is sometimes possible that even if after applying the Box-Cox transformation the series does not appear to be stationary, diagnostics from ARIMA modeling can then be used to decide if differencing or seasonal differencing might be useful to to remove polynomial trends or seasonal trends respectively. After that the result might be an ARMA model that is stationary. If diagnostics confirm the orders p an q for the ARMA model, the AR and MA parameters can then be estimated.

In [8]:
```
# Box-Cox Transformations
eth_month['close_box'], lmbda = stats.boxcox(eth_month.Close)
print("Dickey-Fuller test: p=%f" % adfuller(eth_month.close_box)[1])
```

```
Dickey-Fuller test: p=0.731393
```

The p-value indicates that series is still not stationary.

Differencing

When building models to forecast time series data (like ARIMA), another pre-processing step is differencing the data (calculating sequentially xt−xt−1) until we get to a point where the series is stationary. Models account for oscillations but not for trends, and therefore, accounting for trends by differencing allows us to use the models that account for oscillations.

Once the model has been constructed, we can account for trends separately, by adding the trends component-wise.

Seasonal differentiation

One method of differencing data is seasonal differencing, which involves computing the difference between an observation and the corresponding observation in the previous year.

In [9]:
```
# Seasonal differentiation (12 months)
eth_month['box_diff_seasonal_12'] = eth_month.close_box - eth_month.close_box.shift(12)
print("Dickey-Fuller test: p=%f" % adfuller(eth_month.box_diff_seasonal_12[12:])[1])
```

```
Dickey-Fuller test: p=0.237069
```

The p-value indicates that series is still not stationary.

In [10]:
```
# Seasonal differentiation (3 months)
eth_month['box_diff_seasonal_3'] = eth_month.close_box - eth_month.close_box.shift(3)
print("Dickey-Fuller test: p=%f" % adfuller(eth_month.box_diff_seasonal_3[3:])[1])
```

```
Dickey-Fuller test: p=0.069912
```

The p-value indicates that series is stationary as the computed p-value is lower than the significance level alpha = 0.05.

Regular differentiation

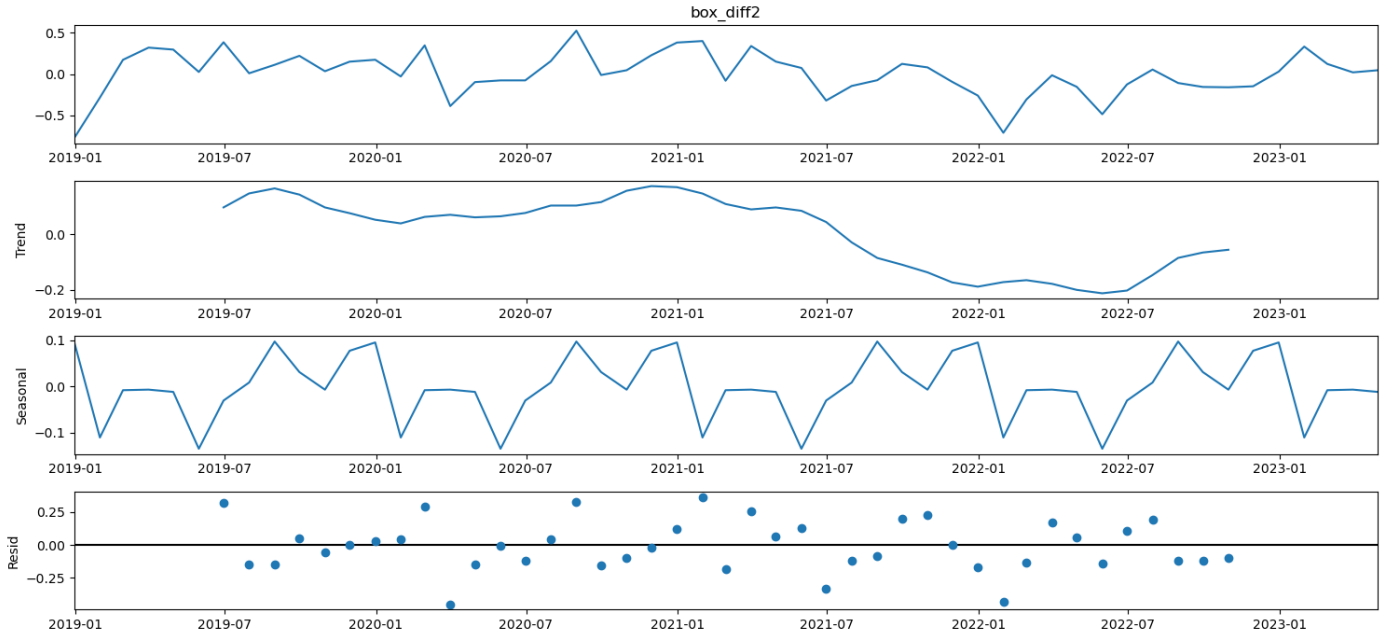
Sometimes it may be necessary to difference the data a second time to obtain a stationary time series, which is referred to as second order differencing.

In [11]:
```
# Regular differentiation
eth_month['box_diff2'] = eth_month.box_diff_seasonal_12 - eth_month.box_diff_seasonal_12

# STL-decomposition
seasonal_decompose(eth_month.box_diff2[13:]).plot()
print("Dickey-Fuller test: p=%f" % adfuller(eth_month.box_diff2[13:])[1])

plt.show()
```

```
Dickey-Fuller test: p=0.000004
```



In [12]:
```
eth_month.Close
```

Out[12]:
```
Date
2017-11-30     379.732093
```

```
2017-12-31     640.209291
2018-01-31    1103.646004
2018-02-28     873.116318
2018-03-31     625.761325
                  ...
2022-12-31    1237.105890
2023-01-31    1466.950026
2023-02-28    1624.605630
2023-03-31    1675.357851
2023-04-30    1916.605385
Freq: M, Name: Close, Length: 66, dtype: float64
```

The p-value indicates that series is stationary as the computed p-value is lower than the significance level alpha = 0.05.

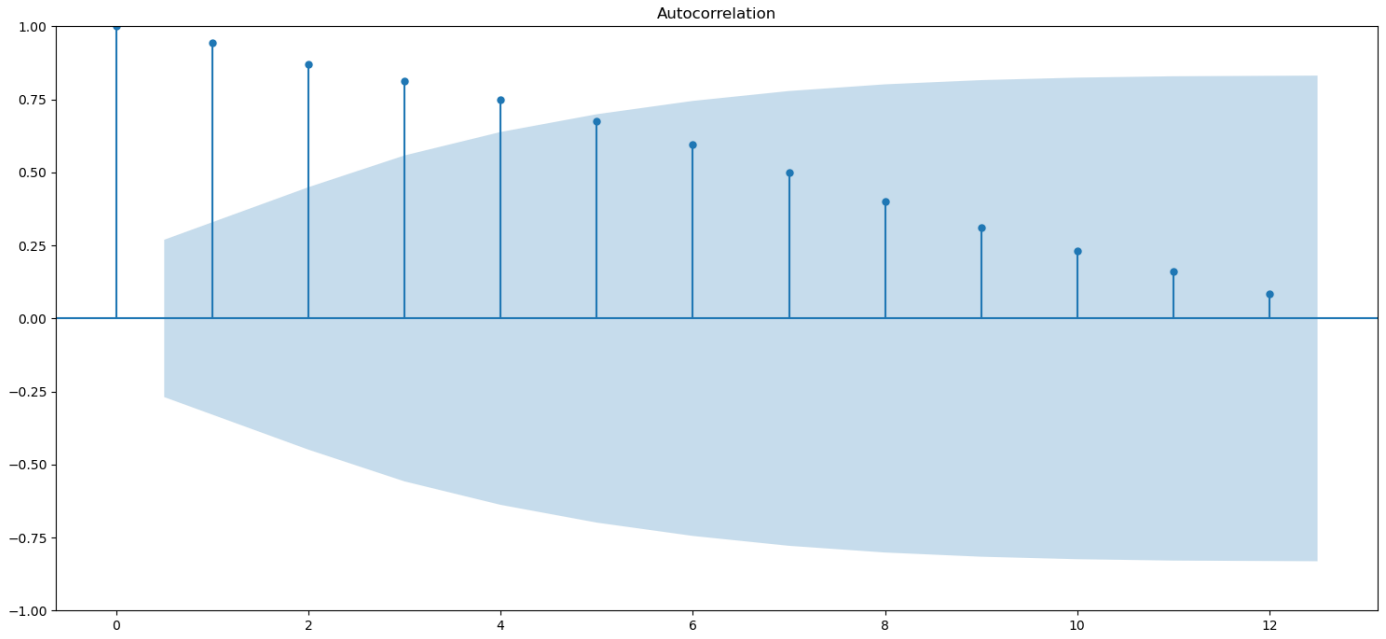
Autocorrelation

Autocorrelation is the correlation of a time series with the same time series lagged. It summarizes the strength of a relationship with an observation in a time series with observations at prior time steps.

We create autocorrelation factor (ACF) and partial autocorrelation factor (PACF) plots to identify patterns in the above data which is stationary on both mean and variance. The idea is to identify presence of AR and MA components in the residuals.

In [13]:
```
#autocorrelation_plot(eth_month.close)
plot_acf(eth_month.Close[13:].values.squeeze(), lags=12)

plt.tight_layout()
plt.show()
```



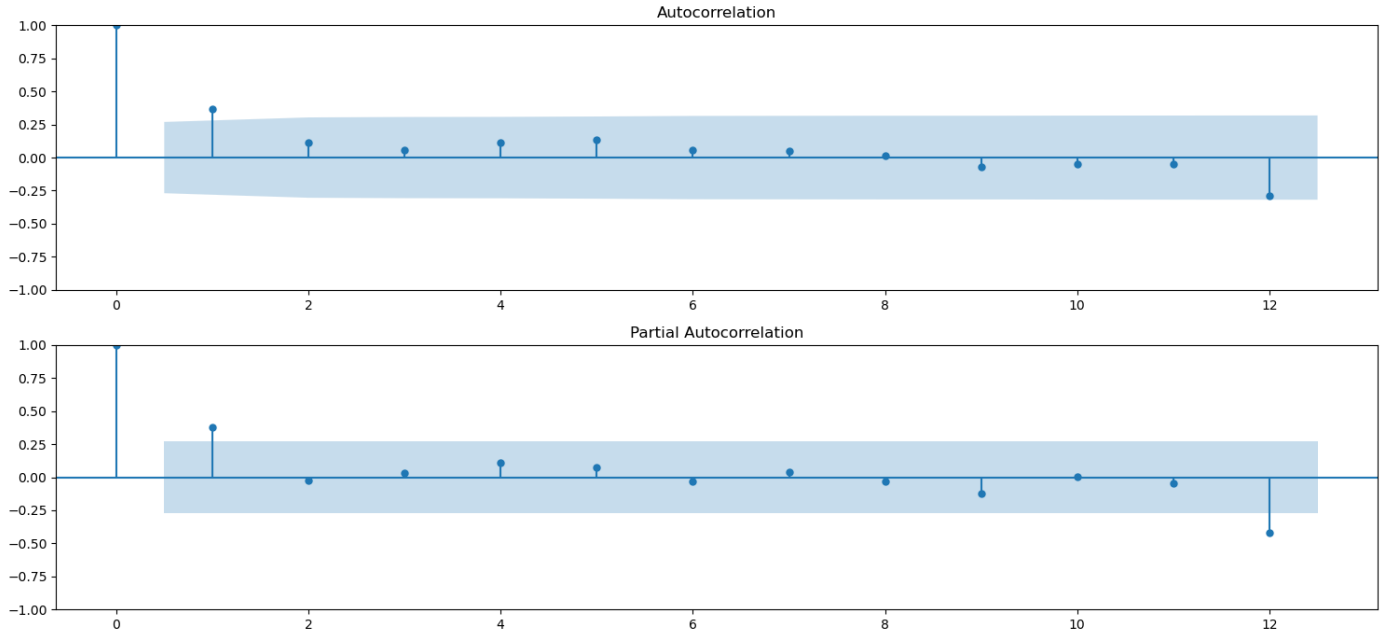
There is a positive correlation with the first 10 lags that is perhaps significant for the first 2-3 lags.

A good starting point for the AR parameter of the model may be 3.

Lets try out autocorrelation on the differences...

In [14]:
```
# Initial approximation of parameters using Autocorrelation and Partial Autocorrelation
ax = plt.subplot(211)
# Plot the autocorrelation function
#sm.graphics.tsa.plot_acf(eth_month.box_diff2[13:].values.squeeze(), lags=48, ax=ax)
```

```
plot_acf(eth_month.box_diff2[13:].values.squeeze(), lags=12, ax=ax)
ax = plt.subplot(212)
#sm.graphics.tsa.plot_pacf(eth_month.box_diff2[13:].values.squeeze(), lags=48, ax=ax)
plot_pacf(eth_month.box_diff2[13:].values.squeeze(), lags=12, ax=ax)
plt.tight_layout()
plt.show()
```



There are not many spikes in the plots outside the insignificant zone (shaded) so there may not be enough information available in the residuals to be extracted by AR and MA models.

There may be a seasonal component available in the residuals at the lags of quarters (3 months) represented by spikes at these intervals. But probably not significant.

ARIMA Model

AutoRegressive Integrated Moving Average

ARIMA models are denoted with the notation ARIMA(p, d, q). These parameters account for seasonality, trend, and noise in datasets:

```
p - the number of lag observations to include in the model, or lag order. (AR)
d - the number of times that the raw observations are differenced, or the
degree of differencing. (I)
q - the size of the moving average window, also called the order of moving
average.(MA)
```

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model. A value of 0 for a parameter indicates to not use that element of the model.

Parameter Selection

We will iteratively explore different combinations of parameters. For each combination we fit a new ARIMA model with SARIMAX() and assess its overall quality.

We will use the AIC (Akaike Information Criterion) value, returned with ARIMA models fitted using statsmodels. The AIC measures how well a model fits the data while taking into account the overall complexity of the model. A model that fits the data very well while using lots of features will be assigned a larger AIC score than a model that uses fewer features to achieve the same goodness-of-fit. Therefore, we are interested in finding the model that yields the lowest AIC value.

In [15]:
```
# Initial approximation of parameters
qs = range(0, 3)
ps = range(0, 3)
d=1
parameters = product(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 = SARIMAX(eth_month.close_box, order=(param[0], d, param[1])).fit(disp=-1)
    except ValueError:
        print('bad parameter combination:', param)
        continue
    aic = model.aic
    if aic < best_aic:
        best_model = model
        best_aic = aic
        best_param = param
    results.append([param, model.aic])
```

In [ ]:
```
```

Note that some parameter combinations may lead to numerical misspecifications and we explicitly disabled warning messages in order to avoid an overload of warning messages. These misspecifications can also lead to errors and throw an exception, so we catch these exceptions and just print out the parameter combinations that cause these issues.

In [16]:
```
# Best Models
result_table = pd.DataFrame(results)
result_table.columns = ['parameters', 'aic']
print(result_table.sort_values(by = 'aic', ascending=True).head())
```

```
  parameters         aic
1     (0, 1) -31.768080
4     (1, 1) -30.138384
2     (0, 2) -30.132382
3     (1, 0) -29.833002
6     (2, 0) -28.736864
```

Note the AICs are negative but this is not a problem.

Usually, AIC is positive; however, it can be shifted by any additive constant, and some shifts can result in negative values of AIC. [...] It is not the absolute size of the AIC value, it is the relative values over the set of models considered, and particularly the differences between AIC values, that are important.

Ref: Model Selection and Multi-model Inference: A Practical Information-theoretic Approach (Burnham and Anderson, 2004)

In [17]:
```
print(best_model.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:              close_box   No. Observations:                   66
Model:               SARIMAX(0, 1, 1)   Log Likelihood                17.884
Date:                Sun, 16 Apr 2023   AIC                          -31.768
Time:                        19:31:04   BIC                          -27.419
Sample:                    11-30-2017   HQIC                         -30.052
                         - 04-30-2023
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ma.L1          0.3955      0.119      3.315      0.001       0.162       0.629
sigma2         0.0337      0.006      5.224      0.000       0.021       0.046
===================================================================================
Ljung-Box (L1) (Q):                   0.14   Jarque-Bera (JB):                 0.33
Prob(Q):                              0.71   Prob(JB):                         0.85
Heteroskedasticity (H):               0.38   Skew:                            -0.10
Prob(H) (two-sided):                  0.03   Kurtosis:                         2.71
===================================================================================

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

Analysis of Results

The coef column shows the weight (i.e. importance) of each feature and how each one impacts the time series. The P>|z| column informs us of the significance of each feature weight. Here, each weight has a p-value lower or close to 0.05, so it is reasonable to retain all of them in our model.

When fitting seasonal ARIMA models (and any other models for that matter), it is important to run model diagnostics to ensure that none of the assumptions made by the model have been violated. The plot_diagnostics object allows us to quickly generate model diagnostics and investigate for any unusual behavior.

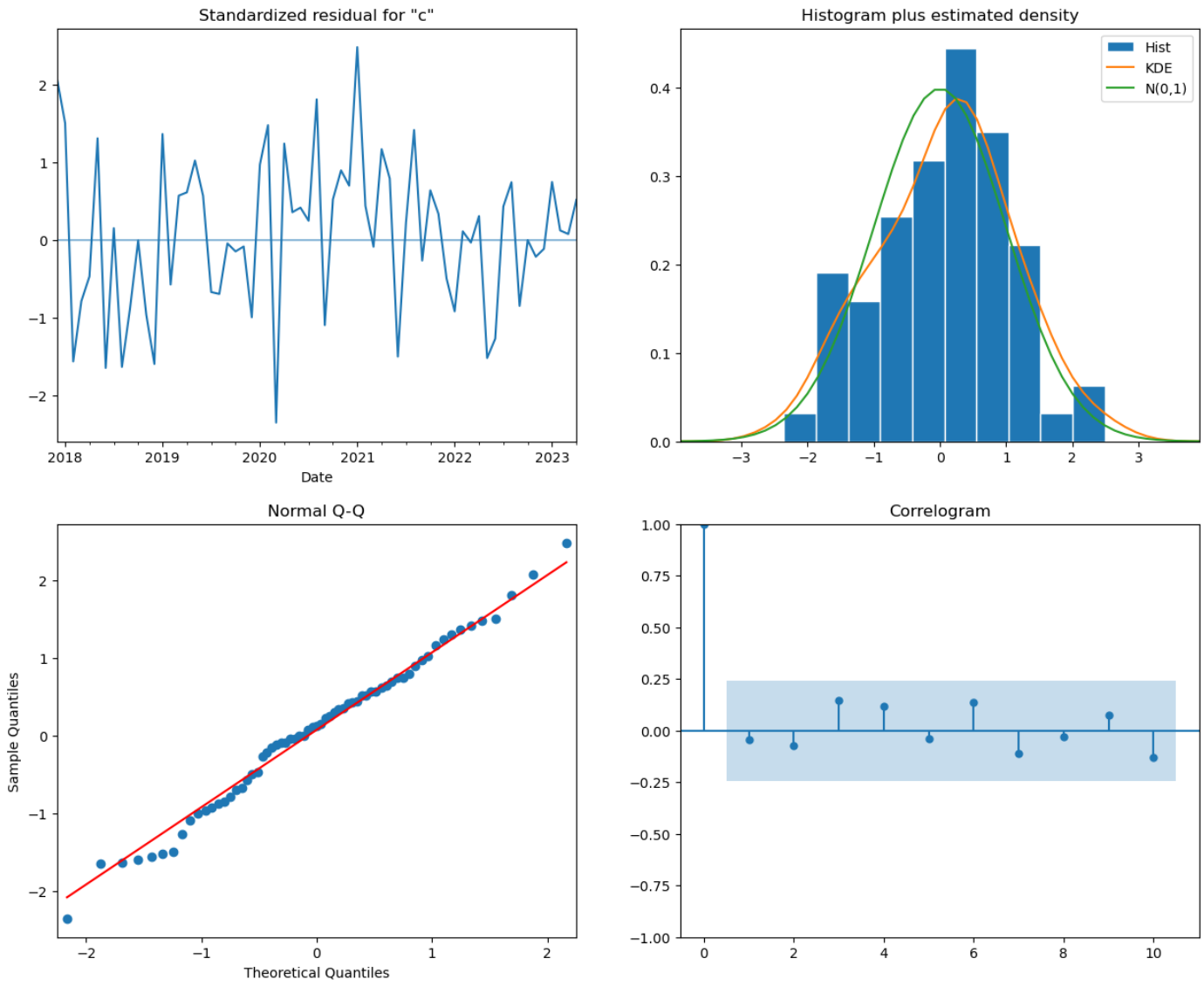
In [18]:
```
print("Dickey-Fuller test:: p=%f" % adfuller(best_model.resid[13:])[1])
```

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

In [ ]:
```

```

In [19]:
```
best_model.plot_diagnostics(figsize=(15, 12))
plt.show()
```



Our primary concern is to ensure that the residuals of our model are uncorrelated and normally distributed with zero-mean. If the seasonal ARIMA model does not satisfy these properties, it is a good indication that it can be further improved.

> In the histogram (top right), the KDE line should follow the N(0,1) line (normal distribution with mean 0, standard deviation 1) closely. This is an indication whether the residuals are normally distributed or not.
>
> In the Q-Q-plot the ordered distribution of residuals (blue dots) should follow the linear trend of the samples taken from a standard normal distribution with N(0, 1). Again, this is an indication whether the residuals are normally distributed.
>
> The standardized residual plot doesn't display any obvious seasonality. This is confirmed by the autocorrelation plot, which shows that the time series residuals have low correlation with lagged versions of itself.

Conclusion: We may consider trying to standardise the distribution further. But lets go ahead and do a prediction anyway...

Prediction

In [20]:
```
# Inverse Box-Cox Transformation Function
def invboxcox(y,lmbda):
   if lmbda == 0:
      return(np.exp(y))
   else:
      return(np.exp(np.log(lmbda*y+1)/lmbda))
```

In [21]:
```
eth_month
```

Out[21]:

Date	Open	High	Low	Close	Adj Close	Volume	close_box	box_diff_season
2017-11-30	373.696317	393.111908	363.283634	379.732093	379.732093	1.225341e+09	5.308340	
2017-12-31	630.583997	667.252580	596.163133	640.209291	640.209291	2.576202e+09	5.719587	
2018-01-31	1093.099893	1163.799714	1024.934606	1103.646004	1103.646004	5.277749e+09	6.139622	
2018-02-28	882.527006	917.850394	825.723679	873.116318	873.116318	2.978337e+09	5.959982	
2018-03-31	640.787129	653.875259	606.506935	625.761325	625.761325	1.732780e+09	5.701788	
...	...	...	...	...	...	...	...	
2022-12-31	1240.294394	1255.038866	1221.342147	1237.105890	1237.105890	5.197763e+09	6.226558	-0.8
2023-01-31	1454.913763	1486.601736	1434.767791	1466.950026	1466.950026	7.256547e+09	6.355624	-0.5
2023-02-28	1623.919765	1654.276873	1595.175406	1624.605630	1624.605630	7.991048e+09	6.432533	-0.4
2023-03-31	1668.488836	1712.653257	1635.392877	1675.357851	1675.357851	9.757034e+09	6.455650	-0.4
2023-04-30	1900.191414	1938.581299	1879.125748	1916.605385	1916.605385	8.883953e+09	6.556426	-0.3

66 rows × 10 columns

In [22]:
```
eth_month.close_box
```

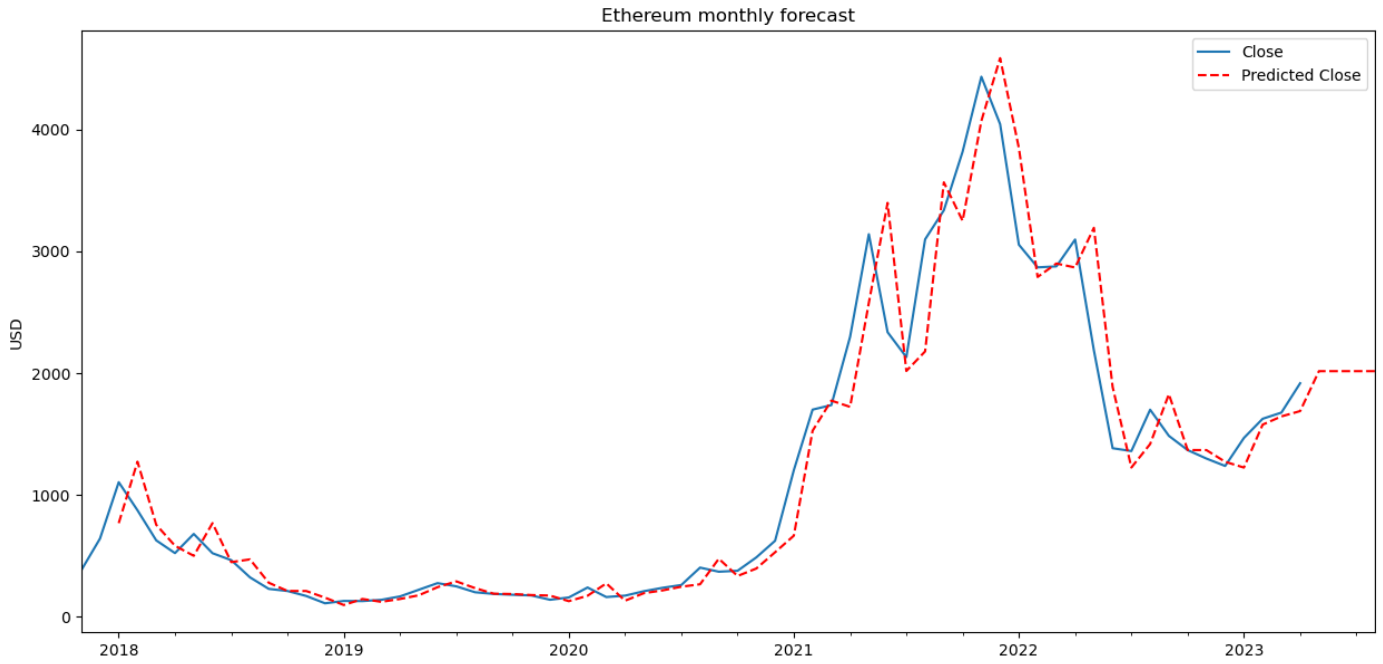
Out[22]:
```
Date
2017-11-30    5.308340
2017-12-31    5.719587
2018-01-31    6.139622
2018-02-28    5.959982
2018-03-31    5.701788
                ...
2022-12-31    6.226558
2023-01-31    6.355624
2023-02-28    6.432533
2023-03-31    6.455650
2023-04-30    6.556426
Freq: M, Name: close_box, Length: 66, dtype: float64
```

In [23]:
```
# Prediction
eth_month_pred = eth_month[['Close']]
#date_list = [datetime(2018, 3, 31), datetime(2018, 4, 30), datetime(2018, 5, 31), datet
#             datetime(2018, 7, 31), datetime(2018, 8, 31), datetime(2018, 9, 30), datet
```

```
#              datetime(2018, 11, 30), datetime(2018, 12, 31)]
date_list = [datetime(2023, 5, 31), datetime(2023, 6, 30), datetime(2023, 7, 31), dateti
future = pd.DataFrame(index=date_list, columns= eth_month.columns)
eth_month_pred = pd.concat([eth_month_pred, future])

#eth_month_pred['forecast'] = invboxcox(best_model.predict(start=0, end=75), lmbda)
eth_month_pred['forecast'] = invboxcox(best_model.predict(start=datetime(2018, 1, 31), e


plt.figure(figsize=(15,7))
eth_month_pred.Close.plot()
eth_month_pred.forecast.plot(color='r', ls='--', label='Predicted Close')
plt.legend()
plt.title('Ethereum monthly forecast')
plt.ylabel('USD')
plt.show()
```



In [24]:
```
eth_month_pred.forecast.tail()
```

Out[24]:
```
2023-04-30    1687.912220
2023-05-31    2015.730684
2023-06-30    2015.730684
2023-07-31    2015.730684
2023-08-31    2015.730684
Name: forecast, dtype: float64
```

Analysis of Results

Prediction

Validation

A simple indicator of how accurate out forecast is is the root mean square error (RMSE). So lets calcualte RMSE for the one-step ahead predictions starting from 2018, through to the end of 2022.

In [109…
```
y_forecasted = eth_month_pred.forecast
y_truth = eth_month_pred['2018-1-31':'2022-11-29'].Close

# Compute the root mean square error
```

```
mse = np.sqrt(((y_forecasted - y_truth) ** 2).mean())
print('Mean Squared Error:',mse.round(2))
```

```
Mean Squared Error: 332.76
```

FB Prophet Model

In [26]:
```
from prophet import Prophet
from prophet.diagnostics import cross_validation, performance_metrics
from prophet.plot import add_changepoints_to_plot, plot_cross_validation_metric
```

In [27]:
```
data= pd.read_csv("ETH-USD.csv", parse_dates=['Date'],date_parser=dateparse)
data
```

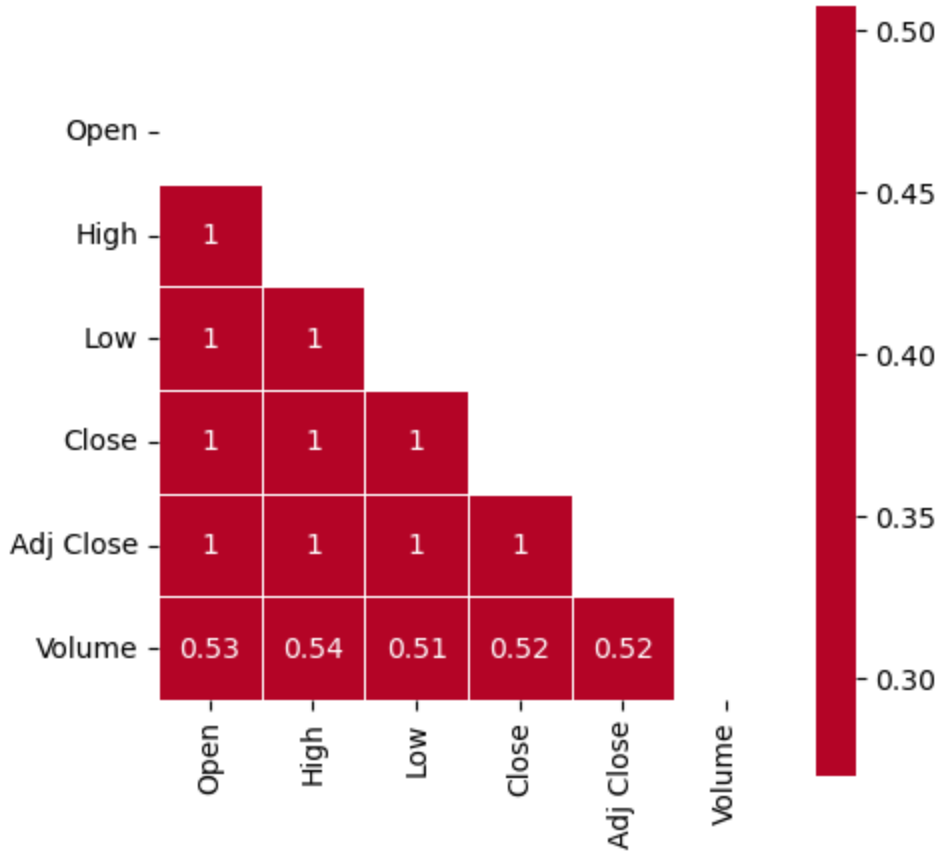
Out[27]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2017-11-09	308.644989	329.451996	307.056000	320.884003	320.884003	893249984
1	2017-11-10	320.670990	324.717987	294.541992	299.252991	299.252991	885985984
2	2017-11-11	298.585999	319.453003	298.191986	314.681000	314.681000	842300992
3	2017-11-12	314.690002	319.153015	298.513000	307.907990	307.907990	1613479936
4	2017-11-13	307.024994	328.415009	307.024994	316.716003	316.716003	1041889984
...	...	...	...	...	...	...	...
1980	2023-04-12	1891.949707	1929.881226	1860.036865	1920.682129	1920.682129	11010714187
1981	2023-04-13	1917.698364	2022.150146	1901.860352	2012.634644	2012.634644	12546950499
1982	2023-04-14	2013.930664	2126.316650	2011.503296	2101.635498	2101.635498	16298099411
1983	2023-04-15	2101.616455	2111.075439	2076.510742	2092.466797	2092.466797	8036468153
1984	2023-04-16	2091.595215	2102.360840	2080.340576	2087.440186	2087.440186	6878934528

1985 rows × 7 columns

In [ ]:
```
```

In [28]:
```
plt.figure(figsize=(5,5))
corr=data[data.columns[1:]].corr()
mask = np.triu(np.ones_like(corr, dtype=bool))
sns.heatmap(data[data.columns[1:]].corr(), mask=mask, cmap='coolwarm', vmax=.3, center=0
            square=True, linewidths=.5,annot=True)
plt.show()
```



In [29]:
```
prophet_df=data[['Date','Close']]
prophet_df.rename(columns={'Date':'ds','Close':'y'},inplace=True)
```

In [30]:
```
prophet_basic = Prophet()
prophet_basic.fit(prophet_df[['ds','y']])
```

```
INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to ove
rride this.
```

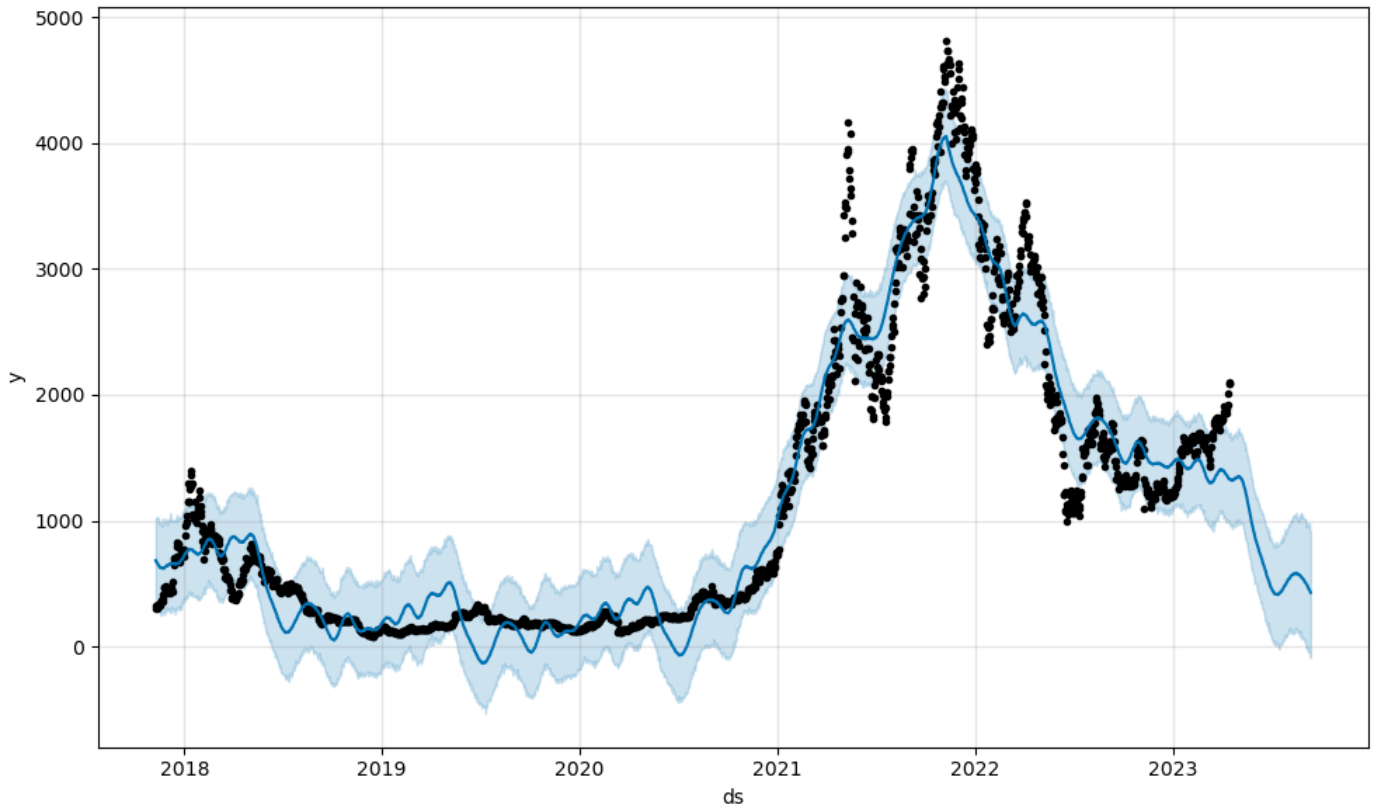
Out[30]:
```
<prophet.forecaster.Prophet at 0x14ce9456310>
```

In [31]:
```
future= prophet_basic.make_future_dataframe(periods=149)
future.tail(2)
```

Out[31]:

	ds
2132	2023-09-11
2133	2023-09-12

In [32]:
```
forecast=prophet_basic.predict(future)
fig1 =prophet_basic.plot(forecast)
```



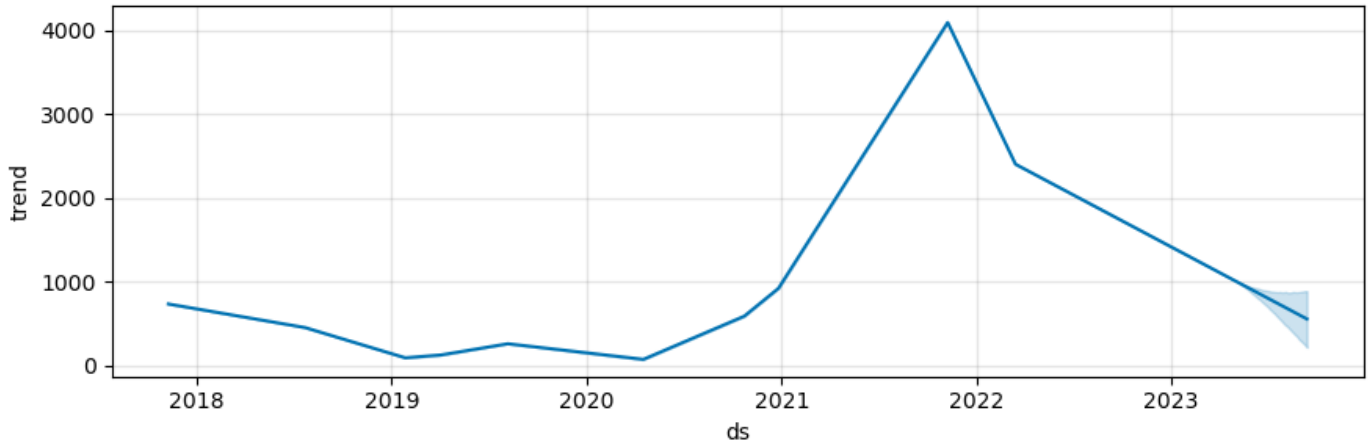
In [33]:
```
forecast
```

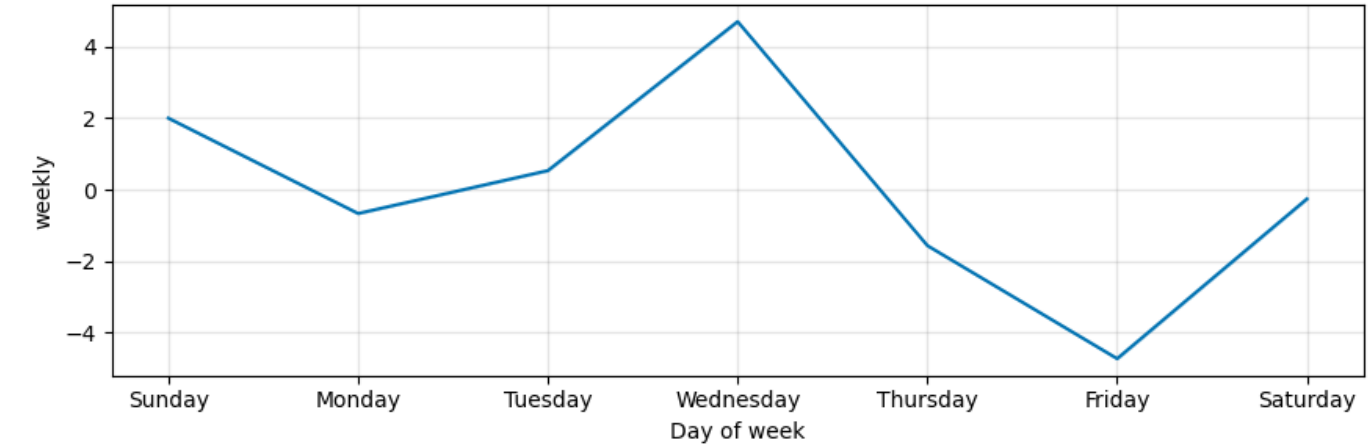
Out[33]:

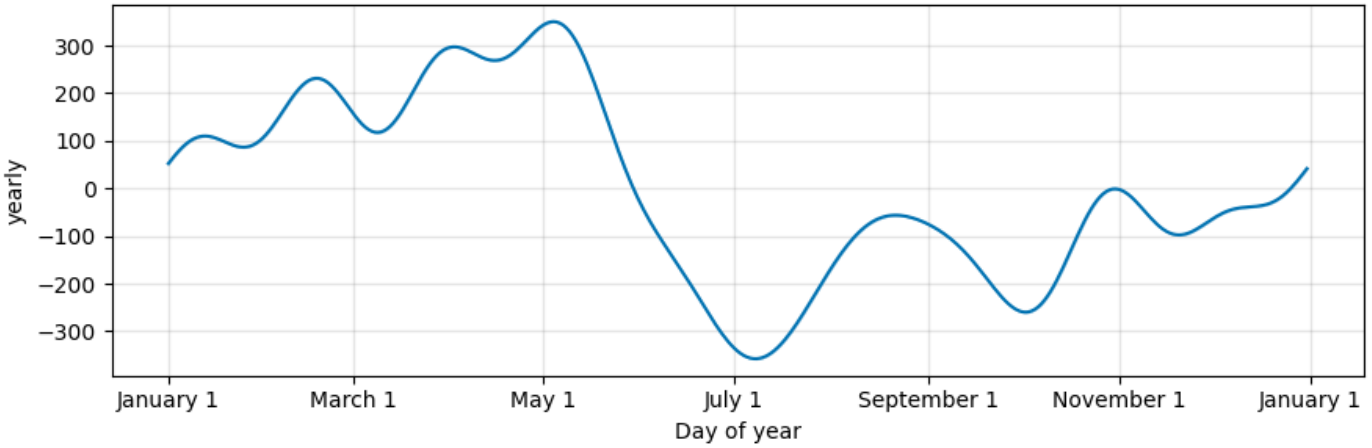
	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	additive_terms_lower
0	2017-11-09	733.233321	335.207904	1020.937388	733.233321	733.233321	-48.409276	-48.409276
1	2017-11-10	732.140618	300.825691	1024.736657	732.140618	732.140618	-58.993966	-58.993966
2	2017-11-11	731.047915	291.610583	1021.256777	731.047915	731.047915	-61.768953	-61.768953
3	2017-11-12	729.955213	293.782230	1031.022829	729.955213	729.955213	-66.407125	-66.407125
4	2017-11-13	728.862510	314.743229	1021.629868	728.862510	728.862510	-75.474866	-75.474866
...	...	...	...	...	...	...	...	...
2129	2023-09-08	568.008510	-48.073816	971.157839	244.505699	888.488551	-107.334279	-107.334279
2130	2023-09-09	564.625000	-37.887012	954.568370	237.217334	888.116370	-108.195392	-108.195392
2131	2023-09-10	561.241491	-45.632068	982.662659	233.041394	886.491171	-111.607722	-111.607722
2132	2023-09-11	557.857981	-87.803148	920.429952	223.503990	886.961078	-120.295827	-120.295827
2133	2023-09-12	554.474472	-85.366821	906.430698	215.457852	887.124748	-125.476379	-125.476379

2134 rows × 19 columns

In [34]:
```
fig1 = prophet_basic.plot_components(forecast)
```







Now lets split the dataset and add remaining features to Prophet model which exhibits functionality of Polynomial regression

In [35]:
```
prophet_df['Open'] = data['Open']
prophet_df['High'] = data['High']
prophet_df['Low'] = data['Low']
prophet_df['Vol'] = data['Volume']
#prophet_df['Change'] = data['Change %']
prophet_df=prophet_df.dropna()
train_X= prophet_df[:1500]
test_X= prophet_df[:]
```

In [36]:
```
prophet_df
```

Out[36]:

	ds	y	Open	High	Low	Vol

0	2017-11-09	320.884003	308.644989	329.451996	307.056000	893249984
1	2017-11-10	299.252991	320.670990	324.717987	294.541992	885985984
2	2017-11-11	314.681000	298.585999	319.453003	298.191986	842300992
3	2017-11-12	307.907990	314.690002	319.153015	298.513000	1613479936
4	2017-11-13	316.716003	307.024994	328.415009	307.024994	1041889984
...	...	...	...	...	...	...
1980	2023-04-12	1920.682129	1891.949707	1929.881226	1860.036865	11010714187
1981	2023-04-13	2012.634644	1917.698364	2022.150146	1901.860352	12546950499
1982	2023-04-14	2101.635498	2013.930664	2126.316650	2011.503296	16298099411
1983	2023-04-15	2092.466797	2101.616455	2111.075439	2076.510742	8036468153
1984	2023-04-16	2087.440186	2091.595215	2102.360840	2080.340576	6878934528

1985 rows × 6 columns

In [37]:
```
pro_regressor= Prophet()
pro_regressor.add_regressor('Open')
pro_regressor.add_regressor('High')
pro_regressor.add_regressor('Low')
pro_regressor.add_regressor('Vol')
```

Out[37]:
```
<prophet.forecaster.Prophet at 0x14ce8cb0070>
```

In [38]:
```
pro_regressor.fit(train_X)
```

```
INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to ove
rride this.
```

Out[38]:
```
<prophet.forecaster.Prophet at 0x14ce8cb0070>
```

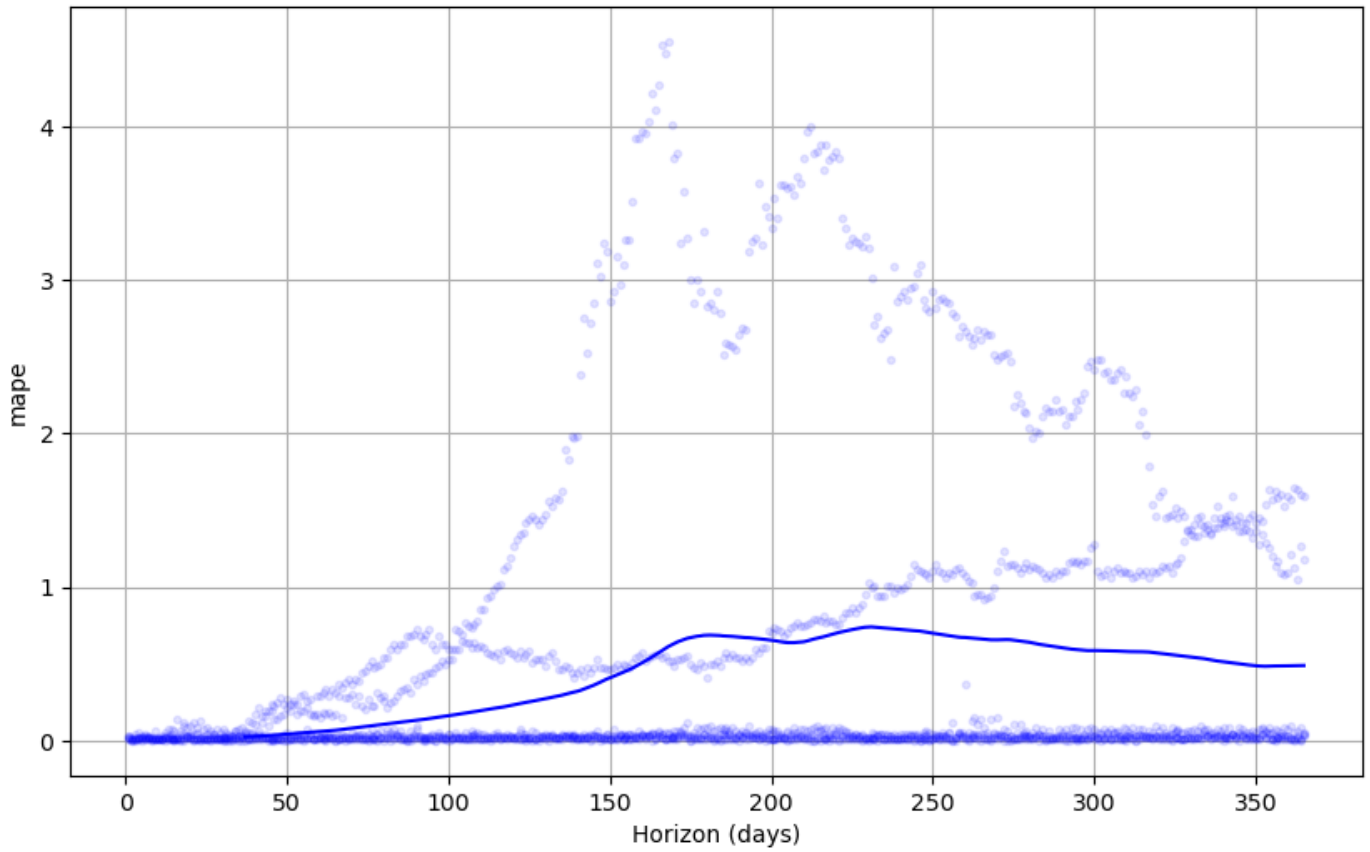
In [39]:
```
forecast_data = pro_regressor.predict(test_X)
pro_regressor.plot(forecast_data);
```

In [40]:
```
df_cv = cross_validation(pro_regressor, initial='100 days', period='180 days', horizon =
pm = performance_metrics(df_cv, rolling_window=0.1)
display(pm.head(),pm.tail())
fig = plot_cross_validation_metric(df_cv, metric='mape', rolling_window=0.1)
plt.show()
```

```
INFO:prophet:Making 6 forecasts with cutoffs between 2018-07-01 00:00:00 and 2020-12-17
00:00:00
WARNING:prophet:Seasonality has period of 365.25 days which is larger than initial windo
w. Consider increasing initial.
  0%|          | 0/6 [00:00<?, ?it/s]
```

	horizon	mse	rmse	mae	mape	mdape	smape	coverage
0	37 days	196.280897	14.010028	7.763468	0.023019	0.015556	0.022675	0.885845
1	38 days	202.407612	14.227003	7.979446	0.023685	0.015556	0.023377	0.881279
2	39 days	222.868832	14.928792	8.345828	0.024552	0.015957	0.024268	0.876712
3	40 days	238.182962	15.433177	8.715885	0.025736	0.016081	0.025512	0.863014
4	41 days	241.905497	15.553311	8.896330	0.026906	0.016123	0.026785	0.860731

	horizon	mse	rmse	mae	mape	mdape	smape	coverage
324	361 days	32701.985150	180.836902	127.700367	0.486298	0.048025	0.489405	0.292237
325	362 days	32396.764600	179.991013	126.293990	0.486494	0.046231	0.488415	0.296804
326	363 days	32238.162293	179.549888	125.991979	0.486685	0.046605	0.487696	0.296804
327	364 days	32201.879572	179.448822	125.708590	0.487395	0.046605	0.487408	0.296804
328	365 days	32159.023795	179.329372	125.913799	0.487787	0.046605	0.487271	0.292237


mape
0
1
2
3
4
0
50
100
150
200
250
300
350
Horizon (days)