SBI Stock Closing Price Forecast with Uncertainity

By Edagotti Naresh,P.Alipeer,K.Prashanth

In [1]:
```
## required packages
import numpy as np
import pandas as pd
import datetime as dt
import matplotlib.pyplot as plt
import pandas_datareader.data as pdr
import plotly.graph_objects as go
import tensorflow as tf
import tensorflow_probability as tfp
import statsmodels.tsa.stattools as sts_tools
import statsmodels as sm
import statsmodels.api
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from math import sqrt
from scipy.stats import norm
from tensorflow_probability import sts
from statsmodels.tsa.stattools import adfuller
```

Load Data

In [2]:
```
## dates for which stock data will be collected
start = dt.datetime(2017, 6, 1)
end = dt.datetime(2022, 6, 1)
```

In [3]:
```
## State bank of India stocks
sbi= pdr.get_data_yahoo("SBIN.NS", start = start, end = end, interval='d')
sbi
```

Out[3]:

	High	Low	Open	Close	Volume	Adj Close
Date						
2017-06-01	291.399994	284.600006	287.950012	287.450012	12186700.0	280.443756
2017-06-02	290.500000	286.350006	289.899994	287.049988	12004368.0	280.053497
2017-06-05	289.750000	286.750000	288.049988	287.250000	7917410.0	280.248627
2017-06-06	292.950012	286.600006	292.000000	287.299988	12346121.0	280.297394
2017-06-07	291.500000	287.200012	288.200012	290.549988	10864355.0	283.468201
...	...	...	...	...	...	...
2022-05-26	470.100006	452.500000	456.850006	468.899994	17055257.0	468.899994
2022-05-27	475.000000	467.500000	471.399994	468.950012	10977001.0	468.950012
2022-05-30	476.899994	471.100006	473.000000	474.600006	9365470.0	474.600006
2022-05-31	476.399994	465.000000	474.000000	468.100006	15441579.0	468.100006
2022-06-01	472.000000	464.700012	468.000000	468.299988	9424008.0	468.299988

1236 rows × 6 columns

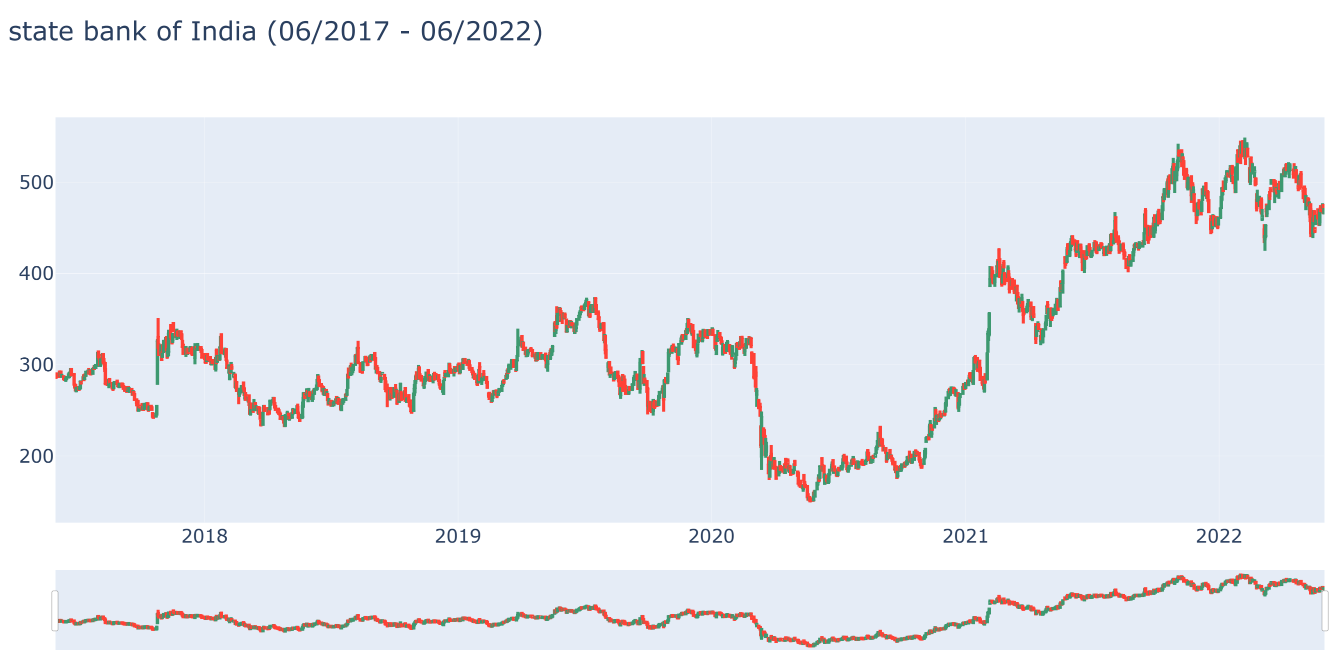
Description:

- **Open** is the price of the stock at the beginning of the trading day (it need not be the closing price of the previous trading day).
- **High** is the highest price of the stock on that trading day.
- **Low** is the lowest price of the stock on that trading day.
- **Close** is the price of the stock at closing time. The closing price is the 'raw' price which is just the cash value of the last transacted price before the market closes.
- **Volume** indicates how many stocks were traded.
- **Adj Close** is the updated stock closing price that accurately reflects the stock's value after accounting for any corporate actions. It is considered to be the true price of that stock and is often used when examining historical returns or performing a detailed analysis of historical returns.

Exploratory Analysis

Candlestick Plot

In [4]:
```
candlestick = go.Candlestick(
    x = sbi.index,
    open = sbi['Open'],
    high = sbi['High'],
    low = sbi['Low'],
    close = sbi['Close']
)
fig = go.Figure(data=[candlestick])
fig.update_layout(title_text='state bank of India (06/2017 - 06/2022)', xaxis_rangeslider_visible=True)
fig.show()
```

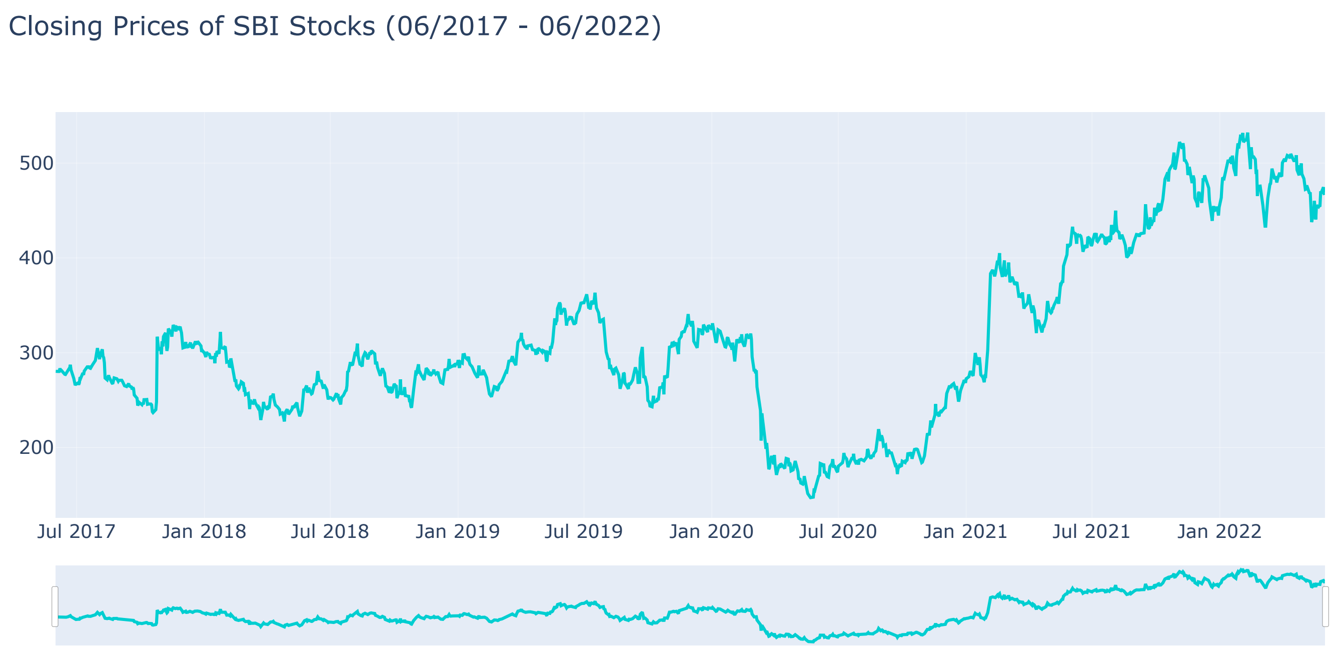


In the above candlestick plots:

- green candlestick indicates a day where the closing price was higher than the open (a gain)
- red candlestick indicates a day where the open was higher than the close (a loss)

Trends of Close Prices of the different Corporations

In [5]:
```
fig1 = go.Figure()
fig1.add_trace(go.Scatter(x=sbi.index, y=sbi['Adj Close'], line_color='darkturquoise'))
fig1.update_layout(title_text='Closing Prices of SBI Stocks (06/2017 - 06/2022)', xaxis_rangeslider_visible=True)
fig1.show()
```



SBI had a relatively consistent growth until 2018, followed by an increase in the closing prices mid 2018, which was then followed by a dip around year end, a slight increase and then another dip. It seems like these stock closing prices might be following mean reversion.

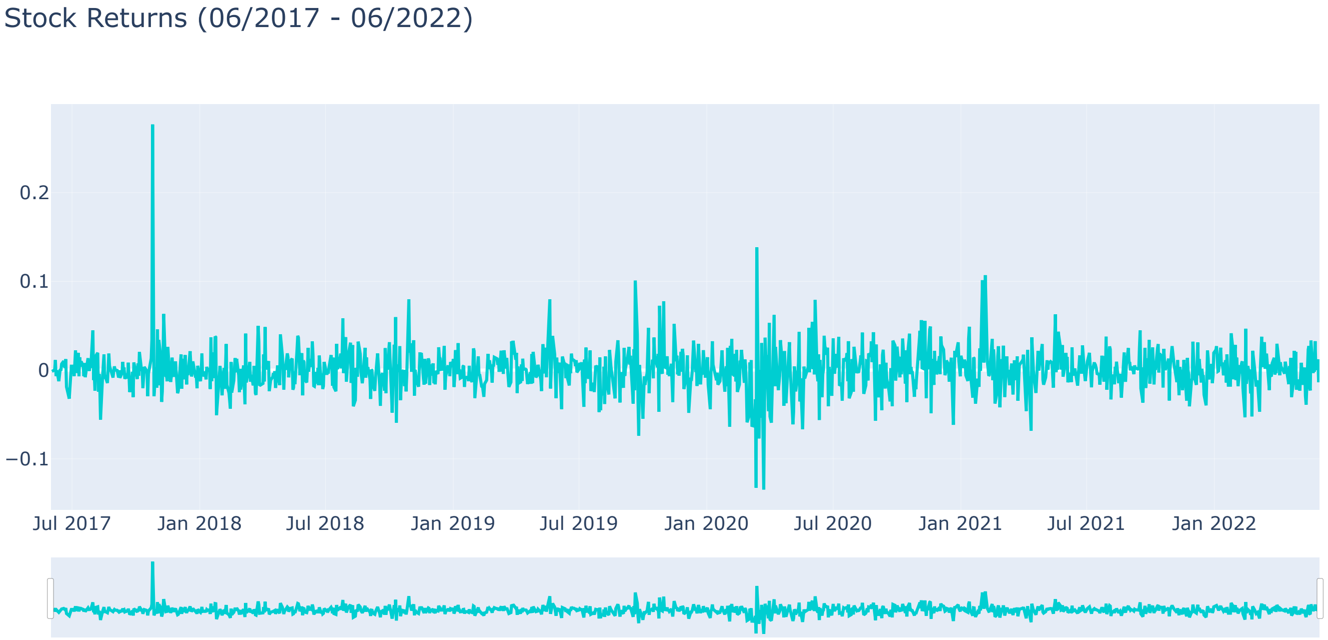
Trends of Stock Returns of the different Corporations

When trading, we are more concerned about the relative change of an asset rather than its absolute price.

In [6]:
```
## Daily returns in closing price
def add_daily_change_column(df, col_name):
    df['Daily Returns'] = df[col_name].pct_change()
    return df

amzn = add_daily_change_column(sbi, 'Adj Close')

fig2 = go.Figure()
fig2.add_trace(go.Scatter(x=sbi.index, y=sbi['Daily Returns'], name="SBI", line_color='darkturquoise'))
fig2.update_layout(title_text='Stock Returns (06/2017 - 06/2022)', xaxis_rangeslider_visible=True)
fig2.show()
```



The relative change in the daily stock prices seems to follow a white noise (stationary distribution). Conducting an augmented dickey-fullter test on this series can confirm if we can assume this is a stationary distribution.

AutoCorrelation in Closing Prices

In [7]:
```
## Augmented Dickey-Fuller Test
adfuller = sts_tools.adfuller(sbi['Daily Returns'][1:], maxlag = 1)
print(adfuller)
## ADF p-value
adfuller[1]
```

```
(-35.473960176480716, 0.0, 0, 1234, {'1%': -3.435660336370594, '5%': -2.863885022214541, '10%': -2.568018522153254}, -5701.734617802278)
```

Out[7]:
```
0.0
```

As the p-value of the ADF hypothesis test is less than 0.05, we have enough statistical evidence to reject the null hypothesis and conclude that the relative change in closing prices is stationary.

Note: The p-value is not exactly 0 but a very very small number, but due to round-off precision error in python it is outputted as 0

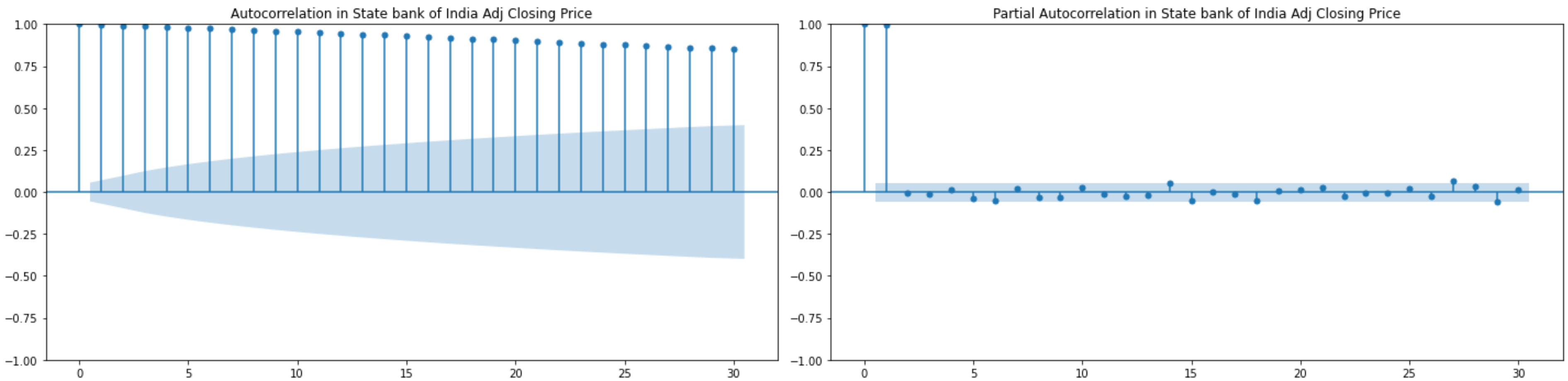
In [8]:
```
max_lags = 30
fig, axes = plt.subplots(1, 2, figsize=(20,5))

plot_acf(sbi['Adj Close'], alpha=0.05, lags=max_lags, ax = fig.axes[0], title = 'Autocorrelation in State bank of India Adj Closing Price')
plot_pacf(sbi['Adj Close'], alpha=0.05, lags=max_lags, ax = fig.axes[1], title = 'Partial Autocorrelation in State bank of India Adj Closing Price')

fig.tight_layout()
```

```
C:\Users\Naresh\AppData\Roaming\Python\Python38\site-packages\statsmodels\graphics\tsaplots.py:348: FutureWarning:

The default method 'yw' can produce PACF values outside of the [-1,1] interval. After 0.13, the default will change tounadjusted Yule-Walker ('ywm'). You can us
e this method now by setting method='ywm'.
```



Autocorrelation is a measure of the correlation in a time-series with a lagged version of itself.

Partial Autocorrelation is similar to autocorrelation, however it removes the effects of previous time points. For instance, for autocorrelation of order=3, parital autocorrelation would remove the effects of lags 1 and 2. Hence, partial autocorrelation can provide us with the order of the AutoRegressive Model for the time-series, as it tells us on how many previous steps does the current value depend on.

Seasonality Trends

In [9]:
```
plt.rcParams['figure.figsize'] = 20, 10

res = sm.tsa.seasonal.seasonal_decompose(sbi['Adj Close'], model='additive', period=5) ## weekly seasonality
res.plot()
plt.show()
```

Modeling

In [10]:
```
SEED = 684

## Build structural time series model
## Includes components for local linear trend, weekly seasonality and autoregressive model
def build_model(observed_time_series,lags,observation_noise_scale_prior):
    linear_trend = tfp.sts.LocalLinearTrend(
        observed_time_series = observed_time_series
    )

    dayofweek_season = sts.Seasonal(
        num_seasons = 5,
        num_steps_per_season = 1,
        observed_time_series = observed_time_series,
        constrain_mean_effect_to_zero = False,
        name = "dayofweek_season"
    )

    autoregressive = sts.Autoregressive(
        order = lags,
        observed_time_series = observed_time_series
    )

    model = sts.Sum(
        [dayofweek_season, linear_trend, autoregressive],
        observed_time_series = observed_time_series,
        observation_noise_scale_prior = observation_noise_scale_prior
    )
    return model

## To fit the model to data, we define a surrogate posterior and
## fit it by minimizing the negative variational evidence lower bound (ELBO)
def fit_variational_posterior(model, variational_posteriors, observed_time_series):
    elbo_loss = tfp.vi.fit_surrogate_posterior(
        target_log_prob_fn = model.joint_log_prob(observed_time_series = observed_time_series),
        surrogate_posterior = variational_posteriors,
        optimizer = tf.optimizers.Adam(learning_rate = 0.1),
        num_steps = 100,
        seed = SEED
    )
    return elbo_loss

## Plot ELBO curve
def plot_elbo_loss(elbo_loss):
    fig = go.Figure(data=go.Scatter(x = np.arange(len(elbo_loss)), y = elbo_loss))
    fig.update_layout(title_text='ELBO Plot')
    fig.update_xaxes(title_text='Number of Iterations')
    fig.update_yaxes(title_text='-ve ELBO')
    fig.show()
## Forecast for `num_forecast_steps` days
def forecast(model, samples, observed_time_series, num_forecast_steps):
    forecast_dist = tfp.sts.forecast(
        model,
        observed_time_series = observed_time_series,
        parameter_samples = samples,
```

```
        num_steps_forecast = num_forecast_steps
    )
    forecast_mean = forecast_dist.mean().numpy().flatten()
    forecast_std = forecast_dist.stddev().numpy().flatten()
    return {'mean': forecast_mean, 'scale': forecast_std}

## Forecast the next 20 days using one-step-prediction which will give
## predictive distribution over observations at each time T, given observations up through time T-1.
def forecast_onestep_prediction(model, samples, observed_time_series, num_forecast_steps):
    forecast_dist = tfp.sts.one_step_predictive(
        model,
        observed_time_series = observed_time_series,
        parameter_samples = samples
    )
    forecast_mean = forecast_dist.mean().numpy()[-num_forecast_steps:]
    forecast_std = forecast_dist.stddev().numpy()[-num_forecast_steps:]
    return {'mean': forecast_mean, 'scale': forecast_std}

## Plot Forecast
def plot_forecast(actual_dates, actual, forecast_dates, prediction, prediction_uncertainity, title='Forecast'):
    prediction_lb = prediction - 1.96*prediction_uncertainity
    prediction_ub = prediction + 1.96*prediction_uncertainity
    fig = go.Figure()
    fig.add_trace(go.Scatter(
        x=actual_dates,
        y=actual,
        name='Ground Truth',
        mode='lines+markers',
        line_color='darkturquoise'
    ))
    fig.add_trace(go.Scatter(
        x=forecast_dates,
        y=prediction,
        name='Forecast',
        mode='lines+markers',
        line_color='orange'
    ))
    fig.add_trace(go.Scatter(
        x = forecast_dates,
        y = prediction_ub,
        fill = None,
        mode='lines',
        line_color='orange',
        showlegend=False,
        name='Forecast UB'
    ))
    fig.add_trace(go.Scatter(
        x = forecast_dates,
        y = prediction_lb,
        fill='tonexty',
        mode='lines',
        line_color='orange',
        showlegend=False,
        name='Forecast LB'
```

```
    ))
    fig.update_layout(title_text=title, xaxis_rangeslider_visible=True)
    fig.show()

## Extract inferred value of model Parameters
def extract_model_params(model, samples):
    model_params = {}
    for param in model.parameters:
        model_params[param.name] = {
            "point_estimate": np.mean(samples[param.name], axis=0),
            "uncertainity": np.std(samples[param.name], axis=0)
        }
    return pd.DataFrame.from_dict(model_params).T
```

In [29]:

```
## Number of days to be forecasted
num_forecast_steps = 20

## Extract training data
sbi_data = sbi['Adj Close']
sbi_train_data = sbi_data[:-num_forecast_steps]
sbi_train_data_len = len(sbi_train_data)
print(sbi_train_data)

## Extract noise scale prior for observation noise from the closing price using daily returns
noise_scale = sbi['Daily Returns'].std()
#noise_scale_prior = tfp.distributions.Normal(np.float64(0.0), np.float64(1.0))
data=tfp.sts.regularize_series(sbi_train_data,frequency=None)
sbi_train_data=data
data1=tfp.sts.regularize_series(sbi_data,frequency=None)
sbi_data=data1
```

```
Date
2017-06-01    280.443756
2017-06-02    280.053497
2017-06-05    280.248627
2017-06-06    280.297394
2017-06-07    283.468201
                 ...
2022-04-27    489.609100
2022-04-28    499.701599
2022-04-29    488.673676
2022-05-02    483.455139
2022-05-04    472.279541
Name: Adj Close, Length: 1216, dtype: float64
```

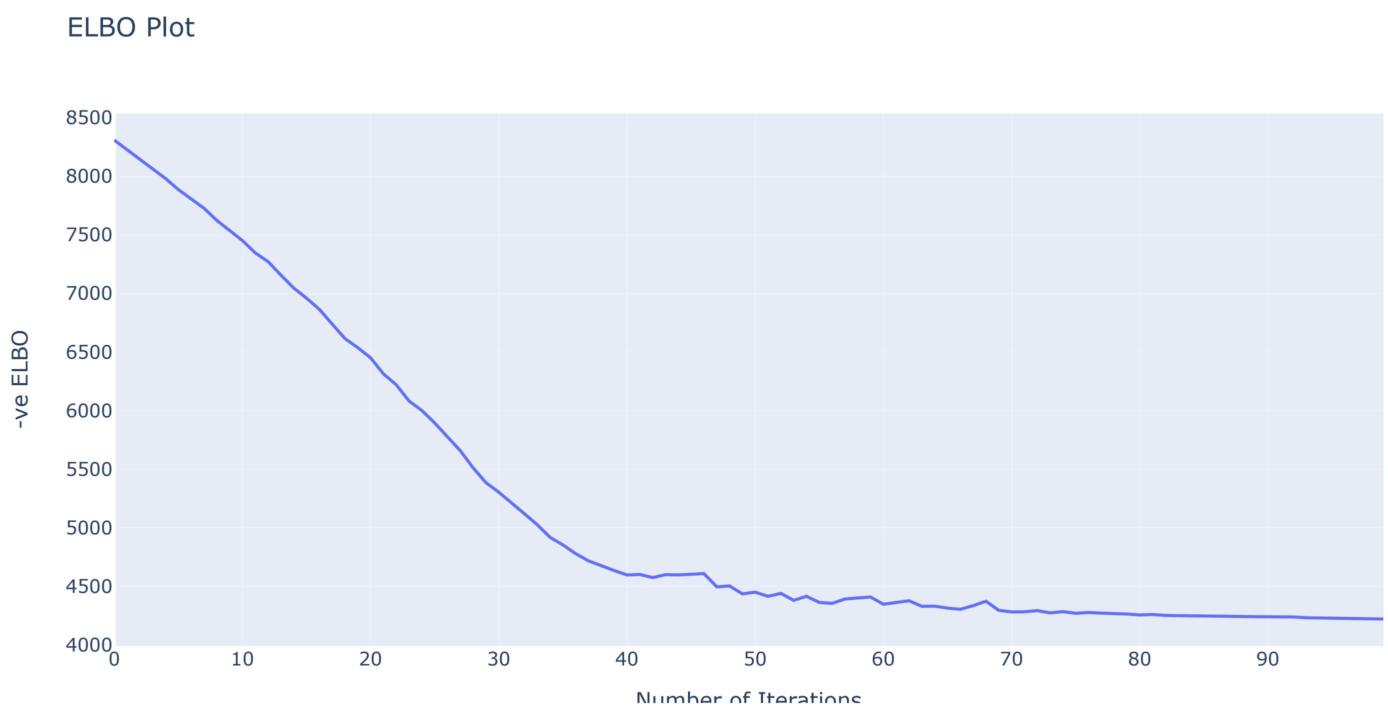
In [12]:
```
## Build Model
model_lag1=build_model(sbi_train_data,1,None)

## Build variational surrogate posterior
variational_posteriors_lag1 = tfp.sts.build_factored_surrogate_posterior(model = model_lag1, seed = SEED)

## To fit the model to data, we define a surrogate posterior and
## fit it by minimizing the negative variational evidence lower bound (ELBO)
elbo_loss_lag1 = fit_variational_posterior(model_lag1, variational_posteriors_lag1,sbi_train_data)

## Plot ELBO
plot_elbo_loss(elbo_loss_lag1)
```

```
WARNING:tensorflow:From C:\Users\Naresh\AppData\Local\Temp\ipykernel_13880\59531330.py:34: StructuralTimeSeries.joint_log_prob (from tensorflow_probability.pyth
on.sts.structural_time_series) is deprecated and will be removed after 2022-03-01.
Instructions for updating:
Please use `StructuralTimeSeries.joint_distribution(observed_time_series).log_prob`
WARNING:tensorflow:From C:\Users\Naresh\AppData\Roaming\Python\Python38\site-packages\tensorflow_probability\python\distributions\distribution.py:342: calling M
ultivariateNormalDiag.__init__ (from tensorflow_probability.python.distributions.mvn_diag) with scale_identity_multiplier is deprecated and will be removed afte
r 2020-01-01.
Instructions for updating:
`scale_identity_multiplier` is deprecated; please combine it into `scale_diag` directly instead.
```



Number of iterations

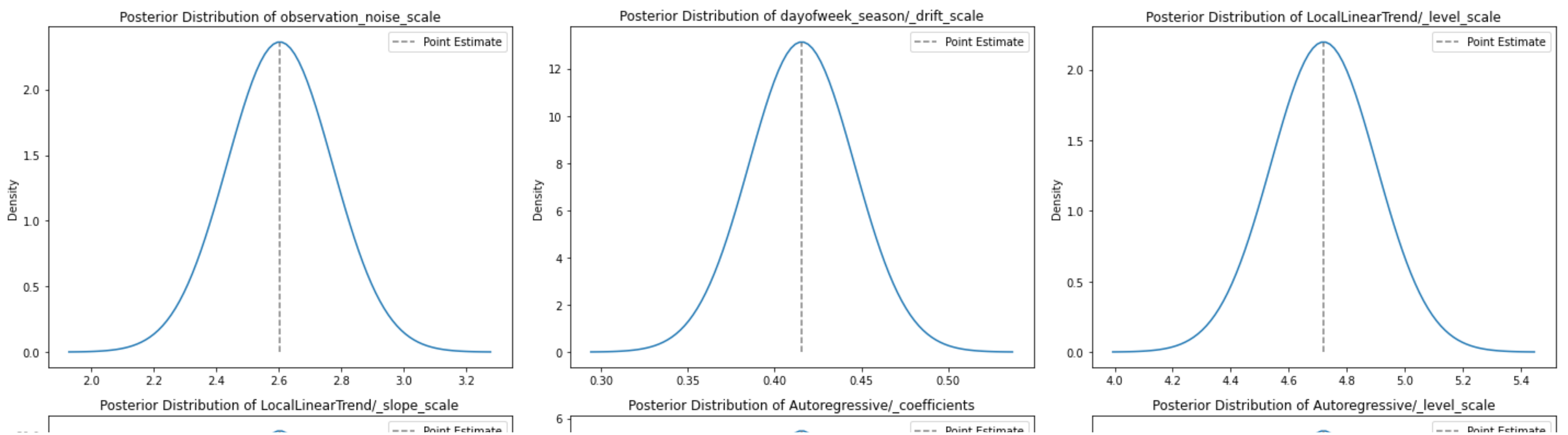
In [13]:
```
## Draw samples from posterior
samples_lag1 = variational_posteriors_lag1.sample(1000)
```

In [14]:
```
model_params = extract_model_params(model_lag1, samples_lag1)
model_params
```

Out[14]:

	point_estimate	uncertainity
observation_noise_scale	2.604042	0.168864
dayofweek_season/_drift_scale	0.415597	0.030381
LocalLinearTrend/_level_scale	4.719924	0.181585
LocalLinearTrend/_slope_scale	0.630676	0.019666
Autoregressive/_coefficients	[0.9010763714290918]	[0.06912388344112749]
Autoregressive/_level_scale	1.932526	0.188139

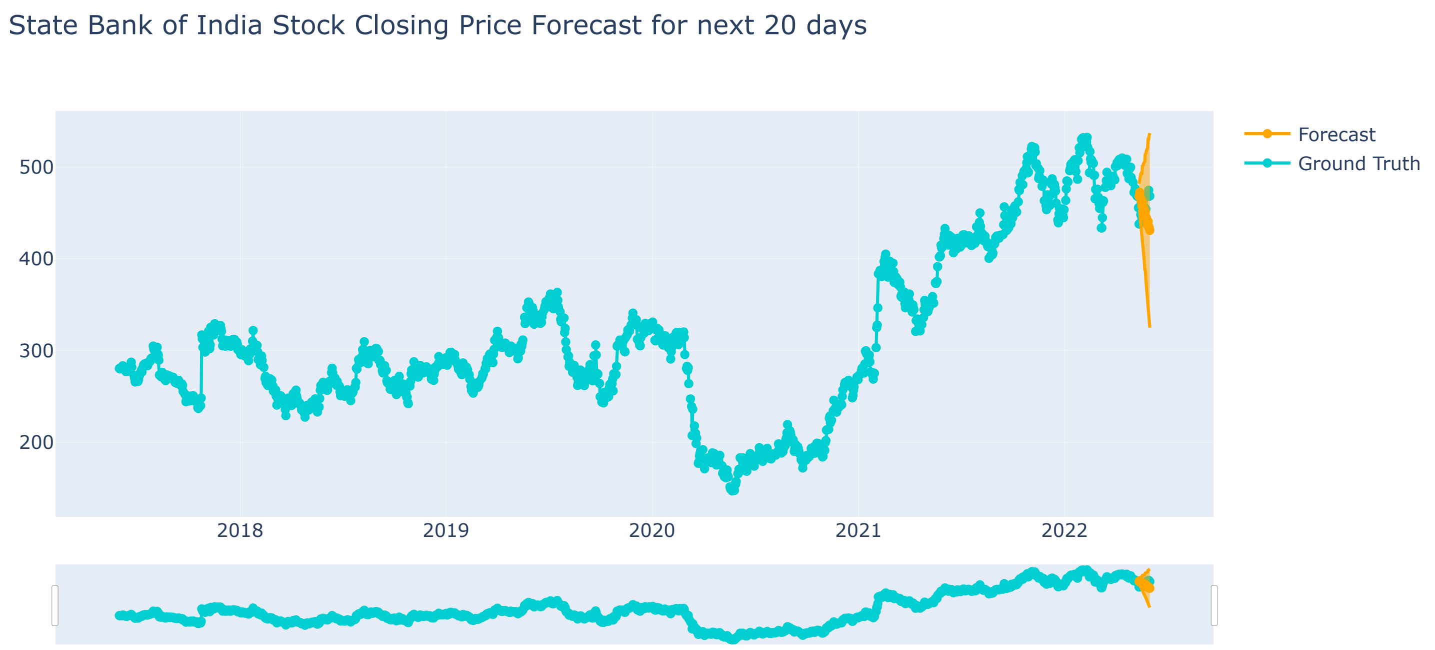
In [15]:
```
fig, axes = plt.subplots(2, 3)
for param, loc, scale, ax in zip(model_params.index, model_params['point_estimate'], model_params['uncertainity'], fig.axes):
    x = np.linspace(loc - 4*scale, loc + 4*scale, 100)
    y = norm.pdf(x, loc, scale)
    ax.plot(x, y)
    ax.vlines(loc, 0, max(y), linestyles = 'dashed', color = 'grey', label='Point Estimate')
    ax.set_title('Posterior Distribution of {}'.format(param))
    ax.set_ylabel('Density')
    ax.legend()
fig.tight_layout()
```



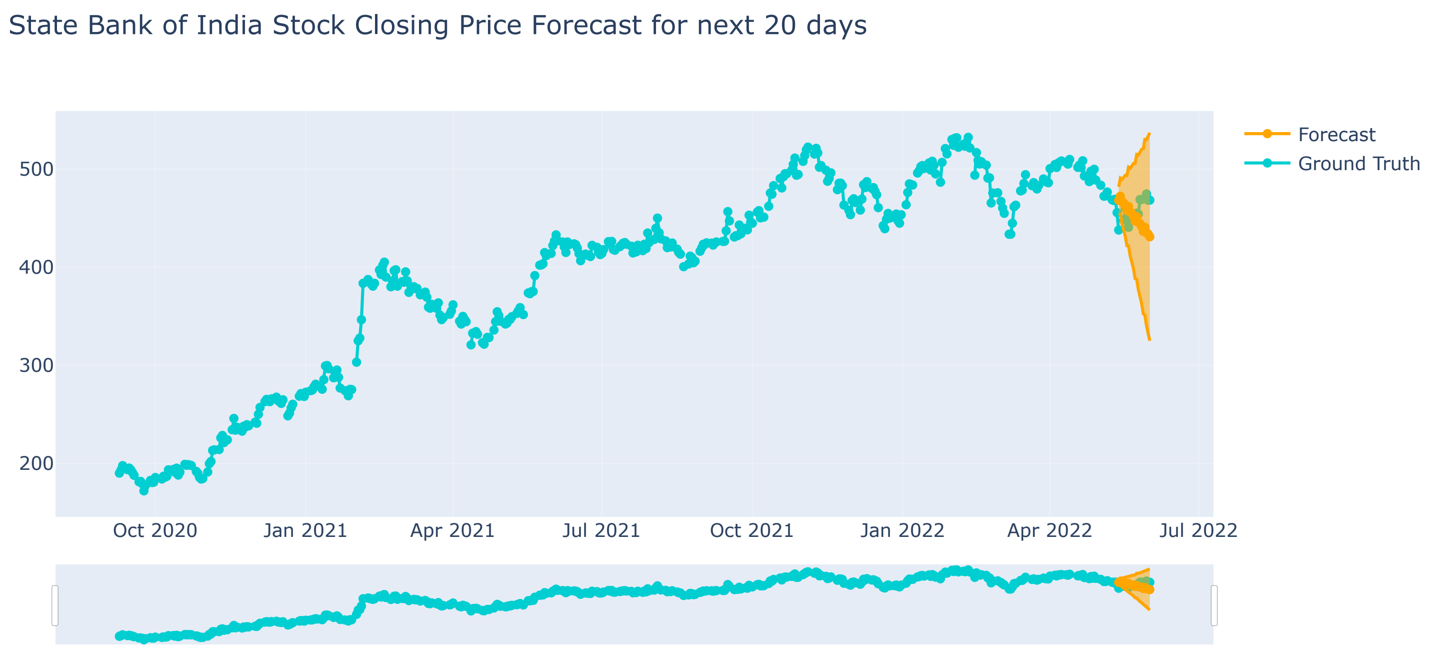
Forecasting

In [ ]:
```
## Forecast next 20 days
forecast_params_lag1 = forecast(model_lag1, samples_lag1, sbi_train_data, num_forecast_steps)
```

In [17]:
```
## Plot Forecast
actual = sbi_data
actual_dates = actual.index
forecast_dates = actual.index[-num_forecast_steps:]
plot_forecast(
    actual_dates, actual,
    forecast_dates, forecast_params_lag1['mean'], forecast_params_lag1['scale'],
    'State Bank of India Stock Closing Price Forecast for next 20 days'
)
```



In [18]:
```
## Plot Forecast
actual = sbi_data[sbi_train_data_len-20:]
actual_dates = actual.index
forecast_dates = actual.index[-num_forecast_steps:]
plot_forecast(
    actual_dates, actual,
    forecast_dates, forecast_params_lag1['mean'], forecast_params_lag1['scale'],
    ' State Bank of India Stock Closing Price Forecast for next 20 days'
)
```



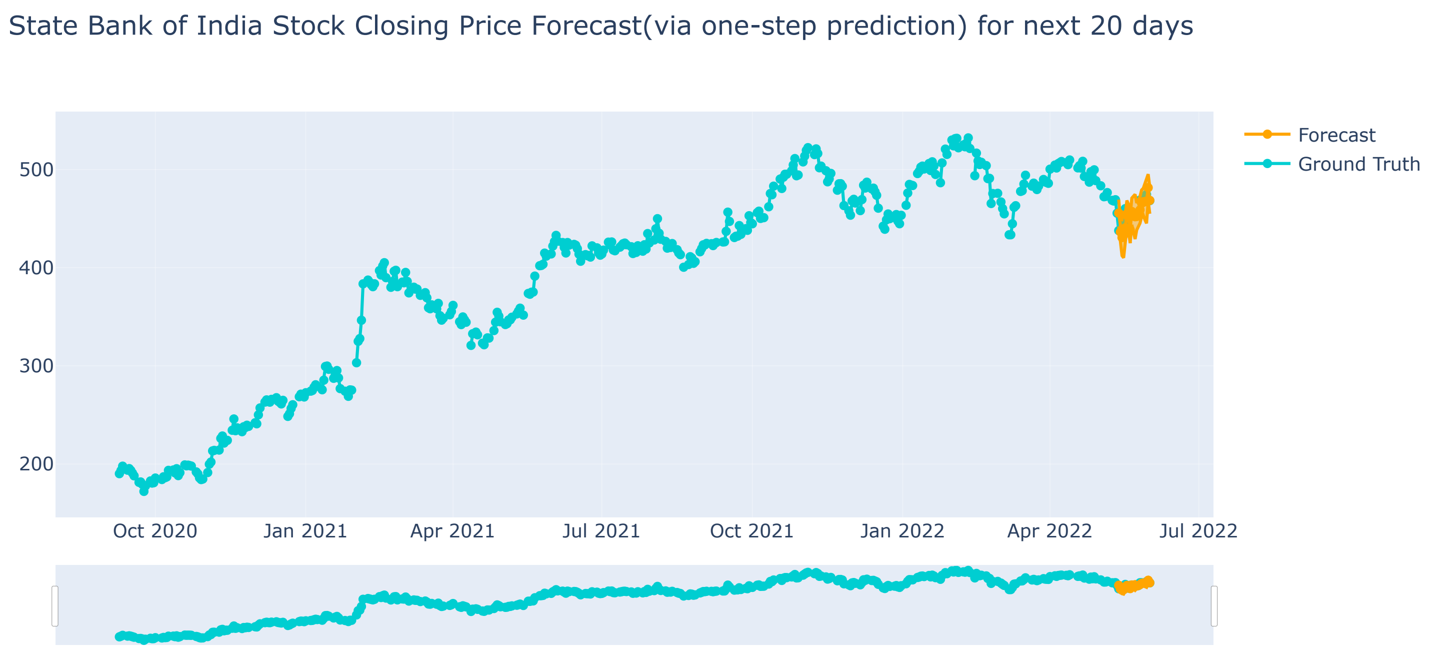
In [27]:
```
## Forecast Estimates and their uncertainty
n_step_forecast = pd.DataFrame(columns = ['Date', 'Actual Value','Forecast Estimate','Forecast Uncertainty'])
n_step_forecast['Date'] = actual[-num_forecast_steps:].index
n_step_forecast['Actual Value'] = actual[-num_forecast_steps:].values
n_step_forecast['Forecast Estimate'] = forecast_params_lag1['mean']
n_step_forecast['Forecast Uncertainty'] = forecast_params_lag1['scale']
n_step_forecast
```

Out[27]:

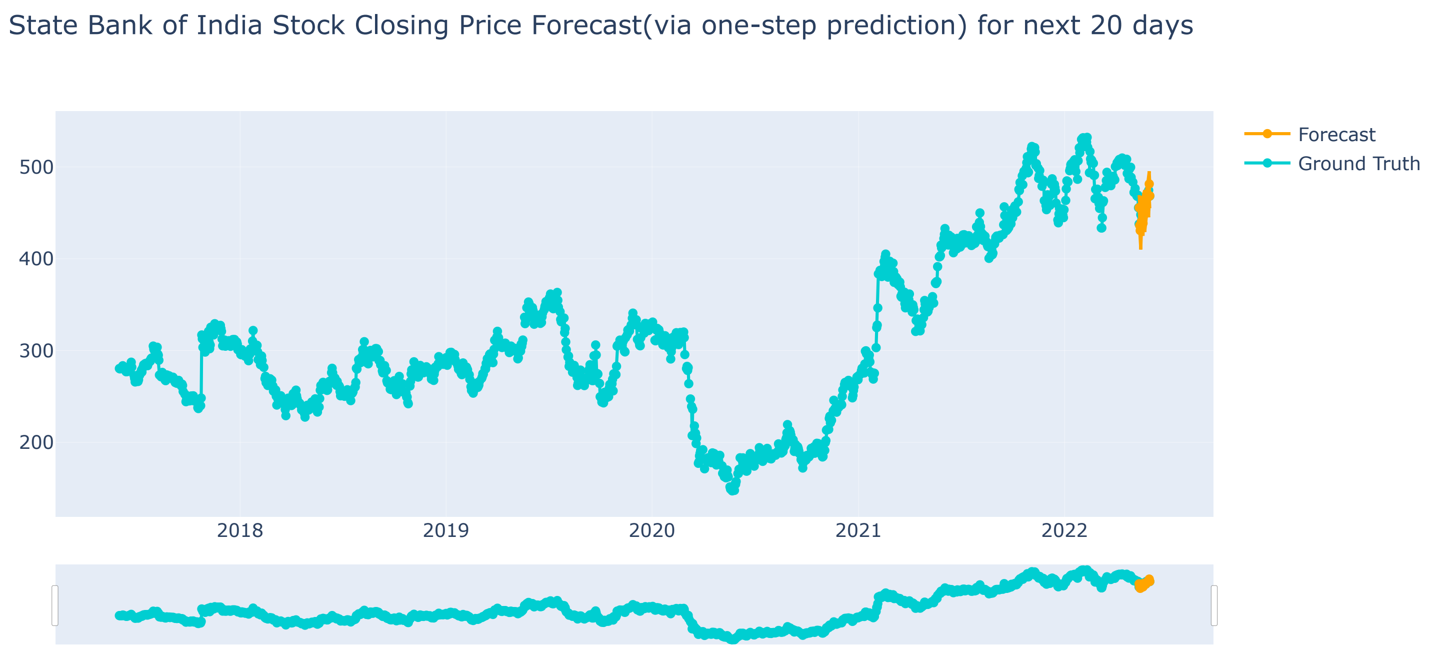
	Date	Actual Value	Forecast Estimate	Forecast Uncertainty
0	2022-05-13	437.817352	468.062793	7.032736
1	2022-05-14	NaN	472.225957	9.411807
2	2022-05-15	NaN	466.643848	11.618172
3	2022-05-16	448.008331	465.382635	13.812337
4	2022-05-17	460.119324	462.579468	15.800871
5	2022-05-18	450.962219	457.594086	18.232837
6	2022-05-19	440.623566	461.739915	20.493222
7	2022-05-20	455.294617	456.143476	22.778561
8	2022-05-21	NaN	454.870259	25.107905
9	2022-05-22	NaN	452.056929	27.335019
10	2022-05-23	453.817657	447.062865	29.938397
11	2022-05-24	454.949982	451.201218	32.437289
12	2022-05-25	454.100006	445.598298	34.978390
13	2022-05-26	468.899994	444.319432	37.559269
14	2022-05-27	468.950012	441.501149	40.066532
15	2022-05-28	NaN	436.502724	42.903487
16	2022-05-29	NaN	440.637220	45.660165
17	2022-05-30	474.600006	435.030877	48.461530
18	2022-05-31	468.100006	433.748960	51.296519
19	2022-06-01	468.299988	430.927950	54.071500

In [28]:
```
## Forecast next 20 days via one-step prediction
onestep_forecast_params_lag1 = forecast_onestep_prediction(model_lag1, samples_lag1, sbi_data, num_forecast_steps)
```

In [21]:
```
## Plot Forecast
actual = sbi_data[sbi_train_data_len-20:]
actual_dates = actual.index
forecast_dates = actual.index[-num_forecast_steps:]
plot_forecast(
    actual_dates, actual,
    forecast_dates, onestep_forecast_params_lag1['mean'], onestep_forecast_params_lag1['scale'],
    'State Bank of India Stock Closing Price Forecast(via one-step prediction) for next 20 days'
)
```



In [22]:
```
## Plot Forecast
actual = sbi_data
actual_dates = actual.index
forecast_dates = actual.index[-num_forecast_steps:]
plot_forecast(
    actual_dates, actual,
    forecast_dates, onestep_forecast_params_lag1['mean'], onestep_forecast_params_lag1['scale'],
    'State Bank of India Stock Closing Price Forecast(via one-step prediction) for next 20 days'
)
```



In [23]:
```
## One step Forecast Estimates and their uncertainty
one_step_forecast = pd.DataFrame(columns = ['Date', 'Actual Value','Forecast Estimate','Forecast Uncertainty'])
one_step_forecast['Date'] = actual[-num_forecast_steps:].index
one_step_forecast['Actual Value'] = actual[-num_forecast_steps:].values
one_step_forecast['Forecast Estimate'] = onestep_forecast_params_lag1['mean']
one_step_forecast['Forecast Uncertainty'] = onestep_forecast_params_lag1['scale']
one_step_forecast
```

Out[23]:

	Date	Actual Value	Forecast Estimate	Forecast Uncertainty
0	2022-05-13	437.817352	455.696930	6.971060
1	2022-05-14	NaN	436.394973	6.985379
2	2022-05-15	NaN	430.715426	9.395971
3	2022-05-16	448.008331	432.839002	11.619119
4	2022-05-17	460.119324	440.582069	7.009992
5	2022-05-18	450.962219	454.817046	6.957615
6	2022-05-19	440.623566	450.808855	6.979675
7	2022-05-20	455.294617	438.647489	6.959781
8	2022-05-21	NaN	457.253819	6.947895
9	2022-05-22	NaN	454.461651	9.369573
10	2022-05-23	453.817657	451.914266	11.613757
11	2022-05-24	454.949982	452.166534	7.029952
12	2022-05-25	454.100006	454.540303	6.944063
13	2022-05-26	468.899994	458.907436	6.949100
14	2022-05-27	468.950012	465.535351	6.947748
15	2022-05-28	NaN	467.183327	6.953854
16	2022-05-29	NaN	467.162645	9.362877
17	2022-05-30	474.600006	467.917078	11.596932
18	2022-05-31	468.100006	481.528619	7.004938
19	2022-06-01	468.299988	468.552714	6.935732

In [24]:
```
## Calculate absolute cumulative error of forecasts
error_values_onestep = pd.Series(
    abs(onestep_forecast_params_lag1['mean'] - actual[-num_forecast_steps:].values)
)
cumsum_onestep = error_values_onestep.cumsum()

error_values = pd.Series(
    abs(forecast_params_lag1['mean'] - actual[-num_forecast_steps:].values)
)
cumsum = error_values.cumsum()
```

In [25]:
```
## Tablular disply of absolute cumulative error of forecast
columns = ['Actual Value','One Step Prediction','N Step Prediction','One Step Error', 'N Step Error']
error_analysis_df = pd.DataFrame(columns= columns)
error_analysis_df['Actual Value'] = actual[-num_forecast_steps:].values
error_analysis_df['One Step Prediction'] = onestep_forecast_params_lag1['mean']
error_analysis_df['N Step Prediction'] = forecast_params_lag1['mean']
error_analysis_df['One Step Error'] = cumsum_onestep
error_analysis_df['N Step Error'] = cumsum
error_analysis_df
```

Out[25]:

	Actual Value	One Step Prediction	N Step Prediction	One Step Error	N Step Error
0	437.817352	455.696930	468.062793	17.879578	30.245441
1	NaN	436.394973	472.225957	NaN	NaN
2	NaN	430.715426	466.643848	NaN	NaN
3	448.008331	432.839002	465.382635	33.048908	47.619745
4	460.119324	440.582069	462.579468	52.586162	50.079889
5	450.962219	454.817046	457.594086	56.440989	56.711756
6	440.623566	450.808855	461.739915	66.626278	77.828106
7	455.294617	438.647489	456.143476	83.273405	78.676965
8	NaN	457.253819	454.870259	NaN	NaN
9	NaN	454.461651	452.056929	NaN	NaN
10	453.817657	451.914266	447.062865	85.176797	85.431758
11	454.949982	452.166534	451.201218	87.960245	89.180522
12	454.100006	454.540303	445.598298	88.400542	97.682230
13	468.899994	458.907436	444.319432	98.393099	122.262792
14	468.950012	465.535351	441.501149	101.807760	149.711655
15	NaN	467.183327	436.502724	NaN	NaN
16	NaN	467.162645	440.637220	NaN	NaN
17	474.600006	467.917078	435.030877	108.490688	189.280784
18	468.100006	481.528619	433.748960	121.919300	223.631830
19	468.299988	468.552714	430.927950	122.172026	261.003868

In [26]:
```
## Plot absolute cumulative error of forecast
fig3 = go.Figure()
fig3.add_trace(go.Scatter(
    x=np.arange(len(cumsum_onestep)),
    y=cumsum_onestep,
    line_color='darkturquoise',
    name = 'One Step Prediction',
    mode='lines+markers'
))
fig3.add_trace(go.Scatter(
    x=np.arange(len(cumsum)),
    y=cumsum, line_color='red',
    name = 'N Step Prediction',
    mode='lines+markers'
))
fig3.update_layout(title_text='Cumulative Absolute Prediction Error')
fig3.update_xaxes(title_text='Days')
fig3.update_yaxes(title_text='Cumulative Error')
fig3.show()
```

One-step-ahead predicitve strategy gives better estimates than N-step.