A Guide to Time Series Forecasting in Python

Time series forecasting is a useful data science technique with applications in a wide range of industries and fields. Here's a guide to getting started with the basic concepts behind it.

Time series forecasting is the task of predicting future values based on historical data. Examples across industries include forecasting of weather, sales numbers and stock prices. More recently, it has been applied to predicting price trends for cryptocurrencies such as Bitcoin and Ethereum. Given the prevalence of time series forecasting applications in many different fields, every data scientist should have some knowledge of the available methods for carrying it out.

A wide array of methods are available for time series forecasting. One of the most commonly used is Autoregressive Moving Average (ARMA), which is a statistical model that predicts future values using past values. This method is flawed, however, because it doesn't capture seasonal trends. It also assumes that the time series data is stationary, meaning that its statistical properties wouldn't change over time. This type of behaviour is an idealized assumption that doesn't hold in practice, however, which means ARMA may provide skewed results.

An extension of ARMA is the Autoregressive Integrated Moving Average (ARIMA) model, which doesn't assume stationarity but does still assume that the data exhibits little to no seasonality. Fortunately, the seasonal ARIMA (SARIMA) variant is a statistical model that can work with nonstationary data and capture some seasonality.

Python provides many easy-to-use libraries and tools for performing time series forecasting. Specifically, the stats library in Python has tools for building ARMA, ARIMA and SARIMA models with just a few lines of code. Since all of these models are available in a single library, you can easily run many experiments using different models in the same script or notebook.

Here, we will look at how to build ARMA, ARIMA and SARIMA models to forecast future prices of Bitcoin (BTC).

TIME SERIES FORECASTING

Time series forecasting is the task of predicting future values based on historical data. Examples across industries include forecasting of weather, sales numbers and stock prices. More recently, it has been applied to predicting price trends for cryptocurrencies such as Bitcoin and Ethereum. Given the prevalence of time series forecasting applications in many different fields, every data scientist should have some knowledge of the available methods for carrying it out.

Reading and Displaying BTC Time Series Data

We will start by reading in the historical prices for BTC using yfinance. Let's install it using a simple pip command in terminal:

pip install yfinance

Let's open up a Python script and import the data-reader:

import yfinance as yfin

Let’s also import the Pandas library itself and relax the display limits on columns and rows:

```
import pandas as pd

pd.set_option('display.max_columns', None)

pd.set_option('display.max_rows', None)
```

We can now import the date-time library, which will allow us to define start and end dates for our data pull:

```
import datetime
```

Now we have everything we need to pull Bitcoin price time series data, let’s collect data.

```
btc = yfin.download(['BTC-USD'], start="2018-01-01",end="2020-12-2")

print(btc.head())
```

Attributes	Adj Close	Close	High	Low	\
Symbols	BTC-USD	BTC-USD	BTC-USD	BTC-USD	
Date					
2018-01-01	13657.200195	13657.200195	14112.200195	13154.700195	
2018-01-02	14982.099609	14982.099609	15444.599609	13163.599609	
2018-01-03	15201.000000	15201.000000	15572.799805	14844.500000	
2018-01-04	15599.200195	15599.200195	15739.700195	14522.200195	
2018-01-05	17429.500000	17429.500000	17705.199219	15202.799805	

Attributes	Open	Volume
Symbols	BTC-USD	BTC-USD
Date		
2018-01-01	14112.200195	1.029120e+10
2018-01-02	13625.000000	1.684660e+10
2018-01-03	14978.200195	1.687190e+10
2018-01-04	15270.700195	2.178320e+10
2018-01-05	15477.200195	2.384090e+10

We see that our data frame contains many columns. Let’s walk through what each of these columns means.

1. Date: This is the index in our time series that specifies the date associated with the price.
2. Close: The last price at which BTC was purchased on that day.
3. Open: The first price at which BTC was purchased on that day.
4. High: The highest price at which BTC was purchased that day.
5. Low: The lowest price at which BTC was purchased that day.
6. Volume: The number of total trades that day.
7. Adj Close: The closing price adjusted for dividends and stock splits.

We’ll use the close price for our forecasting models. Specifically, we will use historical closing BTC prices in order to predict future BTC ones.

```
btc = btc['Close']
```

Let’s write our closing price BTC data to a csv file. This way, we can avoid having to repeatedly pull data using the Pandas data reader.

```
btc.to_csv("btc.csv")
```

Now, let’s read in our csv file and display the first five rows:

```
btc = pd.read_csv("btc.csv")

print(btc.head())
```

```
         Date       BTC-USD
0  2018-01-01  13657.200195
1  2018-01-02  14982.099609
2  2018-01-03  15201.000000
3  2018-01-04  15599.200195
4  2018-01-05  17429.500000
```

In order to use the models provided by the stats library, we need to set the date column to be a data frame index. We also should format that date using the to_datetime method:

```
btc.index = pd.to_datetime(btc['Date'], format='%Y-%m-%d')
```

Let’s display our data frame:

```
del btc['Date']
```

```
                BTC-USD
Date
2018-01-01  13657.200195
2018-01-02  14982.099609
2018-01-03  15201.000000
2018-01-04  15599.200195
2018-01-05  17429.500000
```

Let’s plot our time series data. To do this, let’s import the data visualization libraries Seaborn and Matplotlib:

```
import matplotlib.pyplot as plt import

seaborn as sns
```

Let’s format our visualization using Seaborn:

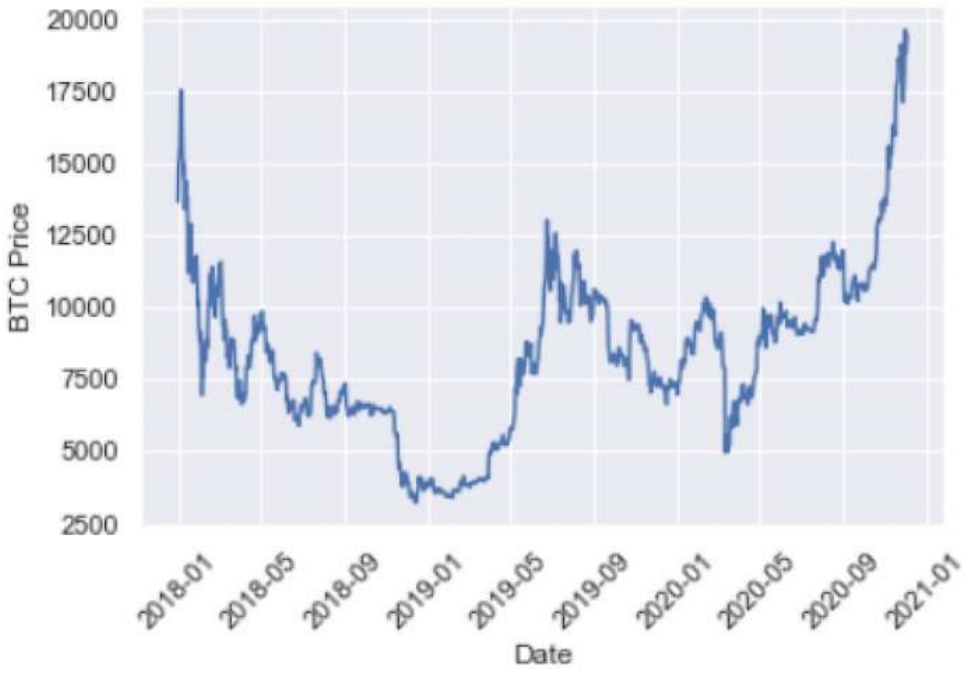
```
sns.set()
```

And label the y-axis and x-axis using Matplotlib. We will also rotate the dates on the x-axis so

that they’re easier to read: plt.ylabel('BTC Price') plt.xlabel('Date') plt.xticks(rotation=45)

And finally, generate our plot with Matplotlib:

```
plt.plot(btc.index, btc['BTC-USD'], )
```



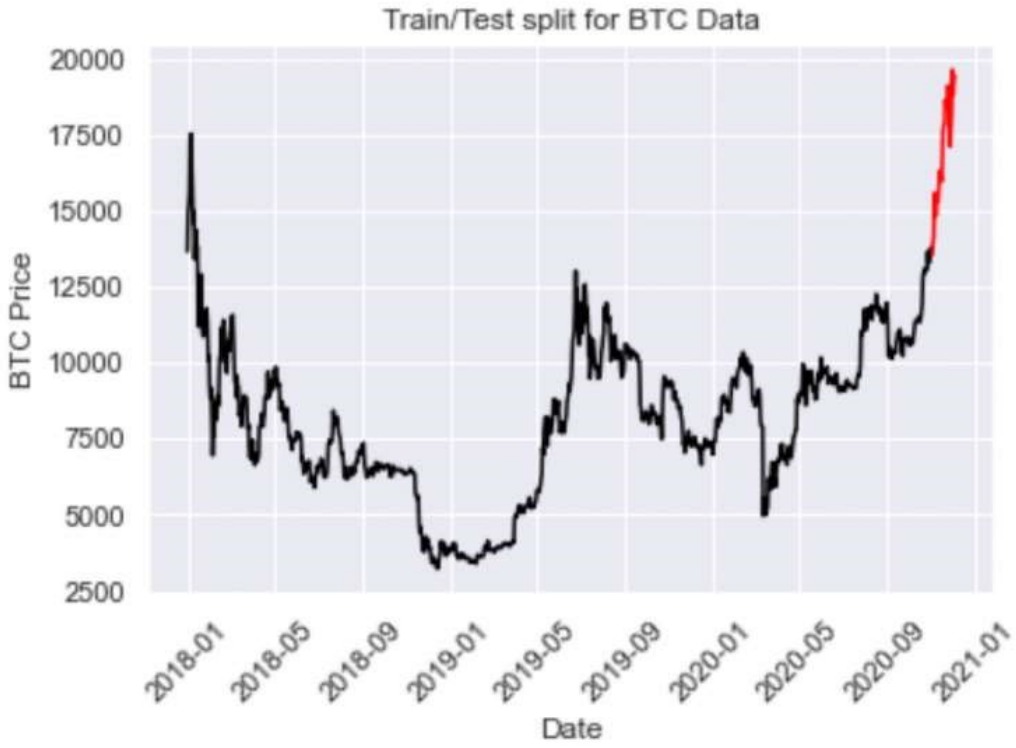
Now we can proceed to building our first time series model, the Autoregressive Moving Average.

Splitting Data for Training and Testing

An important part of model building is splitting our data for training and testing, which ensures that you build a model that can generalize outside of the training data and that the performance and outputs are statistically meaningful.

We will split our data such that everything before November 2020 will serve as training data, with everything after 2020 becoming the testing data:

```
train = btc[btc.index < pd.to_datetime("2020-11-01", format='%Y-%m-%d')]

test = btc[btc.index > pd.to_datetime("2020-11-01", format='%Y-%m-%d')]

plt.plot(train, color = "black") plt.plot(test, color = "red") plt.ylabel('BTC

Price') plt.xlabel('Date') plt.xticks(rotation=45) plt.title("Train/Test split for

BTC Data") plt.show()
```



Autoregressive Moving Average (ARMA)

The term “autoregressive” in ARMA means that the model uses past values to predict future ones. Specifically, predicted values are a weighted linear combination of past values. This type of regression method is similarl to linear regression, with the difference being that the feature inputs here are historical values.

Moving average refers to the predictions being represented by a weighted, linear combination of white noise terms, where white noise is a random signal. The idea here is that ARMA uses a combination of past values and white noise in order to predict future values. Autoregression models market participant behavior like buying and selling BTC. The white noise models shock events like wars, recessions and political events.

We can define an ARMA model using the SARIMAX package:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

Let’s define our input:

```
y = train['BTC-USD']
```

And then let’s define our model. To define an ARMA model with the SARIMAX class, we pass in the order parameters of (1, 0 ,1):

```
ARMAmodel = SARIMAX(y, order = (1, 0, 1))
```

We can then fit our model:

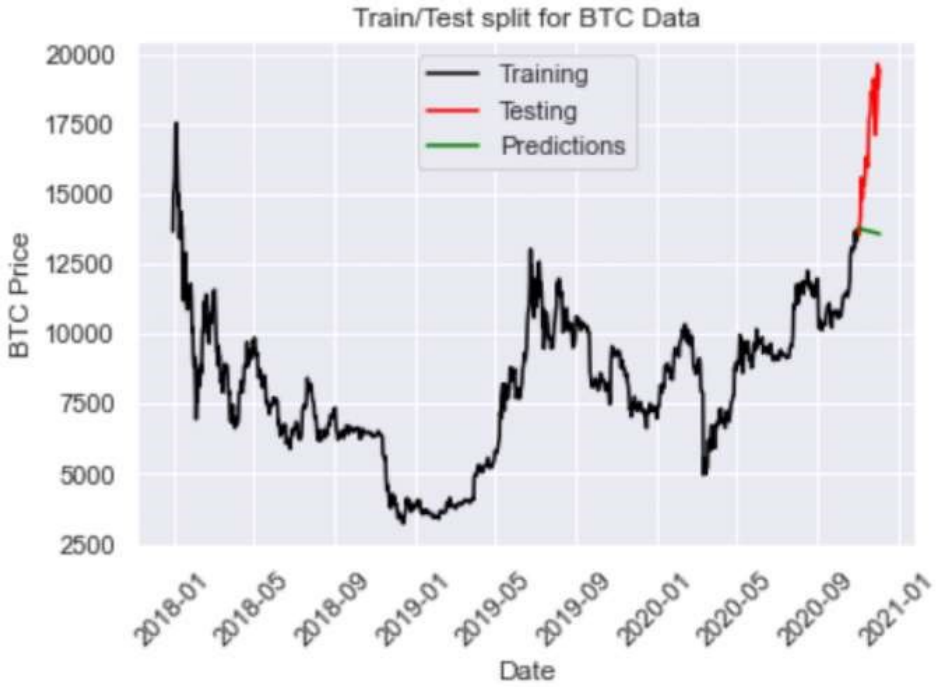
```
ARMAmodel = ARMAmodel.fit()
```

Generate our predictions:

```
y_pred = ARMAmodel.get_forecast(len(test.index))

y_pred_df = y_pred.conf_int(alpha = 0.05)

y_pred_df["Predictions"] = ARMAmodel.predict(start = y_pred_df.index[0], end =
y_pred_df.index[-
1]) y_pred_df.index = test.index

y_pred_out =

y_pred_df["Predictions"]
```

And plot the results: plt.plot(y_pred_out, color='green',

label = 'Predictions') plt.legend()



We can also evaluate the performance using the root mean-squared error:

```
import numpy as np from sklearn.metrics import

mean_squared_error


arma_rmse = np.sqrt(mean_squared_error(test["BTC-USD"].values, y_pred_df["Predictions"]))

print("RMSE: ",arma_rmse)
```

```
RMSE:  3684.5794358524236
```

The RMSE is pretty high, which we could have guessed upon inspecting the plot. Unfortunately, the model predicts a decrease in price when the price actually increases. Again, ARMA is limited in that

it fails for non-stationary time series and does not capture seasonality. Let’s see if we can improve performance with ARIMA

Autoregressive Integrated Moving Average (ARIMA)

Let’s import the ARIMA package from the stats library:

from statsmodels.tsa.arima.model import ARIMA

An ARIMA task has three parameters. The first parameter corresponds to the lagging (past values), the second corresponds to differencing (this is what makes non-stationary data stationary), and the last parameter corresponds to the white noise (for modeling shock events).

Let’s define an ARIMA model with order parameters (2,2,2):

ARIMAmodel = SARIMAX(y, order = (2, 2, 2))

ARIMAmodel = ARIMAmodel.fit() y_pred =

ARIMAmodel.get_forecast(len(test.index))

y_pred_df = y_pred.conf_int(alpha = 0.05)

y_pred_df["Predictions"] = ARIMAmodel.predict(start = y_pred_df.index[0], end = y_pred_df.index[-
1]) y_pred_df.index = test.index y_pred_out =

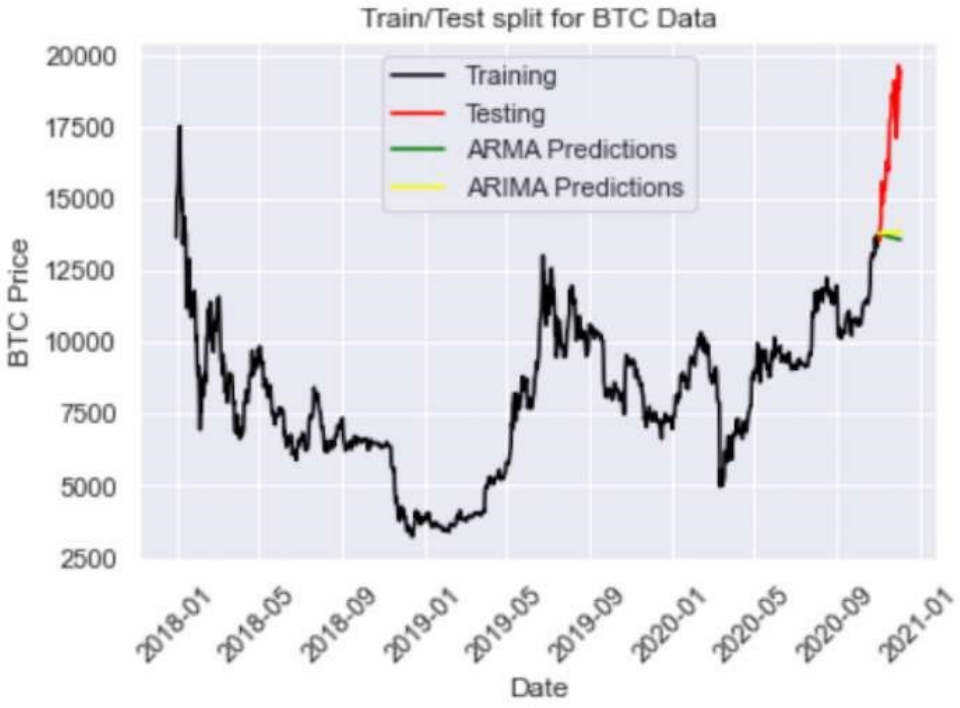
y_pred_df["Predictions"] plt.plot(y_pred_out, color='Yellow',

label = 'ARIMA Predictions') plt.legend()

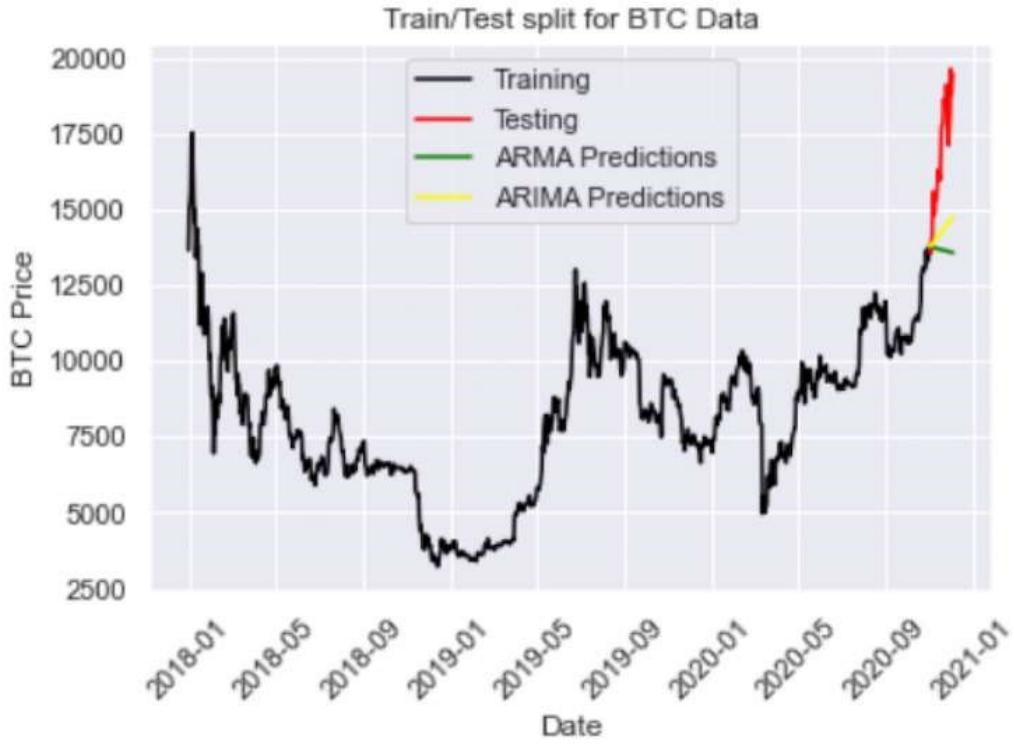
import numpy as np from sklearn.metrics import

mean_squared_error

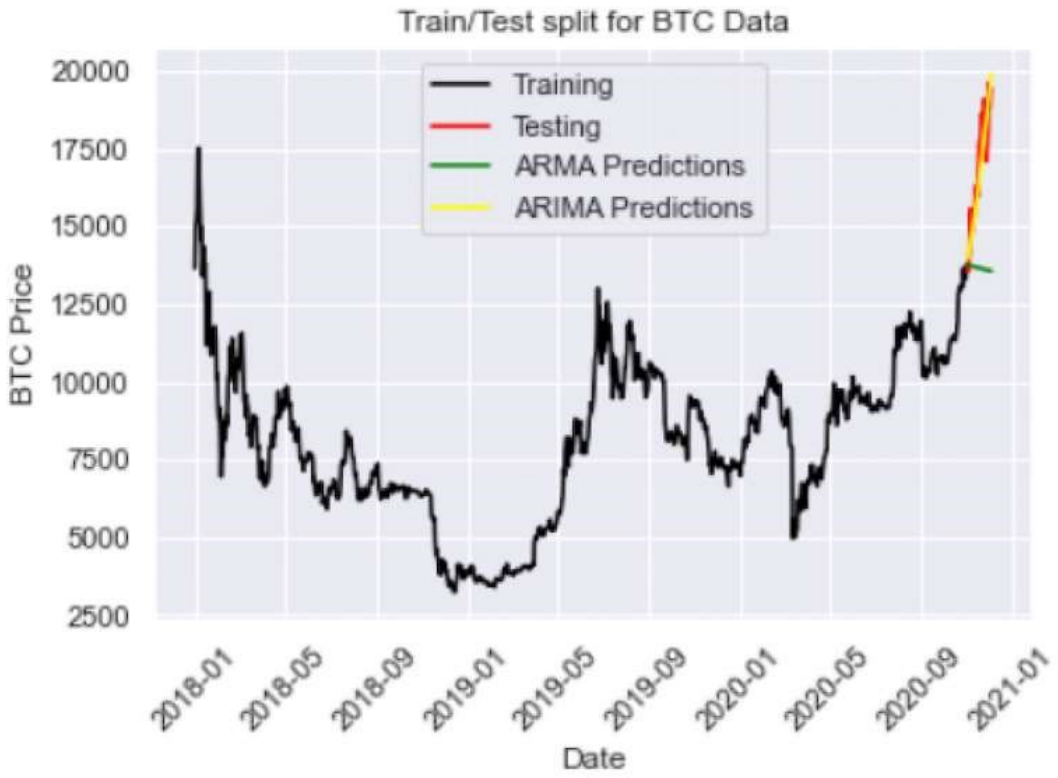
arma_rmse = np.sqrt(mean_squared_error(test["BTC-USD"].values, y_pred_df["Predictions"])) print("RMSE: ",arma_rmse)



We see that the ARIMA predictions (in yellow) fall on top of the ARMA predictions. Let's try increasing the differencing parameter to ARIMA (2,3,2):



We see this helps capture the increasing price direction. Let's try playing with the parameters even further with ARIMA(5,4,2):



And we have an RMSE of 793, which is better than ARMA.

Finally, let's see if SARIMA, which incorporates seasonality, will further improve performance.

Seasonal ARIMA (SARIMA)

Seasonal ARIMA captures historical values, shock events and seasonality. We can define a SARIMA model using the SARIMAX class:

SARIMAXmodel = SARIMAX(y, order = (2,2,2), seasonal_order=(2,2,2,12))

SARIMAXmodel = SARIMAXmodel.fit()

y_pred = SARIMAXmodel.get_forecast(len(test.index))

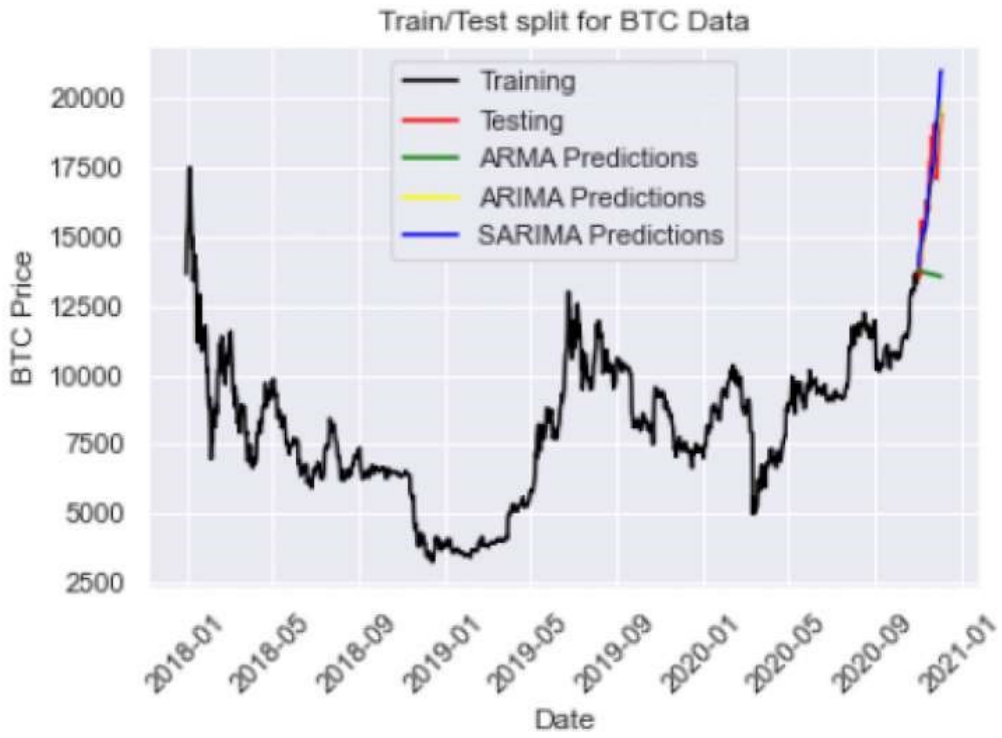
y_pred_df = y_pred.conf_int(alpha = 0.05)

y_pred_df["Predictions"] = SARIMAXmodel.predict(start = y_pred_df.index[0],

end = y_pred_df.index[-1]) y_pred_df.index = test.index y_pred_out =

y_pred_df["Predictions"] plt.plot(y_pred_out, color='Blue', label = 'SARIMA

Predictions') plt.legend()



Here we have an RMSE of 3779, which is much worse than ARIMA. This may be due to lack of hyperparameter tuning. If we play around with the parameters for our SARIMA model we should be able to improve performance even further.

I encourage you to experiment with the hyperparameters to see if you can build a SARIMA model that outperforms ARIMA. Further, you can employ methods like grid search to algorithmically find the best parameters for each model.