Report on Bitcoin Price Prediction Model with LSTM and Facebook Prophet

Shiyi Wang

College of Computing, Georgia Institute of Technology

CS 4675: Advanced Internet Computing Systems and Application Development

Dr. Ling Liu

March 9, 2022

Report on Bitcoin Price Prediction Model with LSTM and Facebook Prophet

In this assignment, I built a Bitcoin Price Prediction Model utilizing Keras Long Short-Term Memory (LSTM) and Facebook Prophet API.

1. Dataset Preparation

1.1 Loading Dataset

The datasets provided in the assignment description were either out-of-date (containing data from 2013 to 2014) or Page 404 Not Found. Therefore, I choose Bitcoin Historical Data from Kaggle. The dataset contains historical bitcoin statistics from January 2012 to March 2021. The dataset records per-minute updates of OHLC (Open, High, Low, and Close price), bitcoin transaction volume, corresponding currency transaction volume, and weighted average price.

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
0	1325317920	4.39	4.39	4.39	4.39	0.455581	2.0	4.39
1	1325317980	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	1325318040	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	1325318100	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	1325318160	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 1 Dataset Briefing

The columns are the following:

1. Open - Open price at start time window.
2. High - High price within time window.
3. Low - Low price within time window.
4. Close - Close price at end of time window.
5. Volume_(BTC) - Volume of BTC transacted in this window.
6. Volume_(Currency) - Volume of corresponding currency transacted in this window.
7. Weighted_Price - VWAP- Volume Weighted Average Price.

1.2 Data Cleaning

I notice that the `Timestamp` column is in Unix format. Therefore, I convert it into readable form and set it as the index column. Correspondingly, I resample the data based on 12-hour frequency. Besides, I notice that some entries are `NaN`. Here, I fill in `NaN` with the previous value since it is a time-based data series.

Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
2011-12-31 00:00:00	4.39	4.390000	4.39	4.390000	0.455581	2.000000	4.390000
2011-12-31 12:00:00	4.49	4.513333	4.49	4.513333	31.620766	141.106779	4.498804
2012-01-01 00:00:00	4.58	4.580000	4.58	4.580000	1.502000	6.879160	4.580000
2012-01-01 12:00:00	4.92	4.920000	4.92	4.920000	10.050000	49.450000	4.920000
2012-01-02 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Figure 2 Dataset Cleaned

1.3 Data Visualizations

`mplfinance` is a matplotlib utilities library for the visualization, and visual analysis, of financial data. I set the `type` to be line since we are dealing with a large set of data. The moving averages `mav` are set to be 30-, 90-, and 180-day period to show the trends at different granularities. Besides, I also plotted the volume distribution along with OHLC to get a better understanding of the data.

```
import mplfinance as mpf
# Here, I need to rename the BTC Volume column to satisfy the format of mplfinance.
df.rename(columns = {'Volume_(BTC)':'Volume'}, inplace = True)
# Let's plot the OHLC with Volume.
kwargs = dict(type='line', mav=(30, 90, 180), volume=True)
mpf.plot(df, **kwargs)
```

Figure 3 Configuration Setting for OHLC

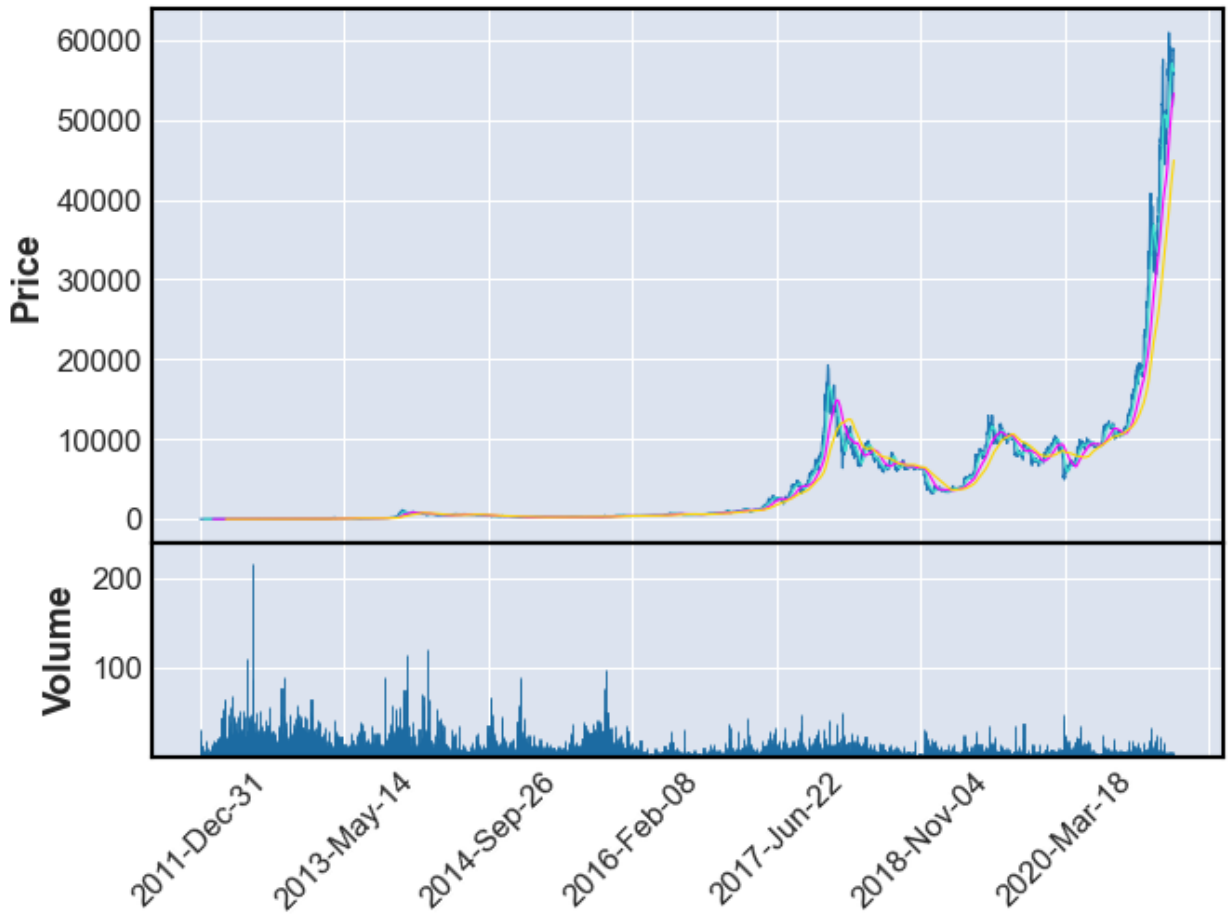

Figure 4 OHLC with Volume Distribution

In this project, I choose to predict Bitcoin's `Weighted_Price` since it filters out the OHLC fluctuation and reduces ambiguity in representing Bitcoin price. Therefore, I drop other unnecessary columns for the sake of simplicity.

Timestamp	Weighted_Price
2011-12-31 00:00:00	4.390000
2011-12-31 12:00:00	4.498804
2012-01-01 00:00:00	4.580000
2012-01-01 12:00:00	4.920000
2012-01-02 00:00:00	4.920000

Figure 5 Finalized Dataset for Prediction

With all preparations done, let's visualize the fluctuations on `Weighted_Price` on a 12-hour frequency.

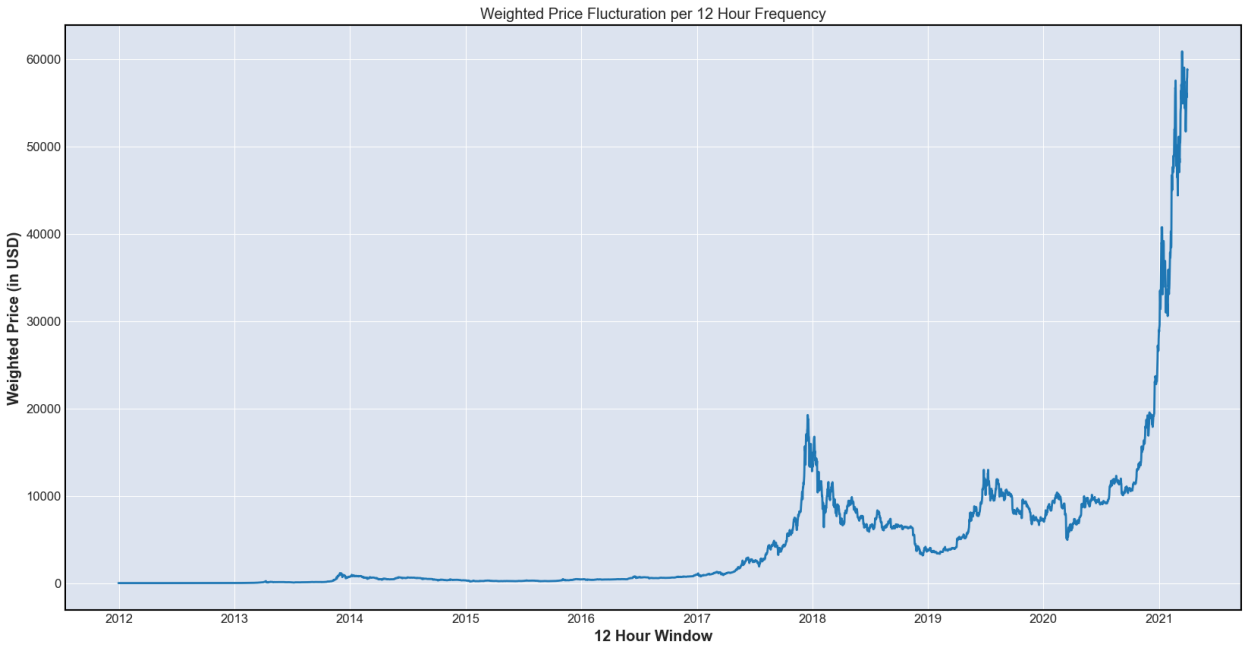

Figure 6 Weighted Price Fluctuation Per 12-hour Frequency

2. Model Implementation

2.1 Hand Trained LSTM Model

The Long Short-Term Memory (LSTM) is a variation of the RNN which solves the Vanishing Gradient Problem. I choose to split 80% of data for training and 20% for testing.

```
training_size = int(df_LSTM.shape[0] * 0.8) - 1
train = df_LSTM.iloc[:training_size, :]
test = df_LSTM.iloc[training_size:, :]
```

Figure 7 Train & Test Split

Besides, I normalized the dataset values by MinMax method. In LSTM, we need to group data. In this case, I group data in 50 * 12 hr. = 25 Days to predict the next group. I process the train and test data as follows.

```
x_train = []
y_train = []

for index in range(steps, train_scaled.shape[0] - steps):
    x_train.append(train_scaled[(index - steps):index, :])
    y_train.append(train_scaled[index, :])

x_train, y_train = np.array(x_train), np.array(y_train)

x_test = []
y_test = []

for index in range(steps, test_scaled.shape[0]):
    x_test.append(test_scaled[(index - steps):index, :])
    y_test.append(test_scaled[index, :])

x_test, y_test = np.array(x_test), np.array(y_test)
```

Figure 8 Train & Test Data Processing

In LSTM model, I need to tune a few hyperparameters. Due to the limited size of my training set, I choose 32 for `batch_size`. Small values provide a learning process with faster convergence with the tradeoff of noise in the training process. I set up my model below.

```
model = Sequential()
model.add(LSTM(units=50, return_sequences = True, input_shape =
(x_train.shape[1],x_train.shape[2])))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences = True))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences = True))
model.add(Dropout(0.2))
model.add(LSTM(units=50))
model.add(Dropout(0.2))
model.add(Dense(units=1))
model.compile(loss="mse", optimizer="adam")
model.fit(x_train, y_train, batch_size = batch_size, epochs = epochs)
```

Figure 9 LSTM Model Setup

Below is the screenshot for the last 10 epochs in 200 epochs. We can see that the loss stabilized at around 2.5e-04.

```
Epoch 190/200
166/166 [==============================] - 14s 82ms/step - loss: 2.2748e-04
Epoch 191/200
166/166 [==============================] - 12s 71ms/step - loss: 2.2771e-04
Epoch 192/200
166/166 [==============================] - 14s 82ms/step - loss: 2.4914e-04
Epoch 193/200
166/166 [==============================] - 14s 82ms/step - loss: 2.4844e-04
Epoch 194/200
166/166 [==============================] - 12s 75ms/step - loss: 2.3149e-04
Epoch 195/200
166/166 [==============================] - 13s 79ms/step - loss: 2.5677e-04
Epoch 196/200
166/166 [==============================] - 13s 78ms/step - loss: 2.4998e-04
Epoch 197/200
166/166 [==============================] - 13s 76ms/step - loss: 2.9344e-04
Epoch 198/200
166/166 [==============================] - 12s 75ms/step - loss: 2.5332e-04
Epoch 199/200
166/166 [==============================] - 13s 76ms/step - loss: 2.3100e-04
Epoch 200/200
166/166 [==============================] - 13s 77ms/step - loss: 2.4047e-04
```

Figure 10 Last 10 Epoch Loss

In Keras, there is a function called `model.summary()` that provides nice summary of the model construct. My LSTM Model consists of 5 hidden 50-neuron-layers with MSE (Mean Square Error) loss function, Adam optimizer, and 200 Epochs.

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm_8 (LSTM)                (None, 50, 50)            10400

 dropout_8 (Dropout)          (None, 50, 50)            0

 lstm_9 (LSTM)                (None, 50, 50)            20200

 dropout_9 (Dropout)          (None, 50, 50)            0

 lstm_10 (LSTM)               (None, 50, 50)            20200

 dropout_10 (Dropout)         (None, 50, 50)            0

 lstm_11 (LSTM)               (None, 50)                20200

 dropout_11 (Dropout)         (None, 50)                0

 dense_2 (Dense)              (None, 1)                 51

=================================================================
Total params: 71,051
Trainable params: 71,051
Non-trainable params: 0
_________________________________________________________________
```

Figure 11 Model Summary

Now we can compute prediction known as `y_hat`.

```
prediction = model.predict(x_test)
prediction = scaler.inverse_transform(prediction)
y_test = test[steps:].reset_index(drop=True)
```

Figure 12 Compute Prediction

Finally, we can visualize the predicted `Weighted_Price` using LSTM.

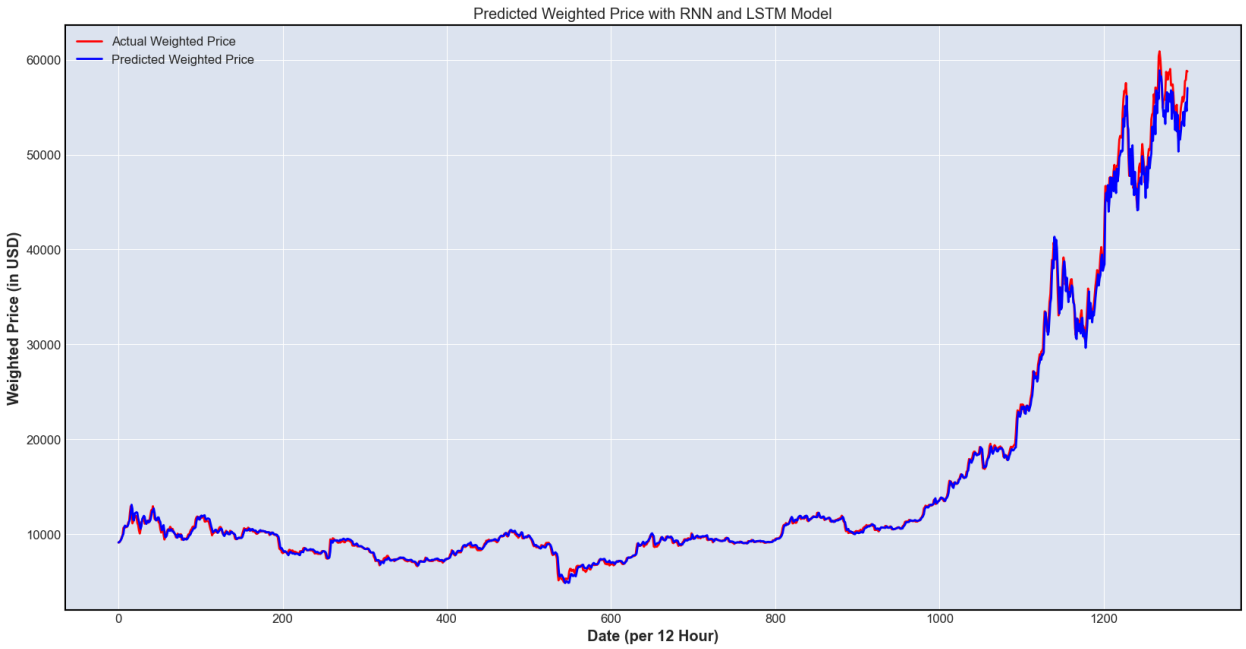

Figure 13 LSTM - Prediction with RNN and LSTM Model

This is not enough. Here I use `forecast` to store the next 50 steps prediction. Besides, I use `sliding_window` to store the next timeframe. First, I prepare sliding window by convert the array to NumPy array for dimension matches.

```
sliding_window.append(inputs[:steps, :])
sliding_window = np.array(sliding_window)
sliding_window = np.reshape(
    sliding_window, (sliding_window.shape[0], sliding_window.shape[1], 1))
```

Figure 14 Sliding Window Setup

Now, let's slide the "window".

```
for step in range(50):
    y_hat = model.predict(sliding_window)
    forecast.append(y_hat[0, :])
    y_hat = y_hat.reshape(1, 1, 1)
    sliding_window = np.concatenate(
        (sliding_window[:, 1:, :], y_hat), axis=1)
```

Figure 15 Slide the Window

Finally, let's visualize the future prediction.

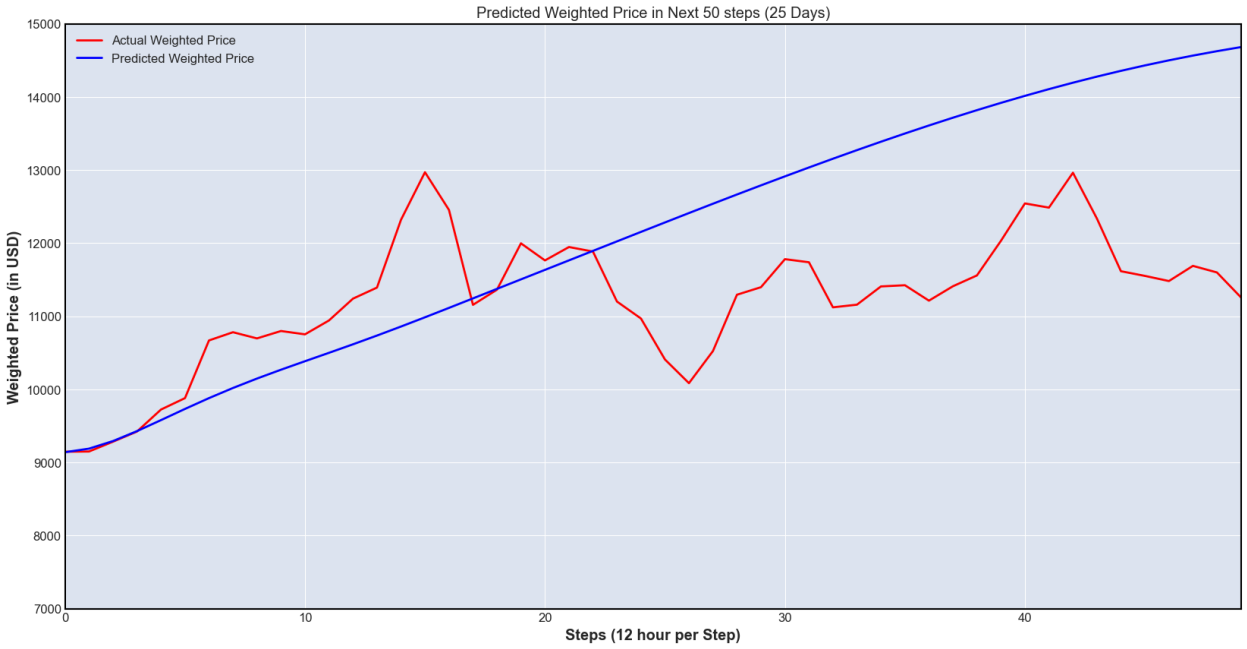

Figure 16 Prediction in Next 50 steps (25 Days)

2.2 Facebook Prophet API

Facebook Prophet is a R and Python-based forecasting procedure. It is provides fully automated forecasts that can fine-tuned manually by data scientists. First, I fit and train the model to the data per documentation requirements.

	ds	yhat	yhat_lower	yhat_upper
0	2011-12-31 00:00:00	140.542034	-699.334549	988.267184
1	2011-12-31 12:00:00	138.489882	-728.142676	938.505376
2	2012-01-01 00:00:00	135.172902	-750.516080	969.955146
3	2012-01-01 12:00:00	133.127187	-751.329924	959.387410
4	2012-01-02 00:00:00	129.870302	-695.964321	1022.260383
...	...	...	...	...
5465	2019-07-17 12:00:00	9901.053035	8859.096116	10859.307872
5466	2019-07-18 12:00:00	9957.491484	8977.071555	11021.108229
5467	2019-07-19 12:00:00	10012.345082	8976.001618	11054.144262
5468	2019-07-20 12:00:00	10065.448328	9039.838517	11120.330888
5469	2019-07-21 12:00:00	10116.681909	9110.307481	11049.749650

5470 rows × 4 columns

Figure 17 Fit and Train Facebook Prophet

Now, I can forecast and visualize the future.

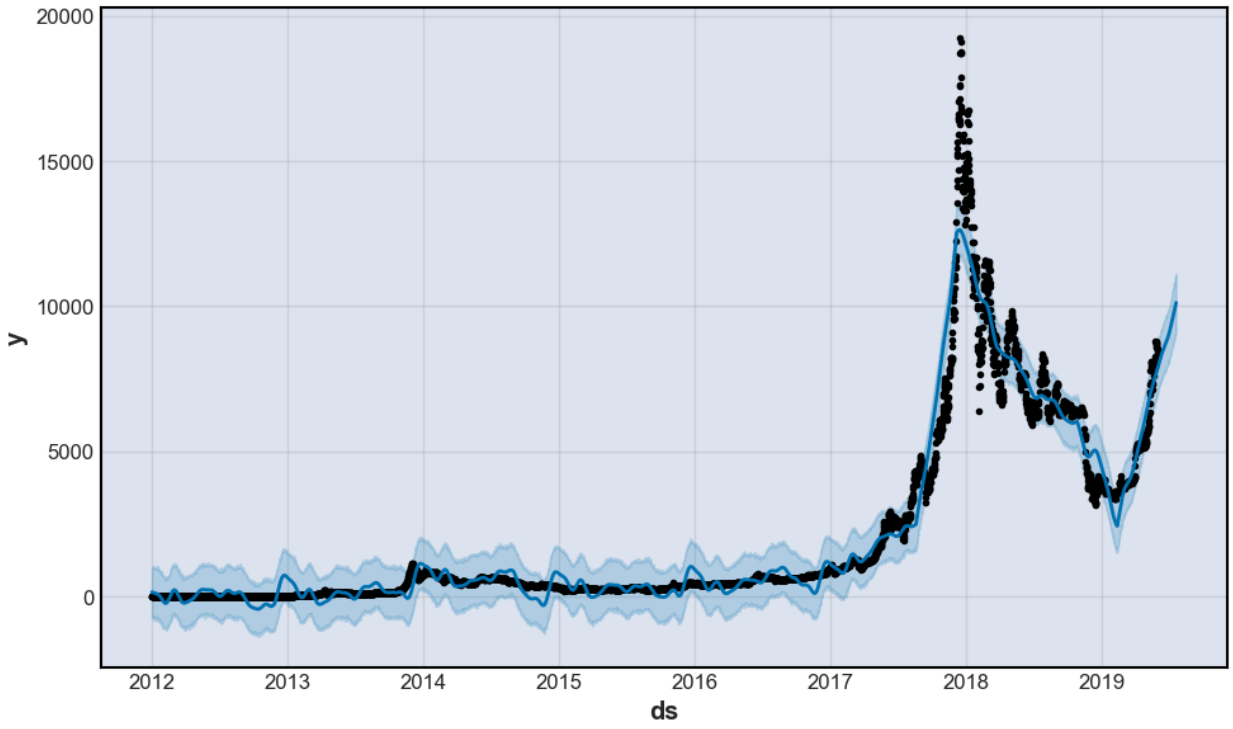

Figure 18 Facebook Prophet Forecast

I also added lines to show detected changepoints. However, there are too many changepoints which indicates that the price is fragile. Therefore, the diagram is not as informative as others.

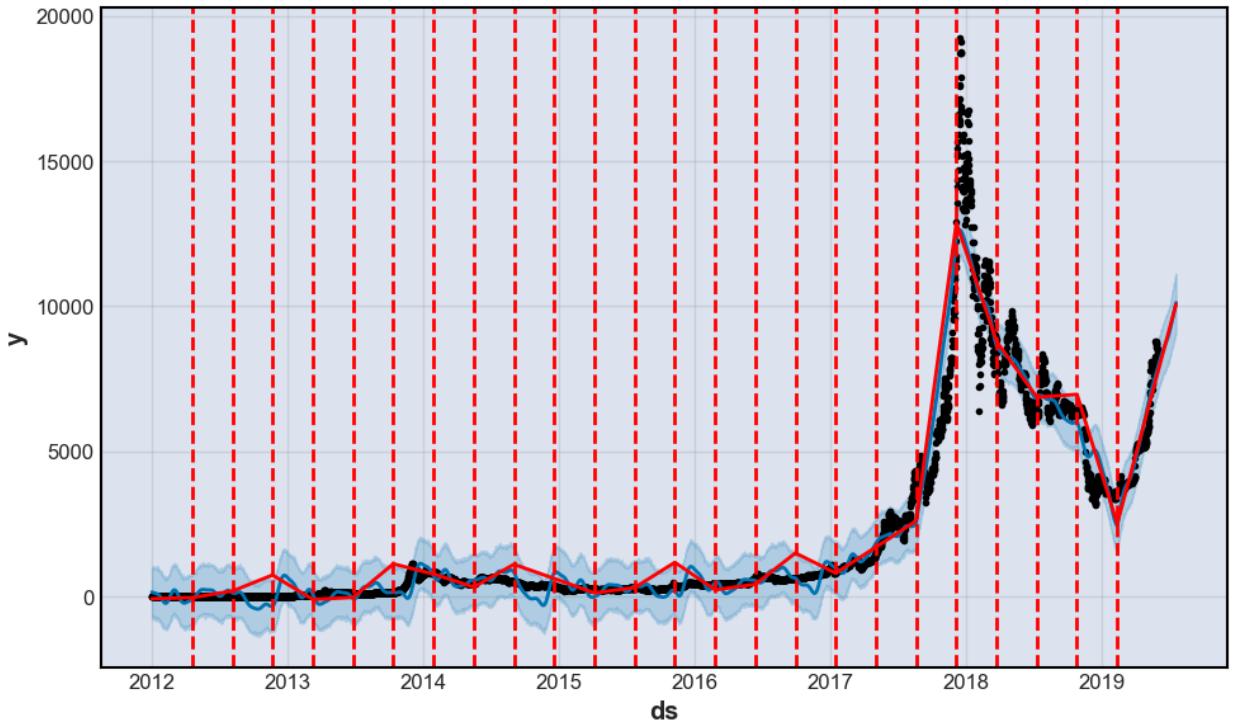

Figure 19 Facebook Prophet Changepoints

3. Discussion

The Long Short-Term Memory (LSTM) can lead to the decrease in the gradient for each level in a regular RNN because of the Recurring Weight close to 0. In my model, I use a time step of 50 meaning the input of time t gives a prediction of time t+50. The LSTM model predicts the Bitcoin price accurately.

In finance, there a phenomenon called Brownian Motion, which makes future values of stock prices independent of the past. In our case, it would be impossible to make long term predictions for Bitcoin price. In the future prediction, the LSTM gives a correct increasing trend

in Bitcoin price in the next 50 steps but failed to capture the exact fluctuations. In this case, the fluctuation can be 2000+ USD dollars.

On the other hand, Facebook Prophet provide a clear forecast direction that the Bitcoin price will increase in the following years which can be validated with the real data. In the changepoint plot, it demonstrates the volatility of Bitcoin price.

4. Future work and Extensions

My current models are trained, fit, and visualized. However, the comparisons are made based on qualitative measures such as eyeball observation. It lacks a way to numerically evaluate the performance across different methodologies. One way to test the performance is through trading simulations. For example, in my LSTM model, if the predicted weighted price in the future 50 steps is increasing, then we can make a buy at the current price and sell in the future. By keep maintaining the budget record, I can show the short-term reward and in term shows the performance.

Another perspective is to consider more public APIs for time series data evaluation and prediction. Besides Facebook Prophet API, there are other libraries such as AutoTS, tsfresh, etc. which can also be incorporated into evaluation.

5. Bonus Section: Special Tricks

In the preparation phase, I spent a lot of time in researching Anaconda, Matplotlib, and other machine learning library dependency relationships. Sometimes, skills are transferrable and independent of the context. I want to dedicate this special section to share some tricks I used in data visualization and dependency solutions.

1. The most notable issue is the dependency conflict between NumPy and TensorFlow. To use TensorFlow 2.X (in my case 2.6), I need to downgrade my NumPy version to 1.19.5.
2. I can export my environment configurations using Anaconda command: conda env export > environment.yml. In this way, others can create the environment without needing to resolve every single dependency conflicts.
3. In Matplotlib, there are two cool features. The first command `plt.style.use('fivethirtyeight') makes` the plots prettier and the second command `plt.rcParams['figure.dpi'] = 300` improves the image resolution, which bothered me for decades.

References

Brownlee, J. (2017, April 11). *How to Tune LSTM Hyperparameters with Keras for Time Series Forecasting*. Machine Learning Mastery. https://machinelearningmastery.com/tune-lstm-hyperparameters-keras-time-series-forecasting/

Facebook. (2019). *Facebook Prophet*. https://facebook.github.io/prophet/

Gandhi, P. (2021, September 26). *7 libraries that help in time-series problems*. Medium. https://towardsdatascience.com/7-libraries-that-help-in-time-series-problems-d59473e48ddd

GitHub. (2021, December 4). *mplfinance*. GitHub. https://github.com/matplotlib/mplfinance/blob/master/README.md

Kaggle. (2021). *Bitcoin Historical Data*. Kaggle.com. https://www.kaggle.com/mczielinski/bitcoin-historical-data