CRYPTOCURRENCY AND NEURAL NETWORK

LSTM Network for Regression to Predict Future Prices

Neural Network & Time-series price prediction using hourly data

Sarit Maitra Sep 14, 2020 · 6 min read ★

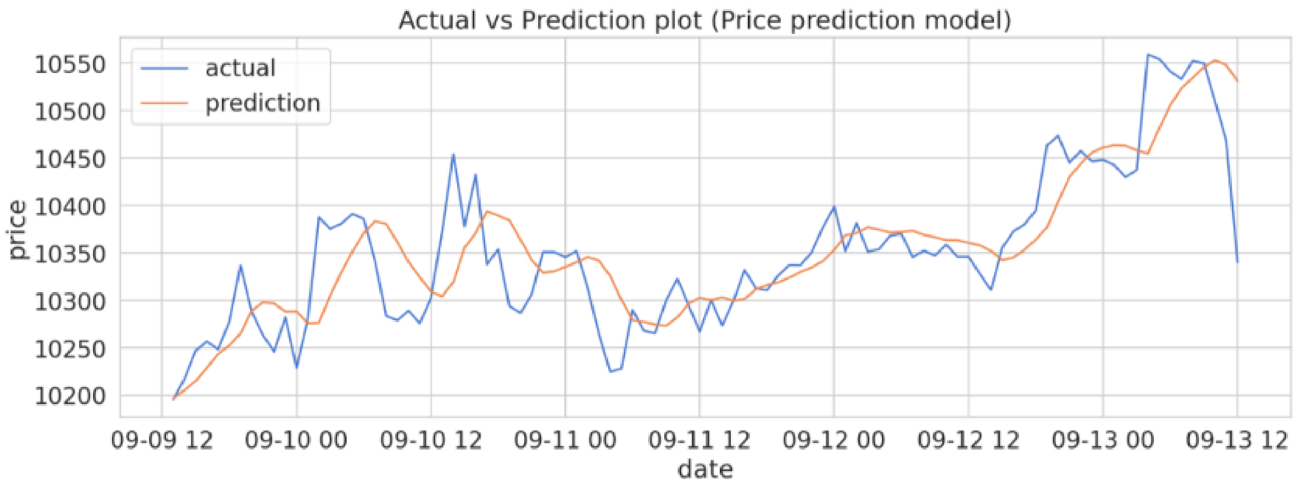


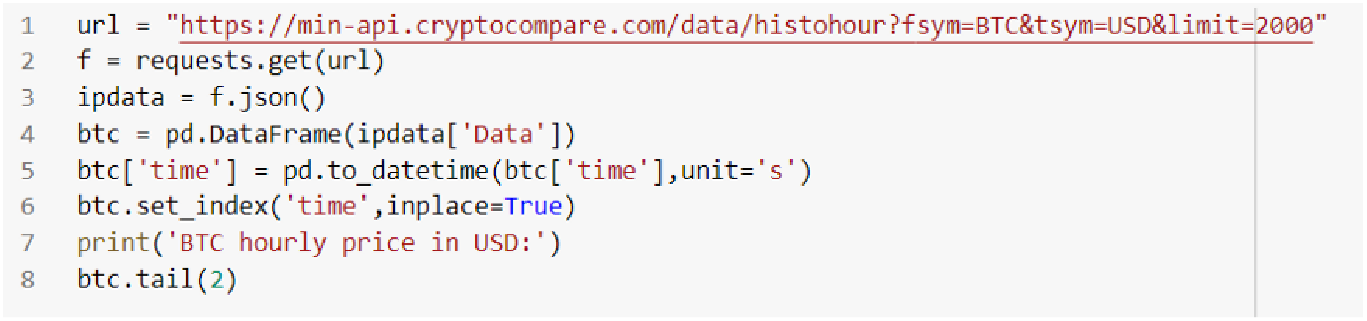
Image by Author

Note from Towards Data Science's editors: *While we allow independent authors to publish articles in accordance with our rules and guidelines, we do not endorse each author's contribution. You should not rely on an author's works without seeking professional advice. See our Reader Terms for details.*

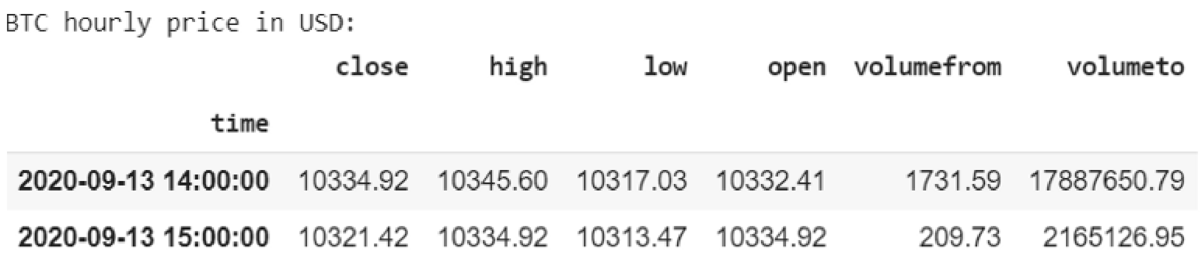
Prediction of stock price is quite a challenging because of highly volatile nature of time series combined with stochastic movement with non-linear . Here, the problem we have in hand is a price prediction issue and we're trying to predict a numerical value defined in a range (from 9000 to 12500 approx). This problem fits the Regression Analysis framework. We shall be using neural network architecture to try to solve the problem here.

We'll build a Deep Neural Network here that does some forecasting for us and use it to predict future price. Let us load the hourly frequency data.

Data loading

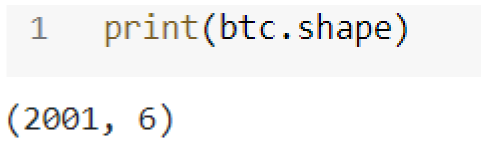


```python
url = "https://min-api.cryptocompare.com/data/histohour?fsym=BTC&tsym=USD&limit=2000"
f = requests.get(url)
ipdata = f.json()
btc = pd.DataFrame(ipdata['Data'])
btc['time'] = pd.to_datetime(btc['time'],unit='s')
btc.set_index('time',inplace=True)
print('BTC hourly price in USD:')
btc.tail(2)
```



```
BTC hourly price in USD:
```

time	close	high	low	open	volumefrom	volumeto
2020-09-13 14:00:00	10334.92	10345.60	10317.03	10332.41	1731.59	17887650.79
2020-09-13 15:00:00	10321.42	10334.92	10313.47	10334.92	209.73	2165126.95



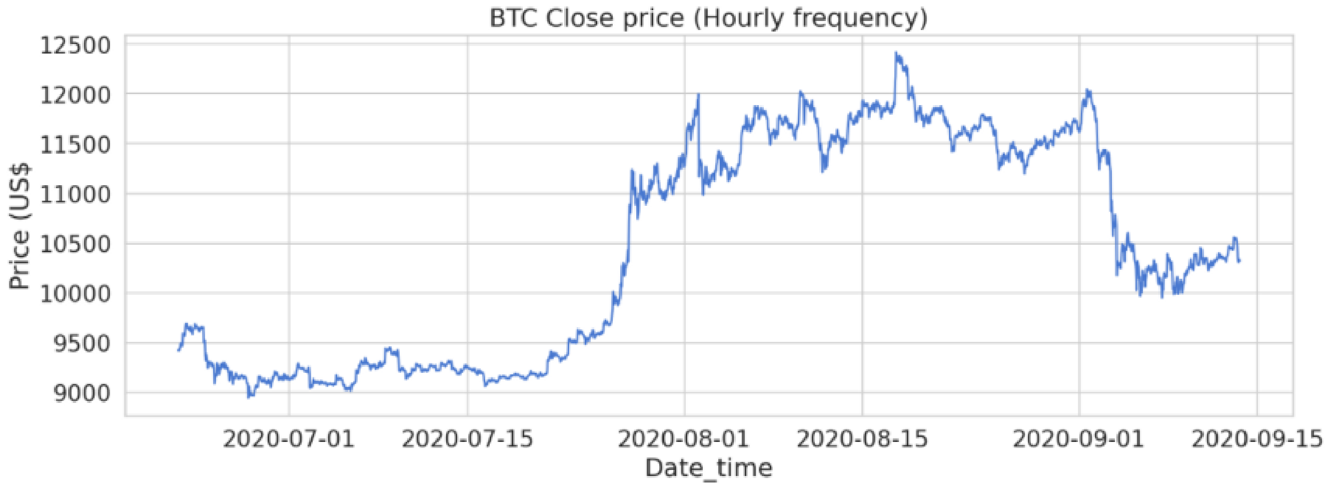
```python
print(btc.shape)
```

```
(2001, 6)
```

We have a total of 2001 data points representing Bitcoin in USD . We're interested in predicting the closing price for future dates.

When we see a time-series, we always want to know if the value of the current time step affects the next one.

```
plt.xlabel ('Date_time')
plt.ylabel ('Price (US$')
plt.show()
```



We shall use LSTM network here which has the ability to capture long-term dependencies in a sequence (e.g. dependency between today's price and that 2 weeks ago). Moreover, uni-variate series is being used here considering only the close price from the series.

Let us normalize the price data using MinMax scaler.

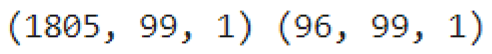
Data normalization

```
# Feature Scaling Normalization
scaler = MinMaxScaler()
# min-max normalization and scale the features in the 0-1 range.
close_price = btc['close'].values.reshape(-1, 1)
# The scaler expects the data to be shaped as (x, y)
scaled_close = scaler.fit_transform(close_price)
scaled_close = scaled_close[~np.isnan(scaled_close)]
# removing NaNs (if any)
scaled_close = scaled_close.reshape(-1, 1)
# reshaping data after removing NaNs
```

Data processing for LSTM

Let’s build some sequences. Sequences work like walk forward validation approach, where initial sequence length will be defined and subsequently will be shifting one position to the right to create another sequence. This way the process is repeated until all possible positions are used.

```
SEQ_LEN = 100
# creating a sequence of 100 hours at position 0.
def to_sequences(data, seq_len):
d = []
for index in range(len(data) - seq_len):
d.append(data[index: index + seq_len])
return np.array(d)
def preprocess(data_raw, seq_len, train_split):
data = to_sequences(data_raw, seq_len)
num_train = int(train_split * data.shape[0])
X_train = data[:num_train, :-1, :]
y_train = data[:num_train, -1, :]

X_test = data[num_train:, :-1, :]
y_test = data[num_train:, -1, :]

return X_train, y_train, X_test, y_test

"""Walk forward validation:
Initial SEQ_LEN is defined above, so, walk forward will be shifting
one position to the right and create another sequence.
The process is repeated until all possible positions are used."""

X_train, y_train, X_test, y_test = preprocess(scaled_close, SEQ_LEN,
train_split = 0.95)
# 5% of the data saved for testing.

print(X_train.shape, X_test.shape)

"""Our model will use 1805 sequences representing 99 hours of Bitcoin
price changes each for training. We shall be predicting the price for
96 hours in the future"""
```

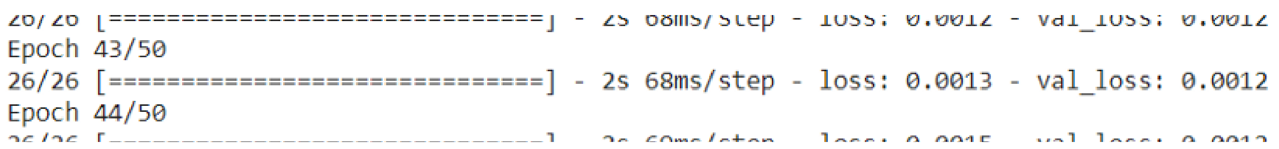


```
(1805, 99, 1) (96, 99, 1)
```

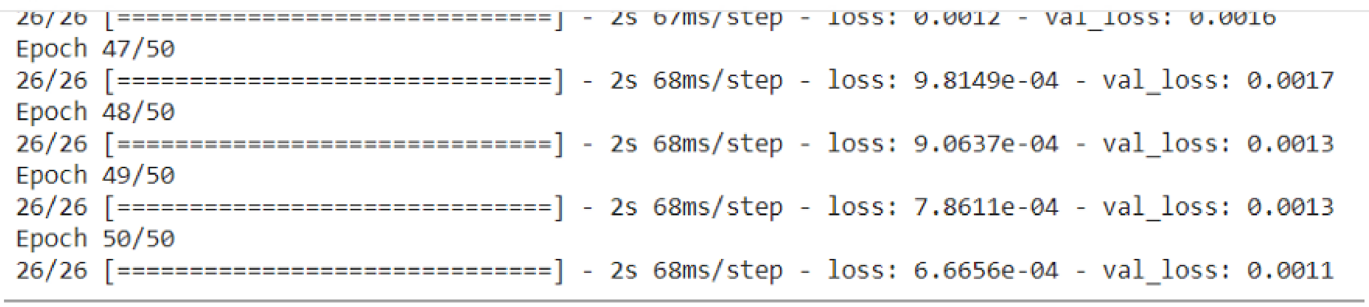
Building LSTM model

```
DROPOUT = 0.2
# 20% Dropout is used to control over-fitting during training
WINDOW_SIZE = SEQ_LEN - 1
model = keras.Sequential()

# Input layer
model.add(Bidirectional(LSTM(WINDOW_SIZE, return_sequences=True),
input_shape=(WINDOW_SIZE, X_train.shape[-1])))

"""Bidirectional RNNs allows to train on the sequence data in forward
and backward direction."""

model.add(Dropout(rate=DROPOUT))

# 1st Hidden layer
model.add(Bidirectional(LSTM((WINDOW_SIZE * 2), return_sequences =
True)))
model.add(Dropout(rate=DROPOUT))

# 2nd Hidden layer
model.add(Bidirectional(LSTM(WINDOW_SIZE, return_sequences=False)))

# output layer
model.add(Dense(units=1))

model.add(Activation('linear'))

"""Output layer has a single neuron (predicted Bitcoin price). We use
Linear activation function which activation is proportional to the
input."""

BATCH_SIZE = 64
model.compile(loss='mean_squared_error', optimizer='adam')
history = model.fit(X_train, y_train, epochs=50,
batch_size=BATCH_SIZE, shuffle=False, validation_split=0.1)
# shuffle not advisable during training of Time Series
```

We can use callbacks option during training to prevent our model from over-fitting too; I have not used call-back, but same can be applied during model fitting phase.



```
26/26 [==============================] - 2s 68ms/step - loss: 0.0012 - val_loss: 0.0012
Epoch 43/50
26/26 [==============================] - 2s 68ms/step - loss: 0.0013 - val_loss: 0.0012
Epoch 44/50
26/26 [==============================] - 2s 69ms/step - loss: 0.0015 - val_loss: 0.0012
```



```
26/26 [==============================] - 2s 67ms/step - loss: 0.0012 - val_loss: 0.0016
Epoch 47/50
26/26 [==============================] - 2s 68ms/step - loss: 9.8149e-04 - val_loss: 0.0017
Epoch 48/50
26/26 [==============================] - 2s 68ms/step - loss: 9.0637e-04 - val_loss: 0.0013
Epoch 49/50
26/26 [==============================] - 2s 68ms/step - loss: 7.8611e-04 - val_loss: 0.0013
Epoch 50/50
26/26 [==============================] - 2s 68ms/step - loss: 6.6656e-04 - val_loss: 0.0011
```

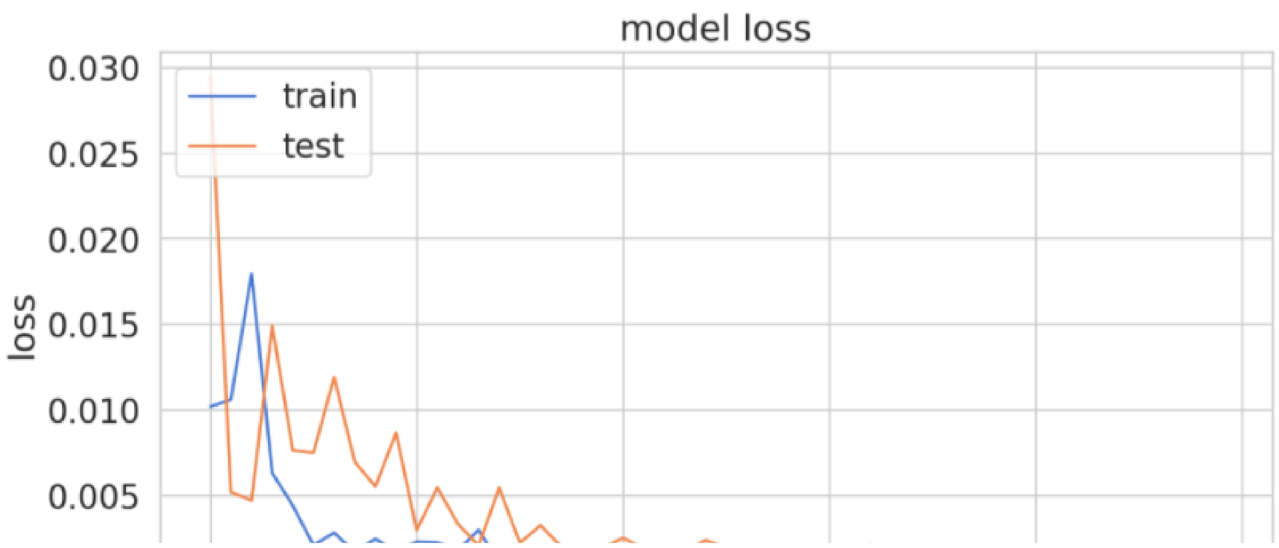
Model Evaluation

A simple way to understand the training process is view the training and validation loss.

```
history.history.keys()
# finding the keys to use for plotting
```

```
dict_keys(['loss', 'val_loss'])
```

```
# history for loss
plt.figure(figsize = (10,5))
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

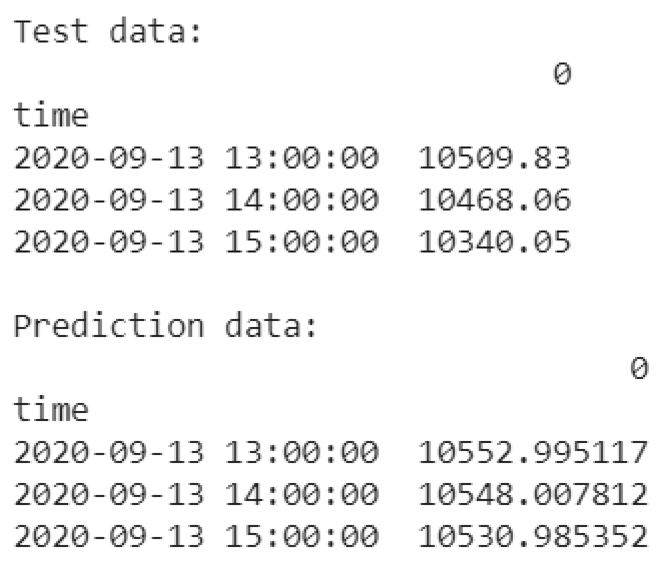


Here, we can see some improvement in both the training error and on the validation error.

Testing

We have some additional data left for testing purpose. Let's get the predictions from the model using those data to validate the goodness of fit of our model.

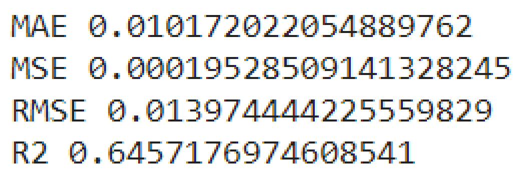
```
# prediction on test data
y_pred = model.predict(X_test)
# invert the test to original values
y_test_inverse = DataFrame(scaler.inverse_transform(y_test))
# assigning datetime
y_test_inverse.index = btc.index[-len(y_test):]
print('Test data:',)
print(y_test_inverse.tail(3)); print();

# invert the prediction to understandable values
y_pred_inverse = DataFrame(scaler.inverse_transform(y_pred))
# assigning datetime
y_pred_inverse.index = y_test_inverse.index
print('Prediction data:',)
print(y_pred_inverse.tail(3))
```



```
Test data:
                            0
time
2020-09-13 13:00:00  10509.83
2020-09-13 14:00:00  10468.06
2020-09-13 15:00:00  10340.05

Prediction data:
                                0
time
2020-09-13 13:00:00  10552.995117
2020-09-13 14:00:00  10548.007812
2020-09-13 15:00:00  10530.985352
```

Accuracy metrics

```
print(f'RMSE {np.sqrt(mean_squared_error(y_test, y_pred))}')
print(f'R2 {r2_score(y_test, y_pred)}')
```

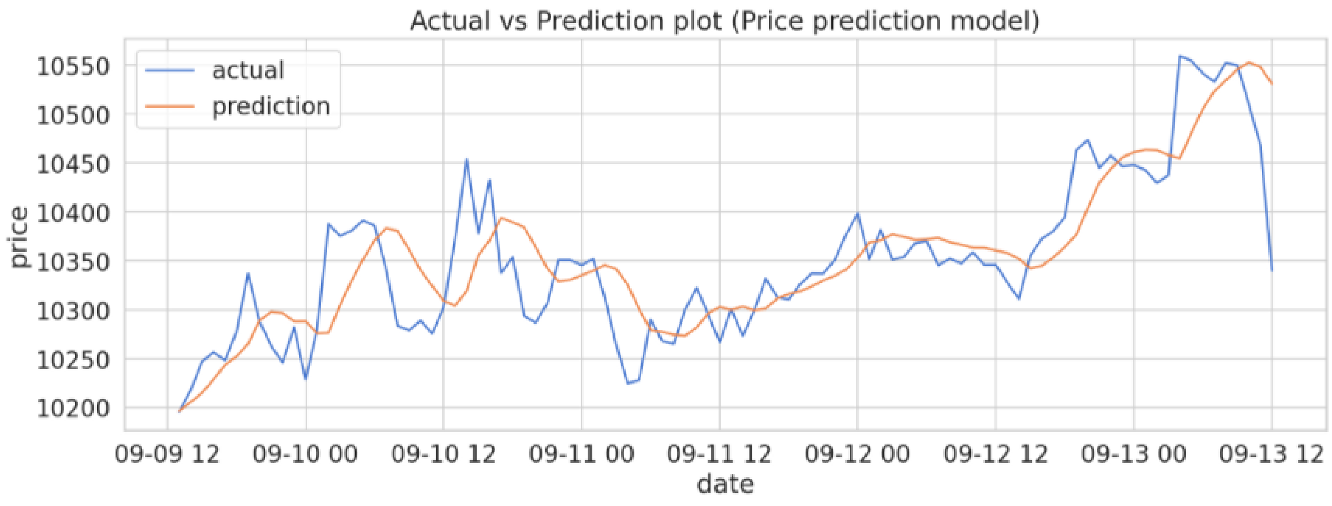


```
MAE 0.010172022054889762
MSE 0.00019528509141328245
RMSE 0.013974444225559829
R2 0.6457176974608541
```

RMSE allows us to penalize points further from the mean. Though the error scores are low but looking at R2 score, there is definitely room for improvement here. The model can be tuned to get better output.

Performance visualization

```
plt.figure(figsize = (15,5))
plt.plot(y_test_inverse)
plt.plot(y_pred_inverse)
plt.title('Actual vs Prediction plot (Price prediction model)')
plt.ylabel('price')
plt.xlabel('date')
plt.legend(['actual', 'prediction'], loc='upper left')
plt.show()
```



terms of we can say that, it generalizes well.

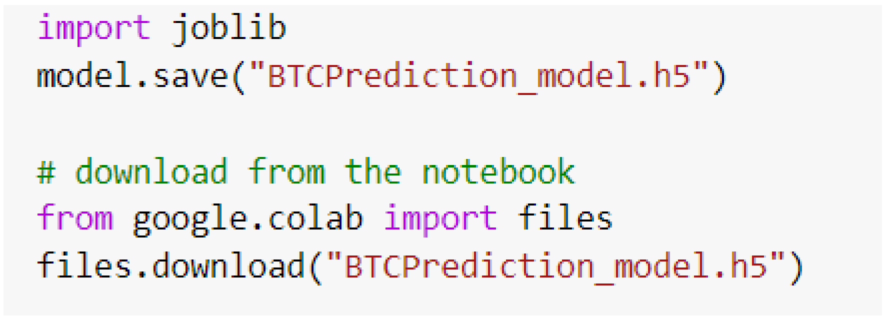
Conclusion & future scope

Predicting stock market returns is a challenging task due to consistently changing stock values which are dependent on multiple parameters which form complex patterns.

Future direction could be:

1. analyzing the correlation between different cryptocurrencies and how would that affect the performance of our model.
2. adding features using technical analysis to check the model performance.
3. adding features from fundamental analysis to check how those affect the mode.
4. adding sentiment analysis from social networking e.g. twitter and new report to check model performance.
5. GRU network can also be tried with different activation e.g. 'softsign" to check the performance.

Multivariate analysis of this nature require a great amount of effort and time on feature engineering, data analysis, model training etc.



```
import joblib
model.save("BTCPrediction_model.h5")

# download from the notebook
from google.colab import files
files.download("BTCPrediction_model.h5")
```

Connect me *here*.

References:

606.

2. *Vanelin Valkov (2015). Hackers guide to machine learning*

Note: *The programs described here are experimental and should be used with caution for any commercial purpose. All such use at your own risk.*

Timeseries Forecasting · Neural Networks · Arificialinterlligence · Predictive Analytics