Bitcoin Price Prediction using Neural Network

MSc Research Project
Data Analytics

Namrata Hemraj Gawali
Student ID: x20177887

School of Computing
National College of Ireland

Supervisor: Prof.Jorge Basilio

Bitcoin Price Prediction using Neural Network

Namrata Hemraj Gawali
x20177887

1 Introduction

In this study, we are predicting the Bitcoin price using different deep learning approaches. This configuration manual lists all of the processes that may be required for replication. It shows the flow from environment setup to data gathering and modeling using different evaluation metrics. Also, the sample codes are added for the understanding of the running of the study.

2 System Configuration

2.1 Hardware Specification

• Model: Lenovo Ideapad Z51, 2015, 1TB GB
• Processor: 2.4 GHz Intel Core i7
• RAM: 8 GB 2401 MHz DDR3
• Graphics: Tropo XT2

2.2 Software Specification

• To Access Google Drive, required a Gmail account.
• Google Colaboratory was used to perform the research implementation. Google offers free cloud based servers for running Python code, however with certain restrictions. Google Collab Professional could be used with higher GPU and RAM.

3 Install Packages

We have a requirements.txt file which we need to upload in the colab enviornment and run the below code to get the required packages along with versioning.

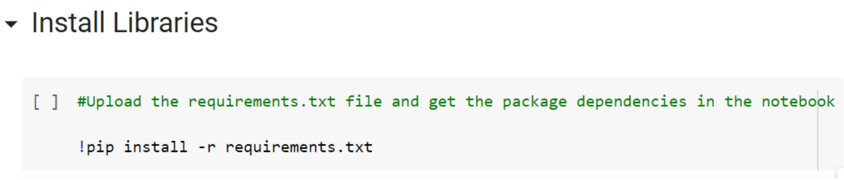

Figure 1: Install Packages

4 Data Gathering

4.1 Google Colaboratory Environment Setup

The experiments have been carried out in Google Colaboratory platform. We have used the dataset from Kaggle using the API. For this purpose, we have to get a token file **kaggle.jason** from the **Account** tab in Kaggle as shown below. After, we need to create a notebook using google colab and upload the **kaggle.jason** file in the enviornment. Then, we have to use the snippet shown in Figure 2 to access the datasets from Kaggle.

Alternatively, we can also upload the dataset file on the google drive and access the data with our own credentials and ID of the file from drive. Then we can use the snippet as shown in figure 3 to load the data. As the data size is big it can't be uploaded on moodle but it can be downloaded from below link. `https://www.kaggle.com/mczielinski/bitcoin-historical-data/download`

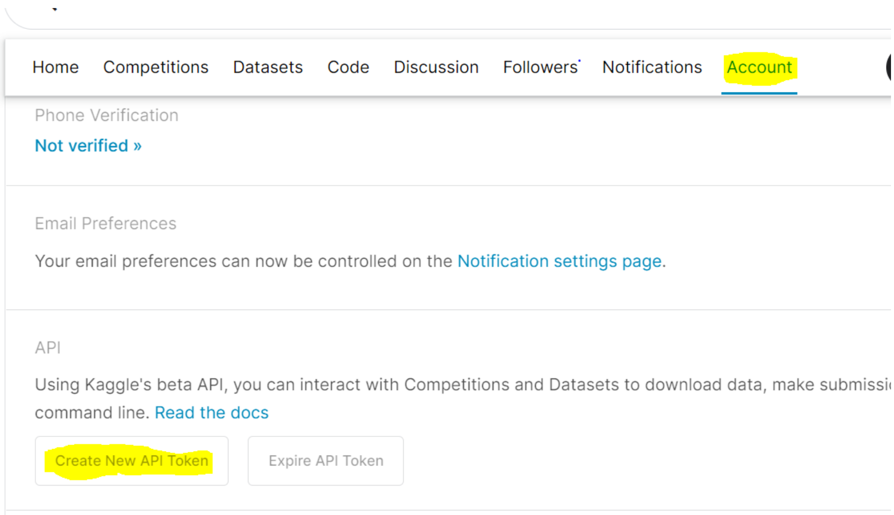

Figure 2: Downloading token for Kaggle API

▾ Get the data from Kaggle API

```
[ ] #mount the drive to the notebook
    from google.colab import drive
    drive.mount('/content/drive')

[ ] #install kaggle library
    !pip install kaggle

[ ] !mkdir ~/.kaggle/

[ ] ! cp kaggle.json ~/.kaggle/

[ ] ! chmod 600 ~/.kaggle/kaggle.json

    !kaggle datasets download -d mczielinski/bitcoin-historical-data

[ ] #type 'y' when asked for accessing the file for replacing
    !unzip '/content/bitcoin-historical-data.zip'
```

Figure 3: Mounting the drive and reading the data through Kaggle API

▾ Get the data from Drive

```
#uncomment the code below if using the dataset from drive

# !pip install -U -q PyDrive

#import libraries
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials

#authenticating the credentials for google colab
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)

# download the file containing the bitcoin data from drive
# replace the id with id of file you want to access
downloaded = drive.CreateFile({'id':'1oAy86Btu6tNFu8XljumpJxYMhAdNRjwn'})
downloaded.GetContentFile('bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv')
```

Figure 4: Reading the data from Drive

4.2 Loading the data

After we have successfully connected to the Kaggle or Drive we can read the data in csv by using the below command

▾ Loading the downloaded data

```
[ ] #import pandas
    import pandas as pd
    import gc

    #create dataframe using the csv donloaded which contains bitcoin price by minutes
    bitcoin_data = pd.read_csv('bitstampUSD_1-min_data_2012-01-01_to_2021-03-31.csv')
```

```
[ ] bitcoin_data.head()
```

Figure 5: Loading the data

5 Data Preparation

5.1 Resampling the data

We have transformed the data for EDA and modelling purpose with the use of code below.

```
[135] #check for null values
      bitcoin_data.isna().sum()
```

```
[136] #create date column from the timestamp
      bitcoin_data["Timestamp"] = pd.to_datetime(bitcoin_data["Timestamp"],unit="s",origin="unix")#.dt.date
```

```
[137] #set the index as date

      bitcoin_data.set_index(bitcoin_data["Timestamp"],drop=True,inplace=True)

      #resample the dataset to daily as we have null values for some timestamps
      bitcoin_daily = bitcoin_data.resample("d").mean()
      bitcoin_daily.head(3)
```

```
[139] #locate null values in dataset

      is_NaN = bitcoin_daily.isnull()
      row_has_NaN = is_NaN.any(axis=1)
      rows_with_NaN = bitcoin_daily[row_has_NaN]

      print(rows_with_NaN)
```

```
[140] #replcae the null values in the data with 1 as we don't want to loose any date in the index
      bitcoin_daily.interpolate(method='linear',inplace=True)
```

5.2 Financial Modeling data

Get the data from FinancialModelling API using the below snippet.

```
[151] #import library and run below snippet obtained from the documentation of the website
      import requests

      #!/usr/bin/env python

      try:
          # For Python 3.0 and later
          from urllib.request import urlopen
      except ImportError:
          # Fall back to Python 2's urllib2
          from urllib2 import urlopen

      import certifi
      import json

      #write a function to ge the jason parsed data
      def get_jsonparsed_data(url):
          """
          Receive the content of ``url``, parse it as JSON and return the object.

          Parameters
          ----------
          url : str

          Returns
          -------
          dict
          """
          response = urlopen(url, cafile=certifi.where())
          data = response.read().decode("utf-8")
          return json.loads(data)

      url = ("https://financialmodelingprep.com/api/v3/historical-price-full/crypto/BTCUSD?apikey=b9e530f97b0bd7d52a3db9cefaaaf4ab")
      print(get_jsonparsed_data(url))
```

```
[152] #get bitcoin data from api
      BTC_data = get_jsonparsed_data(url)
      BTC_data = BTC_data['historical']

      #drop the data into dataframe
      BTC_data = pd.DataFrame.from_dict(BTC_data)
      BTC_data.head(3)
```

```
[153] #set the index as date
      BTC_data.set_index('date',inplace=True)
      #Keep only the close column
      BTC_data = BTC_data[['close']]
      #Rename the column name to BTC
      BTC_data.columns = ['BTC']
      BTC_data.head()
```

```
[154] #merge stocks data with Bitcoin
      Stocks_BTC = BTC_data.merge(Stocks_data, how='inner',right_index = True, left_index=True)

      #Drop NA since we have nan values for weekends. S&P500 only trades business days
      Stocks_BTC.dropna(inplace=True)

      print(Stocks_BTC.head())
```

5.3 Data Scaling and Reshaping

We have scaled the data using MinMaxScalar and reshaped to 3D using code below along with the train test split.

```
[165] #reshaping the data to single column array with float data type

      #reshape the dataset: -1 in reshape function is used when you dont know or want to explicitly t
      tr_dataset = tr_dataset.reshape(-1, 1)

      #change type
      tr_dataset = tr_dataset.astype("float32")
      tr_dataset.shape
```

```
[166] # Feature Scaling
      from sklearn.preprocessing import MinMaxScaler

      #scaling the data using minmaxscalar to normalize the data from (0,1)
      scaler = MinMaxScaler(feature_range = (0, 1))
      data_scaled = scaler.fit_transform(tr_dataset)
      data_scaled
```

```
[168] # Creating a data structure with 3200 samples and 10 time_step using for loop
      X_train = []
      y_train = []
      time_step = 10

      #create the dataset with
      for i in range(time_step, data_scaled.shape[0]):
          X_train.append(data_scaled[i-time_step:i, 0])
          y_train.append(data_scaled[i, 0])
      X_train, y_train = np.array(X_train), np.array(y_train)

      print("X_train shape: ", X_train.shape)
      print("y_train shape: ", y_train.shape)

      gc.collect()
```

```
[169] # Reshaping the data to 3D (Samples, Timesteps, number of features)
      X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))

      #check the shape of training data
      print("X_train shape: ",X_train.shape)
      print("y_train shape: ",y_train.shape) #target variable

      gc.collect()
```

```
[161] #consider 95% data for training and 5% for testing
      train_size = int(len(bitcoin_daily) * 0.95)
      test_size = len(bitcoin_daily) - train_size
      #create dataframes for training and testing
      df_train = bitcoin_daily.iloc[0:train_size, :]
      df_test = bitcoin_daily.iloc[train_size:len(bitcoin_daily), :]
      print("Train size: {}, Test size: {}". format(len(df_train), len(df_test)))

      gc.collect()
```

5.4 Seasonal Decomposition

We have performed the stationarity test and checked the seasonal decomposition as below for monthly data.

```
#Seasonal Decompose for monthly data
ax, fig = plt.subplots(figsize=(15,8), sharex=True)

#get decomposition of target
decomposition_btc = sm.tsa.seasonal_decompose(bitcoin_monthly["Auto_Diff"][1:])

#different plots to check Trend, Seasonal decomposition, Residuals
plt.subplot(411)
plt.plot(bitcoin_monthly["Weighted_Price"], label="Weighted Price")
plt.title("Observed",loc="left", alpha=0.75, fontsize=12)

plt.subplot(412)
plt.plot(decomposition_btc.trend, label="Trend")
plt.title("Trend",loc="left", alpha=0.75, fontsize=12)

plt.subplot(413)
plt.plot(decomposition_btc.seasonal, label="Seasonal")
plt.title("Seasonal",loc="left", alpha=0.75, fontsize=12)

plt.subplot(414)
plt.plot(decomposition_btc.resid, label="Residual")
plt.title("Residual",loc="left", alpha=0.75, fontsize=12)

plt.tight_layout(pad=2)

plt.text(x=datetime.date(2011, 6, 30), y=67000,
         s="Decomposition of the stationary monthly weighted price",fontsize=18, alpha=0.75)

plt.tight_layout(pad=1)

gc.collect()
```

Figure 6: Seasonal Decomposition

6 Model Implementation

Below are the code snippets for all the four models

```
[171] #Training the data

      # Initialising the RNN with four layers
      regressor = Sequential()

      # Adding the first RNN layer with tanh activation, 10 units and some Dropout
      regressor.add(SimpleRNN(units = 10, activation="relu", return_sequences = True, input_shape = (X_train.shape[1], 1)))
      regressor.add(Dropout(0.2))

      # Adding a second RNN layer  with relu activation, 50 units and some Dropout
      regressor.add(SimpleRNN(units = 50, activation="relu", return_sequences = True))
      regressor.add(Dropout(0.2))

      # Adding a third RNN layer  with tanh activation and 50 units and some Dropout
      regressor.add(SimpleRNN(units = 50, activation="tanh", return_sequences = True))
      regressor.add(Dropout(0.2))

      # Adding a fourth RNN layer and some Dropout regularization
      regressor.add(SimpleRNN(units = 100))
      regressor.add(Dropout(0.2))

      # Adding the output layer
      regressor.add(Dense(units = 1))

      # Compiling the RNN
      regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')

      # Fitting the RNN to the Training set
      regressor.fit(X_train, y_train, epochs = 100, batch_size = 32)

      gc.collect()
```

Figure 7: RNN model

```
[187] #LSTM model starts
      from keras.layers import Dropout

      model = Sequential()
      #add an LSTM layer with 256 units
      model.add(LSTM(256, input_shape = (1, time_step)))
      #adding dropout of 0.2
      model.add(Dropout(0.2))
      #output layer
      model.add(Dense(1))
      model.compile(loss = "mean_squared_error",optimizer="adam")
      model.fit(trainX, trainY, epochs=100, batch_size=50)
```

Figure 8: LSTM model

```
#fit the model with optimum parameters
model = sm.tsa.statespace.SARIMAX(df_train["Weighted_Price"], order=results_dataframe.reset_index().pdq[0], seasonal_order=results_dataframe.reset_index().s_pdq[0],
            enforce_invertibility=False,enforce_stationarity=False).fit()
print(model.summary().tables[1])
```

Figure 9: SARIMA model

```
[230] #fit the model with optimum parameters
      model = sm.tsa.statespace.SARIMAX(Stocks_bitcoin_quarterly["Weighted_Price"], order=results_dataframe.reset_index().pdq[0],
                                        seasonal_order=results_dataframe.reset_index().spdq[0],
                  enforce_invertibility=False,enforce_stationarity=False, exog = Stocks_bitcoin_quarterly["sp500"]).fit()
```

Figure 10: SARIMAX model

7 Optimizing parameters and Predictions

Below are the code snippets for all the four models

```
def sarimax_function(data,pdq,s_pdq):

    """
    The function uses a brute force approach to apply all possible pdq combinations and evaluate the model
    """

    result_list = []
    for param in pdq:
        for s_param in s_pdq:

            model = sm.tsa.statespace.SARIMAX(data, order=param, seasonal_order=s_param,
            enforce_invertibility=False,enforce_stationarity=False)

            results = model.fit()
            result_list.append([param,s_param,results.aic])
            print("ARIMA Parameters: {} x: {}. AIC: {}".format(param,s_param,results.aic))

    return result_list,results
```

```
202] #consider 95% data for training and 5% for testing
     train_size = int(len(bitcoin_monthly) * 0.95)
     test_size = len(bitcoin_monthly) - train_size
     #create dataframes for training and testing
     df_train = bitcoin_monthly.iloc[0:train_size, :]
     df_test = bitcoin_monthly.iloc[train_size:len(bitcoin_monthly), :]
     print("Train size: {}, Test size: {}". format(len(df_train), len(df_test)))
```

```
203] #get the SARIMA results for all the combination of p,d,q
     result_list,results = sarimax_function(df_train["Weighted_Price"],pdq,seasonal_pdq)

     gc.collect()
```

Figure 11: Optimize the PDQ parameters

```
[175] from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

      #create a function to evaluate the data based on the results
      def evaluate_metrics(real, prediction):
          mae = mean_absolute_error(real, prediction)
          mse = mean_squared_error(real, prediction)
          rmse = np.sqrt(mean_squared_error(real, prediction))
          R2_score = r2_score(real, prediction)

          print(f"R2_score \t: {R2_score}")
          print(f"MAE \t\t: {mae}")
          print(f"MSE \t\t: {mse}")
          print(f"RMSE \t\t: {rmse}")
```

```
[176] #check the evaluation
      evaluate_metrics(real_bitcoin_price,predicted_bitcoin_price)
```

Figure 12: Evaluation Metrics

```
#predict the test data usin the trained model
predicted_bitcoin_price = regressor.predict(X_test)

#transforming back the target to real prices
predicted_bitcoin_price = scaler.inverse_transform(predicted_bitcoin_price)

# Visualising the results
date = df_test.index
plt.figure(figsize=(10,8))
plt.plot(date,real_bitcoin_price, color = 'green', label = 'Real Bitcoin Price')
plt.plot(date,predicted_bitcoin_price, color = 'blue', label = 'Predicted Bitcoin Price')
plt.title('Bitcoin Price Prediction')
plt.xlabel('Time')
plt.ylabel('Bitcoin Price')
plt.legend()
plt.show()

gc.collect()
```

Figure 13: Predicting the test data and Visualization

8 Ploting the data

We have used below code snippets to plot the data for EDA, predictions and forecasting.

```python
[144] #load libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import warnings

#plot the bitcoin prices by the time

#index for plot
date = bitcoin_daily.index
#set fig size
plt.figure(figsize=(10,8))
plt.plot(date,bitcoin_daily.iloc[:, 3].values, label = 'Close', ls="--")
plt.plot(date,bitcoin_daily.iloc[:, 2].values, label = 'Low', ls="--")
plt.plot(date,bitcoin_daily.iloc[:, 1].values, label = 'High', ls="--")
plt.plot(date,bitcoin_daily.iloc[:, 0].values, label = 'Open', ls="--")
plt.xlabel("Date")
plt.ylabel("Prices")
plt.title("Bitcoin Prices by Year")
plt.legend()
plt.show()
```

Figure 14: Check the trend of different measures

```python
[145] #import Seaborn library
import seaborn as sns

#create a function for plotting subplots for the same column
def multiple_plot(x, title,color):
    fig, ax = plt.subplots(3,1,figsize=(15,10),sharex=True)
    sns.distplot(x, ax=ax[0],color=color)
    ax[0].set(xlabel=None)
    ax[0].set_title('Histogram + KDE')
    sns.boxplot(x, ax=ax[1],color=color)
    ax[1].set(xlabel=None)
    ax[1].set_title('Boxplot')
    sns.violinplot(x, ax=ax[2],color=color)
    ax[2].set(xlabel=None)
    ax[2].set_title('Violin plot')
    fig.suptitle(title, fontsize=25)
    plt.tight_layout(pad=5.0)
    plt.show()
```

Figure 15: Function for Multiple plots for Weighted Price

```python
[209] #PLot the actual vs predicted values
plt.figure(figsize=(10,6))
plt.plot(df_monthly_prediction["Weighted_Price"], label="Real Bitcoin Price", color="green")
plt.plot(df_monthly_prediction["Forcasting"], label="Predicted Bitcoin Price", color="blue", ls = '--')
plt.legend()
plt.xlabel("Dates")
plt.ylabel("Prices")


gc.collect()
```

Figure 16: Plot for Predictions

Forecasting the data using SARIMA

```
[210] #create monthly datetimeindex for 1 year

      future_dates = [df_monthly_prediction.index[-1] + DateOffset(months = x)for x in range(1,12)]
      future_dates = pd.to_datetime(future_dates)  +  MonthEnd(0)
      future = pd.DataFrame(index=future_dates)
      df_monthly_prediction = pd.concat([df_monthly_prediction,future])


      gc.collect()
```

```
[211] df_monthly_prediction.tail()
```

```
#fit the model with optimum parameters and 100% data
model = sm.tsa.statespace.SARIMAX(bitcoin_monthly["Weighted_Price"], order=results_dataframe.reset_index().pdq[0],
                                  seasonal_order=results_dataframe.reset_index().s_pdq[0],
            enforce_invertibility=False,enforce_stationarity=False).fit()
print(model.summary().tables[1])
```

```
[213] #Future Prediction
      df_monthly_prediction["Future_forecast"] = model.predict(start=pd.to_datetime("2021-03-31"),end=pd.to_datetime("2022-02-28"))

      gc.collect()
```

Figure 17: Future Forecasts