# Short-term Bitcoin Market Prediction Report

**Mohammad Amin Abbaszadeh**
Under supervision of Dr. Edie Miglio
April 2023

*Numerical Analysis for Machine Learning Course*

**Polytechnic University of Milan**

# Contents

# 1 Introduction

This document provides a comprehensive description of the implementation of a publication named *"Short-term bitcoin market prediction via machine learning"*. The publication utilizes different machine learning models such as GRU, LSTM, FNN, Logistic Regression, Gradient boosting classifier, and Random Forest for a binary classification task.

The feature set used for the prediction tasks in the publication is a combination of Bitcoin returns, Blockchain-based, Asset-based, and Sentiment/interest-based features. These features are minutely updated data from March 4 to December 10, 2019.
In order to have Asset-based features we should have the Bloomberg developer account that I have requested for it. Still, I have not received any response from them, thereby I excluded this set of features from the implementation.
Regarding Blockchain-based features, the feature values are not available minutely. For example, by calling this URL for the Mempool growth feature you can see that there are only 5 or 6 values available per day, so I also excluded this feature set from the implementation. I searched on the web to get these two feature sets from other sources but I could find any resource for it.
To access tweets text related to Bitcoin we should have a Twitter developer account. I have requested it however my request is declined. To this end, I have used a Python package to obtain the tweets' text which is explained in section 2.1.1.
Therefore, Bitcoin returns and Sentiment/interest-based features are selected for our prediction task.

The process of implementation consists of different phases to execute consecutively. Firstly, the data must be collected and processed after that all the feature values must be merged and reshaped into two different datasets for feeding to non-memory or memory models. Besides, the class label for every sample must be calculated.
Secondly, the models must be trained on their related datasets prepared in the previous phase, and hyperparameters must be tuned to get the best performance or prediction accuracy of each model.

In order to evaluate the effectiveness of various models for our intended prediction tasks, each model has been trained and tested on its respective training and testing datasets, tailored to the specific requirements of the model. The document elaborates on each of these components individually, allowing for a comparison of the performance of different models.

# 2 Implementation

## 2.1 Extracting Sentiment/interest-based features

For this section refer to *"Tweets_features.ipynb"* and *"Tweets_Resample.ipynb"* files.

### 2.1.1 Fetching Data

For calculating Twitter sentiment, Twitter sentiment weighted with strength of emotion, and number of tweets features we should have the tweets' text at the first stage. To this end, we leverage a Python package named "snscrape" to fetch the texts of tweets since the Twitter API is not accessible at the moment.
Due to the fact that we get the texts by a generator, having all English Tweets in the given period that include the hashtag bitcoin and do not contain pictures or URLs is challenging and time-consuming. Hence, another constraint is added to the tweets which is tweets must have at least one fave. Ultimately, the total number of *101102* is ready for the next stage which is processing the tweets' text.

### 2.1.2 Pre-processing Tweets' Text

In this stage, various preprocessing techniques for the collected Tweet texts are applied. First, usernames, non-English characters, and additional whitespace are removed. Following, contractions are replaced by their standard form (e.g., replace "isn't" with "is not"). With this aim, a dictionary of all possible contractions and expansions is made manually. In the next step, lemmatization to the Tweet texts is attained to replace inflected word forms with respective word lemmas (e.g., replace "bought" with "buy"). For this purpose, we take advantage of the "nltk" tool in Python.

Eventually, the Google Natural Language API to generate sentiment and estimates of the strength of emotion for each Tweet is used. For using Google NLP tools a procedure must be progressed. 1. Create a Google Cloud Platform account 2. Enable the Natural Language API 3. Create a service account 4. Get API credentials . After going through all the above steps and having a private key as a JSON file we are enabled to make use of the NLP tools of Google for our goal.

Thereafter, the values of sentiment score and sentiment magnitude are calculated per tweet. This process is a bit time-consuming so that can take about 6 hours long for the whole set of our tweets. The challenge that I faced in this process is that at some point the Google API could not proceed with a tweet text and recognize the language of the text as non-English, so I had to restart the process from the beginning, such tweets which had a total number of 15 tweets are skipped.

In the final stage, the sentiment features are resampled to one-minute frequency and sum the sentiment score and weighted sentiment score for each minute as well as counting the number of features per minute. We will add zero values for the minutes in our time frame that there is no data available.

There are also other ways to do sentiment analysis on our tweets such as using "TextBlob" python package to get the sentiment polarity and using the number of exclamation marks as well as capitalized words to get sentiment magnitude.

### 2.2 Asset-based & Blockchain-based Features

There is a Python library named "yfinance" that can be used to get the features. However, the data provided by this library is daily, which is not the case for our work. The Python file named *"Asset_Blockchain.ipynb"* includes a Snippet of code that presents the usage of this library. In the file there is also a script for fetching the Number of Bitcoin Transactions and Mempool growth feature values however these data are daily which is not helpful for our work.

### 2.3 Bitcoin Returns Features & Labels Computing

The bitcoin values of the year 2019 are obtained from Kaggle website. The return values over the horizons(1min, 5min, 15min, 60min, and 1 week) mentioned in the publication are calculated for every time step in the time interval.
Given the Bitcoin returns, it is possible to calculate the label of every observation with respect to a specific horizon in this way that for a given observation the label assigned to it is 0 if the Bitcoin return is less than the mean value of returns. Otherwise, the label would be 1. Please refer to "*Gen_memorydata.ipynb*" file.

### 2.4 Generating Time Series Dataset for Models With Memory

After having the whole feature values for every observation and the corresponding label, the dataset for training memory models must be generated. To this end, we first merge the bitcoin returns ( a specific period return as well as a 1-week return) and the three sentiment features, which totally we get 5 features. Then, we standardize the data.

To clarify more, we have a minutely time series dataset starting from "2019-03-04 00:00:00" to "2019-12-10 23:59:59" that for each minute we have five different values. In order to have the final dataset to feed to the models we must reshape it to (# samples, # time_steps, # features). The number of time steps(look_back period) is 120.

To be more explicit, starting from the first observation at time $t = 0$ to the observation at time $t+119$ the five feature values are collected by a sliding window of length

120 to shape up a sample, and the label for this sample is the class of the observation at time step *t+120*. The window is slid forward to the extent that it includes the last minute in our date frame. Finally, we have a dataset of dimensions (405720, 120, 5) and labels of size 405720 including only 1 and 0.

### 2.5 Generating Time Series Dataset for Models Without Memory

Starting from the base dataset that includes all feature values per minute we can make a dataset for models without memory. The significant difference between the dataset shape here with the previous case is that in non-memory models, the model accepts a two-dimensional array as input whereas for the memory models, a three-dimensional array is required.

We extract additional features by computing return values over several time intervals before a time step. The time intervals are as following: { [0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 10], [10, 20], [20, 40], [40, 60], [60, 80], [80, 100], [100, 120] }. Thereby, for every time step, we have 12 different Bitcoin return features as well as a 1 week Bitcoin return. Furthermore, Regarding the sentiment features, they are summed up respected to each interval. In the end, we will have 49 features.

### 2.6 Models with Memory

GRU (Gated Recurrent Unit) and LSTM (Long Short-Term Memory) are both types of recurrent neural networks (RNNs) that are designed to address the vanishing gradient problem encountered in traditional RNNs. Both GRU and LSTM have memory cells that store information over time, but they differ in their architecture and computation.

GRUs have fewer parameters than LSTMs and are easier to train. They have two gates, the update gate and the reset gate, which determine how much information from the previous time step should be retained in the memory cell. The update gate decides which information from the previous time step should be carried forward, while the reset gate decides which information should be discarded in favor of new information from the current time step. This makes GRUs faster and computationally more efficient than LSTMs.

LSTMs have three gates, the input gate, the forget gate, and the output gate, which work together to control the flow of information into and out of the memory cell. The input gate determines how much new information from the current time step should be added to the memory cell, while the forget gate decides which information from the previous time step should be discarded. The output gate controls how much information from the memory cell should be output to the next layer. LSTMs are more expressive

and have more capacity to capture long-term dependencies in the data.

In summary, GRUs are simpler and faster than LSTMs, but may have less capacity to capture complex long-term dependencies in the data. LSTMs are more expressive and better suited for tasks that require modeling of long-term dependencies, but are slower and require more training data. The choice between GRU and LSTM depends on the specific requirements of the task and the available resources.

In this section, the process behind building and training the models is presented. For all models, we use the first five ninths of the data (approximately 5 months) to generate a training set. The subsequent one-ninth of the data (approximately 1 month) forms a validation set for hyperparameter tuning, including regularization techniques, such as early stopping. The remaining one-third of the data (approximately 3 months) is used to test our models. The class balance of samples in the training set is 0.87 for the positive class and 1.17 for the other class.

#### 2.6.1 Long Short-Term Memory (LSTM)

This type of model is robust for extracting knowledge from time series data. The model is implemented in Python leveraging the "Keras" and "Tensorflow" tools.
The architecture of the model is as follows based on the paper: **1. Input 2. Batch Normalization 3. LSTM 4. Dropout 5. One Dense Neuron 6. Sigmoid 7. Output**. Starting from the Batch Normalization layer, due to the fact that the data are standardized before feeding models, I omitted this layer in my implementation because it is somehow the first layer.

A significant difference between the paper architecture of this model and the model implemented here is the number of LSTM blocks. Since the number of 15 features is used, the optimal number of LSTM blocks is 256 blocks based on the parameter tuning grid mentioned in the paper. In this implementation, we only use 5 features, so a less complex model is needed to learn the pattern hidden in the data.

The setting and configuration of the model and the hyperparameters tried to conform with paper as much as possible at the beginning. A remarkable point that is observed during the training of the model is that the metric of "val_binary_accuracy" is not improved from the first epoch until the last epoch in the most different configurations of the model whereas the metrics of the "val_loss" as well as "train_loss" are improved normally. In some cases, the "train_binary_accuracy" is stable during learning, and in other cases, it is increasing normally.

In order to address this issue, at first, it is surmised that the model is overfitting, so different techniques to prevent the overfitting are utilized such as adding L1 or L2

regularization to the LSTM layer and increasing the dropout layer percentage. However, these modifications did not lead to a substantial improvement.

The other method, that is used to impede the overfitting is to augment data. To achieve this, several different approaches were used such as augmenting the data by adding some noise to the data manually or flipping the time series samples and adding the new flipped samples to the dataset. A notable result did not conclude after augmenting data.

Having the below values for parameters, we obtain an accuracy of **51.62%** for the 1-minute prediction horizon. With different values of hyperparameters accuracy up to **51.84%** were achieved however in these cases the value of val_binary_accuracy was not changing or improving during the training phase at all. Therefore, the above setting for our neural network is preferred as the best model. For further consideration please refer to parameter tuning csv files regarding the parameter tuning search grid.

- # LSTM blocks = 20
- Learning Rate = 0.0001
- Optimizer = Adam
- Regularizer = 0.5
- Dropout = 0.5
- Batch size = 5000

So far, the parameter **"return_sequence"** of the LSTM layer were *True* in the code and the output of the LSTM layer were not flatten before forwarding to the next layer. In this case, the accuracy of almost 48% to 52% obtained for 1-minute prediction horizon while tuning the hyperparameters of the model, which is complying with the paper results. Since, we are not aiming stacking several LSTM layer, the **"return_sequence"** parameter must be *false* or the output of the LSTM layer must become flatten. Once the parameter has been set to *false*, a significant improvement in accuracy of the model compared with the paper result achieved. Likewise, a sensible result regarding the trend of "val_binary_accuracy" concluded.

It is essential to emphasize that, besides the above modification, using *"tanh"* activation function rather than *"sigmoid"* and decreasing the number of batch size form 5000 to 128 enhanced the model.

At the end, we could get much advanced results with following configuration.

- # LSTM blocks = 20

- Learning Rate = 0.0005
- Optimizer = Adam
- Regularizer = 0
- Dropout = 0.5
- Batch size = 128

Moreover, a prominent alternation in the architecture of the model that may improve the result is that adding a dense layer before the dense layer with one neurons. This alternation is also tested, so a dense layer with 50 neurons is added and after retraining the models the comparison of the models performance in both cases is represented in Table 1.

Table 1: The comparison of the performance of the LSTM layer with an additional dense layer

| – | **1min** | **5min** | **15min** | **60min** |
|---|---|---|---|---|
| With addition dense layer | 52.59% | 82.02% | 85.55% | 91.33% |
| Without addition dense layer | 52.53% | 82.46% | 85.53% | 90.05% |

As it is obvious, the additional dense layer does effect remarkably, nevertheless, it is preferred to use the model with additional dense layer as the final one.

#### 2.6.2 Gated Recurrent Unit (GRU)

The fundamental difference in the implementation of the GRU models compared with the LSTM models is replacing the LSTM layer with a GRU layer. Working with the LSTM and GRU model were simultaneously, so the issue of low accuracy on test set and stable "val_binary_accuracy" were common in both of the models if we configure the model as it is done in the paper.

At the beginning, considering the 1-minute prediction task, to address these issues, several approaches have been done. Such as adding a dense layer with 50 number of neurons before the last one-neuron layer, changing batch size, using the "tanh" activation function instead of "sigmoid" for the GRU layer, and adding a dense layer of 50 neurons before the last dense layer with 1 neuron. Eventually, with following setting a logical result derived.

- # GRU blocks = 20
- Learning Rate = 0.001
- Optimizer = Adam
- Regularizer = 0

- Dropout = 0.5
- batch size = 128

Having the above values for parameters, we obtain an accuracy of **52.42%** for the 1-minute prediction horizon. For further consideration please refer to parameter tuning csv files regarding the parameter tuning search grid.

## 2.7 Non-Memory Models

In this section, the implementation of the rest of the models including Feed Forward Neural Networks, Gradient Boosting Classifier, Logistic Regression, and Random Forest are discussed.
It is notable to mention that the models here do not use the same dataset used in the previous section. As already mentioned, there is a different shape of data regarding models with no memory.The split of data to train, validation, and test sets is same as in the previous case.

After loading the dataset, it is important to standardize it using "StandarScaler" in the "sklearn" package of Python. We first fit the scaler on the train data using "fit_transform", then we transform the validation and test data using transform. The scaler ensures that all features have zero mean and unit variance.

### 2.7.1 Feed Forward Neural Networks

In light of the fact that the number of 15 features are used in principle in the paper whereas 5 features are utilized here so the number of hidden dense layers may play an important role to get a reasonable accuracy. Therefore, as well as hyperparameter tuning the number of hidden dense layers is changed in the different training settings. Regarding the number of dense neurons in each layer, it is half of the number of neurons in the previous layer in the case of having several hidden dense layers.
After training the model in different settings considering the 1-minute prediction we could get the accuracy of **54.11%** taking into account the trend of va_binary_accuray.

- learning rate = 0.005
- # hidden layers = 5
- regularizer = 0
- # neurons in the first hidden dense layer= 512
- dropout = 0.7

- optimizer = 'SGD'
- batchsize = 128

For further consideration please refer to parameter tuning csv files regarding the parameter tuning search grid.

#### 2.7.2 Gradient Boosting Classifier(GBC)

Gradient Boosting Classifier is a type of machine learning algorithm used for classification tasks. It is based on the idea of boosting, which involves combining multiple weak classifiers to form a strong classifier. Gradient Boosting Classifier is a specific implementation of the gradient boosting algorithm that uses decision trees as the weak learners.
The model is built by "xgboos" package. For the 1-minute prediction horizon, we get the accuracy of **50.16%** whit the following set of parameters.

- max_depth = 30
- eta = 0.5
- objective = 'binary:logistic'
- eval_metric = 'logloss'
- lambda = 0.5
- alpha = 0.5

Describing the above parameters, "max_depth" is the maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit. The "eta" is the learning rate. The "eval_metric" is the evaluation metric for the validation data, so we use "logloss" because our problem is a classification. The 'lambda' and 'alpha' parameters are L2 and L1 regularization terms on weights respectively.

#### 2.7.3 Random Forest (RF)

Random Forest is an ensemble learning algorithm for classification tasks. It constructs multiple decision trees using bootstrap samples and random feature subsets and then combines their predictions to make a final prediction. Random Forest Classifier is implemented here taking advantage of the "sklearn" library.
With the 1-minute prediction horizon, we get the accuracy of **48.16%** on the test set of data using the following parameters.

- min_samples_leaf= 0.2
- n_estimators= 100

The "min_samples_leaf" parameter denotes the minimum number of samples required to be at a leaf node. Here we have the minimum fraction of instances per leaf = 20%. The "n_estimators" denotes the number of trees in the forest.

#### 2.7.4 Logistic Regression (LR)

Logistic regression is a statistical method used to model the relationship between a binary dependent variable and one or more independent variables. It estimates the probability of the dependent variable taking on a particular value (0 or 1) based on the values of the independent variables.
The model is coded using the "sklearn" package. For the 1-minute horizon, we get the accuracy of **48.15%** on the test set. The process of training the model is done in several settings by defining the penalty parameter to L1 or L or using different solvers, however, a significant improvement in the accuracy has not been observed.

## 3 Result

Table 2: Accuracy of machine learning models on several time horizons

| **Prediction Horizon** | **LSTM** | **GRU** | **FNN** | **LR** | **GBC** | **RF** |
|---|---|---|---|---|---|---|
| 1 minute | 52.59 | 52.42 | 54.11 | 48.15 | 50.16 | 48.16 |
| 5 minute | 82.02 | 82.60 | 54.12 | 52.08 | 51.42 | 50.75 |
| 15 minute | 85.55 | 85.60 | 54.64 | 53.09 | 51.70 | 51.25 |
| 60 minute | 91.33 | 90.98 | 53.69 | 53.16 | 51.84 | 50.77 |

The accuracy of models on different prediction horizons are provided in the Table 2. Two important observations are deduced based on the table.

First, the accuracy of the models with memory outperform significantly rather than models without memory, which is expected so since our data is time series. RNN( Recurrent Neural Network) models perform better than other models for time series data because they can handle sequences of variable length and capture the temporal dependencies in the data, making them suitable for modeling time series data. Additionally, RNN models can learn and remember past information, making them effective for predicting future values based on past patterns.

Secondly, the accuracy increases with longer prediction horizons, which is more evident in RNN models. Because, the GRU and LSTM can remember data from past while other models cannot, leads to much higher accurate prediction in 5min, 15min, and 60min horizon compared with other models.

To sum up, models without memory cannot perform accurately in bitcoin market and the best model in this context is LSTM model.