Crypto Currency Prediction

A PROJECT REPORT

SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS

FOR THE AWARD OF DEGREE

OF

BACHELOR OF TECHNOLOGY

IN

SOFTWARE ENGINEERING

Submitted by:

Aman Gupta (2K16/SE/007)

Himanshu Nain (2K16/SE/030)

Under the supervision of

MR. Nipun Bansal

Department of Computer Science & Engineering

Delhi Technological University

(Formerly Delhi College of Engineering)

Bawana Road, New Delhi-110042

DEC, 2019

Department of Computer Science & Engineering

Delhi Technological University

(Formerly Delhi College of Engineering)

Bawana Road, New Delhi-110042

CANDIDATE'S DECLARATION

We, Aman Gupta (2K16/SE/007), Himanshu Nain (2K16/SE/030) of B.Tech., hereby declare that the project report titled "Crypto Currency Prediction" which is submitted by us to the Department of Computer Science & Engineering (Software Engineering), Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of degree of Bachelor of Technology is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.

Place: Delhi

Date: 15/12/2019

Aman Gupta

Himanshu Nain

Department of Computer Science & Engineering

Delhi Technological University

(Formerly Delhi College of Engineering) Bawana Road,

New Delhi-110042

CERTIFICATE

I hereby certify that project project titled “Crypto Currency Prediction” which is submitted by Aman Gupta (2K16/SE/007), to the Department of Computer Science & Engineering, Delhi Technological University, Delhi in partial fulfillment of the of the requirement for the award for the award of degree of Bachelor of Technology, is a record of project work under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Mr. Nipun Bansal
Assistant Professor
(Supervisor)

Department of Computer Science & Engineering

Delhi Technological University

(Formerly Delhi College of Engineering)

Bawana Road, New Delhi-110042

CERTIFICATE

I hereby certify that project project titled "Crypto Currency Prediction" which is submitted by Himanshu Nain (2K16/SE/030), to the Department of Computer Science & Engineering, Delhi Technological University, Delhi in partial fulfillment of the of the requirement for the award for the award of degree of Bachelor of Technology, is a record of project work under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

Mr. Nipun Bansal
Assistant Professor
(Supervisor)

ABSTRACT

The data on the net is generated at much faster rate than a decade ago and such data possess a great value and worth if harnessed properly. Users are getting more and more involved on social media platforms and discussion forums and hence a large amount of data containing reviews and opinions is generated. The data generated from such reviews is of prime importance to both the product or service buyer and also the seller of the product or service. Much work has been done in this direction which accepts reviews from user and hence generates an aspect based comprehensive output containing an aspect and its associated reviews.

Our project aims at quantifying those aspects rather than displaying the reviews associated with that aspect. So for each aspect we search through all such reviews available to generate a net averaged positive, negative and neutral score. Because humans have this tendency to understand a quantitative measure much better and make decisions. For each aspect we produce an output indicating how many percent of reviews gives positive, negative or neutral feedbacks about that.

The idea though is easy to interpret but not much discussed and hence our project focuses on the human capability to understand the quantitative measures much better and therefore a averaged score for each aspect of product or service is generated which was not the case with the previous works done in this direction.

Evidently the large amount of data is generated such data though possess immense value but at the same time this valuable data differs from the traditional data in terms of its structure and characteristics like for example usage of slangs and emoticons to convey emotions like sarcasm, happiness etc.

The Idea behind our work in this direction is to identify and explain a given product from each and every dimension possible.

ACKNOWLEDGEMENT

With this project towards its completion, we would like to extend our sincere gratitude towards Mr. Nipun Bansal(Assistant Professor, CSE Department) who acted as a direct guide during the entire course. We are sincerely grateful for their invaluable insight, the time and effort they dedicated to this work and, especially, for their contagious enthusiasm and optimism.

We are also grateful to the Department of Computer Science & Engineering at Delhi Technological University for presenting us with this opportunity which was essential in enhancing learning and promoting research culture among us.

We would like to thank our families for their unconditional support and faith in our endeavors. Their prayers and wishes have played a major role in the successful completion of this project.

Aman Gupta Himanshu Nain

CONTENTS

LIST OF FIGURES

Figure 3.4 Results – Bidirectional Vanilla

CHAPTER 1

INTRODUCTION

1.1 GENERAL

A cryptocurrency is a digital or virtual currency that is secured by cryptography, which makes it nearly impossible to counterfeit or double-spend. Many crypto currencies are decentralized networks based on blockchain technology—a distributed ledger enforced by a disparate network of computers. A defining feature of crypto currencies is that they are generally not issued by any central authority, rendering them theoretically immune to government interference or manipulation.

- A cryptocurrency is a new form of digital asset based on a network that is distributed across a large number of computers. This decentralized structure allows them to exist outside the control of governments and central authorities.
- The word “cryptocurrency” is derived from the encryption techniques which are used to secure the network.
- Blockchains, which are organizational methods for ensuring the integrity of transactional data, is an essential component of many crypto currencies.
- Many experts believe that blockchain and related technology will disrupt many industries, including finance and law.
- Crypto currencies face criticism for a number of reasons, including their use for illegal activities, exchange rate volatility, and vulnerabilities of the infrastructure underlying them. However, they also have been praised for their portability, divisibility, inflation resistance, and transparency.

Crypto currencies are systems that allow for the secure payments online which are denominated in terms of virtual "tokens," which are represented by ledger entries internal to the system. "Crypto" refers to the various encryption algorithms and cryptographic techniques that safeguard these entries, such as elliptical curve encryption, public-private key pairs, and hashing functions.

Advantages

Crypto-currencies hold the promise of making it easier to transfer funds directly between two parties, without the need for a trusted third party like a bank or credit card company. These transfers are instead secured by the use of public keys and private keys and different forms of incentive systems, like Proof of Work or Proof of Stake.

In modern cryptocurrency systems, a user's "wallet" or account address, has a public key, while the private key is known only to the owner and is used to sign transactions. Fund transfers are completed with minimal processing fees, allowing users to avoid the steep fees charged by banks and financial institutions for wire transfers.

Disadvantages

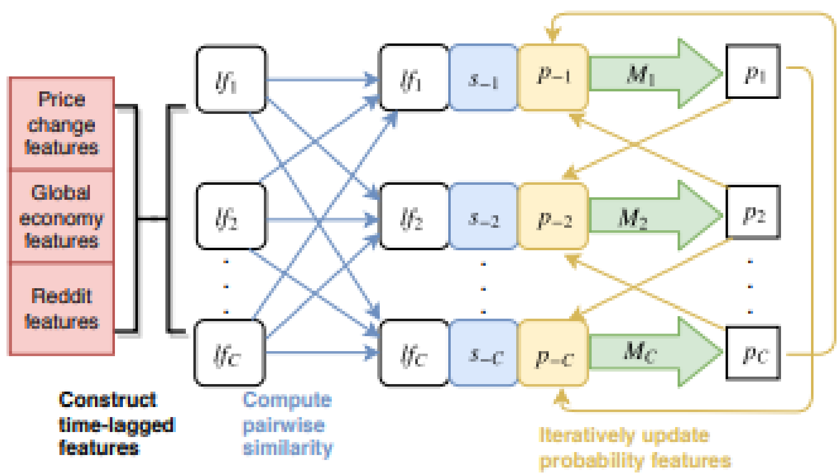
The semi-anonymous nature of cryptocurrency transactions makes them well-suited for a host of illegal activities, such as money laundering and tax evasion. However, cryptocurrency advocates often highly value their anonymity, citing benefits of privacy like protection for whistleblowers or activists living under repressive governments. Some crypto-currencies are more private than others.

Bitcoin, for instance, is a relatively poor choice for conducting illegal business online, since the forensic analysis of the Bitcoin blockchain has helped authorities to arrest and prosecute criminals. More privacy-oriented coins do exist, however, such as Dash, Monero, or ZCash, which are far more difficult to trace.

Although there are some studies that deal with both the task of predicting stock market price movements, as well as the development of profitable trading strategies based on those predictions, it is important to verify the applicability of such studies in new and emerging markets; in particular the cryptocurrency market. This market is characterized by high volatility, no closed trading periods, relatively smaller capitalization, and high market data availability. The financial feasibility of the cryptocurrency market in relation to other markets has been documented and the algorithms upon which the crypto-currencies operate have been validated in other fields as well. The cryptocurrency market seems to behave independently from the other financial markets, but there is a strongly influenced by Asian economies. Part of the appeal behind this market is that the technology used for mining cryptocurrency provides feasible alternative to more traditional markets such as gold. These characteristics have attracted a considerable amount of capital, however up to now there are few studies that have attempted to create profitable trading strategies in the cryptocurrency market. Another point of interest in the cryptocurrency market is the large-scale of available public sentiment data, particularly from social networks.

1.2 RELATED WORK

Our work is related to Chongyang Bai, Tommy White, Linda Xiao, V.S. Subrahmanian’s work on **CryptoCurrency Prediction Engine**. Using available training corpus from coinmarketcap.com, they designed and experimented a number of methods for prediction.



They built their model on the idea of collective classification where, instead of predicting the price of each of the 21 cryptocurrencies individually, we try to predict them simultaneously. However, their collective classification algorithm is novel in many respects — first through the use of similarity metrics to compute pairwise similarities between the feature vectors of every pair of cryptocurrencies, and second, through the use of probabilities of prices going up rather than raw predictions (up vs. down). The C2P2 algorithm is shown in detail as Algorithm 1 and is visualized in Figure below. Informally speaking, the algorithm works as follows to predict whether the prices of cryptocurrencies will go up on day d.

Algorithm 1:

Input : Features $\boldsymbol{f}_{c,d-L}, \ldots \boldsymbol{f}_{c,d-1}$, learned classifier M_c $\forall c \in [1, \ldots C]$, similarity function $S(\cdot, \cdot)$, lag L, convergence threshold ϵ, maximum iteration number I

Output: $\boldsymbol{p}_d = (p_{1,d}, \ldots p_{C,d})$. Predicted probabilities of prices going up for all C cryptocurrencies on day d

/* sampling from uniform distribution */

$\boldsymbol{p}_d = (p_{1,d}, \ldots p_{C,d}) \sim U(0, 1)$

for $c \in [1, \ldots C]$ **do**

  /* construct time-lagged features $\boldsymbol{lf}$ */

  $\boldsymbol{lf}_{c,d} = (\boldsymbol{f}_{c,d-L}, \boldsymbol{f}_{c,d-L+1}, \ldots \boldsymbol{f}_{c,d-1})$

end

/* compute pairwise similarity of $\boldsymbol{lf}$ */

for $c \in [1, \ldots C]$ **do**

  $\boldsymbol{s}_{-c,d} = \underset{\forall i \neq c}{\text{concat}}(S(\boldsymbol{lf}_{i,d}, \boldsymbol{lf}_{c,d}))$

end

$iter = 0$

do

  $iter = iter + 1$

  $\boldsymbol{p}'_d = \boldsymbol{p}_d$

  Set $\boldsymbol{p}_{-c,d} = \underset{\forall i \neq c}{\text{concat}}(p_{i,d})$ for each $c \in [1, \ldots, C]$

  for $c \in [1, \ldots C]$ **do**

    /* model M_c predicts for coin c by concatenating 3 sets of features */

    $p_{c,d} = M_c(\boldsymbol{lf}_{c,d}, \boldsymbol{s}_{-c,d}, \boldsymbol{p}_{-c,d})$

  end

while $\|\boldsymbol{p}_d - \boldsymbol{p}'_d\|_2 > \epsilon$ *or* $iter < I$;

return $\boldsymbol{p}_t$

1.2.1 Related work

Cryptocurrency price prediction in current literature is usually framed as a regression problem, a market simulation to calculate ROI, or as a classification problem in predicting the sign of future price change. Because cryptocurrencies are not managed by a central bank or government, they do not subscribe to the classical economic theories of supply and demand. Instead, additional features, ranging from digital currency specific features to social media trends, are extracted to better predict the price. Bitcoin price prediction is the topic of most papers in the area of cryptocurrency price prediction. References leveraged blockchain features to predict Bitcoin price with varying degrees of success. Sin et al. achieved 64% classification accuracy in predicting the sign of price change using an ensemble of neural networks tuned with genetic algorithms. Jang et al. produced a regression with a Mean Average Percent Error (MAPE) of 0.0138 using Bayesian Neural Networks. McNally et al. reported 53% accuracy in price change sign prediction using an LSTM neural network.

1.2.2 CONCLUSION

The problem of predicting the up/down movements of cryptocurrencies is of great interest to both the financial industry and to individual consumers. They developed the C2P2 algorithm that has two innovations: (i) the use of similarities between cryptocurrency feature vectors, and (ii) that uses tentative predictions about (C − 1) cryptocurrency's up/down price movements to predict that of the Cth cryptocurrency. In addition, they used Reddit data for their predictions. They test C2P2 on the 21 cryptocurrencies with the highest market capitalization (according to coinmarket.com) and show that C2P2 beats out two recent competitors with substantial lifts which are statistically significant.

1.3 Applications

1.

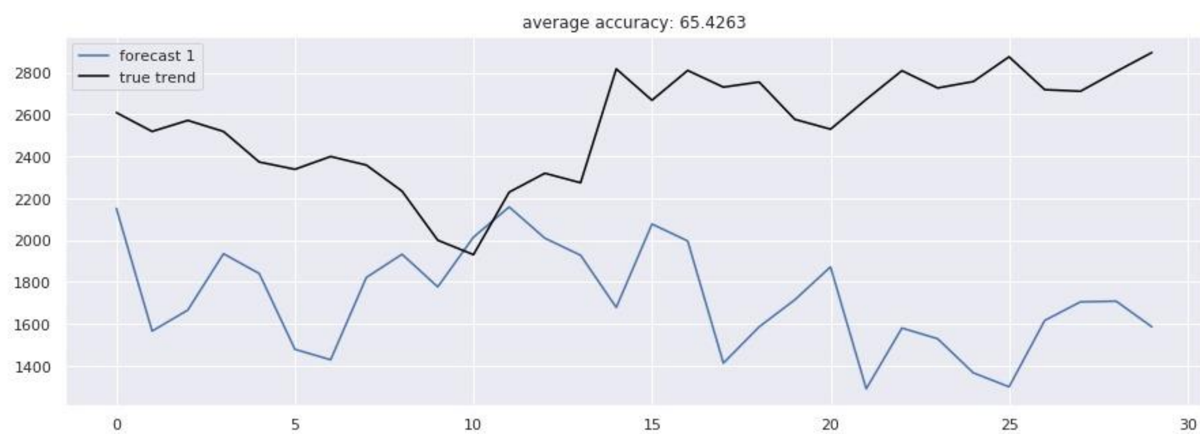


Figure 1.2 : Application use case 1 Vanilla

2.

Figure 1.3 : Application use case 2- GRU

CHAPTER 2

OUR METHOD

2.1 DATASET

We used the dataset available on www.coinmarketcap.com. The data is available in csv format. It consists of seven columns and 1363 values of BTC. The various columns in the dataset are:

1. date
2. open
3. high
4. low
5. close
6. Volume
7. Market cap

Figure 2.1 : Original Dataset

Date	Open*	High	Low	Close**	Volume	Market Cap
11-Nov-13	325.41	351.27	311.78	342.44	0	4,101,635,027
12-Nov-13	343.06	362.81	342.8	360.33	0	4,317,726,291
13-Nov-13	360.97	414.05	359.8	407.37	0	4,883,103,453
14-Nov-13	406.41	425.9	395.19	420.2	0	5,038,817,795
15-Nov-13	419.41	437.89	396.11	417.95	0	5,013,561,020
16-Nov-13	417.28	450.26	415.57	440.22	0	5,282,849,105
17-Nov-13	440.96	500.58	440.24	492.11	0	5,907,842,064
18-Nov-13	496.58	703.78	494.94	703.56	0	8,449,069,629
19-Nov-13	712.76	806.11	456.39	584.61	0	7,022,949,161
20-Nov-13	577.98	599.65	448.45	590.83	0	7,100,077,964
21-Nov-13	594.32	733.4	577.29	722.43	0	8,684,240,726
22-Nov-13	724.07	780.85	668.13	771.44	0	9,276,681,716
23-Nov-13	771.7	844.97	771.7	797.82	0	9,597,335,799
24-Nov-13	795.63	807.36	722.87	774.25	0	9,317,034,156
25-Nov-13	773.02	810.68	754.43	799.11	0	9,619,646,225
26-Nov-13	805.73	928.54	800.8	928.1	0	11,176,110,593
27-Nov-13	923.85	1,001.96	891.68	1,001.96	0	12,069,885,699
28-Nov-13	1,003.38	1,077.56	962.17	1,031.95	0	12,435,823,060
29-Nov-13	1,042.01	1,146.97	1,000.64	1,131.97	0	13,646,039,846
30-Nov-13	1,129.37	1,156.14	1,106.61	1,129.43	0	13,620,389,321
1-Dec-13	1,128.92	1,133.08	801.82	955.85	0	11,531,708,948
2-Dec-13	951.42	1,055.42	938.41	1,043.33	0	12,591,271,606
3-Dec-13	1,046.40	1,096.00	1,011.21	1,078.28	0	13,018,451,838
4-Dec-13	1,077.58	1,156.12	1,070.16	1,151.17	0	13,903,428,351
5-Dec-13	1,152.73	1,154.36	897.11	1,045.11	0	12,626,914,509
6-Dec-13	1,042.38	1,042.38	829.45	829.45	0	10,025,645,095
7-Dec-13	835.32	854.64	640.22	698.23	0	8,442,962,249
8-Dec-13	697.31	802.51	670.88	795.87	0	9,627,500,113
9-Dec-13	793.8	921.93	780.9	893.19	0	10,808,715,488
10-Dec-13	892.32	997.23	892.32	988.51	0	11,966,951,486
11-Dec-13	989.07	1,001.58	834.23	878.48	0	10,638,546,534
12-Dec-13	882.78	901.94	844.95	873.26	0	10,578,889,955
13-Dec-13	874.98	941.79	860.05	892.58	0	10,816,819,988
14-Dec-13	899.85	904.65	858.36	872.6	0	10,578,987,915
15-Dec-13	875.29	886.16	825	876.12	0	10,625,517,651
16-Dec-13	880.33	882.25	668.25	705.97	0	8,565,534,010
17-Dec-13	706.37	754.83	630.88	682.12	0	8,279,623,719
18-Dec-13	678.2	679.32	420.51	522.7	0	6,347,159,168
19-Dec-13	519.06	707.23	502.89	691.96	0	8,406,397,153
20-Dec-13	694.22	729.16	595.33	625.32	0	7,599,576,492
21-Dec-13	619.9	654.27	579.17	605.66	0	7,363,114,611
22-Dec-13	601.78	666.74	585.64	617.18	0	7,505,757,423
23-Dec-13	613.06	680.91	611.04	673.41	0	8,192,487,202
24-Dec-13	672.36	684.39	645.71	665.58	0	8,100,042,042
25-Dec-13	666.31	682.7	649.48	682.21	0	8,305,173,374

For our purpose, we modified the dataset. We removed the column `volume` and the `market cap` column values.

	A	B	C	D	E
1	Date	Open*	High	Low	Close**
2	11-Nov-13	325.41	351.27	311.78	342.44
3	12-Nov-13	343.06	362.81	342.8	360.33
4	13-Nov-13	360.97	414.05	359.8	407.37
5	14-Nov-13	406.41	425.9	395.19	420.2
6	15-Nov-13	419.41	437.89	396.11	417.95
7	16-Nov-13	417.28	450.26	415.57	440.22
8	17-Nov-13	440.96	500.58	440.24	492.11
9	18-Nov-13	496.58	703.78	494.94	703.56
10	19-Nov-13	712.76	806.11	456.39	584.61
11	20-Nov-13	577.98	599.65	448.45	590.83
12	21-Nov-13	594.32	733.4	577.29	722.43
13	22-Nov-13	724.07	780.85	668.13	771.44
14	23-Nov-13	771.7	844.97	771.7	797.82
15	24-Nov-13	795.63	807.36	722.87	774.25
16	25-Nov-13	773.02	810.68	754.43	799.11
17	26-Nov-13	805.73	928.54	800.8	928.1
18	27-Nov-13	923.85	1,001.96	891.68	1,001.96
19	28-Nov-13	1,003.38	1,077.56	962.17	1,031.95
20	29-Nov-13	1,042.01	1,146.97	1,000.64	1,131.97
21	30-Nov-13	1,129.37	1,156.14	1,106.61	1,129.43
22	1-Dec-13	1,128.92	1,133.08	801.82	955.85
23	2-Dec-13	951.42	1,055.42	938.41	1,043.33
24	3-Dec-13	1,046.40	1,096.00	1,011.21	1,078.28
25	4-Dec-13	1,077.58	1,156.12	1,070.16	1,151.17
26	5-Dec-13	1,152.73	1,154.36	897.11	1,045.11
27	6-Dec-13	1,042.38	1,042.38	829.45	829.45
28	7-Dec-13	835.32	854.64	640.22	698.23
29	8-Dec-13	697.31	802.51	670.88	795.87
30	9-Dec-13	793.8	921.93	780.9	893.19
31	10-Dec-13	892.32	997.23	892.32	988.51
32	11-Dec-13	989.07	1,001.58	834.23	878.48
33	12-Dec-13	882.78	901.94	844.95	873.26
34	13-Dec-13	874.98	941.79	860.05	892.58
35	14-Dec-13	899.85	904.65	858.36	872.6
36	15-Dec-13	875.29	886.16	825	876.12
37	16-Dec-13	880.33	882.25	668.25	705.97
38	17-Dec-13	706.37	754.83	630.88	682.12
39	18-Dec-13	678.2	679.32	420.51	522.7
40	19-Dec-13	519.06	707.23	502.89	691.96
41	20-Dec-13	694.22	729.16	595.33	625.32
42	21-Dec-13	619.9	654.27	579.17	605.66
43	22-Dec-13	601.78	666.74	585.64	617.18
44	23-Dec-13	613.06	680.91	611.04	673.41
45	24-Dec-13	672.36	684.39	645.71	665.58
46	25-Dec-13	666.31	682.7	649.48	682.21

Sheet1 Sheet2 Sheet3

Figure 2.2 : Modified Dataset

2.2 PROPOSED METHODOLOGY

The basic flow of our proposed method is as follows:

1. Simple Average

Forecasts are produced by taking the average of all previously observed values.

2. Moving Average

The moving average of a period (extent) *m* is a series of successive averages of *m* terms at a time. The data set used for calculating the average starts with first, second, third and etc. at a time and *m* data taken at a time.

3. Weighted Average

In moving Average Forecast, the weights given to the m values were all equal. If we consider the case where these weights can be different, this type of forecasting is called weighted moving average.

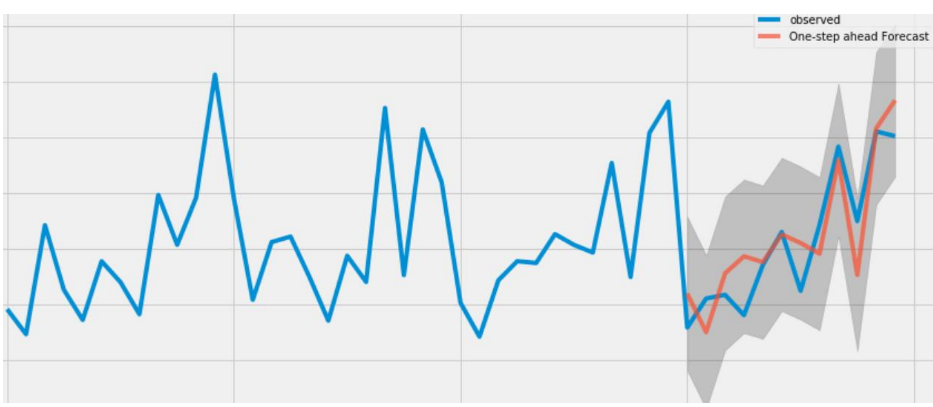


Figure 2.3 Time Series Analysis

4. Linear Regression

Linear regression is a linear model, e.g. a model that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x).

When there is a single input variable (x), the method is referred to as simple linear regression. When there are multiple input variables, literature from statistics often refers to the method as multiple linear regression.

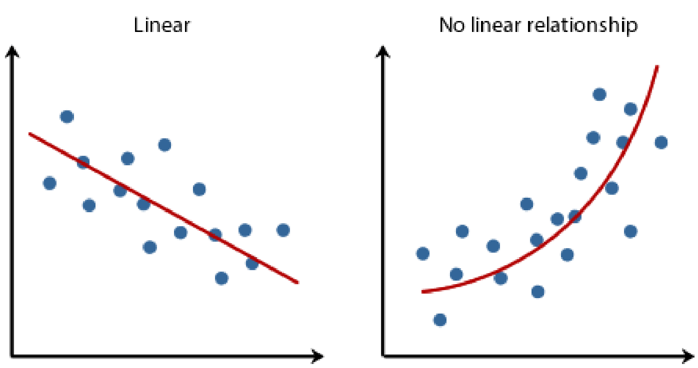


Figure 2.4 : Linear Regression

5. Support Vector Regression

Support vector regression (SVR) is characterized by the use of kernels, sparse solution, and VC control of the margin and the number of *support vectors*. Although less popular than SVM, SVR has been proven to be an effective tool in real-value function estimation. As a supervised-learning approach, SVR trains using a symmetrical loss function, which equally penalizes high and low misestimates.

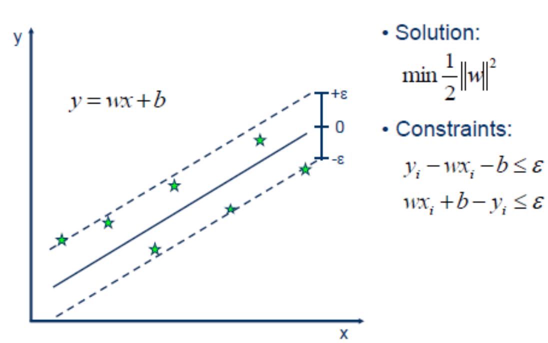


Figure 2.5: Support Vector Regression

6. RNN

Recurrent Neural Network (RNN) are a type of Neural Network where the output from previous step are fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is Hidden state, which remembers some information about a sequence.

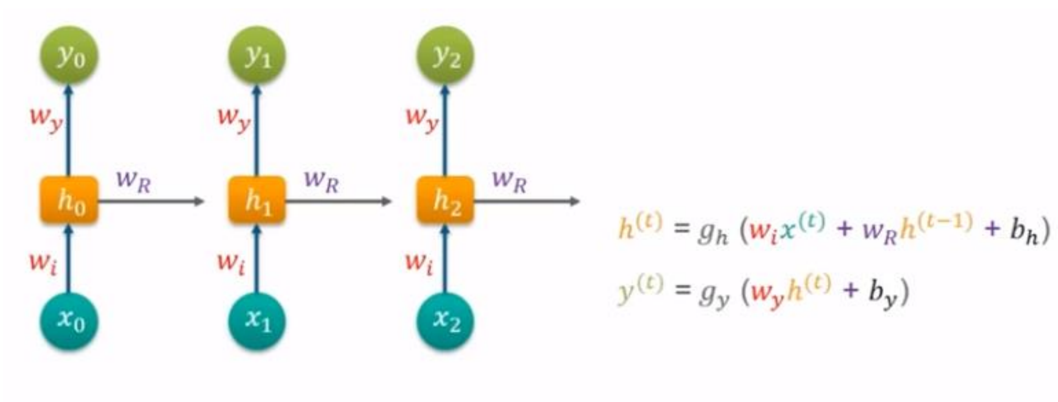


Figure 2.6: RNN

7. LSTM

LSTM was created as the solution to short-term memory. It has internal mechanisms called gates that can regulate the flow of information. These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions.

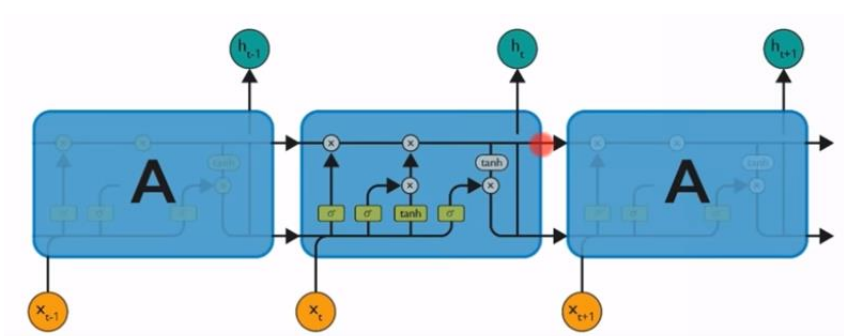
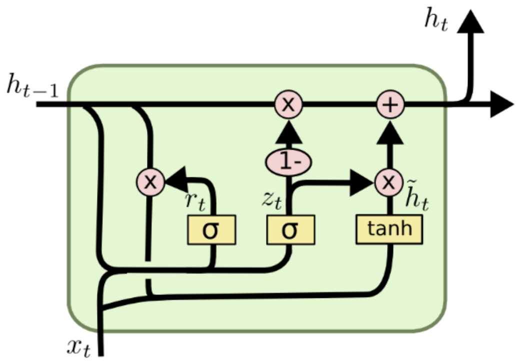


Figure 2.7: LSTM

8. GRU



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 2.8: GRU

The GRU is like a long short-term memory (LSTM) with forget gate but has fewer parameters than LSTM, as it lacks an output gate.[GRU's performance on certain tasks of polyphonic music modeling and speech signal modeling was found to be similar to that of LSTM. GRUs have been shown to exhibit even better performance on certain smaller datasets.

9. Vanilla

It is the standard backpropagation learning algorithm introduced by D.E. Rumelhart and J.L. McClelland and is implemented in Stuttgart Neural Network Simulator(SNNS). It is the most common learning algorithm. Its definition reads as follows:

This algorithm is also called online backpropagation because it updates the weights after every training pattern.

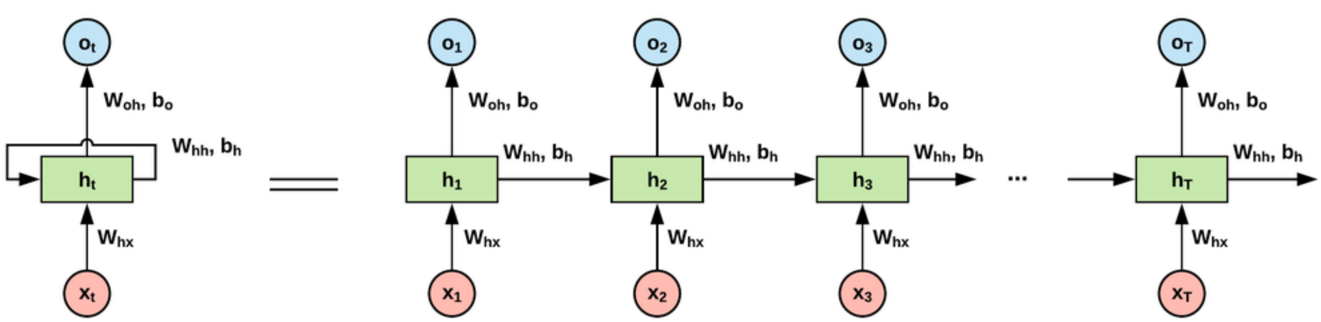


Figure 2.9: Vanilla NN

10. Bidirectional LSTM

Bidirectional LSTMs are an extension of traditional LSTMs that can improve model performance on sequence classification problems.

In problems where all timesteps of the input sequence are available, Bidirectional LSTMs train two instead of one LSTMs on the input sequence. The first on the input sequence as-is and the second on a reversed copy of the input sequence. This can provide additional context to the network and result in faster and even fuller learning on the problem.

Unidirectional LSTM only preserves information of the past because the only inputs it has seen are from the past.
Using bidirectional will run your inputs in two ways, one from past to future and one from future to past and what differs this approach from unidirectional is that in the LSTM that runs backwards you preserve information from the future and using the two hidden states combined you are able in any point in time to preserve information from both past and future.

11. Bidirectional GRU

Bidirectional GRU’s are a type of bidirectional recurrent neural networks with only the input and forget gates. It allows for the use of information from both previous time steps and later time steps to make predictions about the current state.

CHAPTER 3

RESULTS

3.1 RESULT

In this report, we proposed a number of techniques for predicting Value of Cryptocurrency. Although machine learning has been successful in predicting stock market prices through a host of different time series models, its application in predicting cryptocurrency prices has been quite restrictive. The reason behind this is obvious as prices of cryptocurrencies depend on a lot of factors like technological progress, internal competition, pressure on the markets to deliver, economic problems, security issues, political factor etc. Their high volatility leads to the great potential of high profit if intelligent inventing strategies are taken. Our experimental results indicate that some of the proposed techniques are very promising in performing their tasks.

1. GRU

Figure 3.1: GRU NN

2. Vanilla

Figure 3.2: Vanilla NN

3. Bidirectional GRU

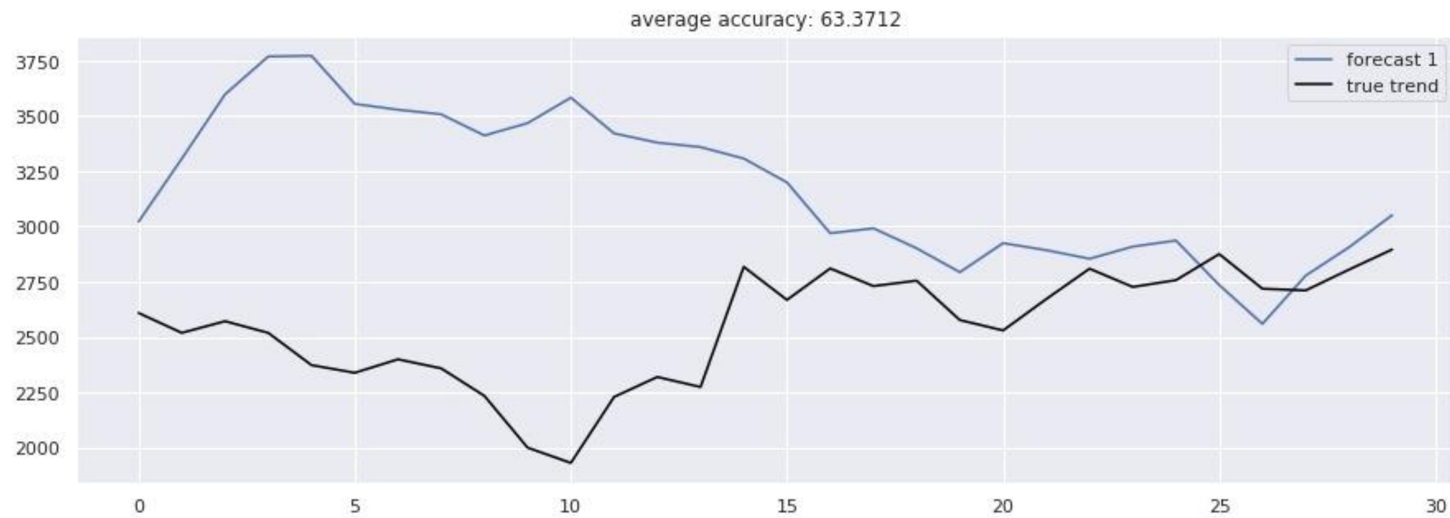


Figure 3.3: Bidirectional GRU

4. Bidirectional Vanilla

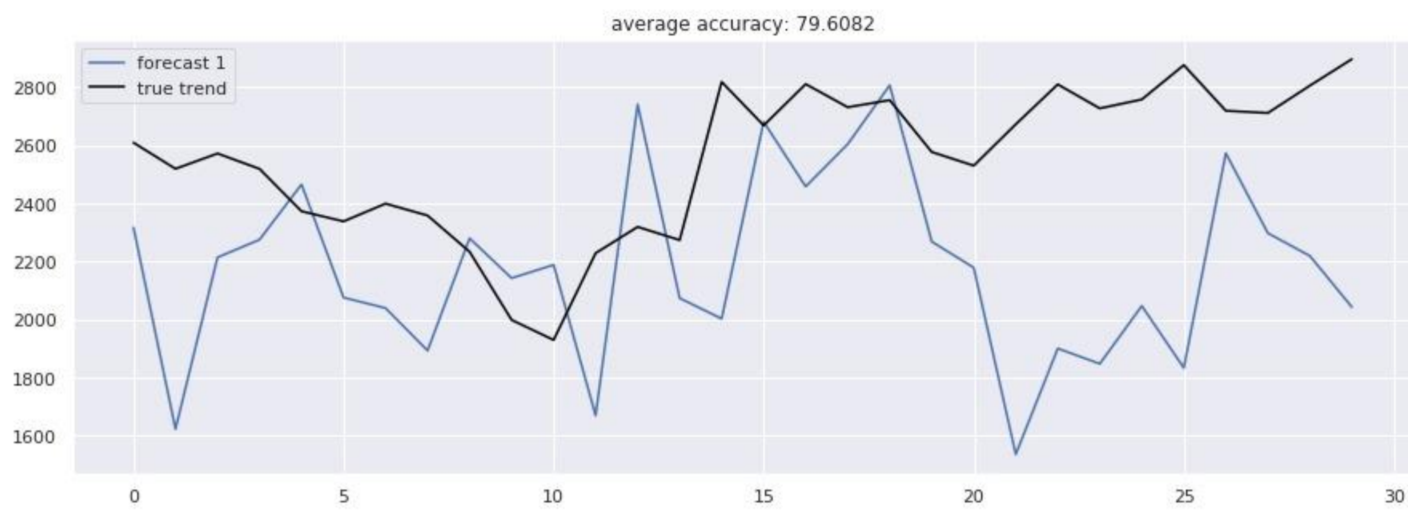


Figure 3.4: Bidirectional Vanilla

5. Bidirectional Gru Seq2seq

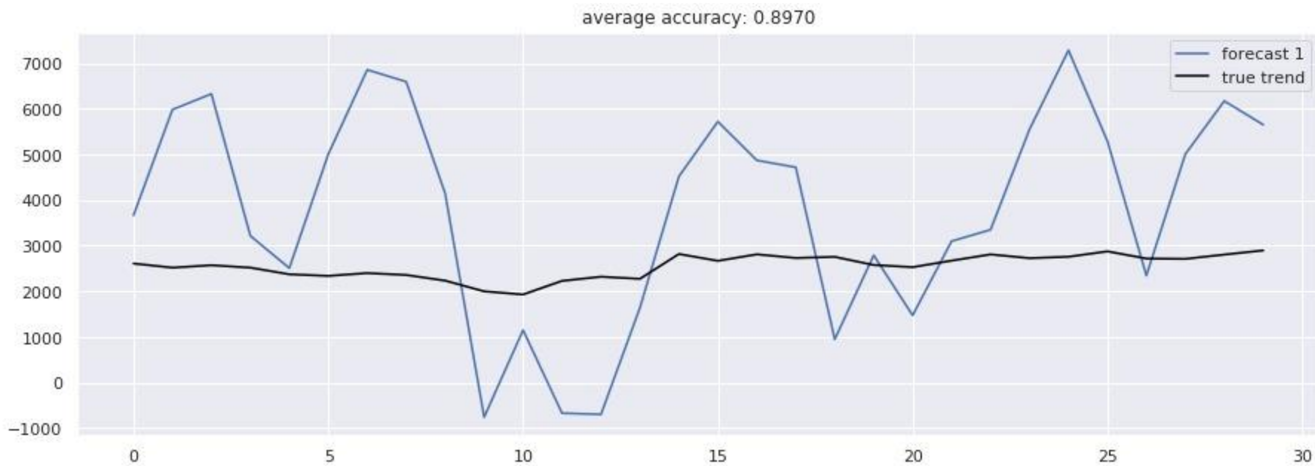


Figure 3.5: Bidirectional Gru Seq2seq

6. Gru-Seq2seq

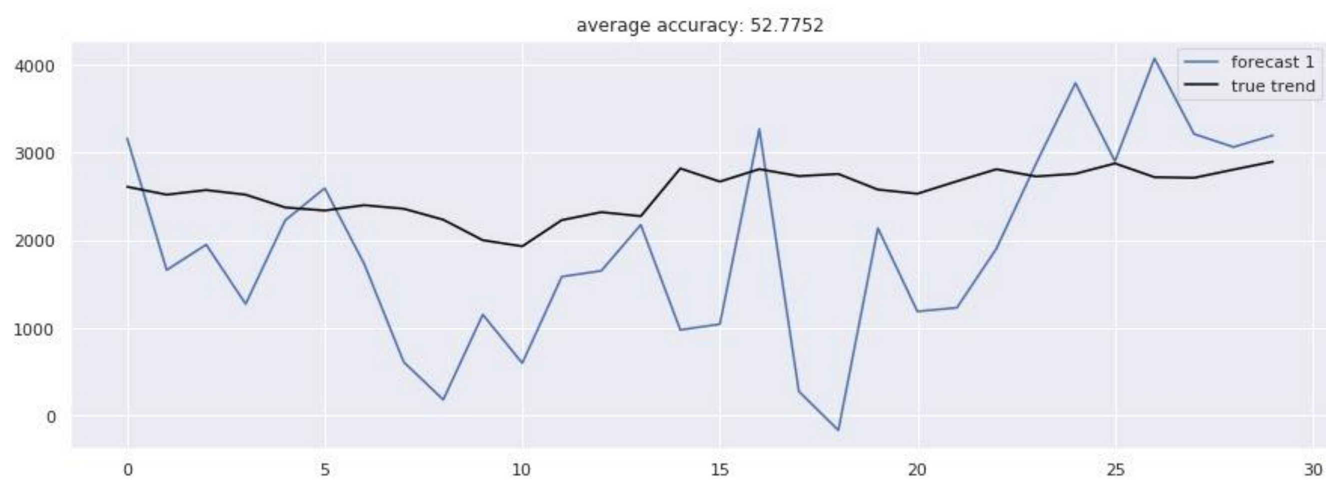


Figure 3.6: Bidirectional Gru Seq2seq

7. Lstm Seq2seq Vae

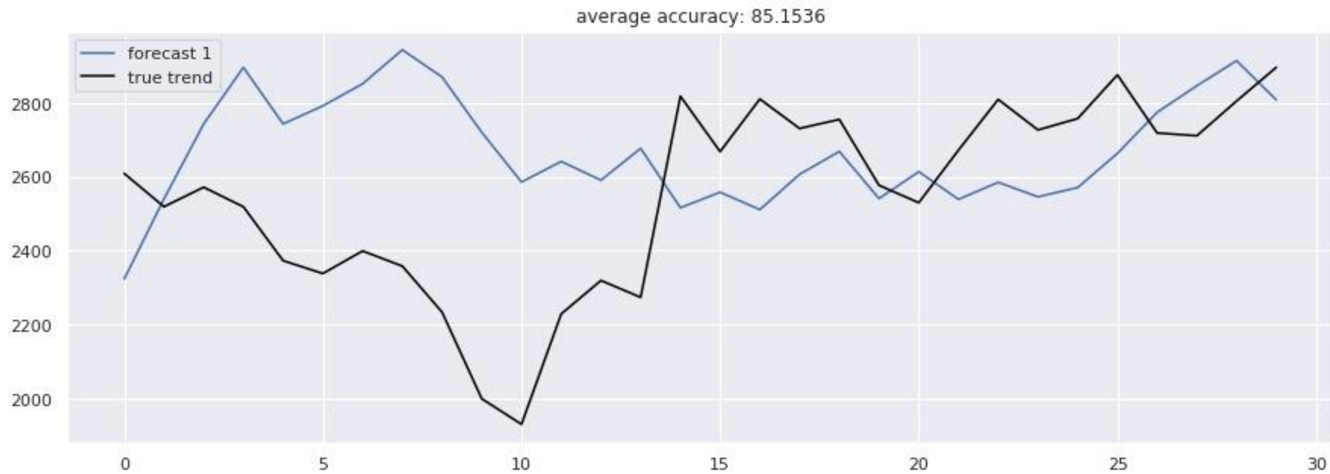


Figure 3.7: LSTM Seq2seq

8. Bidirectional LSTM Seq2seq

Figure 3.8: Bidirectional LSTM Seq2seq

APPENDICES

Appendix 1 : Preprocess

```
In [1]: import pandas as pd
        from sklearn.preprocessing import StandardScaler
        from sklearn.preprocessing import MinMaxScaler
        from collections import deque
        import numpy as np
        import random
```

```
In [2]: data = pd.read_csv("Stellar data crypto.csv")
```

```
In [3]: data = pd.DataFrame({"close":data["Close**"],
                             "volume":data["Volume"]  })
```

```
In [4]: scale= StandardScaler()                                   # scaler 1, using variance and standard variation
        scale2= MinMaxScaler()                                    # scaler 2,using min max value, range (0,1)
                                    #''' scale values of close'''
        a=[]
        for val in data["close"].values:
            a.append([val])
        a = np.array(a)
        a.reshape(-1,1)
                                           #print(scale.fit_transform(a))
        b = scale2.fit_transform(a)
        data["close"] = b
                                           #data['future'] = data['close'-].shift(-FUTURE_PERIOD_PREDICT)
                                           #data.dropna(inplace=True)
                                   #''' scale values of volume'''
        c=[]
        for val in data["volume"].values:
            c.append([val])
        c = np.array(c)
        c.reshape(-1,1)
        d = scale2.fit_transform(c)
        data["volume"] = d
```

Appendix 2: Linear Regression

```
from sklearn.linear_model import LinearRegression
from sklearn import metrics
linreg = LinearRegression()
ypred = []
for i in range(0,727):
    y=[]
    x=[]
    for j in range(i,i+3):
        y.append([data.close[j]])
        x.append([data.volume[j]])
    y = np.array(y)
    y = y.reshape(-1,1)
    x = np.array(x)
    x = x.reshape(-1,1)
    linreg.fit(x,y)
    ypred.append(linreg.predict([[data.volume[i+3]]]))
```

Appendix 3 : Naïve approach

```
ypred=[]
yactual = []
diff=[]
for i in range(0,729):
    ypred.append(data.close[i])
for i in range(1,730):
    yactual.append(data.close[i])
for i in range(0,729):
    diff.append(abs(yactual[i]-ypred[i]))
```

```
def classify(diff):
    if float(0.008) > float(diff):
        return 1
    else:
        return 0
result = []
result.append(list(map(classify, diff)))
```

```
false = 0
true = 0
for a in result:
    for b in a:
        if b == 1:
            true= true+1
        else:
            false= false+1
```

```
print false
true
```

```
253

476
```

Appendix 4 : simple average

simple average

In [28]:
```
prediction=[]
for i in range(3,729):
    value = 0
    for j in range(0,(i-1)):
        value = value + data.close[j]
    value = (value/(i-1))
    prediction.append(value)

abs_difference=[]
for i in range(3,726):
    abs_difference.append(abs(data.close[i]-prediction[i-3]))
```

In [29]:
```
abs_result = []
abs_result.append(list(map(classify, abs_difference)))

abs_false = 0
abs_true = 0
for a in abs_result:
    for b in a:
        if b == 1:
            abs_true= abs_true+1
        else:
            abs_false= abs_false+1
```

In [30]:
```
abs_true
```

Out[30]:
```
13
```

Appendix 5: Moving Average

In [12]:
```
prediction=[]
for i in range(0,726):
    value=0
    for j in range(i,i+3):
        value = value + data.close[j]
    value = value/3
    prediction.append(value)

abs_difference=[]
for i in range(3,726):
    abs_difference.append(abs(data.close[i]-prediction[i-3]))
```

In [13]:
```
abs_result = []
abs_result.append(list(map(classify, abs_difference)))

abs_false = 0
abs_true = 0
for a in abs_result:
    for b in a:
        if b == 1:
            abs_true= abs_true+1
        else:
            abs_false= abs_false+1
```

In [14]:
```
abs_true
```

Out[14]:
```
110
```

Appendix 6: Weighted Moving Average

```
In [18]: prediction=[]
         for i in range(0,726):
             value = 0.1* data.close[i] + 0.3* data.close[i+1] + 0.6* data.close[i+2]
             prediction.append(value)

         abs_difference=[]
         for i in range(3,726):
             abs_difference.append(abs(data.close[i]-prediction[i-3]))
```

```
In [19]: def classify(abs_difference):
             if float(threshold) > float(abs_difference):
                 return 1
             else:
                 return 0
         abs_result = []
         abs_result.append(list(map(classify, abs_difference)))

         abs_false = 0
         abs_true = 0
         for a in abs_result:
             for b in a:
                 if b == 1:
                     abs_true= abs_true+1
                 else:
                     abs_false= abs_false+1
```

```
In [20]: abs_true
```

```
Out[20]: 117
```

Appendix 7: Support Vector Regression

In [5]:
```
diff_rbf=[]
diff_lin=[]
diff_poly=[]
for i in range(0,727):
    diff_rbf.append(abs(data.close[i+3]-ypred_rbf[i]))
    diff_lin.append(abs(data.close[i+3]-ypred_lin[i]))
    diff_poly.append(abs(data.close[i+3]-ypred_poly[i]))
def classify(diff):
    if float(0.002) > float(diff):
        return 1
    else:
        return 0
result_linear = []
result_rbf = []
result_poly = []
result_linear.append(list(map(classify, diff_lin)))
result_rbf.append(list(map(classify, diff_rbf)))
result_poly.append(list(map(classify, diff_poly)))
```

In [6]:
```
false_rbf = 0
true_rbf = 0
for a in result_rbf:
    for b in a:
        if b == 1:
            true_rbf= true_rbf+1
        else:
            false_rbf= false_rbf+1
false_lin = 0
true_lin = 0
for a in result_linear:
    for b in a:
        if b == 1:
            true_lin= true_lin+1
        else:
            false_lin= false_lin+1
false_poly = 0
true_poly = 0
for a in result_poly:
    for b in a:
        if b == 1:
            true_poly= true_poly+1
        else:
            false_poly= false_poly+1
```

In [7]:
```
print true_poly
print true_lin
print true_rbf
```

```
189
189
189
```

Appendix 8: GRU

```
import warnings

if not sys.warnoptions:
    warnings.simplefilter('ignore')

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from datetime import datetime
from datetime import timedelta
from tqdm import tqdm
sns.set()
tf.compat.v1.random.set_random_seed(1234)

df = pd.read_excel('BTC data.xlsx')
df.head()
```

	Date	Open*	High	Low	Close**	Volume	Market Cap
0	2013-11-11	325.41	351.27	311.78	342.44	0	4101635027
1	2013-11-12	343.06	362.81	342.80	360.33	0	4317726291
2	2013-11-13	360.97	414.05	359.80	407.37	0	4883103453
3	2013-11-14	406.41	425.90	395.19	420.20	0	5038817795
4	2013-11-15	419.41	437.89	396.11	417.95	0	5013561020

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
```

```
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.GRUCell(size_layer)

        rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop = tf.contrib.rnn.DropoutWrapper(
            rnn_cells, output_keep_prob = forget_bias
        )
        self.hidden_layer = tf.placeholder(
            tf.float32, (None, num_layers * size_layer)
        )
        self.outputs, self.last_state = tf.nn.dynamic_rnn(
            drop, self.X, initial_state = self.hidden_layer, dtype =
tf.float32
        )
        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100

def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer

def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value = np.zeros((1, num_layers * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
```

```python
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.hidden_layer: init_value,
                },
            )
            init_value = last_state
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value = np.zeros((1, num_layers * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value = last_state

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.hidden_layer: init_value,
            },
```

```
        )
        init_value = last_state
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 8: GRU Seq2Seq

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.GRUCell(size_layer)

        rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop = tf.contrib.rnn.DropoutWrapper(
            rnn_cells, output_keep_prob = forget_bias
        )
        self.hidden_layer = tf.placeholder(
            tf.float32, (None, num_layers * size_layer)
        )
        _, last_state = tf.nn.dynamic_rnn(
            drop, self.X, initial_state = self.hidden_layer, dtype =
tf.float32
        )

        with tf.variable_scope('decoder', reuse = False):
            rnn_cells_dec = tf.nn.rnn_cell.MultiRNNCell(
                [lstm_cell(size_layer) for _ in range(num_layers)],
state_is_tuple = False
            )
            drop_dec = tf.contrib.rnn.DropoutWrapper(
                rnn_cells_dec, output_keep_prob = forget_bias
            )
            self.outputs, self.last_state = tf.nn.dynamic_rnn(
                drop_dec, self.X, initial_state = last_state, dtype =
tf.float32
            )

        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100

def anchor(signal, weight):
```

```
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
```

```
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value = np.zeros((1, num_layers * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.hidden_layer: init_value,
                },
            )
            init_value = last_state
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value = np.zeros((1, num_layers * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
```

```
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value = last_state

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 9: LSTM Seq2Seq

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.LSTMCell(size_layer, state_is_tuple = False)

        rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop = tf.contrib.rnn.DropoutWrapper(
            rnn_cells, output_keep_prob = forget_bias
        )
        self.hidden_layer = tf.placeholder(
            tf.float32, (None, num_layers * 2 * size_layer)
        )
        _, last_state = tf.nn.dynamic_rnn(
            drop, self.X, initial_state = self.hidden_layer, dtype =
tf.float32
        )

        with tf.variable_scope('decoder', reuse = False):
            rnn_cells_dec = tf.nn.rnn_cell.MultiRNNCell(
                [lstm_cell(size_layer) for _ in range(num_layers)],
state_is_tuple = False
            )
            drop_dec = tf.contrib.rnn.DropoutWrapper(
                rnn_cells_dec, output_keep_prob = forget_bias
            )
            self.outputs, self.last_state = tf.nn.dynamic_rnn(
                drop_dec, self.X, initial_state = last_state, dtype =
tf.float32
            )

        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100

def anchor(signal, weight):
```

```
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
```

```
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value = np.zeros((1, num_layers * 2 * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.hidden_layer: init_value,
                },
            )
            init_value = last_state
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value = np.zeros((1, num_layers * 2 * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
```

```
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value = last_state

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 10: LSTM Seq2Seq Vae

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
        lambda_coeff = 0.5
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.LSTMCell(size_layer, state_is_tuple = False)

        rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop = tf.contrib.rnn.DropoutWrapper(
            rnn_cells, output_keep_prob = forget_bias
        )
        self.hidden_layer = tf.placeholder(
            tf.float32, (None, num_layers * 2 * size_layer)
        )
        _, last_state = tf.nn.dynamic_rnn(
            drop, self.X, initial_state = self.hidden_layer, dtype =
tf.float32
        )

        self.z_mean = tf.layers.dense(last_state, size)
        self.z_log_sigma = tf.layers.dense(last_state, size)

        epsilon = tf.random_normal(tf.shape(self.z_log_sigma))
        self.z_vector = self.z_mean + tf.exp(self.z_log_sigma)

        with tf.variable_scope('decoder', reuse = False):
            rnn_cells_dec = tf.nn.rnn_cell.MultiRNNCell(
                [lstm_cell(size_layer) for _ in range(num_layers)],
state_is_tuple = False
            )
            drop_dec = tf.contrib.rnn.DropoutWrapper(
                rnn_cells_dec, output_keep_prob = forget_bias
            )
            x = tf.concat([tf.expand_dims(self.z_vector, axis=0), self.X],
axis = 1)
            self.outputs, self.last_state = tf.nn.dynamic_rnn(
                drop_dec, self.X, initial_state = last_state, dtype =
tf.float32
            )

        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.lambda_coeff = lambda_coeff
```

```
        self.kl_loss = -0.5 * tf.reduce_sum(1.0 + 2 * self.z_log_sigma -
self.z_mean ** 2 -
                             tf.exp(2 * self.z_log_sigma), 1)
        self.kl_loss = tf.scalar_mul(self.lambda_coeff, self.kl_loss)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits) +
self.kl_loss)
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100

def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
```

```
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value = np.zeros((1, num_layers * 2 * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_x = np.random.binomial(1, 0.5, batch_x.shape) * batch_x
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.hidden_layer: init_value,
                },
            )
            init_value = last_state
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
```

```
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value = np.zeros((1, num_layers * 2 * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value = last_state

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 11: Vanilla

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.BasicRNNCell(size_layer)

        rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop = tf.contrib.rnn.DropoutWrapper(
            rnn_cells, output_keep_prob = forget_bias
        )
        self.hidden_layer = tf.placeholder(
            tf.float32, (None, num_layers * size_layer)
        )
        self.outputs, self.last_state = tf.nn.dynamic_rnn(
            drop, self.X, initial_state = self.hidden_layer, dtype =
tf.float32
        )
        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100

def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
```

```
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value = np.zeros((1, num_layers * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.hidden_layer: init_value,
                },
            )
            init_value = last_state
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value = np.zeros((1, num_layers * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
```

```python
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value = last_state

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.hidden_layer: init_value,
            },
        )
        init_value = last_state
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 12: Bidirectional Vanilla

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.BasicRNNCell(size_layer)

        backward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        forward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop_backward = tf.contrib.rnn.DropoutWrapper(
            backward_rnn_cells, output_keep_prob = forget_bias
        )
        forward_backward = tf.contrib.rnn.DropoutWrapper(
            forward_rnn_cells, output_keep_prob = forget_bias
        )
        self.backward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * size_layer)
        )
        self.forward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * size_layer)
        )
        self.outputs, self.last_state = tf.nn.bidirectional_dynamic_rnn(
            forward_backward,
            drop_backward,
            self.X,
            initial_state_fw = self.forward_hidden_layer,
            initial_state_bw = self.backward_hidden_layer,
            dtype = tf.float32,
        )
        self.outputs = tf.concat(self.outputs, 2)
        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100
```

```python
def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value_forward = np.zeros((1, num_layers * size_layer))
        init_value_backward = np.zeros((1, num_layers * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.backward_hidden_layer: init_value_backward,
                    modelnn.forward_hidden_layer: init_value_forward,
                },
            )
            init_value_forward = last_state[0]
            init_value_backward = last_state[1]
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value_forward = np.zeros((1, num_layers * size_layer))
    init_value_backward = np.zeros((1, num_layers * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
```

```python
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value_forward = last_state[0]
    init_value_backward = last_state[1]

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 13: Bidirectional GRU

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.GRUCell(size_layer)

        backward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        forward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop_backward = tf.contrib.rnn.DropoutWrapper(
            backward_rnn_cells, output_keep_prob = forget_bias
        )
        forward_backward = tf.contrib.rnn.DropoutWrapper(
            forward_rnn_cells, output_keep_prob = forget_bias
        )
        self.backward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * size_layer)
        )
        self.forward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * size_layer)
        )
        self.outputs, self.last_state = tf.nn.bidirectional_dynamic_rnn(
            forward_backward,
            drop_backward,
            self.X,
            initial_state_fw = self.forward_hidden_layer,
            initial_state_bw = self.backward_hidden_layer,
            dtype = tf.float32,
        )
        self.outputs = tf.concat(self.outputs, 2)
        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100
```

```
def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value_forward = np.zeros((1, num_layers * size_layer))
        init_value_backward = np.zeros((1, num_layers * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.backward_hidden_layer: init_value_backward,
                    modelnn.forward_hidden_layer: init_value_forward,
                },
            )
            init_value_forward = last_state[0]
            init_value_backward = last_state[1]
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value_forward = np.zeros((1, num_layers * size_layer))
    init_value_backward = np.zeros((1, num_layers * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
```

```
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value_forward = last_state[0]
    init_value_backward = last_state[1]

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 14: Bidirectional GRU Seq2Seq

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.GRUCell(size_layer)

        backward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        forward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop_backward = tf.contrib.rnn.DropoutWrapper(
            backward_rnn_cells, output_keep_prob = forget_bias
        )
        forward_backward = tf.contrib.rnn.DropoutWrapper(
            forward_rnn_cells, output_keep_prob = forget_bias
        )
        self.backward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * size_layer)
        )
        self.forward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * size_layer)
        )
        _, last_state = tf.nn.bidirectional_dynamic_rnn(
            forward_backward,
            drop_backward,
            self.X,
            initial_state_fw = self.forward_hidden_layer,
            initial_state_bw = self.backward_hidden_layer,
            dtype = tf.float32,
        )

        with tf.variable_scope('decoder', reuse = False):
            backward_rnn_cells_decoder = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
            )
            forward_rnn_cells_decoder = tf.nn.rnn_cell.MultiRNNCell(
                [lstm_cell(size_layer) for _ in range(num_layers)],
                state_is_tuple = False,
            )
            drop_backward_decoder = tf.contrib.rnn.DropoutWrapper(
            backward_rnn_cells_decoder, output_keep_prob = forget_bias
            )
```

```
            forward_backward_decoder = tf.contrib.rnn.DropoutWrapper(
                forward_rnn_cells_decoder, output_keep_prob = forget_bias
            )
            self.outputs, self.last_state = tf.nn.bidirectional_dynamic_rnn(
                forward_backward_decoder, drop_backward_decoder, self.X,
                initial_state_fw = last_state[0],
                initial_state_bw = last_state[1],
                dtype = tf.float32
            )
        self.outputs = tf.concat(self.outputs, 2)
        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100

def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value_forward = np.zeros((1, num_layers * size_layer))
        init_value_backward = np.zeros((1, num_layers * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
                    modelnn.backward_hidden_layer: init_value_backward,
                    modelnn.forward_hidden_layer: init_value_forward,
```

```python
                },
            )
            init_value_forward = last_state[0]
            init_value_backward = last_state[1]
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value_forward = np.zeros((1, num_layers * size_layer))
    init_value_backward = np.zeros((1, num_layers * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value_forward = last_state[0]
    init_value_backward = last_state[1]

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
```

```
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

Appendix 15: Bidirectional LSTM Seq2Seq

```
class Model:
    def __init__(
        self,
        learning_rate,
        num_layers,
        size,
        size_layer,
        output_size,
        forget_bias = 0.1,
    ):
        def lstm_cell(size_layer):
            return tf.nn.rnn_cell.LSTMCell(size_layer, state_is_tuple = False)

        backward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        forward_rnn_cells = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
        )
        self.X = tf.placeholder(tf.float32, (None, None, size))
        self.Y = tf.placeholder(tf.float32, (None, output_size))
        drop_backward = tf.contrib.rnn.DropoutWrapper(
            backward_rnn_cells, output_keep_prob = forget_bias
        )
        forward_backward = tf.contrib.rnn.DropoutWrapper(
            forward_rnn_cells, output_keep_prob = forget_bias
        )
        self.backward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * 2 * size_layer)
        )
        self.forward_hidden_layer = tf.placeholder(
            tf.float32, shape = (None, num_layers * 2 * size_layer)
        )
        _, last_state = tf.nn.bidirectional_dynamic_rnn(
            forward_backward,
            drop_backward,
            self.X,
            initial_state_fw = self.forward_hidden_layer,
            initial_state_bw = self.backward_hidden_layer,
            dtype = tf.float32,
        )

        with tf.variable_scope('decoder', reuse = False):
            backward_rnn_cells_decoder = tf.nn.rnn_cell.MultiRNNCell(
            [lstm_cell(size_layer) for _ in range(num_layers)],
            state_is_tuple = False,
            )
            forward_rnn_cells_decoder = tf.nn.rnn_cell.MultiRNNCell(
                [lstm_cell(size_layer) for _ in range(num_layers)],
                state_is_tuple = False,
            )
            drop_backward_decoder = tf.contrib.rnn.DropoutWrapper(
            backward_rnn_cells_decoder, output_keep_prob = forget_bias
            )
```

```
            forward_backward_decoder = tf.contrib.rnn.DropoutWrapper(
                forward_rnn_cells_decoder, output_keep_prob = forget_bias
            )
            self.outputs, self.last_state = tf.nn.bidirectional_dynamic_rnn(
                forward_backward_decoder, drop_backward_decoder, self.X,
                initial_state_fw = last_state[0],
                initial_state_bw = last_state[1],
                dtype = tf.float32
            )
        self.outputs = tf.concat(self.outputs, 2)
        self.logits = tf.layers.dense(self.outputs[-1], output_size)
        self.cost = tf.reduce_mean(tf.square(self.Y - self.logits))
        self.optimizer = tf.train.AdamOptimizer(learning_rate).minimize(
            self.cost
        )

def calculate_accuracy(real, predict):
    real = np.array(real) + 1
    predict = np.array(predict) + 1
    percentage = 1 - np.sqrt(np.mean(np.square((real - predict) / real)))
    return percentage * 100

def anchor(signal, weight):
    buffer = []
    last = signal[0]
    for i in signal:
        smoothed_val = last * weight + (1 - weight) * i
        buffer.append(smoothed_val)
        last = smoothed_val
    return buffer
```

```
def forecast():
    tf.reset_default_graph()
    modelnn = Model(
        learning_rate, num_layers, df_log.shape[1], size_layer,
df_log.shape[1], dropout_rate
    )
    sess = tf.InteractiveSession()
    sess.run(tf.global_variables_initializer())
    date_ori = pd.to_datetime(df.iloc[:, 0]).tolist()

    pbar = tqdm(range(epoch), desc = 'train loop')
    for i in pbar:
        init_value_forward = np.zeros((1, num_layers * 2 * size_layer))
        init_value_backward = np.zeros((1, num_layers * 2 * size_layer))
        total_loss, total_acc = [], []
        for k in range(0, df_train.shape[0] - 1, timestamp):
            index = min(k + timestamp, df_train.shape[0] - 1)
            batch_x = np.expand_dims(
                df_train.iloc[k : index, :].values, axis = 0
            )
            batch_y = df_train.iloc[k + 1 : index + 1, :].values
            logits, last_state, _, loss = sess.run(
                [modelnn.logits, modelnn.last_state, modelnn.optimizer,
modelnn.cost],
                feed_dict = {
                    modelnn.X: batch_x,
                    modelnn.Y: batch_y,
```

```
                    modelnn.backward_hidden_layer: init_value_backward,
                    modelnn.forward_hidden_layer: init_value_forward,
                },
            )
            init_value_forward = last_state[0]
            init_value_backward = last_state[1]
            total_loss.append(loss)
            total_acc.append(calculate_accuracy(batch_y[:, 0], logits[:, 0]))
        pbar.set_postfix(cost = np.mean(total_loss), acc = np.mean(total_acc))

    future_day = test_size

    output_predict = np.zeros((df_train.shape[0] + future_day,
df_train.shape[1]))
    output_predict[0] = df_train.iloc[0]
    upper_b = (df_train.shape[0] // timestamp) * timestamp
    init_value_forward = np.zeros((1, num_layers * 2 * size_layer))
    init_value_backward = np.zeros((1, num_layers * 2 * size_layer))

    for k in range(0, (df_train.shape[0] // timestamp) * timestamp,
timestamp):
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(
                    df_train.iloc[k : k + timestamp], axis = 0
                ),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[k + 1 : k + timestamp + 1] = out_logits

    if upper_b != df_train.shape[0]:
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(df_train.iloc[upper_b:], axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
            },
        )
        output_predict[upper_b + 1 : df_train.shape[0] + 1] = out_logits
        future_day -= 1
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    init_value_forward = last_state[0]
    init_value_backward = last_state[1]

    for i in range(future_day):
        o = output_predict[-future_day - timestamp + i:-future_day + i]
        out_logits, last_state = sess.run(
            [modelnn.logits, modelnn.last_state],
            feed_dict = {
                modelnn.X: np.expand_dims(o, axis = 0),
                modelnn.backward_hidden_layer: init_value_backward,
                modelnn.forward_hidden_layer: init_value_forward,
```

```
            },
        )
        init_value_forward = last_state[0]
        init_value_backward = last_state[1]
        output_predict[-future_day + i] = out_logits[-1]
        date_ori.append(date_ori[-1] + timedelta(days = 1))

    output_predict = minmax.inverse_transform(output_predict)
    deep_future = anchor(output_predict[:, 0], 0.3)

    return deep_future[-test_size:]
```

REFERENCES

1. https://machinelearningmastery.com/time-series-forecasting-methods-in-python-cheat-sheet/

2. https://www.analyticsvidhya.com/blog/2018/10/predicting-stock-price-machine-learningnd-deep-learning-techniques-python/

3. https://iknowfirst.com/

4. https://towardsdatascience.com/machine-learning-techniques-applied-to-stock-price-prediction-6c1994da8001

5. https://pdfs.semanticscholar.org/1594/bfd70a0ef7a02725ee8ae22f09a698164a3d.pdf