ECE447 - NEUROFUZZY COMPUTING

University of Thessaly

Department of Electrical and Computer Engineering

Coding Project

Students :
Anastasios Koutsogiannopoulos (ID: 03321)
Fani Banou (ID: 03322)
Ioannis Sakellariou (ID: 03144)

Date: February 14, 2025

Contents

1 Introduction

The prediction of Bitcoin with accuracy is a difficult and demanding process, as it can be influenced by many factors, whether they are related to statistics, economic conditions, technological developments, or general changes that can occur in the world at any given instance. In this project, we will attempt to predict the price of Bitcoin for a specific time period, using neural networks.

In the context of this study, we will try to demonstrate the progression of our thought process through the initial assumptions and our final observations as we analyze the behavior of each model we implement. First, a detailed data analysis of the provided data set will be performed. We will discuss the existing features in the dataset and add new ones as needed based on the requirements that arise. Additionally, we will show how we properly process the data before feeding them into each neural network.

Next, we will present the two main prediction techniques we used in all our implementations, we decided to use to predict the stock price.

In addition, we will dedicate a section to discussing general admissions and highlighting how we will approach all our implementations to ensure consistency in the results we obtain.

As the next step, we have the main focus of our research, which is the presentation of the implementations we used to solve this problem : **LSTM using Closing Price**, **LSTM using Time-based Features and Seasonal Statistics**, **Time-based Features and Moving Averages**, **LSTM using Differences of Close values** and ARIMA model. In this section, for each implementation we will discuss about the setup, the parameters used and the results obtained.

In conclusion, we will summarize the results and solutions discussed and suggest the most effective one, based on our findings and analysis.

2 Data Analysis

2.1 Dataset Description

The dataset is sourced from the provided link in the assignment description. It covers the period from 01-01-2021 to 01-03-2022, with data recorded at one-minute intervals. The dataset consists of 610,782 rows and 9 columns, which represent the dataset's features.

2.2 Feature Description

The key features in the dataset are categorized into four groups: *Time-related Information, Market Price Data, Trading Volume Data and Trading Pair Symbol*. The *time-related information* includes **unix**, a numerical timestamp that represents the exact moment when the data was recorded, and **date**, a standard date-time format that translates the Unix timestamp into a more readable format.

The *market price data* consists of **open**, which is the initial price of Bitcoin at the start of a given minute. **High** and **low**, representing the maximum and minimum prices reached within that specific minute and **close**, the final recorded price of Bitcoin at the end of the minute.

Additionally, the dataset contains *trading volume data*, including **Volume BTC**, which indicates the quantity of Bitcoin traded in a given minute, and **Volume USD**, representing the corresponding trade volume measured in US dollars. Lastly, the *trading pair symbol*, represented by the **symbol** column, specifies the currency pair being analyzed, which in this case is BTC/USD.

2.3 Data Preprocessing

Now, at this point, we will focus on preparing the dataset before proceeding with the dataset split.

Initially, we take the dataset and convert the 'date' column into a datetime format. Then, we set this 'date' column as the index of the DataFrame. After that, we reverse the order of the data, ensuring that the records are arranged from the oldest to the recent.

Next, we perform the necessary checks for missing values and empty entries in the dataset. It is observed that the provided dataset does not contain any missing values, so no further action is required.

Depending on how we choose to structure the dataset, we apply the appropriate resampling based on the sample frequency. We use features that are mainly based on hourly metrics, so we resample the dataset to an hourly frequency.

The dataset is ready in terms of preprocessing. The next step is to decide on the final features to be used for each model and to perform the dataset split.

2.4 Feature Engineering

Feature engineering is a crucial step in the machine learning pipeline, where we create new features or modify existing ones to improve the performance of our model. In this phase, we focus on identifying and extracting the most relevant information from the raw data to enhance the precision of our model. This process can involve techniques such as creating new variables and applying transformations to existing features.

In this section, we will review the features that were used in the models we experimented with. We will describe what each feature represents, how it is calculated, and why it was chosen for use.

Close

The selection of the **closing price** as a feature was made because it is the target value for prediction and is already available in the dataset.

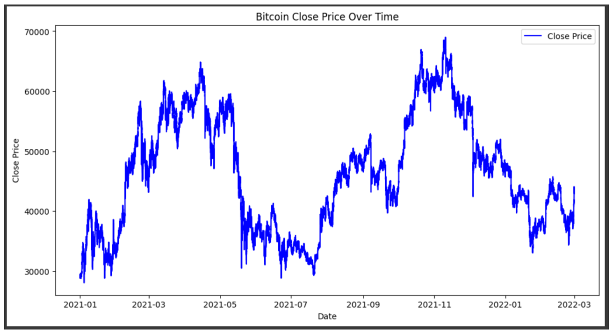

Figure 1: Close values for given Dataset

Diff-Close

This feature is derived by calculating the **differences** between consecutive closing prices. By subtracting the close values from each other, we obtain the price differences. The differences show how the price changes from one period to the next.

Using these differences helps the model recognize patterns, which can improve the prediction of the closing price. That's why it's worth using it as a potential feature. We use it in the deep learning model we implemented, as well as in the ARIMA model. If we want to visualize, the result of subtracting the close values we can see the plot below:

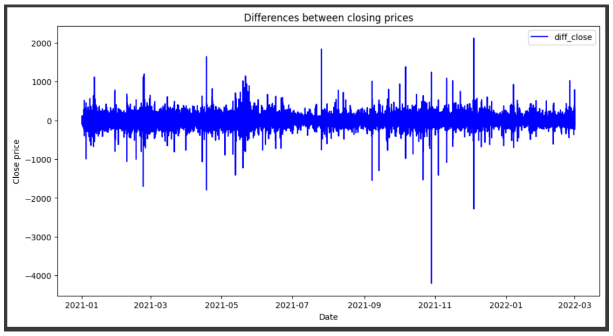

Figure 2: diff_close visulization for given Dataset

Moving Average

The next feature we introduce in our list is the **Simple Moving Average**, which is the average of the previous n values (in our case, the values will be the closing prices). Specifically, if we set $n = 5$, then the average will be calculated from the last 5 closing prices as shown :

$$\text{SMA} = \frac{close_1 + close_2 + close_3 + close_4 + close_5}{5}$$

Moving averages help identify the trend of a stock by showing whether its price is going up or down. The longer the period used for the average, the slower it reacts to price changes. For example, with long-term averages (like 200-records) it changes slower than short-term ones (like 20-records). In our models, we use a combination of the Simple Moving Average with window sizes of 24 and 240 samples in hourly analysis, and 24 and 5 samples in daily analysis.

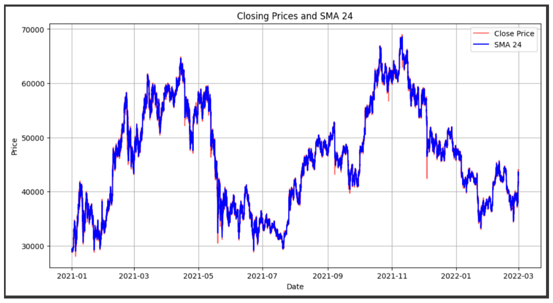

Figure 3: Window size : 24

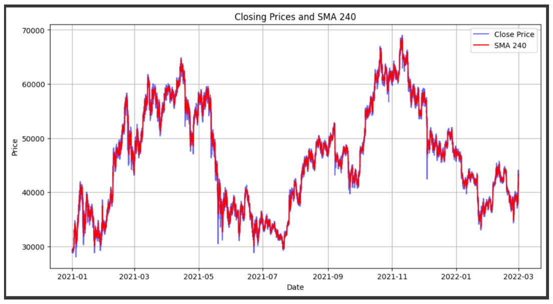

Figure 4: Window size : 240

Shorter moving averages are useful for short-term trading, while longer ones are better for long-term investing. If the moving average is going up, the stock is likely increasing in price, and if it's going down, the stock is likely decreasing. When a short-term moving average crosses above a long-term one, it can signal a price increase, while crossing below may suggest a price drop.

Time-based features

For the analysis of time-based features, we divided the data into different time scales: **by hour, by day, by day of the week, and by month** to capture the seasonal and cyclical trends that influence the closing price. Since time-related variables, such as hour and month, have a cyclical nature, they cannot be treated as simple numerical values. To preserve periodicity, we transformed these time variables using **sine and cosine transformations**. We extracted these features to calculate seasonal statistics and analyze how the closing price changes over different time scales, identifying recurring patterns that can improve the performance of the model.

Seasonal Statistics

Seasonal statistics identify repeating trends in time series data over specific periods, such as **hourly, daily and monthly cycles**. In Bitcoin trading, seasonality is influenced by factors like trading activity, investor behavior, and economic events. Recognizing these patterns helps improve price predictions by capturing predictable variations in price movements.

We use seasonal statistics to forecast the closing price because it helps detect recurring market behaviors, minimizes random variations, and enhances model accuracy. In particular, **seasonal statistics for hours, weekdays, and months are based on the average closing price**, providing insights into how Bitcoin's value changes over different time frames. For example, past data may indicate that Bitcoin tends to have higher average closing prices on Mondays compared to weekends, or that prices are typically more volatile during certain hours of the day when financial markets are open.

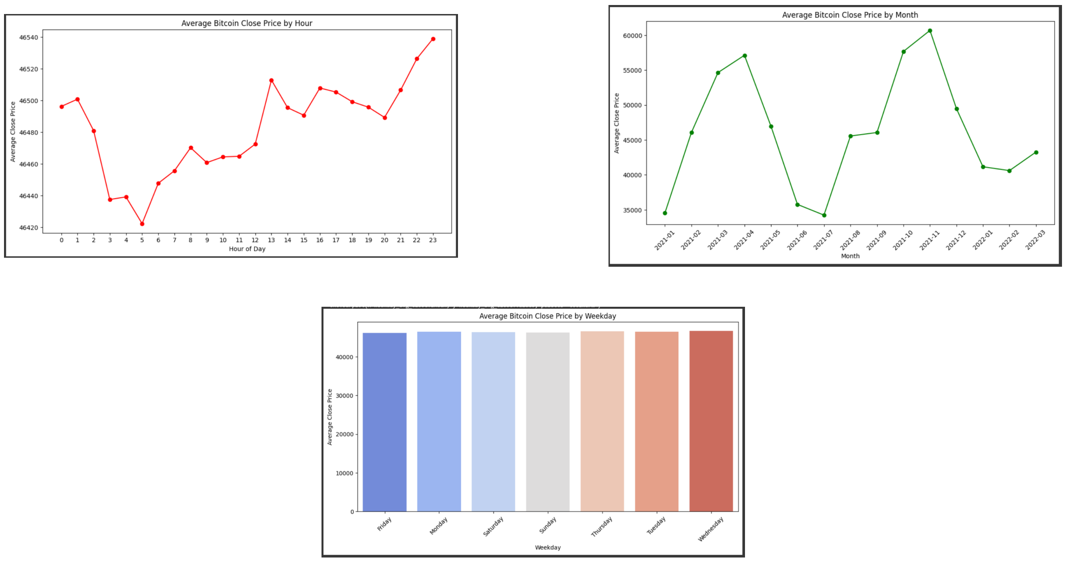

Figure 5: Seasonal statistics for hour, weekday and month

2.5 Dataset Split

Dataset splitting is a fundamental process in machine learning that involves dividing a dataset into distinct subsets to train and evaluate models effectively. The data is divided into two sets: the **training set** and the **test set**. The training set consists of data from **January 1st, 2021 (00:00:00)** to **February 18th, 2022 (23:59:59)**, and it is used to train the model on historical Bitcoin price data. The test set includes data from February 19th, 2022 (00:00:00) to February 28th, 2022 (23:59:59) and is used exclusively for comparing the model's predictions at the end of the process.

It is worth mentioning, during both the training and prediction phases, only the data prior to February 19th, 2022, is used, ensuring that the model doesn't have any knowledge of future data. This separation allows for a more **realistic** evaluation of the model's ability to predict unseen data, simulating a real-world scenario where predictions are made without access to future information. Also, during the training of any model, we set a validation set by selecting a percentage of around $15-25\%$ from the training set. Therefore, we use about 80% of the training set for training and the remaining 20% for validation, while the test set remains untouched.

2.6 Extracting X_train and y_train

We generally use two methods to extract the **X_train** and **y_train** samples, which will be used for training the models.

Predict one Y per-time

In this method, we use **sliding windows** for the X_train values. We initially define an input-window-size and then extract X values. The **input-window-size** corresponds to the length of the data that will be fed into the model each time. At the beginning, we take the first input-window-size values from the train set (and assign to X_train) and assign the (input-window-size $+ 1$)-th value to y_train. After this, when we want to create the new X values, we move the window one position to the right and also shift y one position to the right.

Let's assume that we have a vector with values $x_1, x_2, \cdots, x_8$ and set the input-window-size equal to **3**.

$$X\_train = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_2 & x_3 & x_4 \\ x_3 & x_4 & x_5 \\ x_4 & x_5 & x_6 \\ x_5 & x_6 & x_7 \end{bmatrix} \quad y\_train = \begin{bmatrix} x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \end{bmatrix}$$

Essentially, with this approach, we train the neural network to predict the next value after a given window of past values.

$$x_1 \quad x_2 \quad x_3 \quad \xrightarrow{\text{predict}} \quad [x_4]$$

Predict All In One

Regarding the second method, we also use **sliding windows** to generate X_train. The difference lies in the processing of **y_train**. In this approach, we set the input-window-size again and assign the **first input-window-size values** to X_train (as we did previously). Now, to assign y, we introduce a new variable, the **output-window-size**. The output-window-size determines how many values we expect the model to predict.

Each time, instead of predicting a single value of y, a sequence of y values will be predicted, with a length equal to output-window-size.

So, to create the first assignment to X and y, we take the first input-window-size values from the train set and assign them to X. Then, we take the next output-window-size values and assign them to y. After this, we move the input window by one position and continue this process for the entire train set.

For example if we have a vector with values $\underline{x_1, x_2, \cdots, x_9}$, a <u>input-window-size equal to 3</u> and a <u>output-window-size equal to 2</u>, we have,

$$X\_train = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_2 & x_3 & x_4 \\ x_3 & x_4 & x_5 \\ x_4 & x_5 & x_6 \\ x_5 & x_6 & x_7 \end{bmatrix} \quad y\_train = \begin{bmatrix} x_4 & x_5 \\ x_5 & x_6 \\ x_6 & x_7 \\ x_7 & x_8 \\ x_8 & x_9 \end{bmatrix}$$

Essentially, based on the same input, we predict multiple y values.

$$x_1 \quad x_2 \quad x_3 \quad \xrightarrow{\text{predict}} \quad [x_4 \quad x_5]$$

As we present the setups we created, we will mention each time which of the two methods we are using.

In the next section, we will demonstrate **how**, based on these two methods of extracting X_train and y_train, we make our predictions.

3 Predictions

Depending on how we choose to extract X_train and y_train from train-set and, by extension, how we train our model, we must follow a corresponding approach for making predictions.

3.1 Recursive Approach

If we use the **first method** of X and y extraction, during the prediction process, we provide the neural network with the **last** window from X_train as input to predict a value $y_{1,predicted}$ (it corresponds to $y_{1,actual}$). Once this prediction is made, we shift the window one step forward and repeat the process. To visualize the problem with an example we have:

$$\begin{aligned} [x_{n-2} \quad x_{n-1} \quad x_n] &\xrightarrow{\text{predict}} y_{1,predicted} \\ [x_{n-1} \quad x_n \quad y_{1,predicted}] &\xrightarrow{\text{predict}} y_{2,predicted} \\ [x_n \quad y_{2,predicted} \quad y_{3,predicted}] &\xrightarrow{\text{predict}} y_{4,predicted} \end{aligned}$$

We recursively perform this process for as many steps as we want to predict. In our case, we continue for the entire duration of the test set. This forecast method we call it **”Recursive Approach”**

3.2 All-in-One Approach

As for the **second method** what we need to do, is take the last **[input-window-size]** values from the training data and pass them as input to the model. As we mentioned in section 2.5, we have a variable, output-window-size, which essentially represents how many values we want the model to predict. Regarding our models, the output-window-size is always fixed. After splitting the dataset into train and test sets, we set the **output-window-size to be equal to the length of the test set**. We do this so that our model, based on its training, can predict a sequence of y values (and not one single value as the previous method). In our case, the goal is to predict the last ten days of February. Therefore, the output-window-size will be the same as the number of data points that represent this ten-day period.

For example,if we set a input-window-size equal to 5 we use the red-boxed values to calculate the predicted values:

$$\overset{\text{Training Set}}{\begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_{n-5} & \boxed{x_{n-4}} & \boxed{x_{n-3}} & \boxed{x_{n-2}} & \boxed{x_{n-1}} & \boxed{x_n} \end{bmatrix}} \xrightarrow{\text{predict}} y_{\text{predicted\_set}}$$

,where $\textbf{len}(y_{\textbf{predicted\_set}}) = \textbf{len(test\_data)} = \textbf{output-window-size}$.

4 Parameters and Other Configuration

In this section, we will discuss some key parameters of the models, the approaches we will take to solve the problem, and general observations that apply across all our implementations. These choices help ensure consistency and effectiveness in our results

Data Shuffling

Since Bitcoin price prediction is a problem that falls under **time-series analysis**, it is crucial to maintain the chronological order of the data. This is because time-series data has *temporal dependencies*, meaning that the sequence in which the data points are recorded carries valuable information. By default, many machine learning frameworks shuffle the data before training to improve generalization. However, in time-series problems, shuffling can disrupt the inherent temporal relationships between samples, potentially degrading model performance. To preserve the sequential structure and take advantage of the patterns hidden in the temporal dependencies, we ensure that during training, we set the parameter **shuffle=False**. This guarantees that the data is fed into the model in the exact order it was recorded, allowing the model to learn meaningful trends and patterns over time.

Early-Stopping

We use Early Stopping in all the models implemented in this project. It is a useful technique to avoid overfitting during training and improve the model's performance. We consider it a valuable addition to the training process, as it not only saves computational resources and execution time, but also improves the model's ability to generalize, leading to more reliable predictions on unseen data.

We generally use around 100 iterations in our implementations, with a **patience** value ranging from 15 to 20.

Model evaluation

We choose to run the training and evaluation process iteratively for **10 times** in each solution we propose. At the end, we present the results, which are based on averages, either in graphical representations or through various metrics we use. In this way, we ensure more reliable and consistent results, reducing the impact of random fluctuations during training.

Metrics

In this study, we evaluate the performance of our model using three key error metrics: **Mean Absolute Error** (MAE), **Mean Squared Error** (MSE), and **Root Mean Squared Error** (RMSE).

MAE measures the average absolute difference between predicted and actual values, providing an intuitive interpretation of the error magnitude.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

MSE, on the other hand, gives greater weight to larger errors by squaring the differences, making it sensitive to significant deviations.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

RMSE, as the square root of MSE, retains the same unit as the target variable and gives a more understandable measure of error magnitude.

$$\text{RMSE} = \sqrt{\text{MSE}}$$

By using all three metrics, we gain a comprehensive understanding of the model's accuracy and error distribution.

During the training of the model, the metrics **loss** and **validation loss** are used. The loss refers to the model's error on the training set, while the validation loss corresponds to the error on the validation set, which we define beforehand. We consider it appropriate to store these metrics after each training-prediction process so that, in the end, along with the actual vs predicted graph, we can also plot the loss and validation loss graphs. We should note that, since we run each implementation 10 times and use **EarlyStopping**, each model finishes its training at different iterations. We will plot the mean losses and mean validation losses for each epoch, where the number of epochs is equal to the minimum number of iterations across all trainings.

Finally, it should be emphasized that we present the following plots (**loss-val_loss**) to confirm that the model does not suffer from overfitting in the training set during training. In all the graphs we present, we observe that both metrics gradually decrease, so we conclude that the model is learning correctly.

Scaling

In every neural network we use, we perform scaling on the data to train the model. It is worth noting that we **first** split the data into a train set and a test set and **then** apply scaling. This is important because if we scale the data before splitting, there is a risk of **data leakage**, which could lead to an overestimation of the model's performance. In this way, by using the **MinMaxScaler()** (before splitting), we indirectly provide the model with information about the statistics of the test set (min and max), which affects the training process. Specifically, we use the following normalization:

```
scaler = MinMaxScaler(feature_range=(0, 1))
train_scaled = scaler.fit_transform(train_data)
test_scaled = scaler.transform(test_data)
```

By using **fit_transform**, we perform normalization based on the minimum and maximum values of the train set. Then, using the same scaler, we normalize the test data. The key point here is that we normalize the training data while keeping the test data separate, without affecting the normalization of the train set.

Specifications

We ran our codes on the **Google Colab** platform using the **NVIDIA TESLA (T4) GPU**, which offers us high computational power and efficient performance for deep learning tasks.

Training and Model Parameters

In each implementation we present, we will include a table listing all the hyperparameters of the model used. It is clear, that these parameters vary across different implementations and are not identical throughout. The chosen parameters were determined experimentally after numerous simulations. The final values we provide represent the **best-performing solutions** when compared to each other.

Resampling

The given dataset contains records per minute. Due to the large amount of data (limited resources) and for convenience, we primarily perform resampling per hour. That is, we collect all 60 samples for a specific hour and determine the one that will represent that hour using the average.

Additionally, in the implementation we considered the best (section 7 - 3 versions), we found it important to observe the model's behavior when making predictions based on daily features. Therefore, we also performed resampling per day and compared the results with each other.

5 LSTM using Closing Price

In this section, we will present our first implementation. We use only the close price as the single input feature in the model. We implemented it using both methods (which we previously mentioned regarding X_train and train extraction and prediction), specifically the **recursive** and the **All-in-One** method.

Setup and Parameters

It is important to note that the final parameters we applied were determined through experimentation, following numerous trials, and the best ones were chosen. We use the same setup in both approaches. The following table presents all the key hyperparameters used in our experiment:

Hyperparameter	**Value**
Input Features	Close Price (1 feature)
Input Window Size	60
LSTM Layers	(128, 64, 32) units
Dropout Rate	0.2
Dense Layers	(32, 1) units
Activation Function	ReLU (Hidden), Linear (Output)
Optimizer	Adam ($lr = 0.001$)
Loss Function	Mean Squared Error (MSE)
Evaluation Metric	Mean Absolute Error (MAE)
Batch Size	256
Epochs	50
Validation Split	20%
Early Stopping	Patience = 15

Table 1: Summary of Model Hyperparameters

This model is composed of **three LSTM layers** with 128, 64, and 32 units, respectively, designed to capture and process the sequential patterns in the data. To reduce the risk of overfitting, each LSTM layer is followed by a **Dropout layer** with a dropout rate of 0.2. After these layers, a **Dense layer** with 32 units and a ReLU activation function is used to learn complex patterns in the data. The model concludes with an **output layer**, which can either produce a single prediction for regression approach or multiple outputs if using an all-in-one approach, where the number of neurons corresponds to the length of the test set.

5.1 Recursive Approach

Experimental Results

The figures present the graphical representations of the average results obtained after running the experiment ten times. Regarding the validation loss, we report the average of the minimum loss values observed across these ten runs.

Due to the use of early stopping, the number of epochs varies across the ten repetitions, leading to differences in training duration for each run.

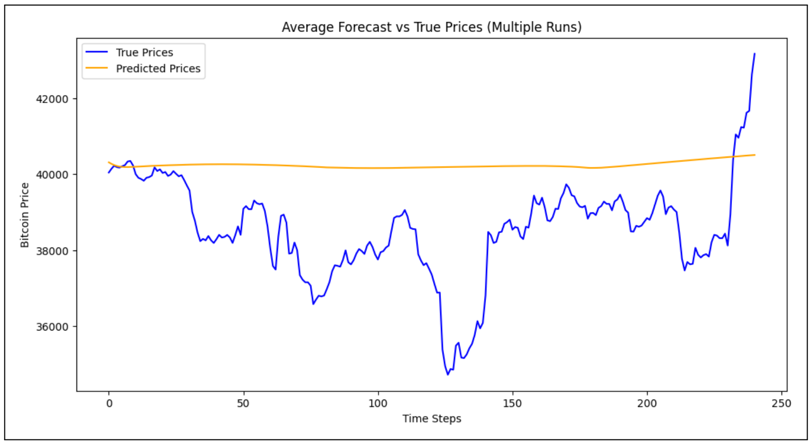

Figure 6: Predicted vs Actual values (Method=Recursive, Model=LSTM, Features=Close)

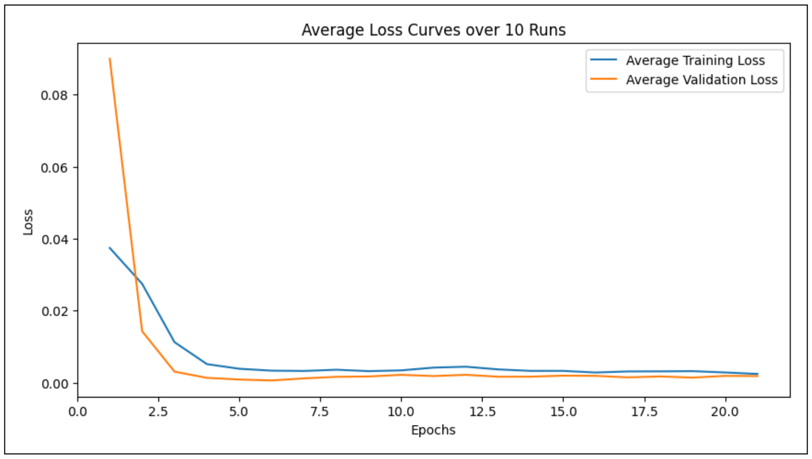

Figure 7: Validation Loss and Training Loss (Method=Recursive, Model=LSTM, Features=Close)

Metric	Average Value (10 Runs)
Mean Absolute Error (MAE)	3066.9017
Mean Squared Error (MSE)	14849968.7572
Root Mean Squared Error (RMSE)	3555.1553

Table 2: Average performance metrics over 10 experimental runs (Method=Recursive, Model=LSTM, Features=Close)

Metric	Time (seconds)
Model Training Time	29.8469
Predicting Time	17.5626

Table 3: Average Execution Times over 10 Runs

5.2 All-in-One Approach

Experimental Results

Similarly, we present the (average) predicted and actual Bitcoin prices for the last ten days. Following that, we have the average values of the loss and validation loss, showing how they evolve during the training, with the final metrics provided at the end.

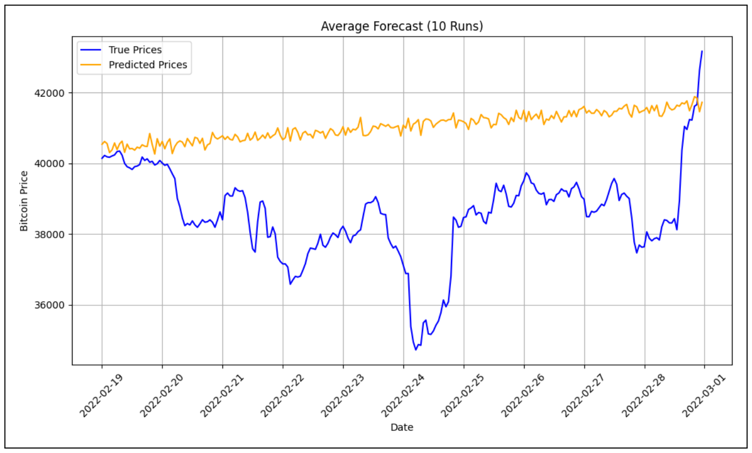

Figure 8: Predicted vs Actual values (Method=All-in-One, Model=LSTM, Features=Close)

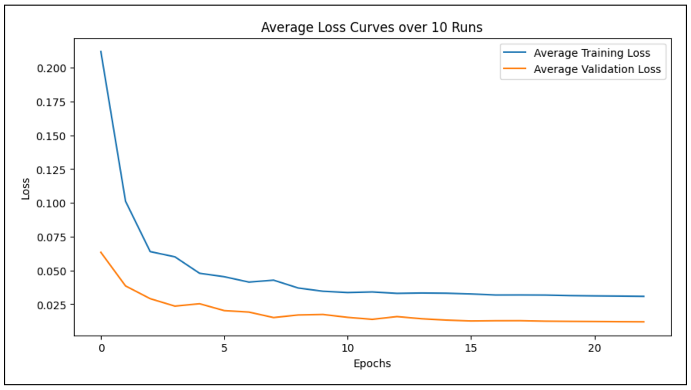

Figure 9: Validation Loss and Training Loss (Method=All-in-One, Model=LSTM, Features=Close)

Metric	**Average Value (10 Runs)**
Mean Absolute Error (MAE)	2577.6335
Mean Squared Error (MSE)	10336230.8259
Root Mean Squared Error (RMSE)	3215.00

Table 4: Average performance metrics over 10 experimental runs (Method=All-in-One, Model=LSTM, Features=Close)

Metric	**Time (seconds)**
Model Training Time	63.3288
Predicting Time	0.4144

Table 5: Average Execution Times over 10 Runs (Method=All-in-One, Model=LSTM, Features=Close)

5.3 Observations

After the results of the two approaches, we conclude that both provide relatively good results. Using the recursive approach, we obtain a MAE of approximately 3066, while with the All-in-One approach, it is equal to 2577. This deviation can be considered acceptable.

In the recursive method, we also observe that the curve is smoother compared to the second one. This may be due to the fact that the first method directly depends on the previous values, which are also the predicted ones.

It is worth noting that, in both cases, the curves we obtained do not exhibit the curvature and sharp fluctuations seen in the real data. This raises questions and concerns, as it may indicate the need to apply additional methods so that the model can better capture potential temporal dependencies.

6 LSTM using Time-based Features and Seasonal Statistics

We continue using the same type of neural network (LSTM), but this time, we modify the input features from closing values to the combination of **time-based features** and **seasonal statistics**. These features were mentioned in section 2.4 Feature Engineering, where time-based features are defined as features exclusively related to time and seasonal statistics as information for the average closing value per each hour, day or month.

Based on the results from the previous section, we mentioned that the model might need adjustments in order to highlight any temporal patterns and display fluctuations in the graph. For this reason, we propose the solution with time-based features and seasonal stats.

Time-based features, such as the **hours** of the day, **days** of the week, and **months**, can help the model recognize temporal dependencies that affect the price. For example, there may be hourly or weekly trends in Bitcoin prices that are not visible from the closing prices alone. Similarly, with seasonal stats, by using the **averages**, the model can better understand temporal patterns and trends, while simultaneously reducing noise.

Before presenting the parameters and results of our implementation, we will briefly explain how we create the dataset with these features and how we make the predictions, as it differs from the previous approaches.

6.1 Feature extraction

Let's start with the time-based features. From this category, we have a total of 7 features: **hour_sin**, **hour_cos**, **weekday_cos**, **weekday_sin**, **month_sin**, **month_cos** and **is_weekend**. Each feature indicates the hour, day of the week, or month we are in, or whether it is a weekend. Each of these factors may influence the closing value. Therefore, every closing value is associated with a set of these characteristics. For example, when we have a closing value, we use the time-based features to determine the hour of the day, the day of the week, and the month it corresponds to, assuming that the closing price can be directly influenced by time. We compute these features for the entire dataset (train + test) so that they are mapped to each closing value.

Regarding the seasonal stats, we have a total of three features. Specifically, **mean hour**, **mean day**, and **mean month**, which represent the average closing value for the given hour, day, and month we are analyzing. These averages are computed **only** based on the training set. This is crucial because if we included the test set as well, we would introduce data leakage.

Summarizing, for each timeslot, we have 7 features related to the time period and 3 features representing the average closing value for that specific timeslot.

6.2 Prediction

Our model takes clusters of records as input and outputs a single closing value each time. Specifically, we define a **window size** at the beginning and use it to segment the training data (X_train). Each record of X_train contains only the 10 features (7 + 3), we mentioned earlier. Each window of X_train has a corresponding y_train, in which its value represents the closing price of the next timeslot. The window moves one step further each time. Once the training process is complete, we need to carefully consider how we predict the ten-day period.

To create *X_test*, which will be used as input for the model, we first take the **last window-size** clusters of features, from the training set. Then, we take the test set* and apply windowing to it, just as we did with *X_train*. Finally, we place the clusters of features we mentioned earlier at the beginning of the segmented test set. This is the *X_test*, and we give it to the neural network. The first window predicts the first (predicted) value, which corresponds to the first value of the test set. The second gives the second value etc. This process is reapeated, until predictions are generated for the entire ten-day period.

As we move further into the test set, the features we use have already been precomputed. The time-based features are assigned based on the corresponding timeslot, while the seasonal stats are determined according to the values obtained from the training set. (* **NOTE**: Under no circumstances do we use the close values of the test set. We only utilize the features associated with them, without providing any direct information about the close values themselves)

6.3 Implementation

Now, it's time to discuss the model parameters we used, as well as the results we obtained.

Setup and Parameters

Hyperparameter	**Value**
Input Features	Time-based & Seasonal features (10 features)
Input Window Size	72
LSTM Layers	(128, 64) units
Dropout Rate	0.4
Dense Layers	(64, 1) units
Activation Function	ReLU (Hidden), Linear (Output)
Optimizer	Adam ($lr = 0.001$)
Loss Function	Mean Squared Error (MSE)
Evaluation Metric	Mean Absolute Error (MAE)
Batch Size	256
Epochs	50
Validation Split	20%
Early Stopping	Patience = 15

Table 6: Summary of LSTM using Time-based Features and Seasonal Statistics

The above model consists of **2 LSTM layers** with 128 and 64 neurons, respectively, which process the temporal dependencies of the data. To prevent overfitting, a **Dropout layer** with a rate of 0.4 follows each LSTM layer. Then, **a Dense layer** with 64 neurons is applied, using ReLU. Finally, the **output layer** consists of a single neuron for regression, providing the final prediction.

Experimental Results

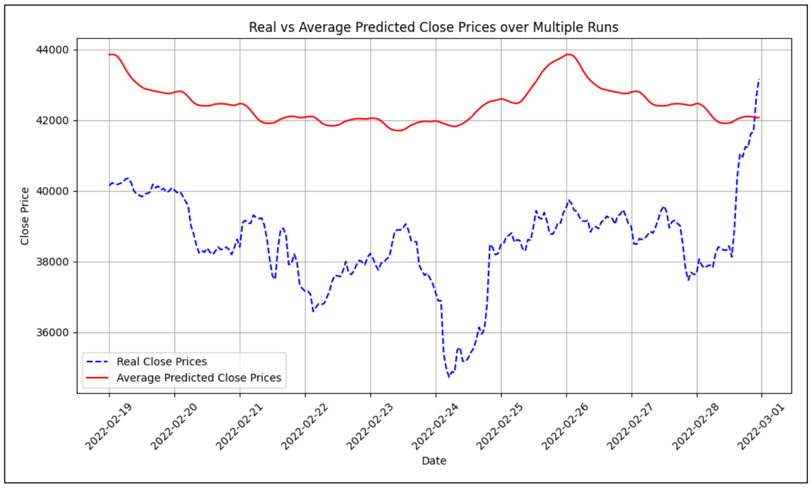

Figure 10: Predicted vs Actual values (Model=LSTM, Features=[mean_hour, mean_day, mean_month, hour_cos, hour_sin, day_cos, day_sin, month_cos, month_sin, is_weekend])

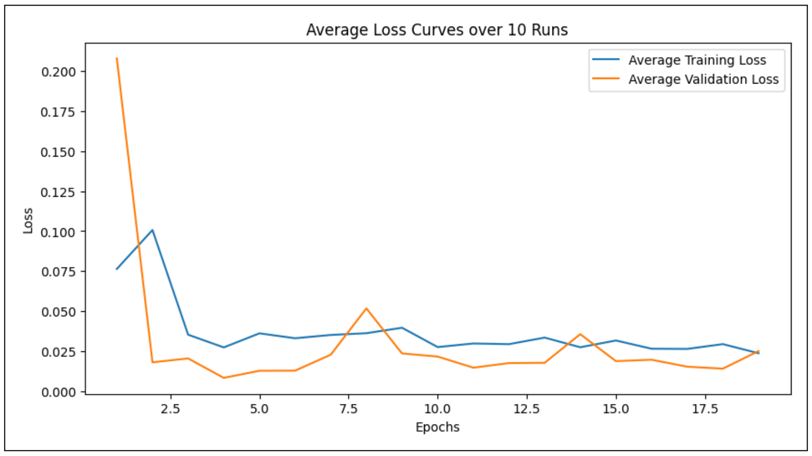

Figure 11: Validation Loss and Training Loss (Model=LSTM, Features=[mean_hour, mean_day, mean_month, hour_cos, hour_sin, day_cos, day_sin, month_cos, month_sin, is_weekend])

Metric	**Average Value (10 Runs)**
Mean Absolute Error (MAE)	3960.9892
Mean Squared Error (MSE)	17381228,8802
Root Mean Squared Error (RMSE)	4169.0801

Table 7: Average performance metrics over 10 experimental runs of LSTM using Time-based Features and Seasonal Statistics

Metric	**Time (seconds)**
Model Training Time	20.0441
Predicting Time	0.6171

Table 8: Average Execution Times over 10 Runs

6.4 Observations

We can observe in Figure 10 the predicted closing value in comparison with the actual values for the ten-day period. We see it slightly higher than the actual values, which corresponds to the higher MAE of 3960, surpassing the two previous ones. However, we can observe that the curve shows more noticeable fluctuations, which is closer to the actual movement of Bitcoin prices. This means that the model is doing a better job of capturing the real ups and downs of the Bitcoin market, compared to the previous models that had smoother, more linear trends.

It is worth noting an observation: when we ran the model for training and prediction once, we sometimes observed quite good results. However, when we ran it 10 times, as shown in the results less consistent. That's why we present results from running the model 10 times, to ensure more reliable and consistent outcomes.

In section 5, we used the closing value as a feature, and in section 6, we replaced it with time-based features and seasonal stats, as the results of the first approach were too flat and did not capture the stock's variations. It turned out that, although the error was higher, the morphology improved significantly, indicating that we should focus more on temporal dependencies.

7 Time-based Features and Moving Averages

As we mentioned above, the seasonal averages, combined with the time-based features, helped our model predict data with a shape that resembles real-world data. At this point, we need to test the Moving Averages. We create two new features that calculate the average of the last 24 hours and the average of the last 240 hours, named **MA_24** and **MA_240**. Each cluster of data now has as features the hour, the day, and the month it belongs to, as well as the moving averages of the last **24** and **240** hours.

Below, we present two approaches. In the first one, we compute the Moving Averages only for the training data, and using the **all-in-one** method we defined, our model predicts the 10-day period we aim for. In the second one, which resembles the **recursive** approach, the model predicts one close value, and based on that, the MA_24 and MA_240 are updated. After shifting the window one step to the right, the process continues until all values of the 10-day period have been predicted.

7.1 CNN-LSTM using All-in-one Approach

This time, we chose to use the above-mentioned features in combination with a **CNN-LSTM model**. In the previous implementation, which was similar to this one (it used statistical and time-based features), we used LSTM, which specializes in understanding temporal dependencies. So, we decided to experiment with the use of CNN-LSTM. But why?

Regarding CNNs, they are useful for recognizing **spatial** patterns and features in a time series, which may not be immediately apparent. Therefore, the combination of extracting local and morphological features, along with analyzing temporal dependencies, can lead to both improved model accuracy and a curve with fluctuations. When we mention fluctuations, we're referring to the CNN's ability to learn from the sharp price spikes and drops in Bitcoin within the train set, helping it produce a more accurate representation of the price movements.

We will use the **all-in-one** approach.

Setup and Parameters

The following table presents all the key hyperparameters used in this model.

Hyperparameter	**Value**
Input Features	MA & Time-based features (8 features)
Input Window Size	120
Convolutional Layers	Conv1D (64 filters, kernel_size=3)
Pooling Layer	MaxPooling1D (pool_size=2)
LSTM Layers	(256, 128) units
Dropout Rate	0.5 (Conv), 0.3, 0.2 (LSTM), 0.2 (Dense)
Dense Layers	(64, output-size) units
Activation Function	ReLU (Hidden), Linear (Output)
Optimizer	Adam ($lr = 0.001$)
Loss Function	Mean Squared Error (MSE)
Evaluation Metric	Mean Absolute Error (MAE)
Batch Size	256
Epochs	100
Validation Split	20%
Early Stopping	Patience = 15

Table 9: Summary of Model Hyperparameters of CNN-LSTM using All-in-One Moving Averages

The above model consists of **two Conv1D layers** with 64 filters and a kernel size of 3, activated by ReLU functions, which help in detecting local temporal patterns in the data. A **MaxPooling1D layer** with a pool size of 2 follows to reduce dimensionality, and a **Dropout layer** with a rate of 0.5 is added to prevent overfitting.

The model also includes **two LSTM layers** with 256 and 128 neurons, respectively, to process the temporal dependencies of the data. Each LSTM layer is followed by **Batch Normalization** to stabilize training and **Dropout layers** with rates of 0.3 and 0.2 for additional regularization.

After the LSTM layers, **a dense layer** with 64 neurons is applied, using ReLU as the activation function. Finally, the **output layer** generates a single prediction for all the data we are seeking (all-in-one approach), with the number of neurons equal to the length of the test set, using a linear activation function.

Experimental Results

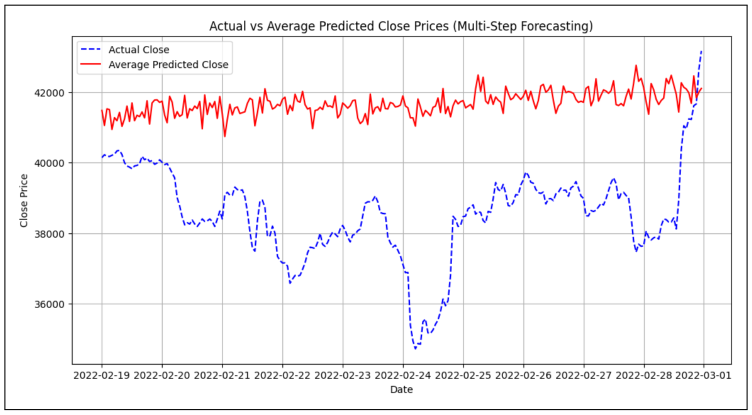

Figure 12: Predicted vs Actual values (Method=All-in-One, Model=CNN-LSTM, Features=[MA_24, MA_240, hour_sin, hour_cos, weekday_sin, weekday_cos, month_sin, month_cos])

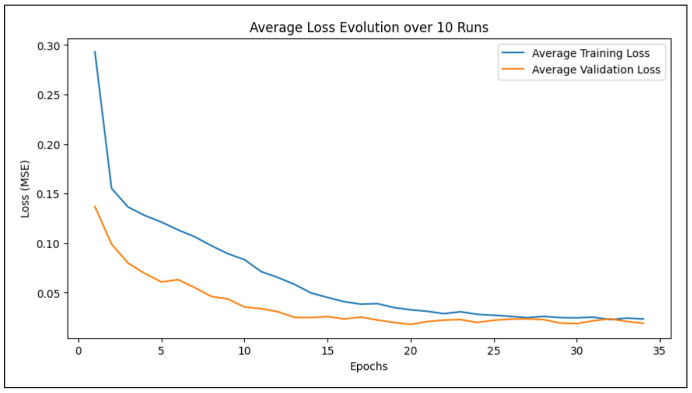

Figure 13: Validation Loss and Training Loss (Method=All-in-One, Model=CNN-LSTM, Features=[MA_24, MA_240, hour_sin, hour_cos, weekday_sin, weekday_cos, month_sin, month_cos])

Metric	Average Value (10 Runs)
Mean Absolute Error (MAE)	3769.8505
Mean Squared Error (MSE)	22229227.1176
Root Mean Squared Error (RMSE)	4053.8297

Table 10: Average performance metrics over 10 experimental runs of CNN-LSTM using All-in-One Moving Averages

Metric	**Time (seconds)**
Model Training Time	46.5175
Predicting Time	0.4304

Table 11: Average Execution Times over 10 Runs of CNN-LSTM using All-in-One Moving Averages

Observations

As observed, the model's MAE is 3769.85, which is indicating a slightly deviation from the actual values. Although the predicted values show some fluctuations, they do not accurately reflect the real market variations. The overall trend of the predicted values remains relatively stable, failing to capture the sharp increases and decreases seen in the actual values.

The results suggest that, despite using multiple features (MA_24, MA_240, etc.), the model struggles to adapt to rapid market changes. Its inability to accurately predict sudden fluctuations indicates the need to explore the recursive forecasting method. This approach could help the model better identify temporal dependencies in the data and improve the overall prediction accuracy.

7.2 CNN-LSTM using Recursive Approach

We will use the same model with some of its parameters varied, by using the recursive approach. We believe that if we stick to the same architecture and predict one value at a time, incorporating them into the next predictions, it will lead to different results.

Setup and Parameters

The following table presents all the key hyperparameters used in this model.

Hyperparameter	**Value**
Input Features	MA & Time-based features (8 features)
Input Window Size	36
Convolutional Layers	Conv1D (64 filters, kernel_size=3)
Pooling Layer	MaxPooling1D (pool_size=2)
LSTM Layers	(128, 128) units
Dropout Rate	0.2
Dense Layers	(64, 1) units
Activation Function	ReLU (Hidden), Linear (Output)
Optimizer	Adam ($lr = 0.001$)
Loss Function	Mean Squared Error (MSE)
Evaluation Metric	Mean Absolute Error (MAE)
Batch Size	256
Epochs	100
Validation Split	20%
Early Stopping	Patience = 20

Table 12: Summary of Model Hyperparameters of CNN-LSTM using Recursive Moving Averages

The model begins with a **Conv1D layer** with 64 filters, a kernel size of 3, activated by the ReLU function to detect local patterns in the sequence data. This is followed by **Batch Normalization** to stabilize learning, a **MaxPooling1D layer** with a pool size of 2 to reduce dimensionality, and a **Dropout layer** with a rate of 0.2 to prevent overfitting.

The model also includes **two LSTM layers** each with 128 units, that process the temporal dependencies of the data. Both LSTM layers are followed by **Batch Normalization and Dropout layers** with a rate of 0.2.

After the LSTM layers, **a dense layer** with 64 neurons is applied, using ReLU as the activation function. Finally, the **output layer** with a single neuron for regression.

Experimental Results

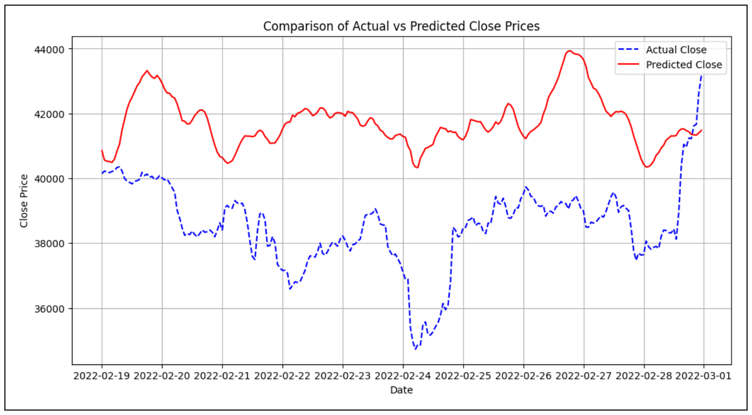

Figure 14: Predicted vs Actual values (Method=Recursive, Model=CNN-LSTM, Features=[MA_24, MA_240, hour_sin, hour_cos, weekday_sin, weekday_cos, month_sin, month_cos])

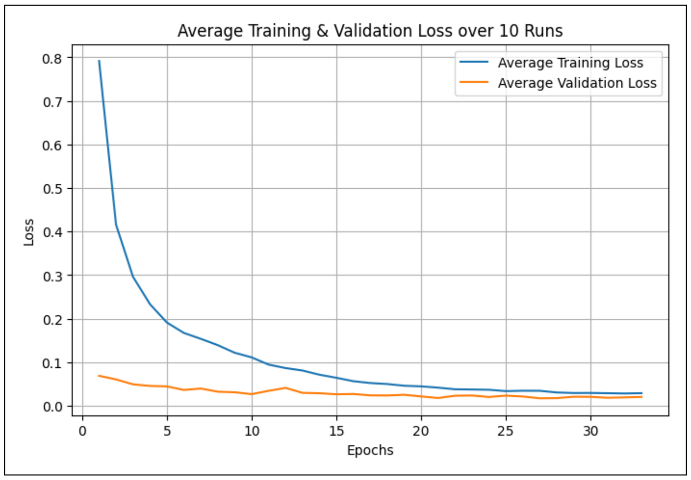

Figure 15: Validation Loss and Training Loss (Method=Recursive, Model=CNN-LSTM, Features=[MA_24, MA_240, hour_sin, hour_cos, weekday_sin, weekday_cos, month_sin, month_cos])

Metric	Average Value (10 Runs)
Mean Absolute Error (MAE)	3807.7775
Mean Squared Error (MSE)	21405819.3624
Root Mean Squared Error (RMSE)	4481.5029

Table 13: Average performance metrics over 10 experimental runs of CNN-LSTM using Recursive Moving Averages

Metric	**Time (seconds)**
Model Training Time	45.0808
Predicting Time	17.3065

Table 14: Average Execution Times over 10 Runs

Observations

We observe that the model's MAE is 3807.78, which does not differ significantly from the previous model, indicating that the average deviation from the actual values remains at similar levels. However, the graph shows a clear improvement in how the predicted values align with the actual market fluctuations. The predictions exhibit more realistic variations and follow the overall trend of the actual prices more closely.

This suggests that, while the average error remains approximately the same, the model's ability to capture the market's dynamic behavior has improved. The recursive method seems to enhance the model's performance, allowing it to better reflect the ups and downs of the actual data. However, there is still room for further improvement. To address this, we will experiment with another LSTM model, aiming to reduce prediction errors.

7.3 LSTM using Recursive Approach

Based on the results of **7.1** and **7.2**, we can conclude that the recursive method outputs more realistic closing values compared to the all-in-one method. We would like to try the combination of MA and time-based features with LSTM to see what result comes out, focusing only on the temporal dependencies.

Setup and Parameters

We begin with the parameters used in this model. These are listed in the table below.

Hyperparameter	**Value**
Input Features	MA & Time-based features (8 features)
Input Window Size	36
LSTM Layers	(128, 128, 128) units
Dropout Rate	0.2
Dense Layers	(64, 1) units
Activation Function	ReLU (Hidden), Linear (Output)
Optimizer	Adam ($lr = 0.001$)
Loss Function	Mean Squared Error (MSE)
Evaluation Metric	Mean Absolute Error (MAE)
Batch Size	256
Epochs	100
Validation Split	20%
Early Stopping	Patience = 20

Table 15: Summary of Model Hyperparameters of LSTM using Recursive Moving Averages

The above model consists of **3 LSTM layers**, each with 128 units, that process the temporal dependencies of the data. To prevent overfitting, a **Dropout layer** with a rate of 0.2 follows each LSTM layer, alongside **Batch Normalization** to stabilize training. Then, **a dense layer** with 64 neurons follows, using ReLU as the activation function, followed by another **dropout layer**, before reaching the final **output layer** with a single neuron for regression.

Experimental Results

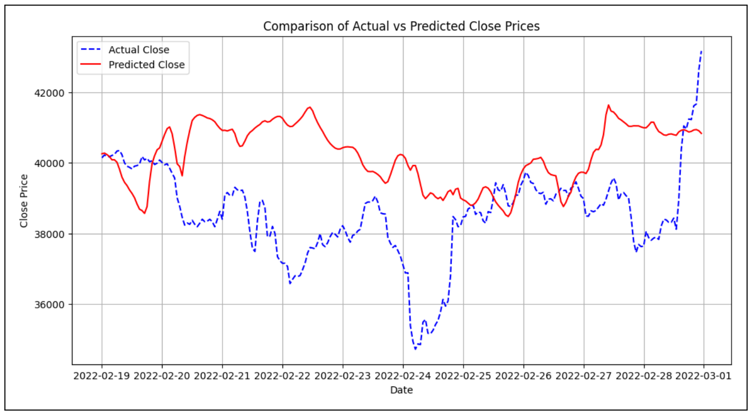

Figure 16: Predicted vs Actual values (Method=Recursive, Model=LSTM, Features=[MA_24, MA_240, hour_sin, hour_cos, weekday_sin, weekday_cos, month_sin, month_cos])

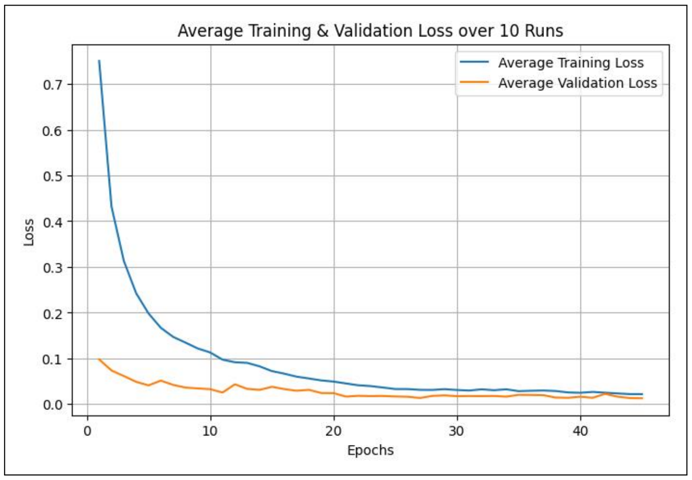

Figure 17: Validation Loss and Training Loss (Method=Recursive, Model=LSTM, Features=[MA_24, MA_240, hour_sin, hour_cos, weekday_sin, weekday_cos, month_sin, month_cos])

Metric	**Average Value (10 Runs)**
Mean Absolute Error (MAE)	2671.7974
Mean Squared Error (MSE)	10436449.8039
Root Mean Squared Error (RMSE)	3136.9178

Table 16: Average performance metrics over 10 experimental runs of LSTM using Recursive Moving Averages

Metric	Time (seconds)
Model Training Time	78.2900
Predicting Time	18.2316

Table 17: Average Execution Times over 10 Runs

Observations

The model demonstrates a significant improvement, as reflected by the reduction in the MAE to 2671.80, compared to previous models. This decrease shows smaller deviations from the actual values, suggesting more accurate predictions.

The model effectively captures the general direction and variability of the real data, with the predicted fluctuations aligning more accurately with the actual price movements. This indicates that the LSTM model, using the recursive method, demonstrates an improved ability to model temporal dependencies and adapt to the dynamic behavior of the market.

Overall, the results suggest that the model not only achieves lower prediction errors but also provides a more reliable representation of Bitcoin price variations.

7.4 Daily Sampling Results

The results of this section showed significant fluctuations and relatively good outcomes compared to the others. Therefore, we find it important to present the results of these 3 implementations for **daily sampling** in order to observe their responses.

It should be noted that the architecture of the neural networks remains the same. However, some parameters need to be adjusted, as we are now performing analysis on a daily basis rather than hourly. Specifically, we are changing the calculation window for the Moving Averages. We will now use **MA5** and **MA24**, meaning the averages for the previous 5 and 24 days, respectively. Additionally, we are adjusting the window-size when extracting the X_train (i.e., the input). For the recursive methods, we observed optimal results with a window-size of 14, while for the all-in-one method, 24 was found to be more effective.

Below, we present for each implementation the graphical representation of the predicted values, along with a summary table of metric results after 10 simulations.

CNN-LSTM using All-in-One

Here we got the plot actual vs predicted closing values and the progression of loss and validation loss over epoch :

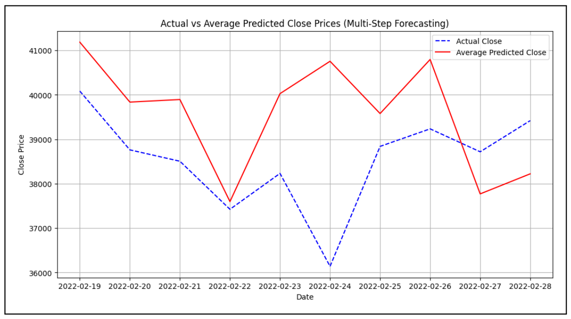

Figure 18: Actual vs Predicted (Daily Sampling)

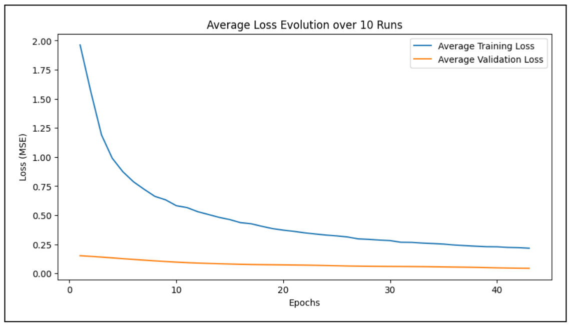

Figure 19: Training and Validation Loss (Daily Sampling)

We also present the training and prediction times.

Metric	**Time (seconds)**
Model Training Time	18.4397
Predicting Time	0.3996

Table 18: Average Execution Times over 10 Runs of CNN-LSTM using All-in-One (Daily Sampling)

CNN-LSTM using Recursive

We present the results for this implementation in the same way. Here we have the plots :

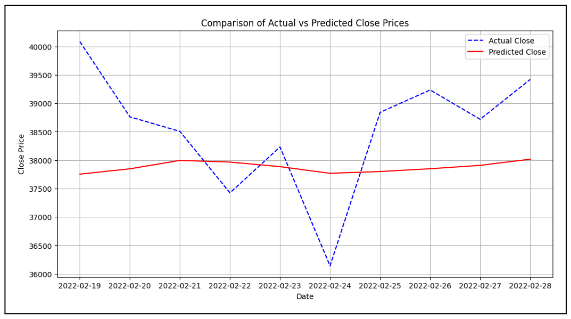

Figure 20: Actual vs Predicted (Daily Sampling)

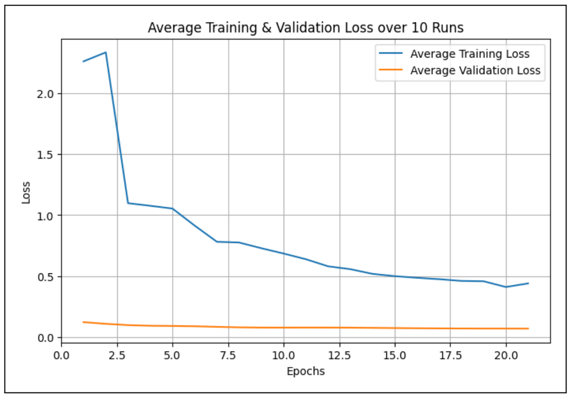

Figure 21: Training and Validation Loss (Daily Sampling)

and the execution times :

Metric	Time (seconds)
Model Training Time	17.9897
Predicting Time	1.1492

Table 19: Average Execution Times over 10 Runs of CNN-LSTM using Recursive (Daily Sampling)

LSTM using Recursive

Finally, the third implementation we give the plots:

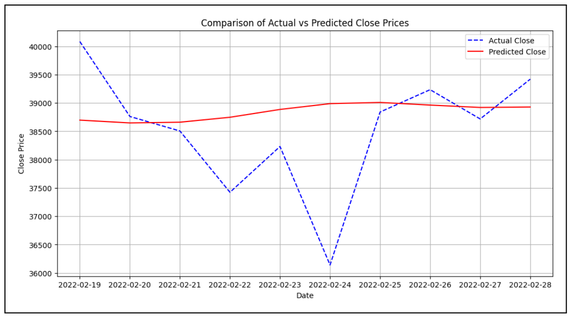

Figure 22: Actual vs Predicted (Daily Sampling)

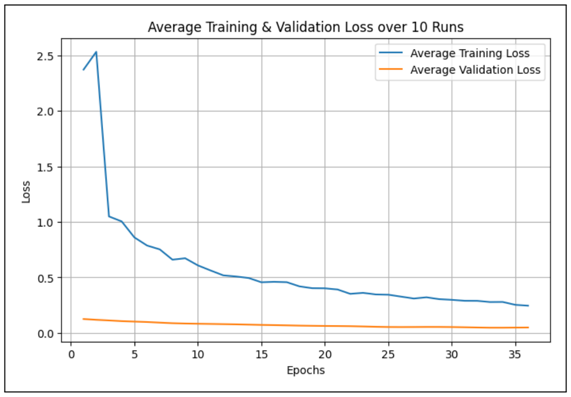

Figure 23: Training and Validation Loss (Daily Sampling)

and the execution time :

Metric	Time (seconds)
Model Training Time	17.1030
Predicting Time	1.0748

Table 20: Average Execution Times over 10 Runs of LSTM using Recursive (Daily Sampling)

Metrics Summary Table

Here we have the summary table of metrics for each method:

Metric	CNN-LSTM (All-in-One)	CNN-LSTM (Recursive)	LSTM (Recursive)
MAE	3178.0803	2805.1925	2199.7086
MSE	16536174.0030	10193204.8767	6709750.2727
RMSE	3879.3454	3008.9859	2434.7025

Table 21: Average performance metrics over 10 experimental runs for all three implementations (Daily Sampling)

Observations

Regarding the graphical representations, we observe that the recursive approaches produce a flatter outcome (which appear similar to each other), in contrast to the all-in-one method, which exhibits more significant fluctuations.

When it comes to deviations, it appears that the LSTM implementation with the recursive approach is the best according to the above metrics, followed by the CNN-LSTM with the recursive approach, while the all-in-one approach has the worst relative performance.

8 ARIMA

The next implementation will be using the ARIMA model. **A**uto**R**egressive **I**ntegrated **M**oving **A**verage is a statistical analysis model that uses time series data to either better understand the data set or to predict future trends.

In this section, we will start by examining the data. Then, we will discuss the best parameters we selected and present graphical representations. Lastly, we will share our observations.

8.1 Stationary Series

One of the main requirements for applying ARIMA is the stationarity of the data. Therefore, before passing the data, we need to check whether this condition holds. Specifically, after keeping only the numerical values of the ”close” column from the training set, we pass them through the **Augmented Dickey-Fuller (ADF) test**, which returns a p-value to check whether the time series is stationary. We have set an upper limit of $\underline{p = 0.05}$ for the series to be considered stationary.

When we first apply the test to our data, we obtain a p-value equal to 0.1595, indicating that the series is not stationary. To address this, we compute the differences between the closing values using the following equation:

$$y_t = Y_t - Y_{t-1} \quad \forall t$$

Now we need to re-examine, if the ADF test gives us $p \leq 0.05$ for the new differenced series. We see, that $p = 0.00$, so we conclude that : $\underline{\textbf{We need to use d = 1}}$, where d is the second parameter of ARIMA and is the number of nonseasonal differences needed for stationarity.

8.2 Parameters and Results

We have now determined that the second parameter d is equal to 1, as the series requires differencing only once to become stationary. The next step is to determine the values for parameters **p** and **q**.

After training and evaluating multiple models, where p and q ranged from 0 to 15, we generated all possible combinations and computed their respective performance metrics. The best parameter combination was (p, d, q) = $(\mathbf{10}, \mathbf{1}, \mathbf{10})$, based on the evaluation metrics:

Metric	**Value**
Mean Absolute Error (MAE)	1632.5718
Mean Squared Error (MSE)	3974029.9059
Root Mean Squared Error (RMSE)	1993.4969

Table 22: Performance metrics of ARIMA (10, 1, 10)

Metric	Time (seconds)
Model Training Time	57.1409
Predicting Time	0.02380

Table 23: Training and inference time of ARIMA(10, 1, 10)

Additionally, we provide a graphical representation of the results to illustrate the model's performance.

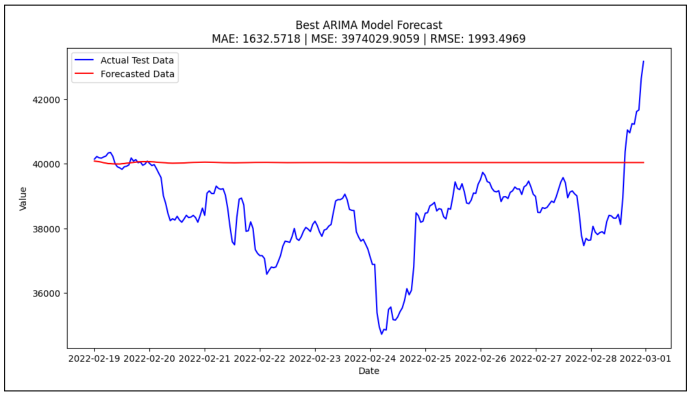

Figure 24: ARIMA model performance

8.3 Observations

The result of the ARIMA technique with parameters (10, 1, 10) shows the same characteristics as the model we implemented in Section 5 (as well as the one we will see in the next section). Specifically, it achieves a relatively low MAE, but it is too linear, without any variability. Therefore, we can conclude that it is not an acceptable solution, as it does not exhibit the characteristics of a curve that represents Bitcoin price movements.

9 LSTM using Differences of Close values

This is our final implementation for the coding project. We chose to see what results we would get by providing a neural network with the same inputs used by ARIMA. As indicated by the title, we will use an **LSTM** neural network and **the differences in closing values** (just like ARIMA calculates when d=1) as input features. This approach is influenced by ARIMA. As mentioned earlier, in time series forecasting it's beneficial to use time-based and statistical features instead of closing values. However, it's worth giving it a try.

In this section, we will demonstrate how we process the data before it enters the neural network and after it for prediction.

Last but not least, we're using the **all-in-one approach**.

9.1 Data Processing

First, we will discuss the data processing before it is fed into the model for training. After performing the train-test split, we take the close values and calculate the differences between consecutive values. This will be the feature we use for training. At the same time, we keep the last value from the training set (last_close_value), which we will use for prediction. It is important to note that we also apply scaling after calculating the differences.

Once we have calculated the differences, we split the data into **X_train** and **y_train** using an all-in-one approach. It should be noted that all values refer to the differences in the closing values.

Finally, during the prediction process, we obtain the diff-close values. Therefore, we need to convert these diff-close values back to the original close values using the **last_close_value** that we mentioned earlier, in the following way:

$$predicted\_close = last\_close\_value + diff\_close$$

9.2 Implementation

Setup and Parameters

We begin with the parameters used in this model. These are listed in the table below.

Hyperparameter	**Value**
Input Features	Difference of Close Values (1 feature)
Input Window Size	24
LSTM Layers	(128, 64, 32) units
Dense Layers	(32, output-size) units
Activation Function	ReLU (Hidden), Linear (Output)
Optimizer	Adam ($lr = 0.001$)
Loss Function	Mean Squared Error (MSE)
Evaluation Metric	Mean Absolute Error (MAE)
Batch Size	128
Epochs	100
Validation Split	20%
Early Stopping	Patience = 15

Table 24: Summary of Model Hyperparameters of LSTM using Differences of Close values

The above model consists of **3 LSTM layers** with 128, 64 and 32 neurons respectively, which process the temporal dependencies of the data. Then, **a dense layer** with 32 neurons follows, using ReLU as the activation function. Finally, the **output layer** generates a single prediction for all the data we are seeking (all-in-one approach), with the number of neurons equal to the length of the test set.

Experimental Results

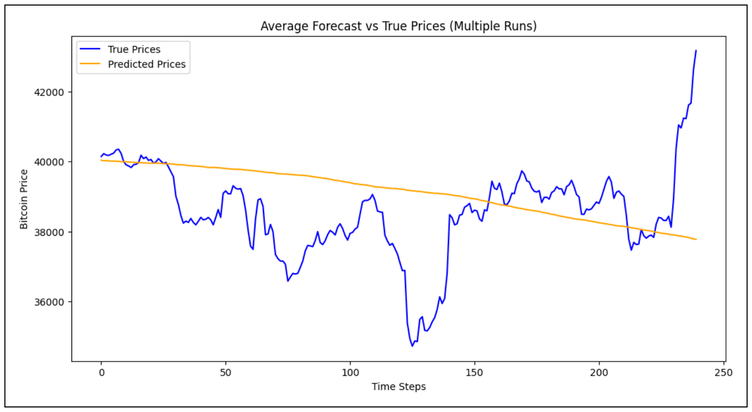

Figure 25: Predicted vs Actual values (Model=LSTM, Features=diff_close)

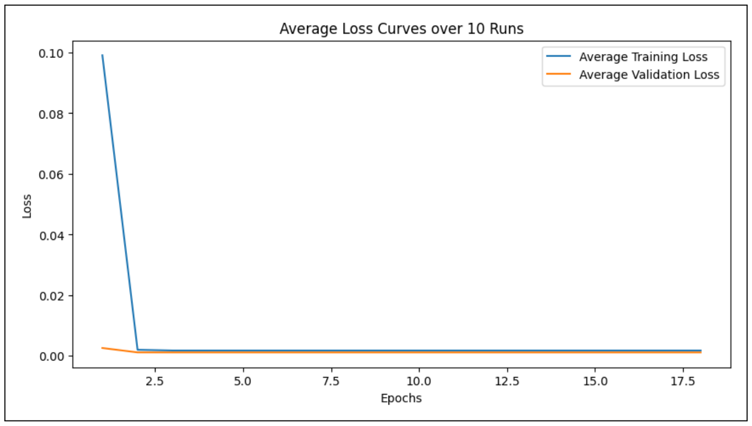

Figure 26: Validation Loss and Training Loss (Model=LSTM, Features=diff_close)

Metric	Average Value (10 Runs)
Mean Absolute Error (MAE)	1224.1342
Mean Squared Error (MSE)	2773016.59750129
Root Mean Squared Error (RMSE)	1665.2377

Table 25: Average performance metrics over 10 experimental runs of LSTM using Differences of Closing Values

Metric	Time (seconds)
Model Training Time	27.2874
Predicting Time	0.8997

Table 26: Average Execution Times over 10 Runs

9.3 Observations

After analyzing the graph in Figure 12, we observe that the Mean Absolute Error (MAE) over 10 runs is approximately 1224, indicating a relatively low average deviation from the actual prices. However, despite this low error, the predicted prices follow a smooth and declining trend, failing to reflect real market behavior. This suggests that the model does not reflect the natural ups and downs of Bitcoin prices, as it smooths out variations instead of accurately representing the sharp upward and downward movements.

One thought that crosses our mind is that by using the differences in closing values, the model could identify patterns in which the price of Bitcoin shows an increase or decrease. However, based on the results, we don't observe any significant fluctuations, and it seems that the diff-close as a feature does not capture the expected patterns effectively.

10 Summary

In this section, we will present the results of all the implemented models examined in this study to evaluate and determine the best solution. The results are displayed in order based on their corresponding Mean Absolute Errors (MAE).

Below is the aggregated table along with the corresponding Mean Absolute Errors (MAE) :

Implementation	Mean Absolute Error
LSTM using Differences of Closing Values	1224.1342
ARIMA	1632.5718
LSTM using Close (All-in-One)	2577.6335
LSTM using Time-based features - MA (Recursive)	2671.7974
LSTM using Close (Recursive)	3066.9017
CNN-LSTM using Time-based Features - MA (All-in-One)	3769.8505
CNN-LSTM using Time-based Features - MA (Recursive)	3807.7775
LSTM using Time-based Features and Seasonsal Stats	3960.9892

Table 27: Summary table of all implementations and their corresponding Mean Absolute Error (MAE)

It is evident from the table above which model is the ”best” if error is our sole criterion. The implementation using **LSTM with Differences of Closing Values** appears to be the best, followed by the **ARIMA** method and then **LSTM using Close Values (All-in-One)**.

However, these three implementations share a common characteristic that we discussed earlier—they produce results (curves) that are very linear, with no signs of significant fluctuations, which is a key feature that typically describes Bitcoin prices. It is logical and expected that if we draw a straight line, it will have a much lower comparison error than a curve that exhibits fluctuations. Therefore, we believe that error should not be the sole criterion for evaluating the models. Instead, the shape of the curve they produce should also be taken into account.

As we move further down the list, we come across the **LSTM using Time-based Features - MA with Recursive approach** implementation. This is the **solution** we propose in the present coding project. It is an implementation that exhibits significant **fluctuations**, effectively simulating the movement of Bitcoin value, while at the same time maintaining a relatively **low error** compared to other implementations that also show variations.

But why we reject the other solutions ?

The next implementation in the list is **LSTM using Close with Recursive approach**, which also produces a very linear result and has a higher error, so it is rejected.

Following this, we have **CNN-LSTM using Time-based Features - MA with All-in-One approach**, which generates results that resemble noise rather than meaningful Bitcoin price predictions. Since the output does not align with the expected price movements, this model is also dismissed, although it gives very good results in daily analysis.

Moving on, we have **CNN-LSTM using Time-based Features - MA with Recursive approach**. It approximates the desired curve quite well and belongs to the same category as our proposed solution. However, since it has a higher error, we reject it as the best choice.

Finally, we have the **LSTM implementation using time-based feature and chronological statistics**. We observe that this result is neither very linear nor does it exhibit significant fluctuations. So, when compared to the proposed solution, this one is rejected.

11 Conclusion

In this coding project, we chose to take an exploratory approach with the goal of testing different combinations of features and neural network architectures in order to find the model that predicts the Bitcoin price most accurately over a finite period of time.

After carefully preprocessing our data (Section 2) and selecting certain techniques, such as shuffling, EarlyStopping etc. that we applied commonly across all implementations (Section 4), we proceeded with the design of the models.

Initially, we started by using only the closing values as features with an LSTM model, which resulted in a very linear outcome that we were not particularly satisfied with. We were concerned and decided to incorporate time into the equation, as we are dealing with time-series prediction. Therefore, we included features that relate to seasonal statistics as well as information about time (time-based features). The result was not as linear as the previous one, so we were encouraged to continue focusing on similar features.

We then proceeded to add the **Moving Average** to the process, along with a variation in the architecture of the neural networks (CNN-LSTM). It was demonstrated that the use of Moving Average and time-based features, combined with an LSTM, yielded very good results, both in terms of error and in the shape of the predicted value. The moving average feature provides the model with the ability to better understand temporal dependencies by using previous records to predict the desired closing price.

Finally, we tested the ARIMA model and the diff-close approach. While they had low error, the results were too linear.

In conclusion, our final proposal is the implementation that uses the LSTM architecture with a combination of Moving Averages and time-based features, following the Recursive approach. This model provides a very good representation of the data while maintaining a relatively low error compared to the other implementations.

The truth is that predicting the price of Bitcoin proved to be a very challenging task, as no one can predict it with high accuracy due to its nature—it changes in an unpredictable way and is influenced by many different factors, given its connection to the financial markets.

12 Submitted Files

In the submitted files you will find:

- The provided report named **CE418_Project_03321_03322_03144**.
- The dataset named bitcoin-dataset.csv. *
- A folder named **PerDayCodes**, which contains subfolders with the Python codes (.py) related to our daily prediction.
- A folder named **PerHourCodes**, which contains subfolders with the Python codes (.py) related to our hourly prediction.
- A folder named **Notebook**, which contains all our notebooks that were run on Google Colab. These notebooks are organized into two folders: **PerDayPrediction** and **PerHourPrediction**.

Within the subfolders of **PerDayCodes** and **PerHourCodes**, all the code is structured into 4 Python files **(main.py, data_loader.py, model_training.py, inference.py)** as required. This structure is designed to achieve flexibility in using different forms of input data, as only a simple wrapper is needed to convert them into data that is consumable by our neural network.

If you want to run, for example, the method from Section 7.1 "Time-based Features and Moving Averages - CNN-LSTM using All-in-one Approach," navigate to **PerHourCodes/MA_TIME_CNNLSTM_AllinOne** and run the file main.py using: *python main.py*.

The corresponding notebooks are provided as they include the graphs from the multiple runs we performed for each method. Regarding the implementation, there is no difference between the code in the notebooks and the code split into separate files.

* **NOTE:** The dataset exceeds the permitted size limits for email transmission. However, if you want to run the scripts, you need to place it in this directory with the name `bitcoin-dataset.csv`. In each `main.py`, the dataset is loaded from the path `"../../bitcoin-dataset.csv"`.

13 Bibliography

[1] Y. Hua, “Bitcoin Price Prediction Using ARIMA and LSTM,” in *E3S Web of Conferences*, vol. 218, 2020. DOI: 10.1051/e3sconf/202021801050.

[2] S. M. Raju and A. M. Tarif, “Real-Time Prediction of Bitcoin Price Using Machine Learning Techniques and Public Sentiment Analysis,” *arXiv preprint*, 2020. DOI: 10.48550/arXiv.2006.14473.

[3] O. G. Darley, A. I. O. Yussuff, and A. A. Adenowo, “Price Analysis and Forecasting for Bitcoin Using Auto Regressive Integrated Moving Average Model,” *Annals of Science and Technology*, vol. 6, no. 2, pp. 47–56, 2021. DOI: 10.2478/ast-2021-0009.

[4] O. Omole and D. Enke, “Deep Learning for Bitcoin Price Direction Prediction: Models and Trading Strategies Empirically Compared,” *Financial Innovation*, vol. 10, no. 1, article no. 117, 2024. DOI: 10.1186/s40854-024-00643-1.

[5] S. Javadi Masoudian, P. M. Kathuria, N. S. Gowda, and T. A. Khan, “Bitcoin Price Prediction Using Long Short Term Memory Neural Networks,” in *2022 International Conference on Artificial Intelligence (ICAI)*, IEEE, 2022. DOI: 10.1109/ICAI55857.2022.9960055.