UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NEURO-FUZZY COMPUTING

TIME SERIES FORECASTING FOR BITCOIN

Final Project

Authors:

Anastasios Fotiadis
Konstantinos Tasiopoulos

Supervisor:

Dimitrios Katsaros

February 2025

Contents

List of Figures

1 Introduction

1.1 Project Goal

In this project, our aim is to develop a neural network-based forecasting model to predict the daily closing price of Bitcoin. The data set provided consists of minute-level Bitcoin price data for the period from January 1, 2021, to March 1, 2022.

Our objective is to train a model using historical price trends and evaluate its performance by predicting Bitcoin prices for the last ten days of February 2022. Since Bitcoin price data are sequential and time-dependent, we will employ **Time Series Forecasting** techniques.

1.2 Dataset Import

To begin our analysis, we first import the data set using the `pandas` library:

```python
import pandas as pd

data = pd.read_csv("bitcoin_data.csv")

data.head()
data.info()
data.describe()
```

The following output represents the result of executing the `data.info()` function, which provides a summary of the dataset, including the number of entries, column names, non-null counts, and data types.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 182301 entries, 0 to 182300
Data columns (total 9 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   unix        182301 non-null  int64
 1   date        182301 non-null  object
 2   symbol      182300 non-null  object
 3   open        182300 non-null  float64
 4   high        182300 non-null  float64
 5   low         182300 non-null  float64
 6   close       182300 non-null  float64
 7   Volume BTC  182300 non-null  float64
 8   Volume USD  182300 non-null  float64
dtypes: float64(6), int64(1), object(2)
memory usage: 12.5+ MB
```

This data set includes columns such as *open price*, *close price*, *volume*, and *timestamp*, which we will preprocess before training our model.

2 Data Preprocessing

Before applying machine learning techniques, the data set needs to be cleaned and formatted:

2.1 Timestamp Conversion

The dataset provides timestamps in Unix format. We convert them into a readable date-time format:

```
data['timestamp'] = pd.to_datetime(data['unix'], unit='s')
data.set_index('timestamp', inplace=True)
```

2.2 Resampling to Daily Data

The original dataset consists of minute-level Bitcoin price data, totaling 610,782 records. However, our goal is to predict the **daily closing price**, which means that the high-frequency data are too granular for our intended purpose.

If we were to train our model on minute-level data, it might learn short-term fluctuations rather than capturing broader trends. This would lead to increased computational complexity without significant benefits for daily forecasting. By resampling the data to **daily closing prices**, we ensure:

- A **Smoother trend**, allowing the model to generalize better.
- **Reduced computation cost**, as fewer data points improve efficiency.
- **Better performance**, since the model can focus on meaningful long-term price patterns.

We resample the data set by aggregating the last recorded price of each day as the **closing price** and calculating the sum of the total volume of Bitcoin.

```
daily_data = data.resample('D').agg({'close': 'last', 'Volume␣BTC
    ': 'sum'})
```

2.3 Handling Missing Data

If there are missing days in our dataset, we interpolate them:

```
daily_data.isna().sum()
daily_data.interpolate(inplace=True)
```

2.4 Feature Engineering

To enhance the model's ability to detect patterns in Bitcoin price movements, we introduce additional **time-based features**. These features help the model understand the correlation between price trends and specific time periods, leading to more accurate predictions.

The following features are extracted from the dataset:

- **Day of the week**: Encodes weekdays numerically, where 0 represents Monday and 6 represents Sunday.
- **Month**: Stores the month as an integer, ranging from 1 (January) to 12 (December).
- **Year**: Represents the calendar year (e.g. 2021, 2022).

Adding these particular characteristics improves our model by capturing **temporal dependencies** and **seasonal patterns**. Each characteristic contributes to the forecasting process in different ways.

- **Day of the week**: Cryptocurrency markets operate 24/7, but trading volume and volatility fluctuate based on the day. Weekends often exhibit lower trading activity because institutional traders are less active on Saturdays and Sundays. Our neural network can learn these cyclical trading behaviors and adjust its predictions accordingly.

- **Month**: The cryptocurrency market follows seasonal trends. Institutional investors adjust their holdings based on quarterly and yearly financial cycles, affecting prices. Additionally, significant events like *Bitcoin halving* (which occurs every four years) influence price trends in specific months. December price spikes are often observed due to increased investing activity from year-end bonuses and holiday income.

- **Year**: Macroeconomic trends evolve yearly due to regulatory changes and market adoption. As Bitcoin matures, new policies, financial instruments, and regulations affect its market behavior. For example, if Bitcoin performed better in 2021 than in 2022 due to regulatory uncertainties, the model can differentiate these behaviors by incorporating the year as a characteristic.

Although the data set already contains a date column, raw timestamps do not provide inherently meaningful insights for time-series forecasting. Recurrent Neural Networks (RNNs) such as LSTMs and Transformers do not process categorical or string-based time representations effectively. Therefore, we encode these time-based features as numerical values to ensure that our model can learn useful temporal patterns.

Time-based features are added to our dataset using the following transformations:

```
daily_data['day_of_week'] = daily_data.index.dayofweek
daily_data['month'] = daily_data.index.month
daily_data['year'] = daily_data.index.year
```

Since these features are cyclical, we apply sine and cosine transformations:

```
import numpy as np

daily_data['day_sin'] = np.sin(2 * np.pi * daily_data['
   day_of_week'] / 7)
daily_data['day_cos'] = np.cos(2 * np.pi * daily_data['
   day_of_week'] / 7)

daily_data['month_sin'] = np.sin(2 * np.pi * daily_data['month']
   / 12)
daily_data['month_cos'] = np.cos(2 * np.pi * daily_data['month']
   / 12)
```

Now that we finished everything needed for the preprocessing of the data, we need to ensure that the dataset is fully processed for the time series forecasting.

```
print(daily_data.isnull().sum())
```

```
close          0
Volume BTC     0
day_of_week    0
month          0
year           0
day_sin        0
day_cos        0
month_sin      0
month_cos      0
dtype: int64
```

No null values found, meaning we have no missing data.

2.5 Data Stationarity Analysis

2.5.1 Checking for Stationarity

To determine whether the dataset is stationary, we perform the Augmented Dickey-Fuller (ADF) test:

```
from statsmodels.tsa.stattools import adfuller

adf_test = adfuller(daily_data['close'])
print(f"ADF␣Statistic:␣{adf_test[0]}")
print(f"P-value:␣{adf_test[1]}")
```

```
ADF Statistic: -2.4494110224214034
P-value: 0.12827316930690108
```

The ADF test helps us understand whether the data's statistical properties remain constant over time. A p-value greater than 0.05 indicates that the data are non-stationary, which means we need to transform it before proceeding with model training.

2.5.2 Handling Non-Stationary Data

Financial time series data often exhibit non-stationary behavior, meaning their statistical properties, such as mean and variance, change over time. Before applying machine learning models, we must transform the data to achieve stationarity. One common approach is **differencing**, which helps remove trends by subtracting each data point from its previous value.

To visualize the trends, we first plot the original closing prices:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12,5))
plt.plot(daily_data.index, daily_data['close'], label='Original␣
   Close␣Prices', color='blue')
plt.title('Bitcoin␣Closing␣Price␣Over␣Time')
plt.xlabel('Date')
plt.ylabel('Price (USD)')
plt.legend()
plt.show()
```

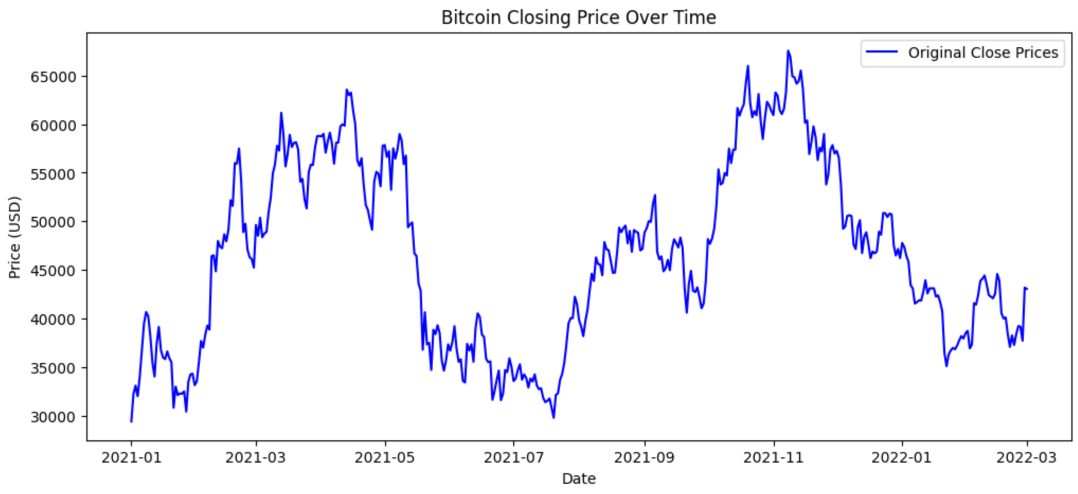

Figure 1: Bitcoin Closing Price Over Time

Observing figure 1, we identify strong upward and downward trends, confirming that the data is non-stationary. We apply the differencing method:

```
daily_data['close_diff'] = daily_data['close'].diff()
daily_data.dropna(inplace=True)
```

2.5.3 Re-Evaluating Stationarity

After applying differencing, we rerun the Augmented Dickey-Fuller (ADF) test to verify stationarity:

```
adf_test_diff = adfuller(daily_data['close_diff'])
print(f"Differenced Data - ADF Statistic: {adf_test_diff[0]}")
print(f"Differenced Data - P-value: {adf_test_diff[1]}")
```

```
Differenced Data - ADF Statistic: -21.53425386425879
Differenced Data - P-value: 0.0
```

Since the p-value is now below 0.05, we confirm that the differenced data is stationary, allowing us to proceed with model development.

2.6 Data Normalization

Neural networks perform better when input data is within a standard range, typically between 0 and 1. Scaling helps prevent numerical differences from dominating the learning process, ensuring stability and improving convergence. We use **MinMaxScaler** to normalize the data:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))
```

```
daily_data[['close_diff_scaled']] = scaler.fit_transform(
   daily_data[['close_diff']])
```

To verify that scaling was successful, we check the minimum and maximum values:

```
print(f"Min:␣{daily_data['close_diff_scaled'].min()},␣Max:␣{
   daily_data['close_diff_scaled'].max()}")
```

```
Min: 0.0, Max: 1.0
```

2.7 Data Splitting

In time series forecasting, it is crucial to split data chronologically to prevent data leakage. The model must learn from past data and predict future values without exposure to future information.

For this project, we split the data set into training and testing sets. Based on the problem's description, we define:

- **Training data**: Before February 19, 2021 - **Test data**: February 19 to February 28, 2021

```
split_date = '2022-02-19'

train_data = daily_data[daily_data.index < split_date].copy()
test_data = daily_data[daily_data.index >= split_date].copy()
```

To visualize the transformation, we compare the original differenced close prices with the scaled version:

```
plt.figure(figsize=(12,5))
plt.plot(daily_data.index, daily_data['close_diff'], label='
   Original␣Differenced␣Close␣Price', color='blue')
plt.plot(daily_data.index, daily_data['close_diff_scaled'], label
   ='Scaled␣Differenced␣Close␣Price', color='red', linestyle='
   dashed')
plt.legend()
plt.title('Comparison␣of␣Original␣and␣Scaled␣Differenced␣Closing␣
   Prices')
plt.show()
```

Examining the results of figure 2, we confirm that normalization preserves the shape of the data while ensuring values remain within a standard range. With all preprocessing steps successfully completed, we now prepare the data for input into the neural network by structuring it into sequential input-output pairs.

3 Creating Sequences for Neural Network Training

Neural networks, particularly Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, require input data in a **sequential format** since they process temporal dependencies. Instead of treating price changes as independent values, the model learns patterns by considering a fixed number of previous days (*lookback window*) to predict the price movement for the next day.

Since we have the closing price column in both differenced and scaled form, we must structure the data into **input-output pairs**, where:

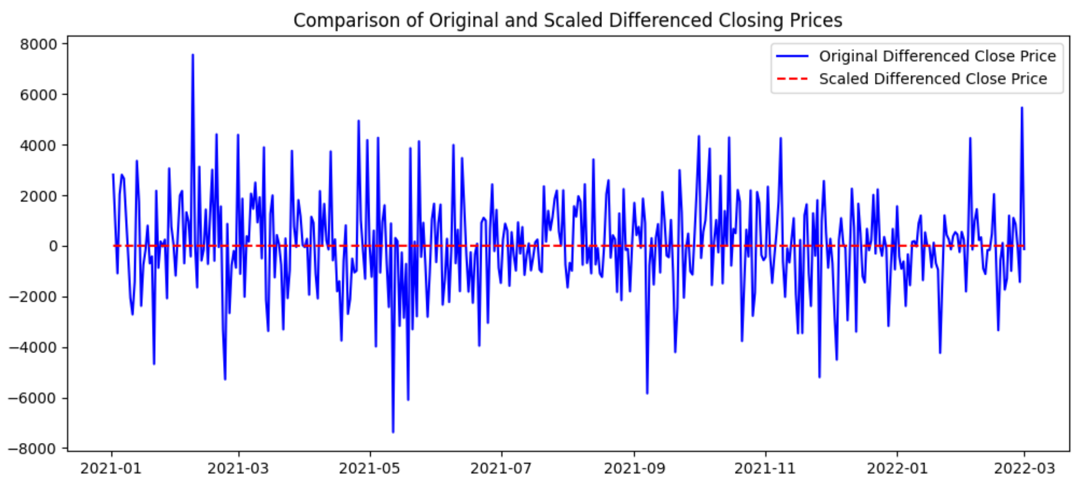

Figure 2: Comparison of Original and Scaled Differenced Closing Prices

- **X (input)**: Contains the past *lookback* days of the `close_diff_scaled` values.
- **Y (output)**: Stores the next day's predicted `close_diff_scaled` value.

The choice of an appropriate lookback period is crucial, as it directly influences the quality and accuracy of the predictions. We observed long-term trends in the preprocessing phase, indicating that a larger lookback window may be more effective. However, selecting an optimal lookback value requires balancing market cycles, data volatility, and computational efficiency.

Given that our data set spans 14 months (**January 1, 2021, to March 1, 2022**, totaling **426 days**), we analyze different lookback values using a comparative approach:

```
df.to_csv("lookback_analysis.csv", index=False)
df.head()
```

Based on this analysis, a lookback period of **30 days** is chosen as the optimal threshold. However, a **60-day window** could also be viable, as it includes additional historical data and aligns with observed trends.

Since our goal is to predict Bitcoin prices for **February 19 - February 28, 2022**, we must generate sequences that allow predictions for these exact dates while ensuring that the lookback requirement is met. Because we selected a 30-day lookback window, the first prediction date (**February 19**) must be based on the prior 30 days (starting from **January 20, 2022**). Thus, the test set must include these past 30 days.

To confirm correctness, we print the dimensions of the dataset.

```
import numpy as np

def create_sequences(data, lookback):
    X, y = [], []
    for i in range(len(data) - lookback):
        X.append(data[i:i+lookback])
        y.append(data[i+lookback])
    return np.array(X), np.array(y)

lookback = 30
```

```
X_train, y_train = create_sequences(train_data['close_diff_scaled
    '].values, lookback)

test_start_date = '2022-01-20'
test_data_extended = daily_data[daily_data.index >=
    test_start_date].copy()

X_test_full, y_test_full = create_sequences(test_data_extended['
    close_diff_scaled'].values, lookback)

X_test = X_test_full[-10:]
y_test = y_test_full[-10:]

print(f"Train␣Shape:␣X={X_train.shape},␣y={y_train.shape}")
print(f"Test␣Shape:␣X={X_test.shape},␣y={y_test.shape}")
```

```
Train Shape: X=(383, 30), y=(383,)
Test Shape: X=(10, 30), y=(10,)
```

4 LSTM Model

Long Short-Term Memory (LSTM) networks are particularly effective for forecasting Bitcoin prices due to their ability to capture long-term dependencies in sequential data. Unlike traditional time-series models such as ARIMA, which assume linear relationships, or simple RNNs, which suffer from the vanishing gradient problem, LSTMs retain historical information through a **gated memory mechanism**. This makes them well-suited for identifying complex trends and patterns in highly volatile markets such as cryptocurrency trading.

While LSTMs are powerful, their accuracy can be influenced by external factors such as:

- Government regulations and policy changes.
- Market sentiment and investor behavior.
- Macroeconomic indicators that are not reflected in the data set.

Although these external factors are beyond the scope of this study, LSTMs provide a **solid foundation for time-series forecasting**. The integration of additional features, such as trading volume, sentiment analysis, or macroeconomic indicators, could further enhance their predictive power.

4.1 LSTM Model Overview

Before defining the LSTM model, we ensure that the training sequences (`X_train`, `y_train`) are properly structured. LSTMs process time-dependent data by taking sequences of past observations as input and learning patterns to predict future values.

Our choice of a **30-day lookback window** is particularly suitable for Bitcoin forecasting, as it captures both **short-term fluctuations** and **medium-term trends**. Cryptocurrencies often exhibit monthly cycles influenced by investor sentiment, economic events, and global financial conditions.

- A shorter lookback window (e.g., 7-14 days) may not provide enough historical context.
- A much longer lookback window (e.g., 90+ days) could introduce outdated patterns and increase overfitting risks.

4.2 Comparison with Other Models

To justify the choice of an LSTM-based model, we compare its advantages with other neural network architectures commonly used for time-series forecasting.

Multi-Layer Perceptron (MLP)

- MLP models treat all inputs as independent, preventing them from capturing temporal dependencies.
- They require extensive **manual feature engineering**, whereas LSTMs learn sequential patterns directly.
- They cannot process sequential data effectively, making them unsuitable for time-series forecasting.

Simple Recurrent Neural Networks (RNNs)

- Standard RNNs struggle with **vanishing gradients**, preventing them from learning long-term dependencies.
- They tend to lose important information from earlier time steps.
- Unlike LSTMs, RNNs lack a **gated memory mechanism** to control information flow over long sequences.

Gated Recurrent Unit (GRU)

- GRUs are a simplified version of LSTMs, requiring fewer parameters and offering faster training.
- While they perform well on short sequences, they struggle to retain information over long time periods.
- LSTMs are preferable for Bitcoin forecasting, as they better capture long-term price trends.

Transformer-Based Models

- Attention-based models (e.g., GPT, BERT) have shown promise for time-series forecasting.
- They excel at capturing long-range dependencies but require **massive datasets and computational power**.
- Given our dataset size, transformers may lead to overfitting, making LSTMs a more practical choice.

4.3 LSTM Model Architecture

The LSTM network we implement consists of:

- **Two LSTM layers** to capture long-term dependencies in sequential data.
- **Dropout layers** to reduce overfitting by preventing reliance on specific neurons.
- **A dense output layer** to transform learned features into final price predictions.

4.3.1 LSTM Layers

The model consists of **two stacked LSTM layers**, ensuring that both short-term and long-term dependencies are effectively learned.

- The **first LSTM layer** learns short-term patterns and forwards meaningful information to the next layer.
- The **second LSTM layer** further refines the learned patterns and strengthens long-term dependencies.

A single LSTM layer would likely fail to capture the complex patterns in highly volatile Bitcoin prices, while using more than two layers would **increase computational costs** without significant gains in performance.

For activation, we use **ReLU (Rectified Linear Unit)** as it improves learning efficiency and helps the model capture Bitcoin's **non-linear fluctuations** effectively.

4.3.2 Dropout Layers

Due to the high volatility of Bitcoin prices, our model includes **dropout layers** after each LSTM layer to **prevent overfitting**.

- **Dropout Rate: 20%**: This means that during training, 20% of the neurons are randomly deactivated.
- **Impact**: The model learns more robust general patterns rather than memorizing specific trends.

4.3.3 Dense Output Layer

The final **fully connected dense layer** is responsible for **converting learned time-series patterns into the final price prediction**.

- Since we are predicting only **one value per day**, a **single neuron** is sufficient.
- We do **not** use an activation function in the output layer because we require raw numerical predictions.

4.3.4 Model Compilation

The model is compiled using:

- **Adam Optimizer**: Adjusts learning rates dynamically for faster and stable convergence.
- **Mean Squared Error (MSE) Loss Function**: Penalizes large prediction errors more significantly.

Mean Squared Error (MSE) We use **MSE** as our loss function, defined as:

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

where:

- n = Total number of samples.
- y_i = Actual price.
- $\hat{y}_i$ = Predicted price.

MSE is preferred as it **penalizes larger errors more heavily**, ensuring that extreme fluctuations are properly considered.

4.3.5 Final Model Definition

Below is the implementation of our **LSTM architecture**:

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

model = Sequential([
    LSTM(50, activation='relu', return_sequences=True,
       input_shape=(lookback, 1)),
    Dropout(0.2),
    LSTM(50, activation='relu'),
    Dropout(0.2),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')

model.summary()
```

4.3.6 Model Summary

Our model consists of **30,651 trainable parameters**, which ensures a balance between **learning complexity** and **computational efficiency**.

```
Model: "sequential_1"

Layer (type)                  Output Shape            Param #
================================================================
lstm_2 (LSTM)                 (None, 30, 50)          10,400
dropout_2 (Dropout)           (None, 30, 50)          0
lstm_3 (LSTM)                 (None, 50)              20,200
dropout_3 (Dropout)           (None, 50)              0
dense_1 (Dense)               (None, 1)               51
================================================================
Total params: 30,651 (119.73 KB)
Trainable params: 30,651 (119.73 KB)
Non-trainable params: 0 (0.00 B)
```

Model Summary Explanation:

- **lstm_2**: The first LSTM layer with 50 units, processing sequences of length 30. It learns short-term dependencies.
- **dropout_2**: A dropout layer applied after the first LSTM to prevent overfitting.
- **lstm_3**: The second LSTM layer, refining patterns and capturing long-term dependencies.
- **dropout_3**: Another dropout layer to enhance generalization.
- **dense_1**: A fully connected layer that outputs the final prediction.

5 Model Training

Once the LSTM model architecture is defined, the next step is to train it using the preprocessed dataset. The training process involves feeding sequences of past Bitcoin price movements to the model and optimizing its parameters through backpropagation.

We train the model using **50 epochs**, striking a balance between learning efficiency and computational cost:

- **Too few epochs (e.g., 10-20)** may result in underfitting, meaning the model does not learn enough from the data.
- **Too many epochs (e.g., 100+)** could lead to overfitting, where the model memorizes the training data instead of generalizing well to unseen data.

The **batch size** is set to **16**, determining the number of samples processed before updating the model's weights. A smaller batch size provides more updates per epoch, improving learning, while a larger batch size enhances computational efficiency.

We also include **validation data** to evaluate the model on unseen test samples after each epoch. The **validation loss** serves as an indicator of how well the model generalizes to new data. If the validation loss starts increasing while the training loss continues to decrease, it could be a sign of overfitting.

Additionally, we record the **total training time** to analyze how long the model takes to train on our dataset. The training duration is measured by capturing the time before and after model training using Python's `time` module. This step is essential for evaluating the computational cost of training and assessing how the model scales with different input data sizes. Later, we compare the recorded training time across different historical window sizes to observe how training performance is affected by increased data volume.

The training process is executed using the following code:

```
history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=16,
    validation_data=(X_test, y_test),
    verbose=1
)
```

To monitor training progress, we log the **first two and last two epochs**:

```
Epoch 1/50
24/24  5s 49ms/step - loss: 0.2126 - val_loss: 0.0462
Epoch 2/50
24/24  1s 23ms/step - loss: 0.0350 - val_loss: 0.0186

...

Epoch 49/50
24/24  1s 24ms/step - loss: 0.0168 - val_loss: 0.0183
Epoch 50/50
24/24  1s 24ms/step - loss: 0.0193 - val_loss: 0.0183

Total Training Time: 56.24 seconds
```

Examining the logs, we observe that the **training loss** consistently decreases, indicating that the model is learning meaningful patterns in the data. Additionally, the **validation loss** stabilizes around **0.0183**, suggesting that the model generalizes well without significant overfitting.

6 Model Performance

To evaluate our model, we use several error metrics from the `sklearn` library. Since we are forecasting Bitcoin closing prices, we need metrics that accurately measure prediction deviations while handling cryptocurrency market volatility. The most suitable choices are:

6.1 Evaluation Metrics

6.1.1 Mean Absolute Error (MAE)

Formula:

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

- y_i: Actual Closing Price
- $\hat{y}_i$: Predicted Closing Price
- n: Number of predictions

MAE measures the average absolute difference between predicted and actual values. It is a straightforward accuracy metric and is particularly useful because it treats all deviations equally, making it robust to extreme fluctuations, which are common in the crypto market.

6.1.2 Root Mean Squared Error (RMSE)

Formula:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{2}$$

RMSE measures the standard deviation of prediction errors while penalizing larger deviations more than MAE. It is useful for detecting large forecasting errors in volatile Bitcoin prices. However, RMSE is more sensitive to outliers than MAE, so it is best used in combination with other metrics.

6.1.3 Mean Absolute Percentage Error (MAPE)

Formula:

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{3}$$

MAPE expresses prediction errors as a percentage of actual prices, making it scale-independent and useful for comparing forecasting accuracy across different price levels. However, it can be unreliable when prices approach zero, as small errors can lead to inflated percentage values.

6.2 Inference Time Measurement

To further analyze our model's efficiency, we measure the **inference time**, which represents how long the model takes to generate predictions. This is crucial for real-world applications, where rapid forecasting is essential for decision-making in volatile cryptocurrency markets.

The total inference time is calculated by recording the time before and after calling `model.predict(X_test)`, providing the overall duration required to predict closing prices for the test period. Additionally, we compute the **average time per prediction**, which helps assess the model's scalability when applied to larger datasets.

```
from sklearn.metrics import mean_absolute_error,
    root_mean_squared_error, mean_absolute_percentage_error
import time

start_time = time.time()
predictions_scaled = model.predict(X_test)

predicted_diff = scaler.inverse_transform(predictions_scaled)
```

```
actual_prices = test_data['close'].iloc[-10:].values
predicted_prices = actual_prices + predicted_diff.flatten()

end_time = time.time()
inference_time = end_time - start_time

mae = mean_absolute_error(actual_prices, predicted_prices)
rmse = root_mean_squared_error(actual_prices, predicted_prices)
mape = mean_absolute_percentage_error(actual_prices,
   predicted_prices)

print(f"MAE:␣{mae}")
print(f"RMSE:␣{rmse}")
print(f"MAPE:␣{mape}")

print(f"Total␣Inference␣Time:␣{inference_time:.4f}␣seconds")
print(f"Average␣Time␣per␣Prediction:␣{inference_time␣/␣len(X_test
   ):.6f}␣seconds")
```

Results:

```
MAE: 160.5609
RMSE: 160.8764
MAPE: 0.0041
Total Inference Time: 0.1971 seconds
Average Time per Prediction: 0.01971 seconds
```

6.3 Performance Evaluation

- **MAE = 160.56**: The model's predictions deviate by approximately $160.56 from the actual Bitcoin prices. This indicates a moderate prediction error, which is reasonable given Bitcoin's volatility.
- **RMSE = 160.87**: Since RMSE and MAE values are close, it suggests that large prediction errors are not dominating performance, making the model fairly consistent.
- **MAPE = 0.4%**: The model's error is only 0.4% relative to Bitcoin's actual price, meaning that while absolute errors seem high in dollar terms, they are relatively small compared to Bitcoin's overall price fluctuations.

6.4 Training Scalability Analysis: Impact of Data Size on Training Time

To understand how training time scales with increasing dataset size, we train the model on progressively larger subsets (20%, 40%, 60%, 80%, and 100%) and record the corresponding training durations.

```
train_sizes = [int(len(X_train) * frac) for frac in [0.2, 0.4,
   0.6, 0.8, 1.0]]
training_times = []
```

```
for size in train_sizes:
    start_time = time.time()
    model.fit(X_train[:size], y_train[:size], epochs=10,
       batch_size=16, verbose=0)
    end_time = time.time()
    training_times.append(end_time - start_time)

import matplotlib.pyplot as plt

plt.figure(figsize=(8,5))
plt.plot(train_sizes, training_times, marker='o', linestyle='-')
plt.xlabel("Training␣Data␣Size")
plt.ylabel("Training␣Time␣(seconds)")
plt.title("Training␣Time␣vs.␣Input␣Data␣Size")
plt.grid(True)
plt.show()
```

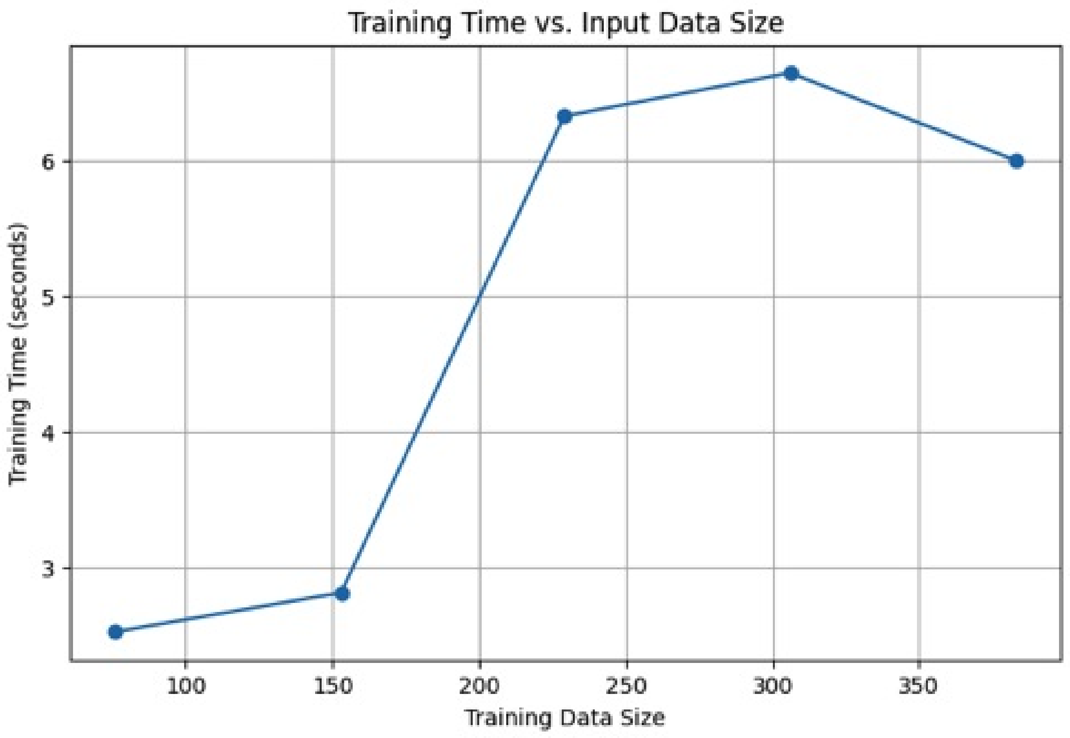

Figure 3: Training Time vs. Input Data Size

Figure Analysis:

- Initially, training time increases gradually.
- After approximately 200 samples, we observe a significant jump, indicating higher computational cost.
- Beyond 300 samples, training time fluctuates slightly, potentially due to hardware limitations or TensorFlow optimizations.
- This analysis highlights the importance of balancing dataset size with computational efficiency.

6.5 Predictions Visualization

To further analyze the model's accuracy, we visualize the actual vs. predicted Bitcoin closing prices for the test period (February 19-28, 2022). This allows us to assess how well the model captures price trends.

```
import matplotlib.pyplot as plt

plt.figure(figsize=(12,5))
plt.plot(test_data.index[-10:], test_data['close'].iloc[-10:].
   values, label="Actual␣Prices", marker='o', linestyle='-')
plt.plot(test_data.index[-10:], predicted_prices, label="
   Predicted␣Prices", marker='x', linestyle='dashed')

plt.legend()
plt.title("Bitcoin␣Price␣Prediction␣(Feb␣19,␣2022␣-␣Feb␣28,␣2022)
   ")
plt.xlabel("Date")
plt.ylabel("Price␣(USD)")
plt.grid(True)

plt.show()
```

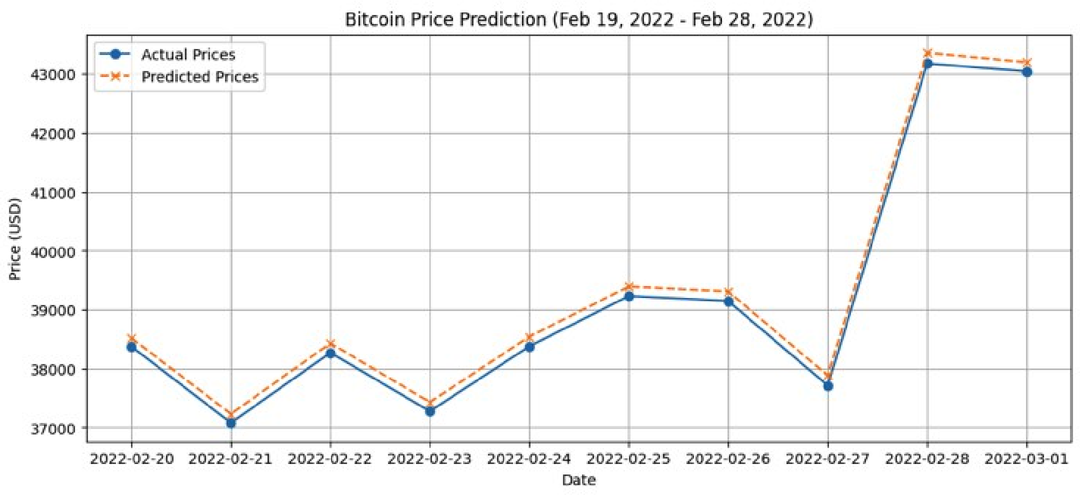

Figure 4: Bitcoin Price Prediction

Observations:

- The model's predictions closely align with actual prices, indicating that it captures trends well.
- A sharp price increase at the end of the test period is tracked, though with slightly higher deviations.
- Some minor deviations exist, as expected, since LSTMs cannot capture external market factors like regulatory news or investor sentiment.
- Overall, the model effectively follows Bitcoin price fluctuations, particularly during the February 19-27 period.

7 Comparing LSTM with ARIMA

To compare these two forecasting methods, we train an **ARIMA** model on our dataset and evaluate the predictions using the same error metrics as LSTM (**MAE, RMSE, MAPE**).

ARIMA (AutoRegressive Integrated Moving Average), as discussed earlier, belongs to traditional statistical models for time-series forecasting. It is commonly used for short-term predictions and assumes a linear relationship in the data.

7.1 ARIMA Training

The **ARIMA** method assumes Bitcoin prices follow a linear trend over time. Unlike LSTM, which can model complex **non-linear dependencies**, ARIMA relies solely on historical price values and their differences for making predictions.

Before training, we select the model parameters (p, d, q):

- **p**: Number of autoregressive (AR) terms.
- **d**: Number of differences applied to ensure stationarity.
- **q**: Number of moving average (MA) terms.

Additionally, to compare computational efficiency, we measure the total **training time** by tracking how long it takes for ARIMA to fit the dataset.

```
import time
from statsmodels.tsa.arima.model import ARIMA

p, d, q = 5, 1, 0

start_time = time.time()

arima_model = ARIMA(train_data['close'], order=(p, d, q))
arima_fit = arima_model.fit()

end_time = time.time()
arima_training_time = end_time - start_time

print(f"Total␣ARIMA␣Training␣Time:␣{arima_training_time:.2f}␣
   seconds")
```

```
Total ARIMA Training Time: 0.14 seconds
```

7.2 ARIMA Performance

Now, we use the trained **ARIMA model** to predict Bitcoin Closing Prices for our test period. Since ARIMA generates predictions sequentially based on past values, we record the **inference time**, tracking how long it takes to generate predictions for the full test period. This helps compare how quickly **ARIMA vs. LSTM** can make forecasts in real-world scenarios.

```
start_time = time.time()

arima_predictions = arima_fit.forecast(steps=len(test_data))

end_time = time.time()
arima_inference_time = end_time - start_time

print(f"Total␣ARIMA␣Inference␣Time:␣{arima_inference_time:.4f}␣
   seconds")
print(f"Average␣ARIMA␣Time␣per␣Prediction:␣{arima_inference_time␣
   /␣len(test_data):.6f}␣seconds")
```

```
Total ARIMA Inference Time: 0.0088 seconds
Average ARIMA Time per Prediction: 0.000804 seconds
```

7.3 ARIMA Model Performance Evaluation

We calculate the same error metrics (**MAE, RMSE, MAPE**) for ARIMA as we did for LSTM.

```
from sklearn.metrics import mean_absolute_error,
   root_mean_squared_error, mean_absolute_percentage_error

import numpy as np
arima_predictions = np.array(arima_predictions)

mae_arima = mean_absolute_error(test_data['close'],
   arima_predictions)
rmse_arima = root_mean_squared_error(test_data['close'],
   arima_predictions)
mape_arima = mean_absolute_percentage_error(test_data['close'],
   arima_predictions)

print(f"ARIMA␣MAE:␣{mae_arima}")
print(f"ARIMA␣RMSE:␣{rmse_arima}")
print(f"ARIMA␣MAPE:␣{mape_arima}")
```

ARIMA Model Performance:

```
ARIMA MAE: 1790.2788
ARIMA RMSE: 2109.4167
ARIMA MAPE: 0.0453
```

7.4 LSTM vs. ARIMA: Performance Comparison

Now that we have evaluated both models, we compare their **prediction accuracy and computational efficiency**. While **LSTM** uses deep learning to capture **non-linear dependencies**, **ARIMA** assumes a **linear** relationship in time-series data. By analyzing performance metrics and time efficiency, we determine which model is better suited for Bitcoin price forecasting.

A comparison table summarizes the differences:

Metric	LSTM	ARIMA
MAE	160.56	1790.28
RMSE	160.87	2109.42
MAPE	0.0040	0.0453
Training Time (s)	49.74	1.55
Inference Time (s)	0.1971	0.0100
Avg Time per Prediction (s)	0.0197	0.00091

Table 1: Performance Comparison: LSTM vs. ARIMA

7.5 LSTM vs. ARIMA: Visual Comparison

To visualize the differences, we plot the actual vs. predicted Bitcoin prices for both models:

```
plt.figure(figsize=(12,5))

plt.plot(test_data.index[-len(predicted_prices):], test_data['
   close'].iloc[-len(predicted_prices):],
         label="Actual␣Prices", color="blue", marker='o')

plt.plot(test_data.index[-len(predicted_prices):],
   predicted_prices,
         label="LSTM␣Predictions", linestyle='dashed', color="
            orange", marker='x')

plt.plot(test_data.index[-len(arima_predictions):],
   arima_predictions,
         label="ARIMA␣Predictions", linestyle='dashed', color="
            green", marker='s')

plt.legend()
plt.title("LSTM␣vs.␣ARIMA␣Bitcoin␣Price␣Predictions␣(Feb␣19␣-␣Feb
   ␣28,␣2022)")
plt.xlabel("Date")
plt.ylabel("Price␣(USD)")
plt.grid(True)
plt.show()
```

Absolute Error Analysis:

To further analyze the performance, we compare the absolute prediction errors for both models.

```
actual_prices_trimmed = test_data['close'].iloc[-len(
   predicted_prices):].values

lstm_errors = abs(actual_prices_trimmed - predicted_prices)
arima_errors = abs(actual_prices_trimmed - arima_predictions[:len
   (predicted_prices)])

plt.figure(figsize=(12,5))
```

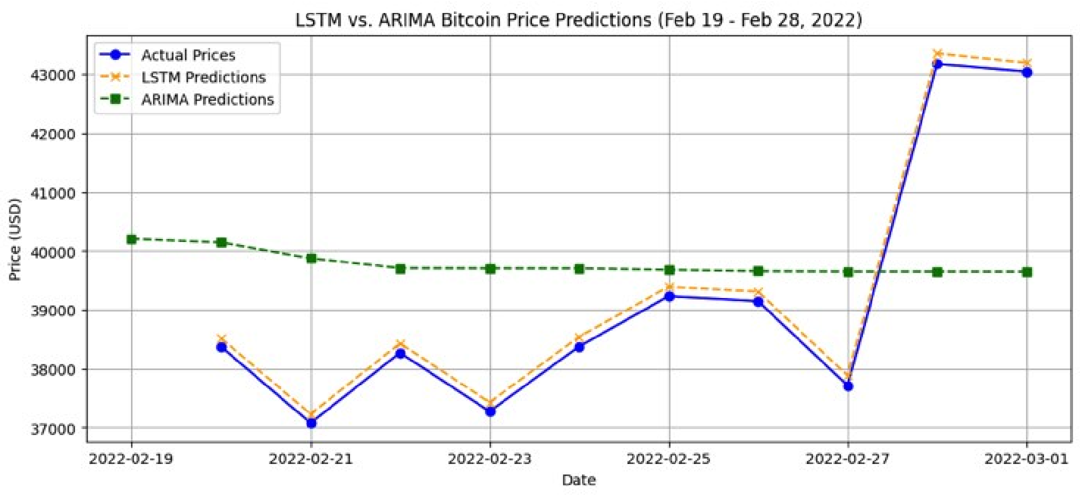

Figure 5: LSTM vs. ARIMA Bitcoin Price Predictions

```
plt.plot(test_data.index[-len(lstm_errors):], lstm_errors, label=
   "LSTM␣Absolute␣Error", color="orange", marker='x')

plt.plot(test_data.index[-len(arima_errors):], arima_errors,
   label="ARIMA␣Absolute␣Error", color="green", marker='s')

plt.legend()
plt.title("Absolute␣Errors:␣LSTM␣vs.␣ARIMA")
plt.xlabel("Date")
plt.ylabel("Absolute␣Error␣(USD)")
plt.grid(True)
plt.show()
```

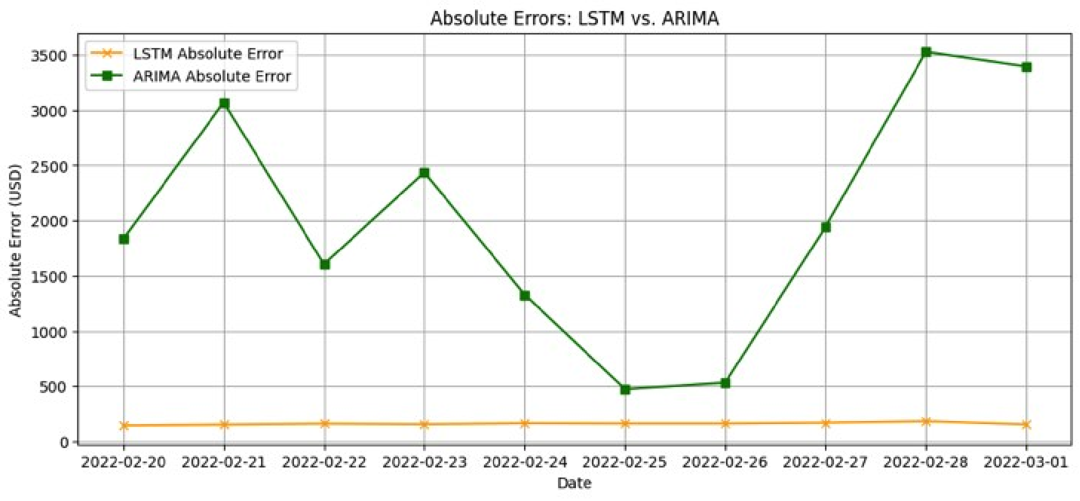

Figure 6: Absolute Errors: LSTM vs. ARIMA

8 Final Comparison and Conclusion

The results from our evaluation highlight significant differences in both **prediction accuracy** and **computational efficiency** between the two forecasting models: **LSTM and ARIMA**.

8.1 Performance Insights

The first plot comparing actual vs. predicted prices demonstrates that the **LSTM model** closely follows the actual Bitcoin price trends, with only minor deviations. In contrast, **ARIMA** predictions remain relatively static, failing to capture Bitcoin's volatile price movements.

This observation is further confirmed by the **absolute error plot**, where ARIMA exhibits significantly higher errors, reaching over **3,500 USD** on certain days, while LSTM maintains consistently lower errors, staying well below **200 USD** throughout the test period.

8.2 Quantitative Comparison

Examining the error metrics, LSTM significantly **outperforms ARIMA**, as shown below:

- **Mean Absolute Error (MAE):** LSTM: **160.56** vs. ARIMA: **1790.28**.
- **Root Mean Squared Error (RMSE):** LSTM: **160.87** vs. ARIMA: **2109.42**.
- **Mean Absolute Percentage Error (MAPE):** LSTM: **0.4%** vs. ARIMA: **4.53%**.

These metrics indicate that **deep learning models like LSTM are far better suited** for modeling the **complex, non-linear** nature of cryptocurrency price movements.

8.3 Computational Efficiency

Despite its superior predictive accuracy, LSTM comes at a higher **computational cost**. Comparing the training and inference times:

- **Training Time:** LSTM: **49.74 seconds** vs. ARIMA: **1.55 seconds**.
- **Inference Time:** LSTM: **0.1971 seconds** vs. ARIMA: **0.01 seconds**.

These results suggest that **ARIMA is more suitable for applications requiring fast forecasts with minimal computational resources**, such as real-time trading applications where speed is prioritized over accuracy.

8.4 Conclusion

Ultimately, LSTM emerges as the superior model for forecasting Bitcoin closing prices. While ARIMA provides faster training and inference, its inability to model non-linearity makes it **ineffective for predicting volatile assets like cryptocurrency**.

On the other hand, LSTM captures **trend fluctuations far more effectively**, making it the **preferred choice for accurate Bitcoin price forecasting**, despite its higher computational requirements. Future improvements may involve combining LSTM with feature engineering techniques such as **sentiment analysis, trading volume indicators, and macroeconomic factors** to further enhance forecasting accuracy.