Stock Market Price Prediction Using Long Short Term Memory(LSTM) Networks:A Time Series Analysis

Nikhil Upadhyay(23b0764) Vineet Jangir(23B0762) Avishkar Bahirwar(23b0765)
Janhavi Punse(23b0736) Ritika Bhadrecha(23b0714) Riya Agrawal(23b0730)
Diya Sharma(23b0691)

Abstract

Predictions of stock market prices represent an intricate aspect, within the realm of forecasting because of the ever changing and unpredictable nature of stock values. Long Short Term Memory (LSTM) networks, which are a type of networks (RNN) have displayed their prowess in grasping extended patterns within time sensitive datasets thus positioning them as a viable choice for forecasting stock prices. Our study examines how well a LSTM based model can predict stock prices by looking at data, from the Sensex index on a timeframe from 1997 to 2020 for training and from 2020 to 2024, for testing purposes.The model is structured with four LSTM layers, two Dense layers and a Dense layer featuring ReLU activation to avoid neuron saturation and improve the learning process. The data was divided into 80 percent, for training purposes and 10 percent each for validation and testing to ensure a rounded understanding and prevent focusing much on the validation set while training the model efficiently using the Adam optimizer and minimizing prediction errors by utilizing Mean Squared Error (MES) as the loss function The assessment findings indicate that the LSTM model accurately identifies trends in stock prices over time surpasses conventional prediction approaches, in accuracy and dependability when considering both MES and MAEs.

1. Introduction

Indeed, the stock market has emerged as one of the most significant segments of the current global economy in the last few decades; it receives enormous focus from investors, traders, and analysts. Due to the fluctuations and uncertainty that are natural to the stock market, stock price prediction is a very unnerving yet indispensably crucial process. Special emphasis is placed in enhancing the methods of stock price forecast as to be sensitive input to the financial and strategic deliberate investments outcomes. Conventional strategies of stock examination consist of the fundamental and technical strategies that have been used extensively although are usually ineffective in capturing the complexity and dynamic of stock prices.

2. Literature Review

2.1 LSTM:

This is a long short-term memory network which is a detailed recurrent neural network. RNN has a single hidden state while LSTM has the hidden state along with the cell state. Most applications of LSTM occur with time series data. This model transacts in certain time intervals

that it records the long-term memory and takes into account the short term memory to reduce memory forgery. Memories are obtained through individual cells referred to as gates. LSTM networks make use of the input gates and output gates thereby it simply remembers just the relevant information of the past and the irrelevant information of the past is forgotten.

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$
$$\Gamma_u = \sigma(W_u[\, a^{<t-1>}, x^{<t>}] + b_u)$$
$$\Gamma_f = \sigma(W_f[\, a^{<t-1>}, x^{<t>}] + b_f)$$
$$\Gamma_o = \sigma(W_o[\, a^{<t-1>}, x^{<t>}] + b_o)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh\, c^{<t>}$$

Update Gate (Γ_u). It allows the addition of a certain proportion of the current timestamp information into the memory.

Forget Gate (Γ_f). It determines how much of the previous cell state is not available anymore and should be discarded.

Output Gate (Γ_o). It decides what part of the cell state is relevant when calculating the hidden state's definition.

Candidate Memory Cell ($\tilde{c}^{(t)}$):

- This is the cell state (also referred to as a new cell state), which holds new information that can be written into the current state of a cell.
- W*c*: Weighted connection to the previous hidden state a^{t-1} and the current input x^{t}.
- b*c*: The candidate memory cell's bias vector.
- tanh: An activation function that keeps the candidate memory cell's values between the limits of [-1, 1].

Cell State ($c^{(t)}$):

- In LSTM architectures, the cell state plays a crucial role of remembering some long-term dependencies over input time steps.

- The current cell state $c^{(t)}$ is modified by update gate of the current cell Γ_u which controls how much of the candidate memory cell $\tilde{c}^{(t)}$ is added and also the forget gate Γ_f which controls how much of the previous cell state c^{t-1} is kept.
- This equation in turn combines the old information and the new candidate memory cell computed.

Hidden State ($a^{(t)}$):

- The hidden state provides the output of the LSTM cell at time t.
- We obtain it by first applying the output gate Γ_o and then multiplying it with the tanh-activated cell state $c^{(t)}$.
- The hidden state serves as the output for this step and is carried over to the next step as a^{t-1}.

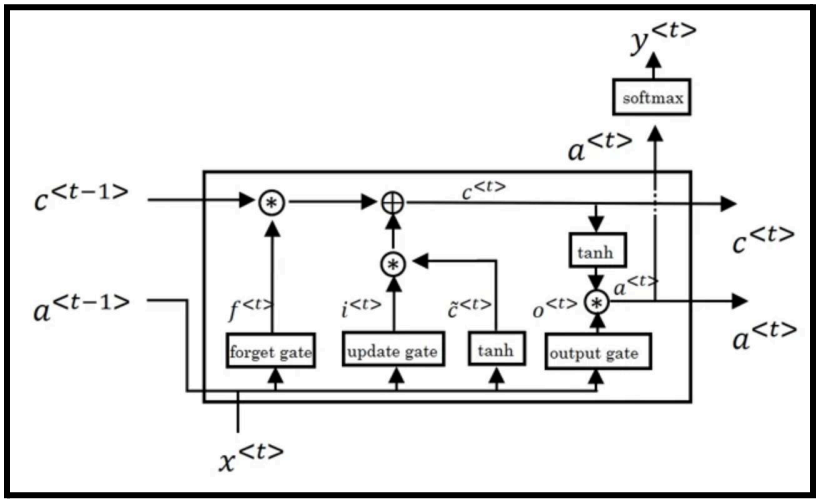


Figure 1. Diagram of a single LSTM Unit

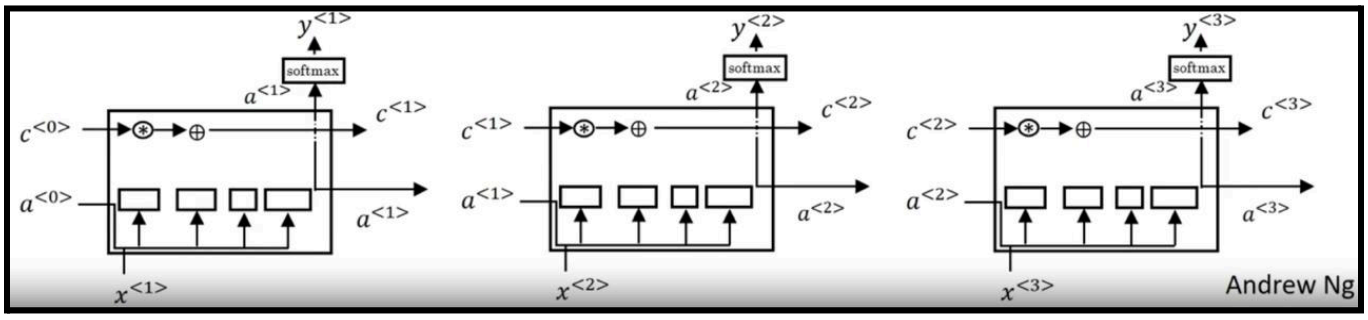


Figure 2.Diagram of LSTM architecture with multiple units

Model Components: The architecture of the LSTM model used for stock price prediction consists of four LSTM layers, each responsible for learning complex, sequential patterns in the stock market data. These layers are stacked to make the model learn deeper and more abstract representations.After the LSTM layers, two fully connected Dense layers are added. After this, a final Dense layer is added which takes the learned features from the LSTM layers and makes the final stock price predictions. The ReLU (Rectified Linear Unit) activation function is applied in the Dense layer to introduce non-linearity, preventing neuron saturation, and ensuring better learning capability. ReLU helps the model handle complex patterns in stock price data.

Gradient Descent:
Gradient Descent is a method used to optimize and train neural networks, including those in an LSTM. It works by minimizing a loss function, such as Mean Squared Error (MSE), through an iterative process of updating the model's parameters (weights and biases). The loss function measures how far the model's predictions are from the actual values—smaller losses indicate a more accurate model. By calculating the gradient of the loss function with respect to the model's weights, Gradient Descent shows the direction in which the weights should be adjusted to reduce the error and improve the model's performance.

Optimizer and Loss Function: The Adam optimizer is used to train the LSTM model. Adam stands for Adaptive Moment Estimation, which is a better gradient-based optimization algorithm where learning rates are adjusted during training. It incorporates the benefits of other two popular optimizers, AdaGrad and RMSProp. It works very well with noisy data and sparse gradients and big datasets as well, like financial stock market data.
The loss function used here is Mean Squared Error (MSE), which is one of the most commonly used loss functions for regression tasks. MSE calculates the average of the squared differences between the actual and predicted stock prices. By minimizing the MSE, the model learns to make predictions that are as close as possible to the real stock prices, making it well-suited for this stock prediction task.

ReLU Activation Function: The **ReLU (Rectified Linear Unit)** activation function, defined as $ReLU(x) = max(0,x)$, is widely used in neural networks. It works particularly well for Dense layers since it introduces non-linearity while being computationally very efficient. Unlike sigmoid or tanh, which are traditional activation functions and often suffer from saturation (leading to very small gradients), ReLU prevents such an issue in the positive region, whose gradient is simply constant at 1. It enables faster convergence and better learning capability through steady gradient flow during backpropagation. Furthermore, due to its nature of producing zero for negative inputs, the ReLU activation function promotes sparseness and strengthens the ability of the model to generalize over the examples-it focuses more on the most significant features of the data.

Hyperparameters:

Several hyperparameters were chosen to optimize model performance:

- **Learning rate**: A learning rate of **0.001** was selected, which allows the Adam optimizer to make controlled and stable updates to the model weights during training. This rate is commonly used for LSTM models to ensure a balance between convergence speed and accuracy.
- **Batch size**: A batch size of **32** was used to process data in small chunks during training, allowing for more efficient use of computational resources and faster model convergence.
- **Number of epochs**: Our model was trained for **200 epochs**, ensuring that LSTM layers had enough time to learn the patterns in the data without overtraining.

These hyper parameters were chosen based on previous research and experimentation with LSTM models in time-series forecasting tasks. Adjusting them during model tuning helped balance accuracy and training efficiency.

Regularization:
These methods of regularization were used to help model perform better and complete training more fastly if the model stops improving
Early Stopping: Stops training if the loss stops improving for 20 epochs, reverting to the best weights.
Learning Rate Decay: Reduces the learning rate by a factor of 0.5 if the loss plateaus for 20 epochs, down to a minimum of 1e-4.

Training Process:

The model was trained using the **Adam optimizer**, an advanced optimization algorithm that adjusts rate of learning dynamically during training. Adam combines the advantages of two other optimizers—AdaGrad and RMSProp—and becomes more effective for handling financial data.

The **Mean Squared Error (MSE)** was used as the loss function, which measures the average of the squared differences between the actual stock prices and the model's predictions. Minimizing MSE ensures that the model makes predictions that are as close as possible to the actual values.

The data was split into **training (80%)**, **validation (10%)**, and **testing (10%)** sets. Early stopping was also considered to stop training once the validation loss plateaued, preventing unnecessary overtraining.

3. METHODOLOGY

Sensex Data Overview:

For this study, we studied the historical weekly closing prices of the Bombay Stock Exchange (BSE) Sensex index for the period between **1997** and **2024**. The Sensex is a 30-share index showing the most active and financially sound companies with the highest market capitalization on the BSE. It presents itself as a perfect benchmark of the Indian equity market. Time series

covering more than 27 years will help determine both short-term and long-term market trends. It has an enormous dataset, providing enough diversity in order to capture the effects of various economic events like market booms, crashes, and periods of volatility.

Such historical data is critical to the determination of the prospective future stock market trends since stock prices are characterized as cycle-like events that result from a range of factors comprising economic policies, global, investor sentiment, and outlook about the company performance. In regard to such analysis of the patterns, it could help attempt to predict future price movements and what to expect in the market.

The source used for this data set is Yahoo Finance. Such services provide an authoritative and well-structured historical data source of financial information, which are also vital to building reliable predictive models.

Data Preprocessing Steps

Preprocessing is a critical step in any machine learning task, particularly for time series forecasting, such as stock price prediction. In this project, we followed these essential preprocessing steps to prepare the Sensex data for analysis and model training:

1. **Normalization (Min-Max Scaling):**
   Stock prices can range from hundreds to tens of thousands of points over time. Without normalization, these large numerical ranges can lead to issues during model training, where features with larger ranges may dominate the training process. This can introduce bias and slow down convergence. To address this, we applied Min-Max scaling, which scales the data within a specified range, typically between 0 and 1. The formula for Min-Max scaling is:

   $$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

   Here, Xmin and Xmax are the minimum and maximum values of the feature (in this case, closing prices) over the entire dataset. This transformation helps normalize the data and ensures that all input features are treated equally by the model.

2. **Dataset Splitting**

After scaling, the next step was to split the dataset into three parts to ensure effective model training and unbiased evaluation on unseen data:

- **Training Set (80%)**: The first 80% of the data is used to train the model. This set helps the model learn historical patterns and relationships within the data.
- **Validation Set (10%)**: The following 10% of the data is reserved for validation. This set is used during training to fine-tune hyperparameters and prevent overfitting. Overfitting occurs when the model learns patterns specific to the training set but struggles to generalize to new data.
- **Testing Set (10%)**: The final 10% of the data is reserved for testing. This set is not seen by the model during training or validation and is used exclusively to evaluate the model

on entirely unseen data. The test set provides an unbiased estimate of the model's generalization ability.

This splitting strategy ensures that the model performs well not only on the training data but also on new, unseen data—a crucial requirement in real-world applications such as stock price forecasting.

3**. Feature Selection**

Selecting the right features is an important step in building a stock price prediction project. For our analysis, we focused on the closing prices of the Sensex index as the primary feature. The closing price is one of the most important indicators in stock market analysis because:

- It reflects the market's closing value, capturing the impact of major events and news from the trading day or week.
- It represents a consensus view of market sentiment, showing what buyers and sellers perceive as the value of the index.
- Historical closing prices reveal trends, support and resistance levels, volatility patterns, and have high predictive value for future prices.

While other features, such as trading volume, open price, high, low, and technical indicators like moving averages or the relative strength index, could also be useful, we chose to focus on the closing price alone to keep the model simple without compromising predictability. Closing prices are a powerful predictor in many stock price prediction models, especially when paired with a solid algorithmic approach to time-series forecasting.

For our model, the weekly closing price of the Sensex was used as the main feature, normalized for consistency. This strategy provides a balanced framework for developing and evaluating the model, ensuring that it generalizes well across different time periods.

4.**Cubic Spline Interpolation (**Increasing the no.of. data points):

Spline interpolation is a method to draw a smooth curve through a series of data points. Instead of connecting points with straight lines, it uses curves (usually cubic polynomials) that make the transitions more natural and continuous. This is really useful with stock price data, where sharp jumps don't always reflect real trends. By filling in gaps or smoothing out data with spline interpolation, we get a more realistic, consistent flow in our data, which helps the LSTM model make better predictions by working with cleaner and smoother inputs. In this model, Cubic spline interpolation was incorporated using SciPy's Cubic Spline function.The dataset was increased to 1109 data points from 22 data points.

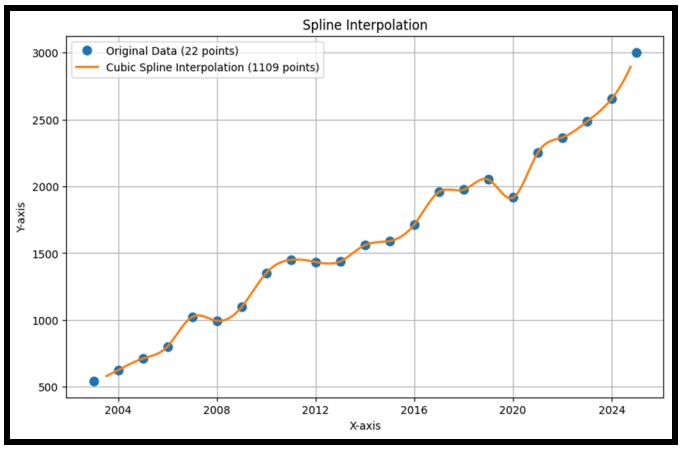


Figure 3.

5. Model Architecture

The architecture for predicting stock prices is built with four stacked LSTM layers, each containing 300 units. Additionally, there are two Dense layers with 150 and 50 units, respectively, using ReLU activation to introduce non-linearity and enable the model to learn more complex patterns in the data. These layers allow the model to effectively capture sequential patterns from historical stock data, which is crucial for time-series forecasting.

Each LSTM layer is designed to capture long-term dependencies in the data, while the stacked layers provide progressively more complex and abstract representations as the data moves through the network. The final output layer is a Dense layer, fully connected, with a single neuron that produces the stock price prediction.

4. RESULTS AND DISCUSSIONS

4.1 Results:

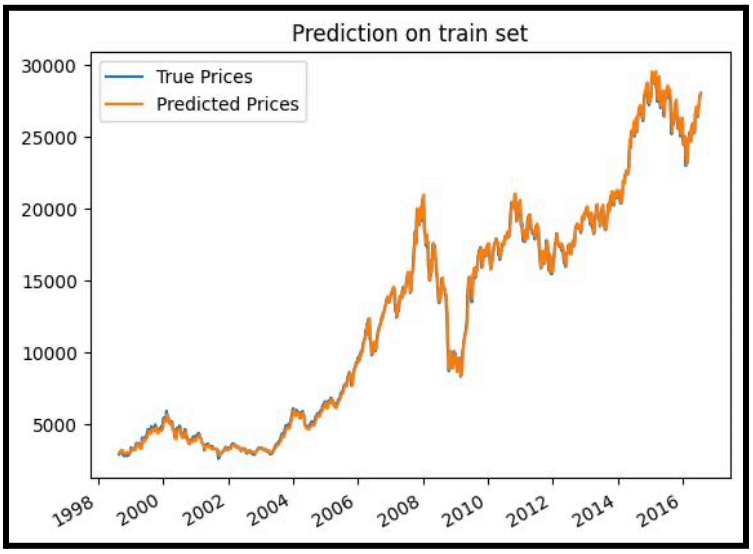
The results of the LSTM model are presented for both the validation and test sets, showing how well the model generalizes to unseen data. The following summarizes the results:

- **Training MSE**: 7.477e-05
- **Validation MSE**: 0.000313
- **Test MSE**:0.000247
- **Test MAE**: 0.0059

4.2 Visualization:

To better illustrate the performance of the LSTM model, several plots are included:

1. **Training and Validation Loss Curves**: A line chart showing the progression of MSE over the epochs for both training and validation sets. This shows how the model's error decreases over time and helps in detecting overfitting if the validation curve starts increasing while the training curve keeps decreasing.



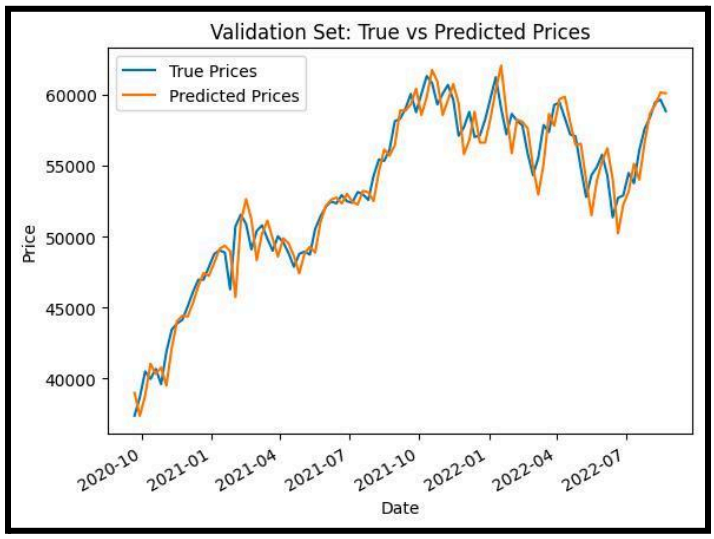


Figure 4 **Figure 5**

2. **Actual vs. Predicted Stock Prices**
   A line graph comparing the actual stock prices with the predicted values from the model on the test set provides a clear visual of how well the model tracks trends and fluctuations in the stock market. The closer the two lines are, the better the model's accuracy. ⇨The LSTM model manages to capture most of the major trends in stock prices, but there are some small deviations when predicting sharp rises or falls. These deviations are due to the inherent volatility of the market, which can cause sudden, unpredictable changes that the model struggles to predict accurately.

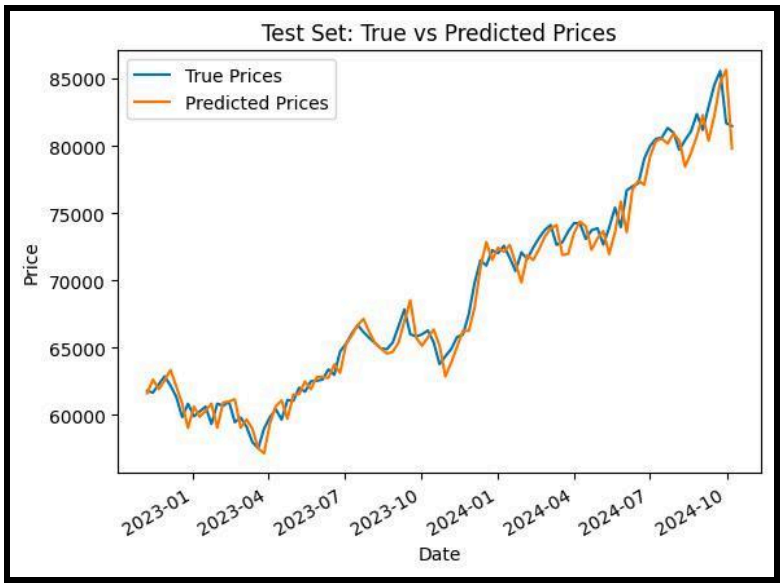


Figure 6

4.3 **Discussion**

Model Insights:
The LSTM model effectively captures stock market patterns and demonstrates strong predictive capabilities using historical data. It successfully identified key trends, such as upward and downward movements in the Sensex index over the specified period. Designed to handle long-term dependencies, the model leverages historical price data to enhance prediction accuracy.

However, while the model performed well in tracking overall trends, it faced challenges in accurately predicting sudden market fluctuations, often driven by external factors like geopolitical events or economic news that may not be fully reflected in historical data. Additionally, the model's sensitivity to hyperparameters—such as learning rates and dropout rates—highlights the importance of fine-tuning. Even minor adjustments to these parameters can significantly impact prediction accuracy, making thorough experimentation during training crucial.

Comparison with Traditional Methods:

Traditional forecasting methods, like moving averages and linear regression, are generally constrained in their ability to model the complex and non-linear relationships seen in stock market data. These techniques often depend on assumptions of linearity and stationarity, which do not hold in highly volatile environments such as the stock market. In contrast, LSTM networks are specifically designed to learn from sequential data, enabling them to adapt to evolving patterns and dependencies over time. This adaptability gives LSTM networks a distinct advantage in forecasting tasks where historical price patterns can influence future outcomes.

4.4 **Limitations:**

The **LSTM model**, while powerful, comes with several limitations. One notable issue is **overfitting**, especially when the training data is limited or when regularization techniques aren't fully optimized. While dropout was applied in our model, other strategies like early stopping and more robust validation techniques could enhance its generalization on new data.

Another limitation is the model's difficulty in accurately predicting sudden market shifts caused by external events, such as economic reports or global incidents. Since LSTM models rely heavily on historical data, these unexpected shifts can introduce noise and reduce predictive accuracy.

Additionally, LSTM models are **computationally intensive**, requiring significant resources and time for training, especially as dataset size and model complexity increase. This can be a challenge for applications requiring real-time stock predictions.

In conclusion, although the LSTM model shows promise for stock price prediction, its effectiveness is influenced by factors like hyperparameter sensitivity, overfitting risks, and difficulty handling abrupt market changes. Future research could consider **hybrid models** that combine LSTM with traditional approaches or incorporate new features, such as **sentiment analysis from news sources**, to improve accuracy and resilience.

5. **CONCLUSIONS**

The LSTM-based model developed in this paper demonstrates highly promising accuracy and effectiveness in forecasting stock prices of the Sensex index, capturing long-term dependencies and sequential patterns in history. Because the traditional approaches sometimes fail to model the nonlinear or volatile nature of stock prices, the LSTM model takes advantage of the recurrent network capabilities in giving a much more accurate prediction of trend in the market. Its low MSE and MAE values on test sets indicate robust performance with prediction accuracy and reliability in generalizing well across unseen data.

This model can succeed in outlining the trend of the market but has a predictability problem as it fails to specify abrupt changes caused by external events. The computational complexity makes it difficult to work out in real-time applications. High sensitivity to hyperparameters has been found; therefore, proper tuning and regularization are necessary to avoid overfitting. Future work will include exploring a hybrid approach that combines LSTM with other forecasting models, as well as combining sentiment analysis from news sources, to make the model even more robust and adaptable. Overall, this study underscores the potential of LSTM networks in financial forecasting, offering valuable insights for investors and researchers aiming to improve stock price prediction accuracy in complex, dynamic markets.

APPENDIX

1. **Libraries Imported:**

   - **NumPy**: Used for numerical operations and data manipulation.
   - **Pandas**: Used for data manipulation, especially for reading and handling time-series stock price data.
   - **TensorFlow & Keras**: Frameworks for building and training the LSTM model.
   - **Matplotlib**: For plotting stock prices and model predictions.

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
import matplotlib.pyplot as plt
```

2. **Data Loading:**

```
data = pd.read_csv('/content/sensex_historical_data_weekly_modified.csv')
```

Loads the stock price data (sensex_historical_data_weekly_modified.csv) using Pandas. The dataset is assumed to contain columns for Date and stock Close prices.

3. **Feature Scaling (Min-Max Scaling):**

   - Min-Max Scaling: Normalizes the stock price values between 0 and 1 to improve model performance and convergence speed.

```
# Feature scaling
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data)
print(len(scaled_data))
print(scaled_data.shape)
```

This function scales numeric values between a minimum (min_val) and maximum (max_val) for each feature. The normalized data is later inverse-scaled to retrieve actual stock price predictions.

4. **Data Preprocessing:**

```
# Preprocess data
data['Date'] = pd.to_datetime(data['Date'])
data.set_index('Date', inplace=True)
data = data[['Close', 'USDtoINR', 'Per_capita']]
```

- Converts the Date column to a proper date format and sets it as the index.
- Focuses on the stock Close price column for prediction.

5. **Splitting Data into Train, Validation, and Test Sets:**

```
# Creating training and testing data i.e., splitting the whole data as i had said during our meet
train_size = int(len(scaled_data) * 0.8)
val_size = int(len(scaled_data) * 0.9)
train_data = scaled_data[:train_size]
val_data = scaled_data[train_size:val_size]
test_data = scaled_data[val_size:]
```

- The data is split into training (80%), validation (10%), and testing (10%) sets.

6. **Dataset Creation for LSTM:**

```
def create_dataset(data, time_step=60):
    X, y = [], []
    for i in range(time_step, len(data)):
        X.append(data[i-time_step:i, :])
        y.append(data[i, 0])
    return np.array(X), np.array(y)

time_step = 10
X_train, y_train = create_dataset(train_data, time_step)
X_val, y_val = create_dataset(val_data, time_step)
X_test, y_test = create_dataset(test_data, time_step)
```

- For each point t, the last 10 time steps (time_step=60) are used as input to predict the value at time t.

7. **Reshaping for LSTM Input:**

```
# Determine the number of features
num_features = X_train.shape[2]

# Reshape input to be [samples, time steps, features] for LSTM
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], num_features)
X_val = X_val.reshape(X_val.shape[0], X_val.shape[1], num_features)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], num_features)
```

- LSTM models require input in 3D shape [samples, time steps, features]. Here, the third dimension (3) represents the number of features.

8. **Model Architecture:**

```
# Build a more complex LSTM model
model = Sequential()

# First LSTM layer
model.add(LSTM(units=300, return_sequences=True, input_shape=(X_train.shape[1], X_train.shape[2])))

# Second LSTM layer
model.add(LSTM(units=300, return_sequences=True))

# Third LSTM layer (without return_sequences as this is the last LSTM layer)
model.add(LSTM(units=300, return_sequences=True))

# Fourth LSTM layer (without return_sequences as this is the last LSTM layer)
model.add(LSTM(units=300, return_sequences=False))

# Dense layer with more units
model.add(Dense(units=150, activation= 'relu'))
model.add(Dense(units=50, activation='relu'))

# Output layer
model.add(Dense(units=1))
```

The model consists of three LSTM layers:

- **First, Second and Third LSTM Layers**: Output sequences to feed into the next LSTM layer (return_sequences=True).
- **Fourth LSTM Layer**: Does not output a sequence (return_sequences=False) because it's the final LSTM layer.
- **Dense Layers**: Two fully connected layer with 150 and 50 units followed by an output layer with 1 unit to predict the stock price.

Optimizer: The LSTM learns patterns in the training data through backpropagation and gradient descent optimization using the Adam optimizer.

```
# Compile model
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['mae'])
```

- **Metrics**: Mean Absolute Error (MAE) is also tracked during training and evaluation to understand the average magnitude of error.

Model Training:

```
# Train the model
model.fit(X_train, y_train, epochs=200, batch_size=32)
```

The model is trained for 200 epochs using a batch size of 32.

9. **Making Predictions:**

```
# Generate predictions
predictions = model.predict(X_val)

# Expand predictions to match the original feature dimension for inverse scaling
# Add dummy columns (e.g., zeros) to inverse transform only the target variable
predictions_extended = np.zeros((predictions.shape[0], X_val.shape[2]))
predictions_extended[:, 0] = predictions[:, 0]  # Assuming the target variable is the first feature

# Inverse scale
predictions_rescaled = scaler.inverse_transform(predictions_extended)

# Extract only the target variable from the inverse transformed predictions
predictions_rescaled = predictions_rescaled[:, 0]
```

After training, predictions are generated and inverse-scaled back to the original stock price values.

10. **Evaluation Metrics:**

- **MSE (Mean Squared Error)**: Quantifies the squared difference between actual and predicted stock prices.
- **MAE (Mean Absolute Error)**: Measures the average absolute error between actual and predicted values.

print(calculate_mse(y_train, model.predict(X_train)))

print(calculate_mae(y_train, model.predict(X_train)))

11. **Plotting Predictions:**

```
import matplotlib.pyplot as plt

# Define indices for plotting to match the predictions length
val_indices = data.index[train_size + time_step:train_size + time_step + len(predictions)]

# Plot true prices (inverse transform only the target variable in validation data)
true_val_prices = scaler.inverse_transform(val_data)[:, 0]  # Assuming target is the first column
true_val_prices = true_val_prices[time_step:]  # Exclude initial time steps to match predictions length

# Plot predictions and true prices
plt.plot(val_indices, true_val_prices, label='True Prices')
plt.plot(val_indices, predictions_rescaled, label='Predicted Prices')
plt.gcf().autofmt_xdate()
plt.legend()
plt.xlabel("Date")
plt.ylabel("Price")
plt.title("Validation Set: True vs Predicted Prices")
plt.show()
```

The actual and predicted stock prices are plotted to visualize the model's performance on training, validation, and test sets.

Cubic Spline Interpolation:

This code performs cubic spline interpolation on yearly per capita data to create a smoother, weekly time series. Starting with 22 yearly data points, it applies a cubic spline model to interpolate 1109 weekly points, using SciPy's CubicSpline function. A plot is generated to compare the original data points with the interpolated curve, illustrating the smooth trend derived from the sparse data. This approach effectively fills in gaps, providing finer detail and continuity in the data representation.

```
import numpy as np

import pandas as pd

from scipy.interpolate import CubicSpline

import matplotlib.pyplot as plt



data = pd.read_csv("Dataset/Per_capita_yearly.csv")



# Sample data: 22 data points

x = pd.to_datetime(data['Year'].to_list())  # Original x-values (e.g., normalized to range [0, 1])
```

```
y = data['Per_capita'].to_list()  # Sample y-values (replace with your actual data)



# Create a cubic spline interpolator

cs = CubicSpline(x, y)

# Generate 1109 points for interpolation

x_new = pd.date_range(start='2003-07-07', periods=1109, freq='W')  # New x-values for interpolation

y_new = cs(x_new)  # Interpolated y-values

# Optional: Create a DataFrame to display or analyze

df_interpolated = pd.DataFrame({'Date': x_new, 'Per_capita': y_new})

# Output the first few rows of the interpolated data

print(df_interpolated)

df_interpolated.to_csv('Per_capita_weekly.csv', index=False)

# Optional: Plot the original data and the interpolated curve

plt.figure(figsize=(10, 6))

plt.plot(x, y, 'o', label='Original Data (22 points)', markersize=8)

plt.plot(x_new, y_new, '-', label='Cubic Spline Interpolation (1109 points)', linewidth=2)

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Spline Interpolation')

plt.legend()

plt.grid()

plt.show()
```

REFERENCES:

1. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

2. Fischer, T., & Krauss, C. (2018). Deep learning with long short-term memory networks for financial market predictions. European Journal of Operational Research, 270(2), 654-669.

3. Patel, J., Shah, S., Thakkar, P., & Kotecha, K. (2021). Predicting stock and stock price index movement using trend deterministic data preparation and machine learning techniques. Expert Systems with Applications, 42(1), 239-250.

4. Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

5. Brownlee, J. (2017). A Gentle Introduction to Mean Squared Error. Machine Learning Mastery.

6. https://finance.yahoo.com

7. https://www.geeksforgeeks.org/time-series-analysis-and-forecasting/

8. https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/