LSTM Stock Price Prediction Project

Neil Slavishak

May 12, 2025

Abstract

This project explores the use of Long Short-Term Memory (LSTM) neural networks for predicting the stock prices of various companies. Using historical daily closing prices from January 2020 to the present, we trained an LSTM model to learn temporal patterns and forecast future stock movements. The data was normalized and split using a rolling window of 30-day sequences, with an 80/20 train-test split. The model was evaluated using root mean squared error (RMSE), demonstrating moderate predictive accuracy in capturing general price trends. While the model effectively follows broader market movements, it tends to underrepresent short-term volatility. A one-month forecast was generated and visualized alongside recent historical data to assess continuity and predictive performance. Limitations include reliance on a single feature (closing price) and the exclusion of macroeconomic indicators or news sentiment. The project highlights the potential and challenges of using deep learning for financial time series forecasting and suggests avenues for improvement through feature expansion and model refinement.

Model Summary (Methods)

To build the LSTM model to predict the moving of the stock prices for these companies, we used PyTorch. PyTorch is an open-source deep learning framework developed by Meta AI that is widely used for building and training neural networks. It features dynamic computation graphs and automatic differentiation, making model development flexible and intuitive. We used PyTorch for this project because it enables the efficient creation, training, and evaluation of the LSTM model used to learn and predict stock price patterns over time. In this section, we break down the code used to build the model.

Importing Libraries

The following code chunk illustrates the libraries used in this project.

```python
#Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import yfinance as yf

import torch
import torch.nn as nn
import torch.optim as optim

from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from math import sqrt
```

This code imports all of the data wrangling and visualization libraries, mathematics libraries, and yfinance for easy data acquisition. It also imports the necessary PyTorch libraries needed for neural network modules and optimization. Finally, it imports StandardScaler to later transform the data, as well as mean squared error to find the root mean squared error to determine the accuracy of the training.

Setting the Device and Loading the Data

```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu
    ')

ticker = 'INTC'
df = yf.download(ticker, '2020-01-01')

scaler = StandardScaler()
df['Close'] = scaler.fit_transform(df[['Close']])
```

This code chunk sets the device to either CUDA, if possible, or the CPU. It then loads in the stock price data using the ticker symbol and then standardizes the data.

Setting Up the LSTM Model

```python
seq_length = 30
data = []

for i in range(len(df) - seq_length):
    data.append(df.Close[i:i+seq_length])

data = np.array(data)
train_size = int(0.8 * len(data))

X_train = torch.from_numpy(data[:train_size, :-1, :]).type(torch.
    Tensor).to(device)
```

```
11 y_train = torch.from_numpy(data[:train_size, -1, :]).type(torch.
       Tensor).to(device)
12 X_test = torch.from_numpy(data[train_size:, :-1, :]).type(torch.
       Tensor).to(device)
13 y_test = torch.from_numpy(data[train_size:, -1, :]).type(torch.
       Tensor).to(device)
```

The code above sets up the data to be used in the LSTM model. Lines 1-5 create the data “windows” used in the LSTM model, where groups of 30 data points are taken from the beginning of the data frame to the end, overlapping so each data point from the beginning to 30 points away from the end starts a window. In LSTM models, the windows are used to predict the next data point after the window, allowing for the creation of continuous data points that mock the actual data. The remaining lines set up the testing and training data, using the common 80/20 split.

Defining the LSTM Model

```
1  class PredictionModel(nn.Module):
2
3      def __init__(self, input_dim, hidden_dim, num_layers,
       output_dim):
4          super(PredictionModel, self).__init__()
5
6          self.num_layers = num_layers
7          self.hidden_dim = hidden_dim
8
9          self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers,
       batch_first=True)
10         self.fc = nn.Linear(hidden_dim, output_dim, bias=True)
11
12     def forward(self, x):
13         h0 = torch.zeros(self.num_layers, x.size(0), self.
       hidden_dim, device=device)
14         c0 = torch.zeros(self.num_layers, x.size(0), self.
       hidden_dim, device=device)
15
16         out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))
17         out = self.fc(out[:, -1, :])
18
19         return out
```

Here, we define the LSTM model. The init method defines the structure. First, it initializes the model and inherits from nn.Module. Then, it stores the number of layers and the hidden units. On line 6, we define an LSTM layer that takes in all of the parameters needed for the LSTM model. Line 7 then creates a fully connected layer that maps the final hidden state to the prediction output.

The forward method defines how input data flows through the model. It first initializes the hidden state and the cell state with zeroes, this allows for these to be passed through the LSTM model with the input. Finally, the last output is fed through the fully connected layer and returned.

Training the Model

```python
1  model = PredictionModel(input_dim=1, hidden_dim=64, num_layers=3,
       output_dim=1).to(device)
2  criterion = nn.MSELoss()
3  optimizer = optim.Adam(model.parameters(), lr=.01)
4
5  num_epochs = 200
6
7  for i in range(num_epochs):
8      y_train_pred = model(X_train)
9      loss = criterion(y_train_pred, y_train)
10
11     optimizer.zero_grad()
12     loss.backward()
13     optimizer.step()
```

This code chunk first creates an instance of the LSTM model with chosen parameters. It also sets up the criterion, which in this case is the mean squared error loss, and the Adam optimizer, which allows for adaptive learning rates. Finally, we chose the number of epochs to be 200. Lines 7-13 loop through the epochs, in which a prediction is made and the loss between the predicted and the actual data is calculated. Afterwards, the gradients are reset, calculated, and then used to update the weights of the model.

Evaluating the Model

```python
model.eval()
y_test_pred = model(X_test)

y_train_pred = scaler.inverse_transform(y_train_pred.detach().cpu()
    .numpy())
y_train = scaler.inverse_transform(y_train.detach().cpu().numpy())
y_test_pred = scaler.inverse_transform(y_test_pred.detach().cpu().
    numpy())
y_test = scaler.inverse_transform(y_test.detach().cpu().numpy())

test_rmse = sqrt(mean_squared_error(y_test[:, 0], y_test_pred[:,
    0]))
```

This code chunk evaluates the model. First, it uses the test set to produce predicted values using the model. Afterwards, the data is then scaled back to its true values, allowing for easier interpretation. Then, the root mean squared error between the true test data and the predicted test data is calculated to determine how effective the model was in predicting the preexisting data.

Forecasting Future Data

```python
forecast_days = 30
last_seq = data[-1]
forecast = []

```

```
5  current_seq = torch.tensor(last_seq, dtype=torch.float32).unsqueeze(0).to(device)
6
7  model.eval()
8  with torch.no_grad():
9      for _ in range(forecast_days):
10         next_val = model(current_seq)
11         forecast.append(next_val.item())
12
13         next_input = next_val.unsqueeze(1)
14         current_seq = torch.cat((current_seq[:, 1:, :], next_input), dim=1)
15
16 forecast = scaler.inverse_transform(np.array(forecast).reshape(-1, 1))
```

This code chunk uses the model to forecast the next 30 days of the stock price. Lines 1-5 set the amount of days to forecast, pick the last sequence from the list of sequences stored in the data array, and then converts the sequence to a PyTorch tensor. Then, lines 8-14 loop through the amount of days and makes a prediction for each day using the model, then updates the sequence being used by adding the next prediction to the end of the sequence. Finally, similar to before, the data is scaled back to its true values.

Plotting Results

```
#Plot the predicitions and error
fig = plt.figure(figsize=(10,8))
gs = fig.add_gridspec(4,1)

ax1 = fig.add_subplot(gs[:3, 0])
ax1.plot(df.iloc[-len(y_test):].index, y_test, color = ’blue’, label = ’Actual Price’)
ax1.plot(df.iloc[-len(y_test):].index, y_test_pred, color = ’green’, label = ’Predicted Price’)
ax1.legend()
plt.title(f"{ticker} Stock Price Prediction")
plt.xlabel(’Date’)
plt.ylabel(’Price’)

ax2 = fig.add_subplot(gs[3,0])
ax2.axhline(test_rmse, color = ’blue’, linestyle = ’--’, label = ’RMSE’)
ax2.plot(df[-len(y_test):].index, abs(y_test-y_test_pred), color = ’red’, label = ’Prediction Error’)
ax2.legend()
plt.title(’Prediction Error’)
plt.xlabel(’Date’)
plt.ylabel(’Error’)

plt.tight_layout()
plt.show()

#Plot the forecast
last_date = df.index[-1]
```

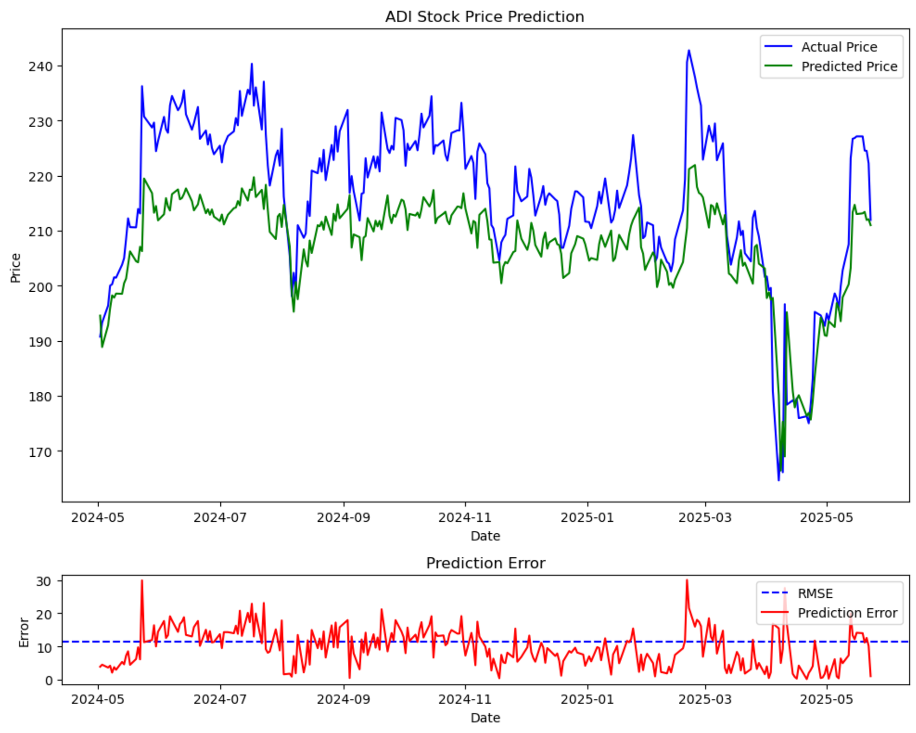
```
future_dates = pd.bdate_range(start=last_date, periods=len(forecast))
forecast_df = pd.DataFrame(forecast, index=future_dates, columns=['Forecast'])
six_months_ago = df.index[-1] - pd.DateOffset(months=6)
recent_df = df[df.index >= six_months_ago]

plt.figure(figsize=(12, 6))
plt.plot(recent_df.index, scaler.inverse_transform(recent_df[['Close']]), label='Actual Prices')
plt.plot(forecast_df.index, forecast_df['Forecast'], label='Forecast', color='green')
plt.axvline(x=last_date, linestyle='--', color='gray', label='Forecast Start')
plt.title(f'{ticker} Forecast')
plt.xlabel('Date')
plt.ylabel('Price')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

The first group of code produces a figure that includes the true data and the predicted data on the same graph, while also illustrating the prediction error of each point in relation to the root mean squared error. The second group of code produces a plot that illustrates the last 6 months of the true data and then expands on it with the forecast.

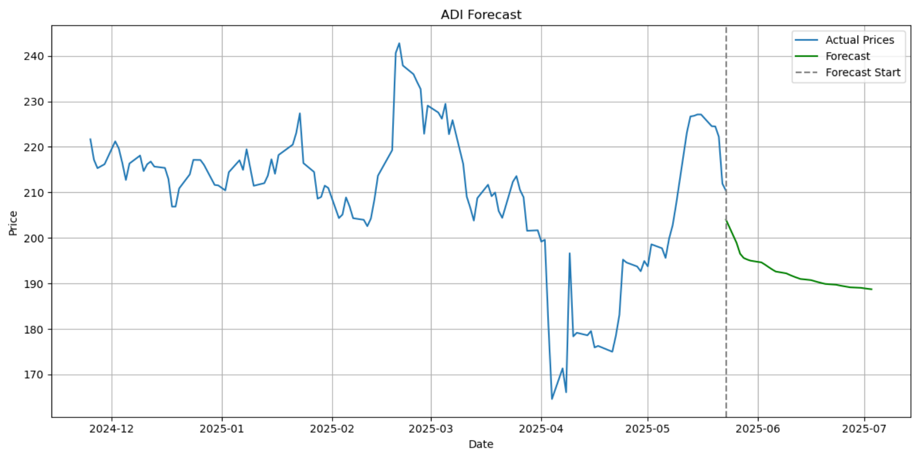
Results

To examine the results of using this model for forecasting, we chose the companies Analog Devices Inc. (ADI), Starbucks Corp (SBUX), and Apple Inc. (AAPL). Each forecast illustrates the trend for the next month of stock prices barring any outside influence.

Analog Devices Inc. (ADI)

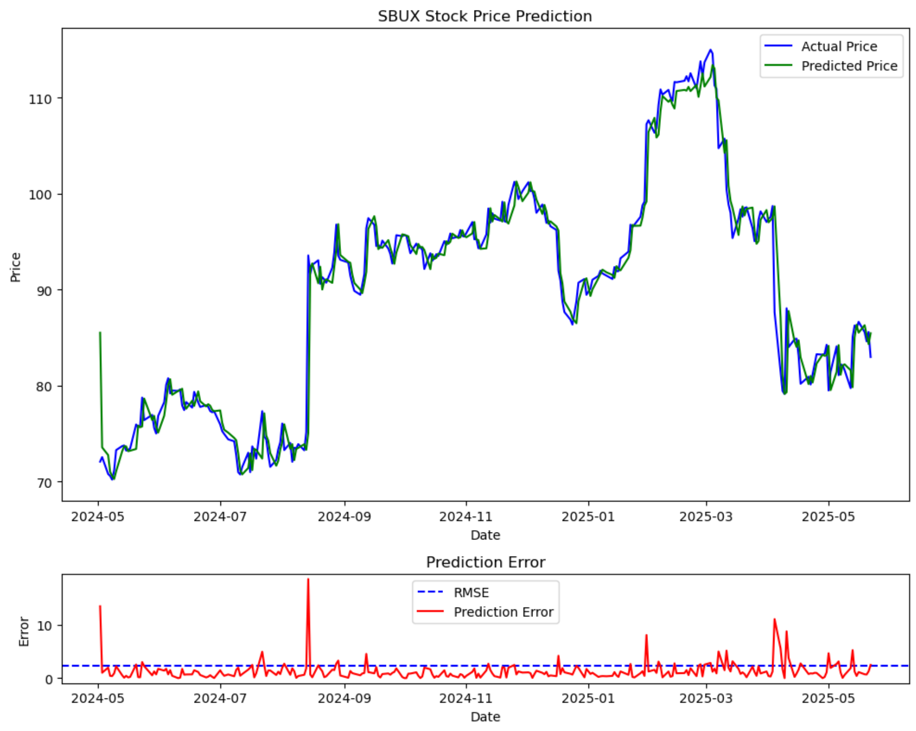


The above image shows the test data compared to the true data, as well as the prediction error for each point. It seems that the model was able to capture the movement of the data, however not the volatility. The root mean squared error came to be 11.36. Using this, we implied that the trends of the data were properly captured, allowing the model to be used to forecast.

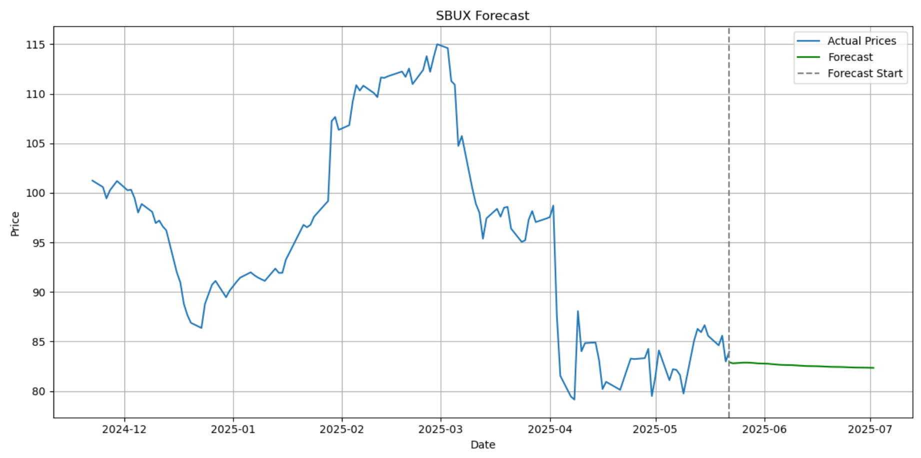


The forecast illustrates a slow decline in the stock price, implying that the stock might continue to drop in the next month. Comparing it to the peak found around March of 2025, this is a reasonable estimate for the future stock price. However, the smoothness of the graph makes it less reasonable for an exact estimate and more reasonable as solely an illustration of the trend.

Starbucks Corp (SBUX)

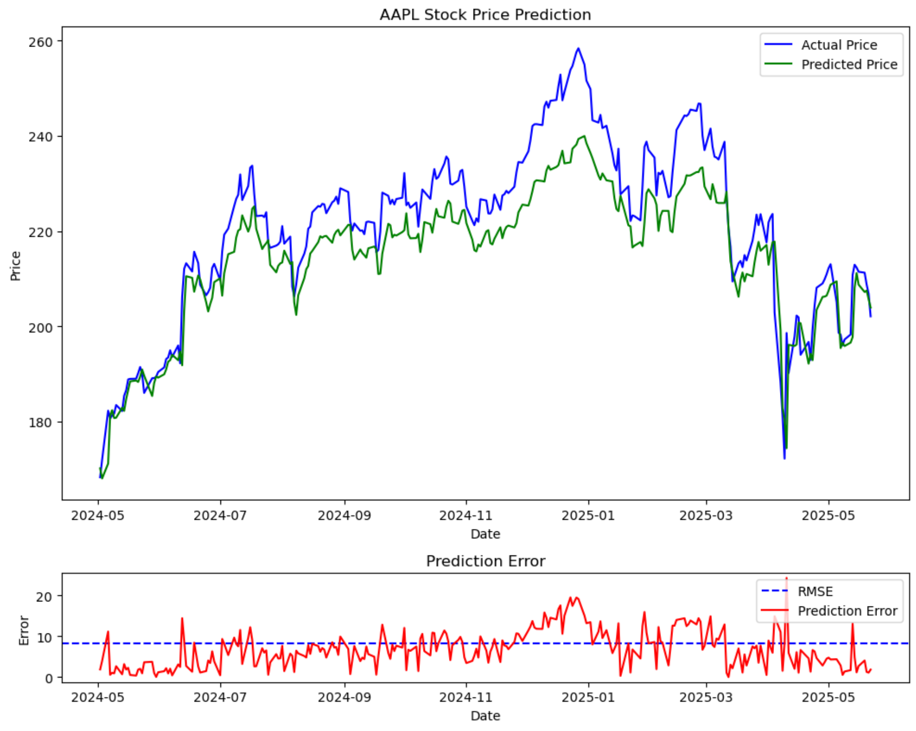


The model appeared to be a much better fit for this stock, in which the majority of the movement and volatility was captured. The various peaks in the prediction error were due to the fact that the model tended to be one step behind, as LSTM models are very reactionary. However, the root mean squared error came out to be 2.31, which was a large improvement.

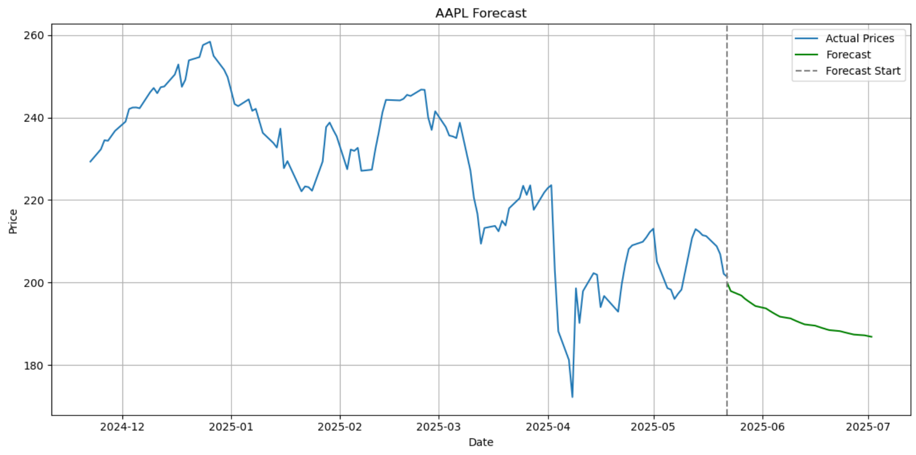


The above forecast illustrates a different result than ADI given that the forecast seems to be quite still. No stock price will ever act this way, however, this may indicate that the stock price might not change too much over the course of the next month, even if there are various spikes. Of course, this is without any external influences, so another plausible inference could be that any major world event in the next month could permanently influence the stock price without much change otherwise.

Apple Inc (AAPL)



The model for AAPL also performed fairly well, again capturing the movement, while still under-performing in relation to the volatility. The root mean squared error for this stock was 8.2, which puts it in between SBUX and ADI in terms of effectiveness. Again, the model seemed to be one step behind the true data, which makes sense due to the sequential structure of the model.



The forecast yet again shows a decline in the stock price over the next month. In the short term, this makes sense, as recent world events have negatively impacted the market as a whole. Again, the forecast only captures the trend, as the graph is too smooth to be accepted as a true forecast for each day.

Limitations

As already discussed, the forecasts do not reflect true data as the graphs are too smooth to truly resemble the volatility of the stock price. A possible way to improve the visual illustration of the forecast would be to add noise just as a way to make the results more plausible, however that could also lead to inaccurate predictions about the stock price on a specific day. Another limitation is that the model does not take into account world events that might drastically affect the market. Also, relying solely on the closing price does not allow for capturing all of the market indicators which could affect results. However, this project is strictly a visual implementation of the LSTM model and not meant to be used to make financial decisions.