```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


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

	Date	Open	High	Low	Close	Adj Close	Volume
0	2015-11-18	463.799988	465.649994	454.975006	456.000000	436.671021	5142766.0
1	2015-11-19	459.450012	469.350006	458.625000	467.375000	447.563873	5569752.0
2	2015-11-20	467.000000	476.399994	462.774994	473.424988	453.357422	5167930.0
3	2015-11-23	475.000000	478.950012	473.100006	476.875000	456.661224	4800026.0
4	2015-11-24	476.500000	485.799988	475.524994	483.850006	463.340515	6768886.0

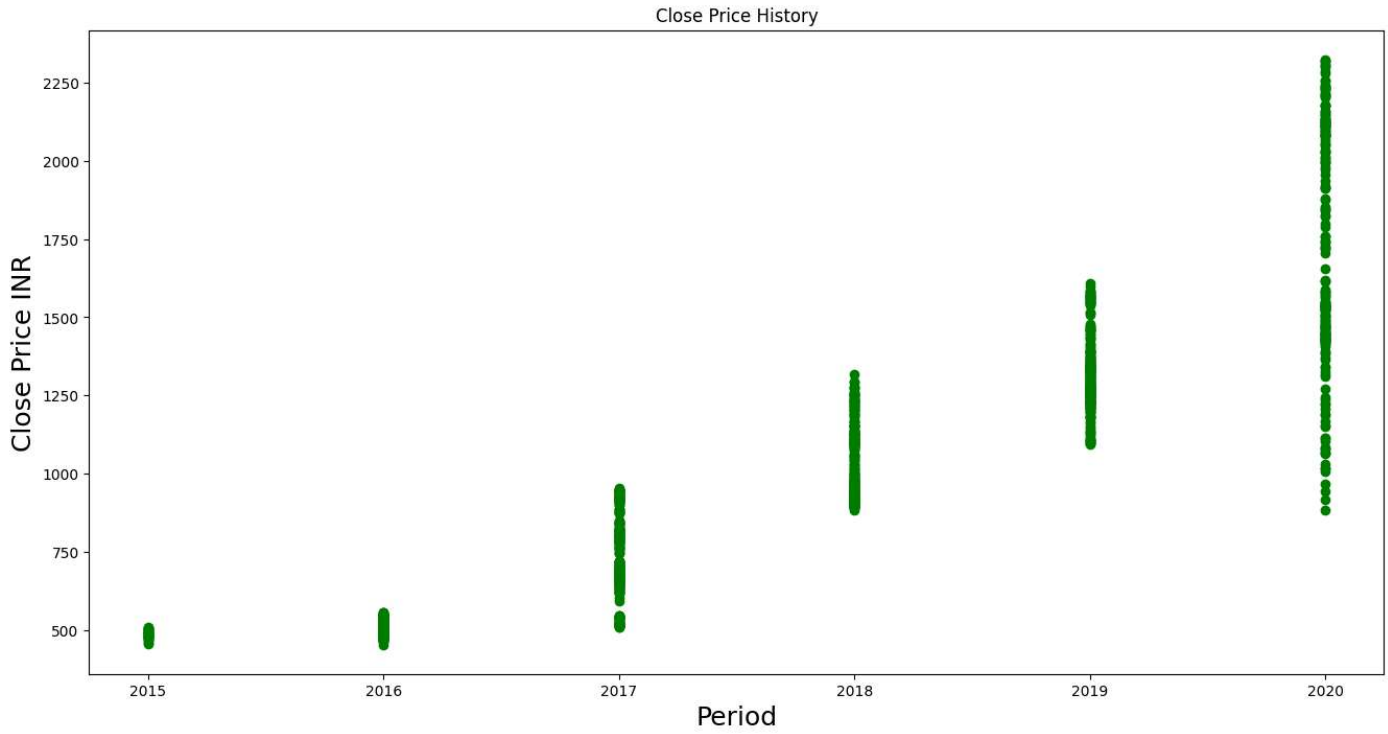
```
# Drop rows with any null values
data.dropna(axis = 0, inplace = True)


del data['Adj Close']
data.shape
```

```
(1232, 6)
```

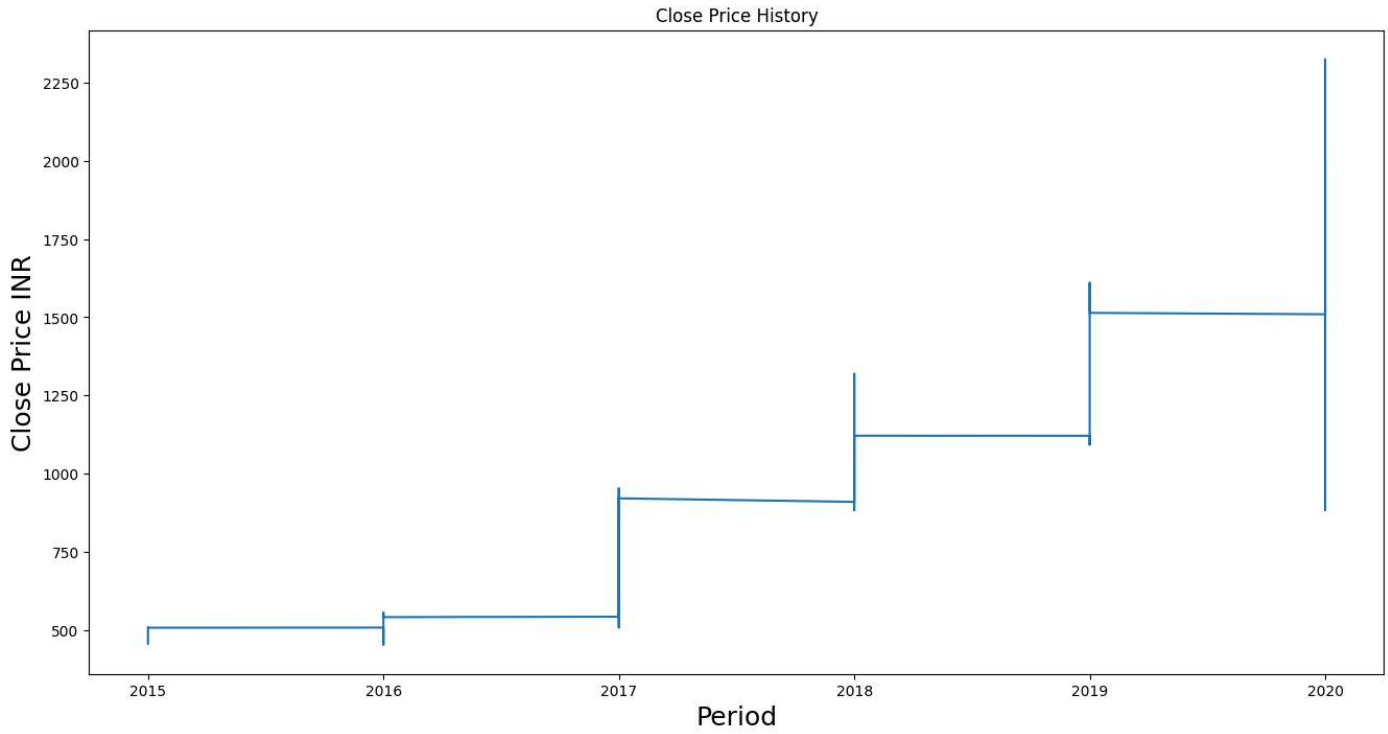
```
# Copy dataset into new dataframe for plots
meta = data.copy()
meta['Date'] = pd.to_datetime(meta['Date'], format='%Y-%m-%d')
meta['Year'] = meta['Date'].dt.year
meta.head()
```

	Date	Open	High	Low	Close	Volume	Year
0	2015-11-18	463.799988	465.649994	454.975006	456.000000	5142766.0	2015
1	2015-11-19	459.450012	469.350006	458.625000	467.375000	5569752.0	2015
2	2015-11-20	467.000000	476.399994	462.774994	473.424988	5167930.0	2015
3	2015-11-23	475.000000	478.950012	473.100006	476.875000	4800026.0	2015
4	2015-11-24	476.500000	485.799988	475.524994	483.850006	6768886.0	2015

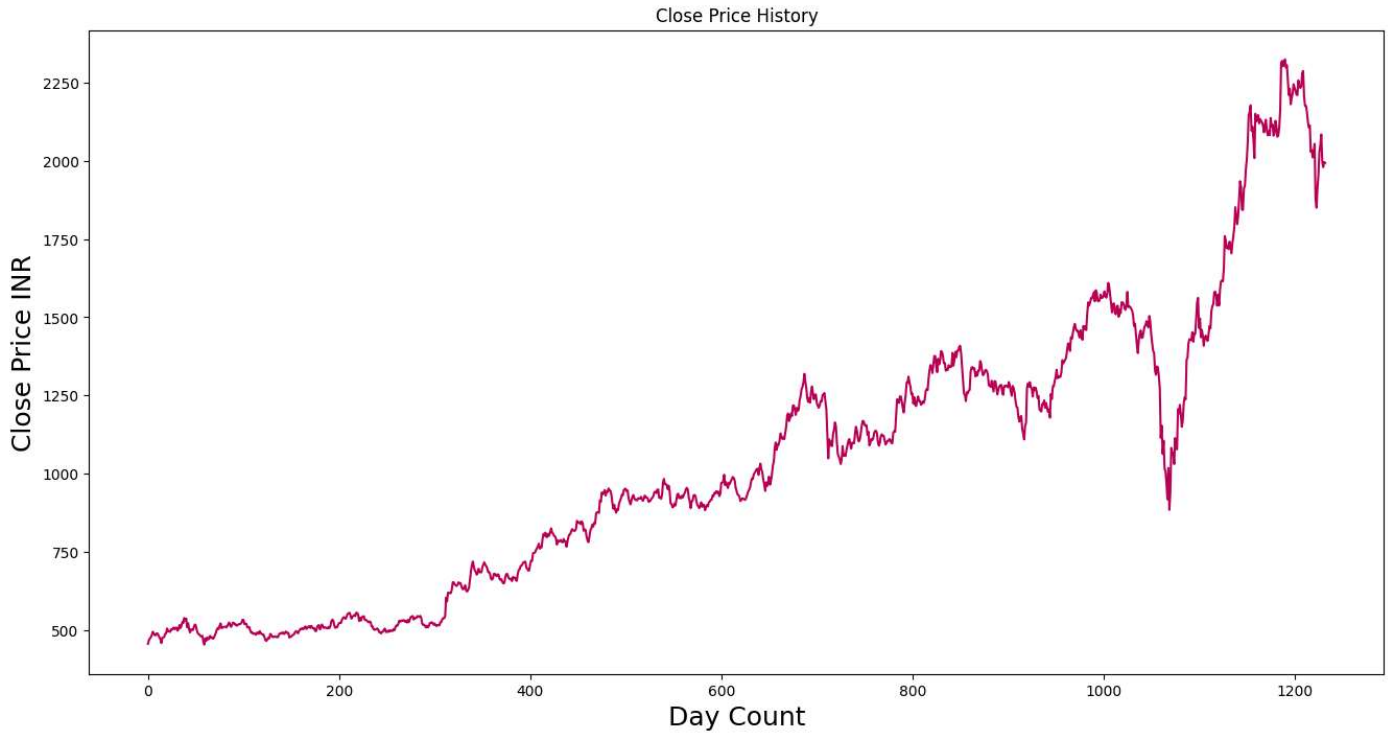
```
# Scatter plot of Close Price vs Year
plt.figure(figsize=(16,8))
plt.title('Close Price History')
plt.scatter(x = meta['Year'], y = meta['Close'], color = 'green')
plt.xlabel('Period', fontsize=18)
plt.ylabel('Close Price INR', fontsize=18)
plt.show()
```



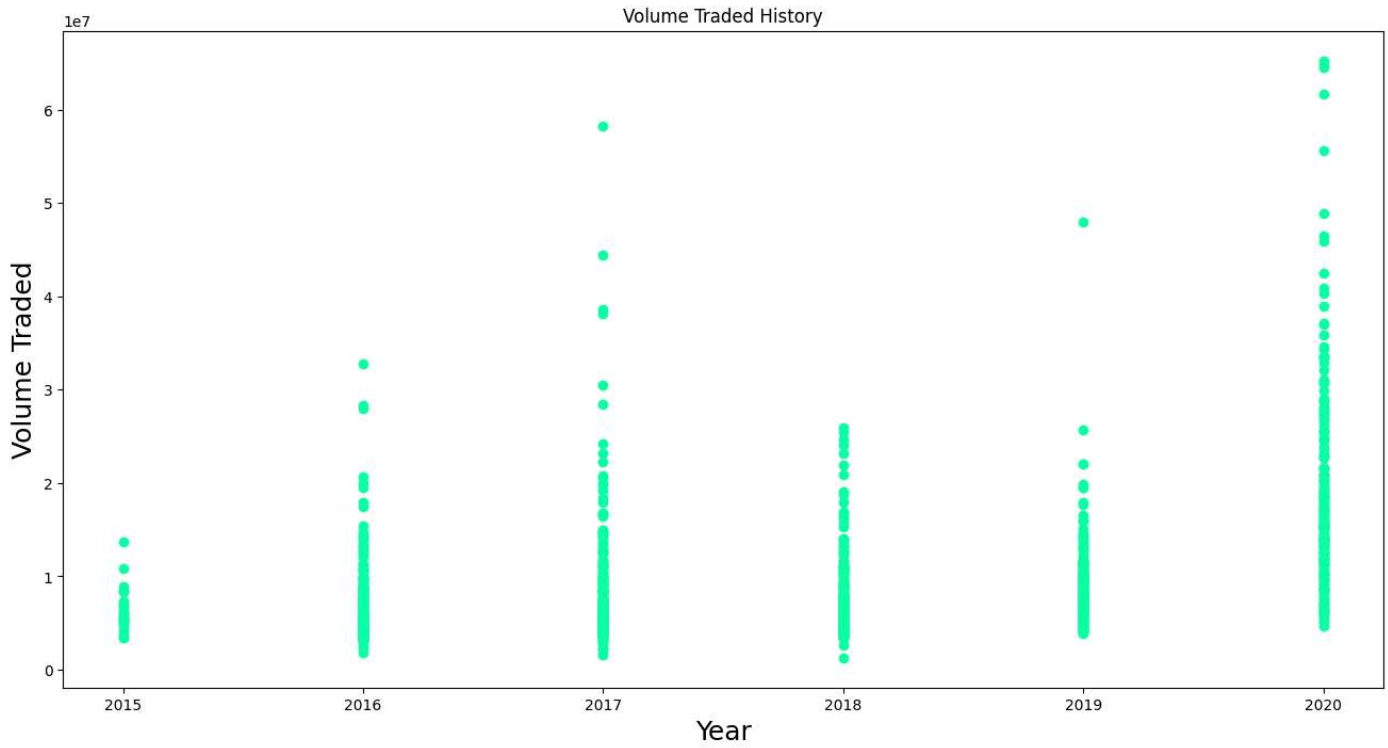

```
# Line plot of Close Price vs Year
plt.figure(figsize=(16,8))
plt.title('Close Price History')
plt.plot(meta['Year'],meta['Close'])
plt.xlabel('Period', fontsize=18)
plt.ylabel('Close Price INR', fontsize=18)
plt.show()
```



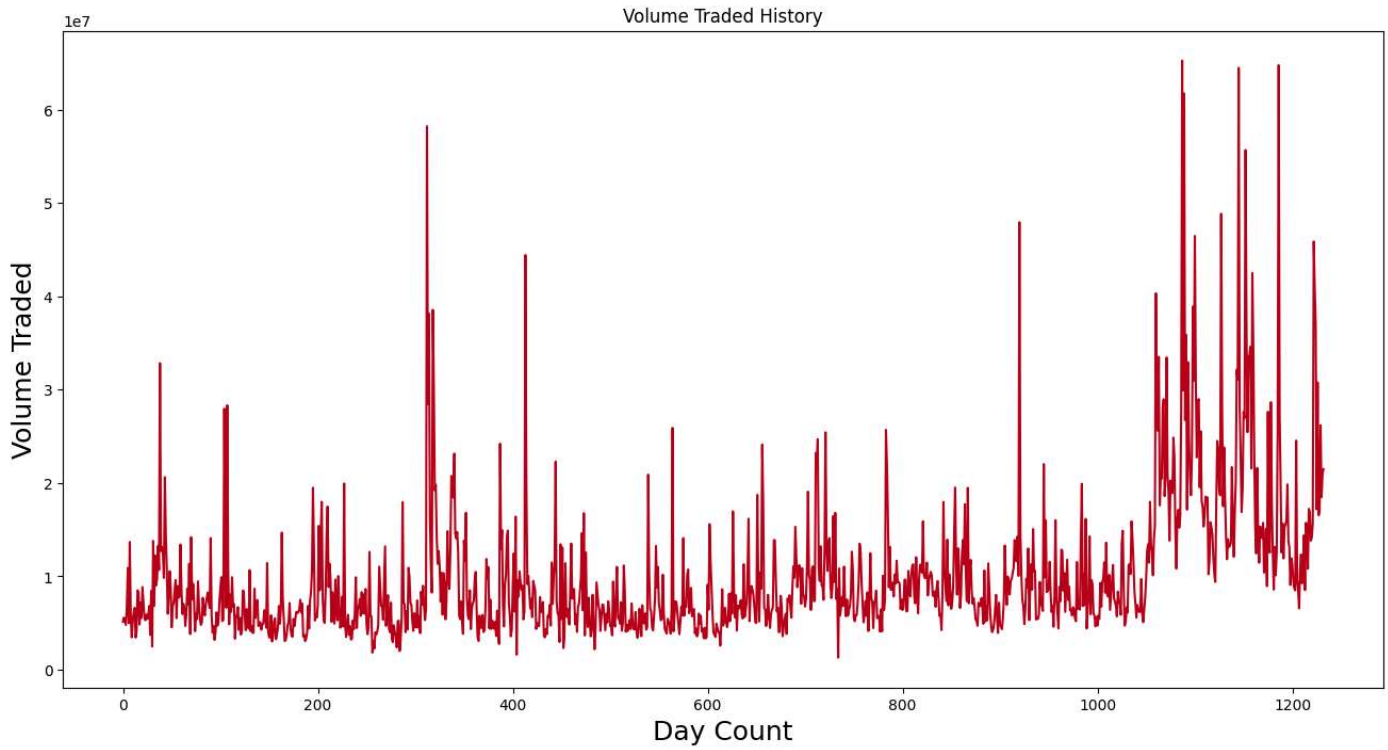

```
# Distribution of Close Price on day basis
plt.figure(figsize=(16,8))
plt.title('Close Price History')
plt.plot(meta['Close'], color = '#ba0459')
plt.xlabel('Day Count', fontsize=18)
plt.ylabel('Close Price INR', fontsize=18)
plt.show()
```




```
# Scatter plot of Volume Traded vs Year
plt.figure(figsize=(16,8))
plt.title('Volume Traded History')
plt.scatter(x = meta['Year'], y = meta['Volume'], color = '#0dffa6')
plt.xlabel('Year', fontsize=18)
plt.ylabel('Volume Traded', fontsize=18)
plt.show()
```




```
# Distribution of Volume Traded on day basis
plt.figure(figsize=(16,8))
plt.title('Volume Traded History')
plt.plot(meta['Volume'], color = '#bd0019')
plt.xlabel('Day Count', fontsize=18)
plt.ylabel('Volume Traded', fontsize=18)
plt.show()
```




```
# Reset index of original dataframe
data.set_index('Date', inplace = True)
data.head()
```

Date	Open	High	Low	Close	Volume
2015-11-18	463.799988	465.649994	454.975006	456.000000	5142766.0
2015-11-19	459.450012	469.350006	458.625000	467.375000	5569752.0
2015-11-20	467.000000	476.399994	462.774994	473.424988	5167930.0
2015-11-23	475.000000	478.950012	473.100006	476.875000	4800026.0
2015-11-24	476.500000	485.799988	475.524994	483.850006	6768886.0

```
# Scale and transform the target and features column using MinMaxScaler
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
X = data[['Open', 'Low', 'High', 'Volume']].copy()
y = data['Close'].copy()

X[['Open', 'Low', 'High', 'Volume']] = scaler.fit_transform(X)
y = scaler.fit_transform(y.values.reshape(-1, 1))


# Function to split dataset into Train and Test dataset
def load_data(X, seq_len, train_size=0.8):
    amount_of_features = X.shape[1]
    X_mat = X.values
    sequence_length = seq_len + 1
    datanew = []

    for index in range(len(X_mat) - sequence_length):
        datanew.append(X_mat[index: index + sequence_length])
```

```
    datanew = np.array(datanew)
    train_split = int(round(train_size * datanew.shape[0]))
    train_data = datanew[:train_split, :]

    X_train = train_data[:, :-1]
    y_train = train_data[:, -1][:,-1]

    X_test = datanew[train_split:, :-1]
    y_test = datanew[train_split:, -1][:,-1]

    X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], amount_of_features))
    X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], amount_of_features))

    return X_train, y_train, X_test, y_test


# Initializing the above function with a lookback window of 22
window = 22
X['close'] = y
X_train, y_train, X_test, y_test = load_data(X, window)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(967, 22, 5)
(967,)
(242, 22, 5)
(242,)
```

```
# Building our LSTM model to make predictions


from sklearn.preprocessing import MinMaxScaler
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
import matplotlib.pyplot as plt


# Check if GPU is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cpu
```

```
df = data['Close'].values.reshape(-1, 1)


df
```

```
array([[ 456.      ],
       [ 467.375   ],
       [ 473.424988],
       ...,
       [1980.      ],
       [1996.400024],
       [1993.25    ]])
```

```
data=df


# Normalize the data
scaler = MinMaxScaler(feature_range=(0, 1))
data_normalized = scaler.fit_transform(data)


def create_sequences(data, seq_length):
    sequences = []
    targets = []
    for i in range(len(data) - seq_length):
        seq = data[i:i+seq_length]
        target = data[i+seq_length]
        sequences.append(seq)
        targets.append(target)
    return np.array(sequences), np.array(targets)
```

```
# Parameters
seq_length = 60  # 60 days of historical data


# Prepare data with sliding window
X, y = create_sequences(data_normalized, seq_length)


# Split data for LSTM, Scoring, and Fine-tuning
lstm_split = int(0.5 * len(X))
scoring_split = int(0.75 * len(X))


X_lstm, y_lstm = X[:lstm_split], y[:lstm_split]
X_scoring, y_scoring = X[lstm_split:scoring_split], y[lstm_split:scoring_split]
X_finetuning, y_finetuning = X[scoring_split:], y[scoring_split:]


# Further split LSTM data into train and test
lstm_train_split = int(0.8 * len(X_lstm))
X_lstm_train, y_lstm_train = X_lstm[:lstm_train_split], y_lstm[:lstm_train_split]
X_lstm_test, y_lstm_test = X_lstm[lstm_train_split:], y_lstm[lstm_train_split:]


# LSTM Model
class LSTMModel(nn.Module):
    def __init__(self, input_size=1, hidden_size=50, num_layers=2, output_size=1):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out

lstm_model = LSTMModel().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(lstm_model.parameters(), lr=0.001)


# Train LSTM model
def train_model(model, train_data, train_targets, epochs=50, batch_size=32):
    train_data = torch.FloatTensor(train_data).to(device)
    train_targets = torch.FloatTensor(train_targets).to(device)
    train_dataset = TensorDataset(train_data, train_targets)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    model.train()
    for epoch in range(epochs):
        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = criterion(outputs, batch_y)
            loss.backward()
            optimizer.step()

        if (epoch + 1) % 10 == 0:
            print(f'Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}')

train_model(lstm_model, X_lstm_train, y_lstm_train)
```

```
Epoch [10/50], Loss: 0.0001
Epoch [20/50], Loss: 0.0002
Epoch [30/50], Loss: 0.0001
Epoch [40/50], Loss: 0.0000
Epoch [50/50], Loss: 0.0001
```

```
# LSTM Predictions and Evaluation
def predict_and_evaluate(model, X, y):
    model.eval()
    with torch.no_grad():
```

```
            X = torch.FloatTensor(X).to(device)
            predictions = model(X).cpu().numpy()

        y = scaler.inverse_transform(y.reshape(-1, 1))
        predictions = scaler.inverse_transform(predictions.reshape(-1, 1))

        mae = np.mean(np.abs(y - predictions))
        mape = np.mean(np.abs((y - predictions) / y)) * 100

        return predictions, mae, mape

    lstm_train_preds, lstm_train_mae, lstm_train_mape = predict_and_evaluate(lstm_model, X_lstm_train, y_lstm_train)
    lstm_test_preds, lstm_test_mae, lstm_test_mape = predict_and_evaluate(lstm_model, X_lstm_test, y_lstm_test)

    print(f"LSTM Train MAE: {lstm_train_mae:.2f}, MAPE: {lstm_train_mape:.2f}%")
    print(f"LSTM Test MAE: {lstm_test_mae:.2f}, MAPE: {lstm_test_mape:.2f}%")
```

```
LSTM Train MAE: 17.09, MAPE: 2.84%
LSTM Test MAE: 17.45, MAPE: 1.86%
```

```
    # Scoring Model
    class ScoringModel(nn.Module):
        def __init__(self, input_size=seq_length+1, hidden_size=32, output_size=1):
            super(ScoringModel, self).__init__()
            self.fc1 = nn.Linear(input_size, hidden_size)
            self.fc2 = nn.Linear(hidden_size, output_size)
            self.relu = nn.ReLU()
            self.sigmoid = nn.Sigmoid()

        def forward(self, x):
            out = self.relu(self.fc1(x))
            out = self.sigmoid(self.fc2(out))
            return out

    scoring_model = ScoringModel().to(device)
    scoring_criterion = nn.MSELoss()
    scoring_optimizer = optim.Adam(scoring_model.parameters(), lr=0.001)
```

```
    # Prepare scoring data
    def prepare_scoring_data(X, y):
        lstm_model.eval()
        with torch.no_grad():
            X_tensor = torch.FloatTensor(X).to(device)
            predictions = lstm_model(X_tensor).cpu().numpy()

        scoring_X = np.concatenate([X.reshape(X.shape[0], -1), predictions], axis=1)
        scoring_y = np.abs(y - predictions.reshape(-1, 1))  # Use absolute error as the score
        return scoring_X, scoring_y

    X_scoring_train, y_scoring_train = prepare_scoring_data(X_scoring[:int(0.8*len(X_scoring))], y_scoring[:int(0.8*len(X_scoring))])
    X_scoring_test, y_scoring_test = prepare_scoring_data(X_scoring[int(0.8*len(X_scoring)):], y_scoring[int(0.8*len(X_scoring)):])
```

```
    # Train scoring model
    train_model(scoring_model, X_scoring_train, y_scoring_train, epochs=30)
```

```
Epoch [10/30], Loss: 0.2813
Epoch [20/30], Loss: 0.2905
Epoch [30/30], Loss: 0.2753
```

```
    # Fine-tuning function
    def fine_tune_lstm(lstm_model, scoring_model, X, y, epochs=10, lr=0.0001):
        fine_tune_optimizer = optim.Adam(lstm_model.parameters(), lr=lr)
        X_tensor = torch.FloatTensor(X).to(device)
        y_tensor = torch.FloatTensor(y).to(device)

        for epoch in range(epochs):
            lstm_model.train()
            total_loss = 0
            for i in range(len(X)):
                fine_tune_optimizer.zero_grad()
                lstm_output = lstm_model(X_tensor[i].unsqueeze(0))
                loss = criterion(lstm_output, y_tensor[i].unsqueeze(0))
```

```
# Get score from scoring model
scoring_input = torch.cat([X_tensor[i].reshape(1, -1), lstm_output.detach()], dim=1)
score = scoring_model(scoring_input)

# Adjust loss based on score
adjusted_loss = loss * (1 + score.item())
adjusted_loss.backward()
fine_tune_optimizer.step()
total_loss += adjusted_loss.item()
```