SOURCE CODE :

```
from django.shortcuts import render
from django.http import JsonResponse
import yfinance as yf
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
import matplotlib.pyplot as plt
import matplotlib
matplotlib.use('Agg')
from io import BytesIO
import base64
import ta
import plotly.graph_objs as go
from sklearn.metrics import mean_squared_error
from datetime import datetime
import plotly.io as pio
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.forms import UserCreationForm
from django.contrib import messages
from django.contrib.auth.forms import AuthenticationForm
from django.contrib.auth.decorators import login_required
```

```python
# Signup view
def signup(request):
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            messages.success(request, "Account created successfully!")
            return redirect('login')
        else:
            messages.error(request, "Please correct the errors below.")
    else:
        form = UserCreationForm()
    return render(request, 'stockpredapp/signup.html', {'form': form})

# Login view
def login_view(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            user = form.get_user()
            login(request, user)
            return redirect('home')  # Redirect to the homepage or dashboard
        else:
            messages.error(request, "Invalid username or password.")
    else:
        form = AuthenticationForm()
    return render(request, 'stockpredapp/login.html', {'form': form})
```

```
# Logout view
def logout_view(request):
    logout(request)
    return redirect('logout_page')
def logout_page(request):
    return render(request,'stockpredapp/logout.html')
 # Redirect to the login page after logout

@login_required
def home(request):
    return render(request, 'stockpredapp/homepage.html')
import requests
from bs4 import BeautifulSoup
import random

### FUNCTION TO SCRAPE STOCK MARKET FACTS ###
def fetch_stock_facts():
    try:
        url = "https://www.moneycontrol.com/news/business/stocks/"
        headers = {"User-Agent": "Mozilla/5.0"}
        response = requests.get(url, headers=headers)
        soup = BeautifulSoup(response.text, "html.parser")

        facts = []
        for news in soup.find_all("li", class_="clearfix"):
            title = news.find("h2")
```

```
            if title:
                facts.append(title.text.strip())

        return facts if facts else ["Stock markets are unpredictable but always rewarding!"]
    except:
        return ["Unable to fetch stock market facts at the moment."]

### FUNCTION TO SCRAPE INVESTMENT PHILOSOPHIES ###
def fetch_market_philosophies():
    try:
        url = "https://www.goodreads.com/quotes/tag/investing"
        headers = {"User-Agent": "Mozilla/5.0"}
        response = requests.get(url, headers=headers)
        soup = BeautifulSoup(response.text, "html.parser")

        philosophies = []
        for quote in soup.find_all("div", class_="quoteText"):
            text = quote.text.strip().split("\n")[0]
            philosophies.append(text)

        return philosophies if philosophies else ["Invest wisely, patience pays!"]
    except:
        return ["Unable to fetch investment philosophies at the moment."]

### FUNCTION TO SCRAPE INVESTMENT STRATEGIES ###
def fetch_investment_strategies():
    try:
```

```
        url = "https://www.investopedia.com/terms/i/investment-strategy.asp"
        headers = {"User-Agent": "Mozilla/5.0"}
        response = requests.get(url, headers=headers)
        soup = BeautifulSoup(response.text, "html.parser")

        strategies = []
        for point in soup.find_all("li"):
            text = point.text.strip()
            if "invest" in text.lower():
                strategies.append(text)

        return strategies if strategies else ["Investing is a marathon, not a sprint!"]
    except:
        return ["Unable to fetch investment strategies at the moment."]

### API VIEW TO FETCH ALL DATA ###
def get_market_data(request):
    data = {
        "stock_facts": random.sample(fetch_stock_facts(), 1),
        "market_philosophies": random.sample(fetch_market_philosophies(), 1),
        "investment_strategies": random.sample(fetch_investment_strategies(), 1),
    }
    return JsonResponse(data)
# Function to fetch data and process predictions
def predict_stock_data(ticker):
    # Step 1: Fetch intraday data
    data = yf.download(tickers=ticker, interval="1m", period="5d")  # Fetch last 5 days of intraday data
```

```
data.columns = data.columns.droplevel(1)  # Keep relevant columns
data = data.dropna()  # Remove missing values

# Step 2: Add technical indicators
data['SMA'] = ta.trend.sma_indicator(data['Close'], window=14)
data['EMA'] = ta.trend.ema_indicator(data['Close'], window=14)
data['RSI'] = ta.momentum.rsi(data['Close'], window=14)
data['MACD'] = ta.trend.macd(data['Close'])
data['BB_high'], data['BB_low'] = ta.volatility.bollinger_hband(data['Close']), ta.volatility.bollinger_lband(data['Close'])
data = data.dropna()  # Drop rows with NaN values due to indicator calculations

# Step 3: Normalize the data
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(data[['Close', 'SMA', 'EMA', 'RSI', 'MACD', 'BB_high', 'BB_low']].values)

# Step 4: Create sequences for LSTM
def create_sequences(data, seq_length):
    x, y = [], []
    for i in range(seq_length, len(data)):
        x.append(data[i-seq_length:i])
        y.append(data[i, 0])  # Predicting the 'Close' price
    return np.array(x), np.array(y)

seq_length = 60  # Use the last 60 minutes for prediction
x, y = create_sequences(scaled_data, seq_length)
```

```
# Step 5: Split into training and testing sets
train_size = int(len(x) * 0.8)
x_train, x_test = x[:train_size], x[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Step 6: Build the LSTM model
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(x_train.shape[1], x_train.shape[2])),
    Dropout(0.2),
    LSTM(50, return_sequences=False),
    Dropout(0.2),
    Dense(25),
    Dense(1)
])

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

# Step 7: Train the model
model.fit(x_train, y_train, batch_size=32, epochs=20, validation_data=(x_test, y_test))

# Step 8: Make predictions
predictions = model.predict(x_test)

# Inverse transform predictions
```

```
    predictions_padded = np.concatenate((predictions, np.zeros((predictions.shape[0], scaled_data.shape[1] - 1))), axis=1)
    predictions = scaler.inverse_transform(predictions_padded)[:, 0]  # Extract the 'Close' column

    # Similarly, adjust y_test for comparison
    y_test_padded = np.concatenate((y_test.reshape(-1, 1), np.zeros((y_test.shape[0], scaled_data.shape[1] - 1))), axis=1)
    y_test_original = scaler.inverse_transform(y_test_padded)[:, 0]  # Extract the 'Close' column

    # Step 9: Evaluate the model
    mse = mean_squared_error(y_test_original, predictions)
    rmse = np.sqrt(mse)
    print(f'Mean Squared Error: {mse}')
    print(f'Root Mean Squared Error: {rmse}')

    # Step 10: Create Plotly figures for visualization
    # Plot 1: Actual vs Predicted Prices
    fig1 = go.Figure()
    fig1.add_trace(go.Scatter(x=data.index[-len(y_test):], y=y_test_original, mode='lines', name='Actual Price'))
    fig1.add_trace(go.Scatter(x=data.index[-len(y_test):], y=predictions, mode='lines', name='Predicted Price'))
    fig1.update_layout(title= ticker+" Intraday Price Prediction", xaxis_title='Time', yaxis_title='Stock Price (INR)')

    # Plot 2: Actual, Predicted, and Future Predictions
    future_predictions = []
    last_sequence = x_test[-1]  # Start with the last sequence from the test set
```

```
    for _ in range(5):  # Predict for the next 5 minutes
        next_prediction = model.predict(last_sequence[np.newaxis, :, :])[0, 0]
        future_predictions.append(next_prediction)
        last_sequence = np.append(last_sequence[1:], [[next_prediction] + [0] * (scaled_data.shape[1] - 1)], axis=0)

    future_predictions_padded = np.concatenate((np.array(future_predictions).reshape(-1, 1), np.zeros((5, scaled_data.shape[1] - 1))), axis=1)
    future_predictions_original = scaler.inverse_transform(future_predictions_padded)[:, 0]  # Extract 'Close'

    fig2 = go.Figure()
    fig2.add_trace(go.Scatter(x=data.index[-len(y_test):], y=y_test_original, mode='lines', name='Actual Price'))
    fig2.add_trace(go.Scatter(x=data.index[-len(y_test):], y=predictions, mode='lines', name='Predicted Price'))
    fig2.add_trace(go.Scatter(x=[data.index[-1] + pd.Timedelta(minutes=i+1) for i in range(5)], y=future_predictions_original, mode='lines', name='Future Predictions', line=dict(dash='dash')))
    fig2.update_layout(title= ticker+" Intraday Price Prediction with Next 5 Minutes", xaxis_title='Time', yaxis_title='Stock Price (INR)')

    # Plot 3: Minute-to-Minute Changes
    actual_changes = np.diff(y_test_original.flatten())  # Actual price changes
    predicted_changes = np.diff(predictions.flatten())  # Predicted price changes
    future_changes = np.diff(np.concatenate([predictions[-1:], future_predictions_original]))

    fig3 = go.Figure()
```

```
    fig3.add_trace(go.Scatter(x=data.index[-len(actual_changes):], y=actual_changes, mode='lines', name='Actual Changes', line=dict(color='green')))
    fig3.add_trace(go.Scatter(x=data.index[-len(predicted_changes):], y=predicted_changes, mode='lines', name='Predicted Changes', line=dict(color='orange')))
    fig3.add_trace(go.Scatter(x=[data.index[-1] + pd.Timedelta(minutes=i+1) for i in range(1, 5)], y=future_changes[:4], mode='lines', name='Future Changes', line=dict(color='red', dash='dash')))
    fig3.update_layout(title="Minute-to-Minute Price Changes", xaxis_title='Time', yaxis_title='Price Change (INR)')

    # Convert Plotly figures to HTML
    plot1_html = pio.to_html(fig1, full_html=False)
    plot2_html = pio.to_html(fig2, full_html=False)
    plot3_html = pio.to_html(fig3, full_html=False)

    return plot1_html, plot2_html, plot3_html, mse, rmse, y_test_original, predictions, future_predictions_original

# View to render the prediction page
def stock_intraday_prediction_view(request):
    ticker=request.GET.get('ticker','WIPRO.NS')
    plot1_html, plot2_html, plot3_html, mse, rmse, y_test_original, predictions, future_predictions = predict_stock_data(ticker)

    # Prepare context for rendering the template
    predo = [(i, r) for i, r in zip(range(1, 5), future_predictions)]

    context = {
```

```
        'ticker':ticker,
        'plot1': plot1_html,
        'plot2': plot2_html,
        'plot3': plot3_html,
        'mse': mse,
        'rmse': rmse,
        'actual_prices': list(y_test_original),
        'predicted_prices': list(predictions),
        'predo':predo
    }

    return render(request, 'stockpredapp/stockintradaypred.html', context)



def stock_moving_average(request):
    # Set today's date for the stock data
    ticker = request.GET.get('ticker', 'WIPRO.NS')
    today = datetime.now().strftime('%Y-%m-%d')

    # Fetch historical stock data for Tesla until today
    stock_data = yf.download(ticker, start='2016-10-01', end=today)

    # Use only the 'Close' column for price prediction
    close_prices = stock_data['Close'].values

    # Normalize the dataset using MinMaxScaler
    scaler = MinMaxScaler(feature_range=(0, 1))
```

```
scaled_data = scaler.fit_transform(close_prices.reshape(-1, 1))

# Split the data into training (80%) and testing (20%) sets
train_size = int(len(scaled_data) * 0.8)
train_data, test_data = scaled_data[:train_size], scaled_data[train_size:]

# Function to create sequences
def create_sequences(data, seq_length):
    x, y = [], []
    for i in range(seq_length, len(data)):
        x.append(data[i-seq_length:i, 0])
        y.append(data[i, 0])
    return np.array(x), np.array(y)

# Create sequences from the training and test data
seq_length = 60  # Use the last 60 days to predict the next day's price
x_train, y_train = create_sequences(train_data, seq_length)
x_test, y_test = create_sequences(test_data, seq_length)

# Reshape the input data to be compatible with LSTM
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

# Build the LSTM model
model = Sequential([
    LSTM(50, return_sequences=True, input_shape=(seq_length, 1)),
    LSTM(50, return_sequences=False),
```

```
    Dense(25),
    Dense(1)
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(x_train, y_train, batch_size=32, epochs=10)

# Make predictions on the test data
predictions = model.predict(x_test)

# Inverse transform the predictions back to original price scale
predictions = scaler.inverse_transform(predictions)

# Inverse transform the actual test data
y_test_scaled = scaler.inverse_transform(y_test.reshape(-1, 1))

# Calculate MSE and RMSE
mse = mean_squared_error(y_test_scaled, predictions)
rmse = np.sqrt(mse)

# Visualization using Plotly
fig = go.Figure()

# Add trace for actual prices
```

```
    fig.add_trace(go.Scatter(x=stock_data.index[-len(y_test):],
y=y_test_scaled.flatten(), mode='lines', name='Actual Price'))


    # Add trace for predicted prices
    fig.add_trace(go.Scatter(x=stock_data.index[-len(y_test):],
y=predictions.flatten(), mode='lines', name='Predicted Price'))


    # Add titles and labels
    fig.update_layout(title=ticker+' Stock Price Prediction', xaxis_title='Date',
yaxis_title='Stock Price (USD)')
    graph1 = fig.to_html(full_html=False)
    # Predict the next 5 days
    last_sequence = scaled_data[-seq_length:].reshape(1, seq_length, 1)  # Use
the last 60 days of data as the input sequence
    next_5_days = []


    for i in range(5):
        prediction = model.predict(last_sequence)  # Predict the next day's value
        next_5_days.append(prediction[0, 0])  # Store the predicted value
        # Update the last sequence with the predicted value for the next iteration
        last_sequence = np.append(last_sequence[:, 1:, :], prediction.reshape(1, 1,
1), axis=1)


    # Inverse transform the 5-day predictions back to original price scale
    next_5_days = scaler.inverse_transform(np.array(next_5_days).reshape(-1,
1))


    # Prepare the future dates for the next 5 days prediction
    future_dates = pd.date_range(start=stock_data.index[-1] +
pd.Timedelta(days=1), periods=5, freq='B')  # Business days
```

```
    future_predictions = next_5_days.flatten()


    # Combine future dates and predictions into a list of tuples
    date_price_pairs = list(zip(future_dates.strftime('%Y-%m-%d').tolist(),
future_predictions.tolist()))
    # Add future predictions trace
    fig.add_trace(go.Scatter(x=future_dates, y=next_5_days.flatten(),
mode='lines', name='Next 5 Days Prediction'))


    # Update the layout for future predictions
    fig.update_layout(title=ticker+' Stock Price Prediction with Future 5 Days',
xaxis_title='Date', yaxis_title='Stock Price (USD)')


    # Convert figure to HTML and pass it to the template
    graph2 = fig.to_html(full_html=False)


    return render(request, 'stockpredapp/stocknormalpred.html', {
        'ticker':ticker,
        'mse': mse,
        'rmse': rmse,
        'graph1': graph1,
        'graph2':graph2,
        'next_5_days': next_5_days.flatten(),
         'date_price_pairs': date_price_pairs,
    })



def stock_tech_analysis(request):
```

```
# Define the stock ticker and timeframe
ticker =  request.GET.get('ticker', 'WIPRO.NS')
start_date = '2025-01-01'
end_date =  datetime.now().strftime('%Y-%m-%d')

# Download stock data
data = yf.download(ticker, start=start_date, end=end_date)
data.columns = data.columns.droplevel(1)
print(data)

# Calculate indicators
data['SMA_50'] = data['Close'].rolling(window=50).mean()
data['SMA_200'] = data['Close'].rolling(window=200).mean()
data['EMA_50'] = data['Close'].ewm(span=50, adjust=False).mean()

print(data[['SMA_50', 'SMA_200']].head(10))



delta = data['Close'].diff(1)
gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
rs = gain / loss
data['RSI'] = 100 - (100 / (1 + rs))

ema_12 = data['Close'].ewm(span=12, adjust=False).mean()
ema_26 = data['Close'].ewm(span=26, adjust=False).mean()
data['MACD'] = ema_12 - ema_26
```

```
data['MACD_signal'] = data['MACD'].ewm(span=9, adjust=False).mean()

data['Middle_BB'] = data['Close'].rolling(window=20).mean()
data['Upper_BB'] = data['Middle_BB'] +
(data['Close'].rolling(window=20).std() * 2)
data['Lower_BB'] = data['Middle_BB'] -
(data['Close'].rolling(window=20).std() * 2)

# Prepare plots
plots = []

# Plot 1: Stock Price and Moving Averages
plt.figure(figsize=(7, 5))
plt.plot(data['Close'], label='Stock Price', color='blue', alpha=0.5)
plt.plot(data['SMA_50'], label='50-Day SMA', color='orange')
plt.plot(data['SMA_200'], label='200-Day SMA', color='green')
plt.plot(data['EMA_50'], label='50-Day EMA', color='red')
plt.title(f'{ticker} Stock Price and Moving Averages')
plt.legend()
plots.append(encode_plot_to_base64())

# Plot 2: RSI
plt.figure(figsize=(7, 5))
plt.plot(data['RSI'], label='RSI', color='purple')
plt.axhline(70, color='red', linestyle='--', label='Overbought')
plt.axhline(30, color='green', linestyle='--', label='Oversold')
plt.title('Relative Strength Index (RSI)')
plt.legend()
```

```
    plots.append(encode_plot_to_base64())

    # Plot 3: MACD
    plt.figure(figsize=(7, 5))
    plt.plot(data['MACD'], label='MACD', color='blue')
    plt.plot(data['MACD_signal'], label='MACD Signal', color='red')
    plt.title('MACD (Moving Average Convergence Divergence)')
    plt.legend()
    plots.append(encode_plot_to_base64())

    # Plot 4: Bollinger Bands
    plt.figure(figsize=(7, 5))
    plt.plot(data['Close'], label='Stock Price', color='blue', alpha=0.5)
    plt.plot(data['Upper_BB'], label='Upper BB', color='green', linestyle='--')
    plt.plot(data['Middle_BB'], label='Middle BB', color='orange', linestyle='--')
    plt.plot(data['Lower_BB'], label='Lower BB', color='red', linestyle='--')
    plt.title('Bollinger Bands')
    plt.legend()
    plots.append(encode_plot_to_base64())

    # Pass plots to the template
    return render(request, 'stockpredapp/stocktechanalysis.html', {'plots':
plots,'t':ticker})

def encode_plot_to_base64():
    buffer = BytesIO()
    plt.savefig(buffer, format='png', bbox_inches='tight')
    buffer.seek(0)
```

```
    encoded_plot = base64.b64encode(buffer.getvalue()).decode('utf-8')
    buffer.close()
    plt.close()  # Close the current figure explicitly
    return f"data:image/png;base64,{encoded_plot}"


def add_technical_indicators(df):
    """
    Add technical indicators to the DataFrame.
    """
    # Moving Averages
    df['SMA_10'] = df['Close'].rolling(window=10).mean()
    df['SMA_20'] = df['Close'].rolling(window=20).mean()

    # Relative Strength Index (RSI)
    delta = df['Close'].diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=14).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=14).mean()
    rs = gain / loss
    df['RSI'] = 100 - (100 / (1 + rs))

    # MACD (Moving Average Convergence Divergence)
    ema_12 = df['Close'].ewm(span=12, adjust=False).mean()
    ema_26 = df['Close'].ewm(span=26, adjust=False).mean()
    df['MACD'] = ema_12 - ema_26
    df['Signal'] = df['MACD'].ewm(span=9, adjust=False).mean()
```

```
    # Drop rows with NaN values due to rolling calculations
    df = df.dropna()

    return df

def predict_next_candles_with_lstm(df, num_predictions=5, lookback=60):
    """
    Predict the next 'num_predictions' candlesticks using an LSTM model with 
technical indicators.
    """
    # Add technical indicators
    df = add_technical_indicators(df)

    # Scale the features
    scaler = MinMaxScaler(feature_range=(0, 1))
    scaled_data = scaler.fit_transform(df[['Close', 'SMA_10', 'SMA_20', 'RSI', 
'MACD', 'Signal']].values)

    # Prepare training data
    X_train, y_train = [], []
    for i in range(lookback, len(scaled_data)):
        X_train.append(scaled_data[i - lookback:i, :])  # Use all features for LSTM
        y_train.append(scaled_data[i, 0])  # Predict the `Close` price

    X_train, y_train = np.array(X_train), np.array(y_train)

    # Build the LSTM model
    model = Sequential()
```

```
    model.add(LSTM(units=50, return_sequences=True,
input_shape=(X_train.shape[1], X_train.shape[2])))
    model.add(LSTM(units=50, return_sequences=False))
    model.add(Dense(units=25))
    model.add(Dense(units=1))
    model.compile(optimizer='adam', loss='mean_squared_error')

    # Train the model
    model.fit(X_train, y_train, batch_size=32, epochs=10, verbose=1)

    # Predict the next 'num_predictions'
    last_lookback_data = scaled_data[-lookback:]
    predictions = []

    for _ in range(num_predictions):
        # Reshape and predict
        input_data = last_lookback_data.reshape((1, lookback,
scaled_data.shape[1]))
        predicted_scaled = model.predict(input_data, verbose=0)
        predicted_price = scaler.inverse_transform([[predicted_scaled[0], 0, 0, 0,
0, 0]])[0, 0]
        predictions.append(predicted_price)

        # Update the last lookback data with the new prediction
        new_row = np.append(predicted_scaled[0], [0, 0, 0, 0, 0])  # Append zeros
for other features
        last_lookback_data = np.vstack((last_lookback_data[1:], new_row))

    # Generate predicted candles
```

```
    predicted_candles = []
    for i, pred in enumerate(predictions):
        open_pred = pred - np.random.rand() * 2
        high_pred = pred + np.random.rand() * 2
        low_pred = pred - np.random.rand() * 2
        predicted_candles.append({
            "x": (df.index[-1] + pd.Timedelta(minutes=5 * (i + 1))).strftime('%Y-
%m-%d %H:%M:%S'),
            "open": open_pred,
            "high": high_pred,
            "low": low_pred,
            "close": pred
        })

    return predicted_candles
def stock_analysis(request):
    return render(request,"stockpredapp/stockanalysis.html")


def calculate_rsi(df, window=14):
    """
    Calculate the Relative Strength Index (RSI).
    """
    delta = df['Close'].diff()
    gain = (delta.where(delta > 0, 0)).rolling(window=window).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(window=window).mean()
    rs = gain / loss
    rsi = 100 - (100 / (1 + rs))
    return rsi
```

```
def add_technical_indicators(df):
    """
    Add multiple technical indicators to the DataFrame.
    """
    # Moving Averages
    df['SMA_10'] = df['Close'].rolling(window=10).mean()  # Simple Moving Average
    df['SMA_20'] = df['Close'].rolling(window=20).mean()
    df['EMA_10'] = df['Close'].ewm(span=10, adjust=False).mean()  # Exponential Moving Average

    # Bollinger Bands
    df['BB_upper'] = df['SMA_20'] + 2 * df['Close'].rolling(window=20).std()
    df['BB_lower'] = df['SMA_20'] - 2 * df['Close'].rolling(window=20).std()

    # RSI
    df['RSI'] = calculate_rsi(df)

    # MACD and Signal Line
    ema_12 = df['Close'].ewm(span=12, adjust=False).mean()
    ema_26 = df['Close'].ewm(span=26, adjust=False).mean()
    df['MACD'] = ema_12 - ema_26
    df['Signal'] = df['MACD'].ewm(span=9, adjust=False).mean()

    # ATR (Average True Range)
    df['High-Low'] = df['High'] - df['Low']
    df['High-Close'] = abs(df['High'] - df['Close'].shift())
```

```
    df['Low-Close'] = abs(df['Low'] - df['Close'].shift())
    df['TR'] = df[['High-Low', 'High-Close', 'Low-Close']].max(axis=1)
    df['ATR'] = df['TR'].rolling(window=14).mean()

    # Stochastic Oscillator
    df['14-high'] = df['High'].rolling(window=14).max()
    df['14-low'] = df['Low'].rolling(window=14).min()
    df['%K'] = (df['Close'] - df['14-low']) / (df['14-high'] - df['14-low']) * 100
    df['%D'] = df['%K'].rolling(window=3).mean()

    # Drop temporary columns
    df = df.drop(columns=['High-Low', 'High-Close', 'Low-Close', 'TR', '14-high',
'14-low'])

    # Drop rows with NaN values due to rolling calculations
    df = df.dropna()

    return df
def predict_next_candles_with_indicators(df, num_predictions=5):
    """
    Predict the next 'num_predictions' candlesticks using linear regression with
technical indicators.
    """
    # Add technical indicators
    df['SMA_10'] = df['Close'].rolling(window=10).mean()  # 10-period Simple
Moving Average
    df['SMA_20'] = df['Close'].rolling(window=20).mean()  # 20-period Simple
Moving Average
    df['RSI'] = calculate_rsi(df)                    # Relative Strength Index
```

```
# Drop rows with NaN values due to rolling calculations
df = df.dropna()

# Features (X) and target (y)
X = df[['Close', 'SMA_10', 'SMA_20', 'RSI']].values
y = df['Close'].values

# Normalize features for better regression performance
scaler = MinMaxScaler()
X = scaler.fit_transform(X)

# Fit the model
model = LinearRegression()
model.fit(X, y)

# Predict the next 'num_predictions' values
last_features = df[['Close', 'SMA_10', 'SMA_20', 'RSI']].iloc[-1].values
last_features_scaled = scaler.transform([last_features])

future_candles = []
for i in range(num_predictions):
    prediction = model.predict(last_features_scaled)[0]

    # Simulate open, high, and low prices based on prediction
    open_pred = prediction - np.random.rand() * 2
    high_pred = prediction + np.random.rand() * 2
```

```
        low_pred = prediction - np.random.rand() * 2

        # Append predicted candle
        future_candles.append({
            "x": (df.index[-1] + pd.Timedelta(minutes=5 * (i + 1))).strftime('%Y-%m-%d %H:%M:%S'),
            "open": open_pred,
            "high": high_pred,
            "low": low_pred,
            "close": prediction
        })

        # Update the last_features with the new prediction for recursive prediction
        last_features_scaled = scaler.transform([[prediction, prediction, prediction, 50]])  # RSI ~ 50 neutral assumption

    return future_candles
@login_required
def stock_search(request):
    if request.method == "POST":
        stock = request.POST.get('stock_symbol')
        period = request.POST.get('period')  # e.g., '1d', '5d', etc.
        interval = request.POST.get('interval')  # e.g., '1m', '5m', '1d', etc.
        start_time = request.POST.get('start_time')
        end_time = request.POST.get('end_time')

        # Fetch live stock data using yfinance
        df = yf.download(tickers=stock, period=period, interval=interval)
```

```
        print(df)

        if df.empty:
            return JsonResponse({"error": f"No data available for {stock}"})

        # Filter data by time range
        df.index = pd.to_datetime(df.index)
        df.index = df.index.tz_convert("Asia/Kolkata")
        print(df.index)
        filtered_df = df.between_time(start_time, end_time)
        print(filtered_df)
        filtered_df.columns = filtered_df.columns.droplevel(1)
        print(filtered_df.columns)

        if filtered_df.empty:
            return JsonResponse({"error": f"No data available between {start_time}
and {end_time} for {stock}"})

        # Predict the next 5 candlesticks based on the historical data
        predicted_candles = predict_next_candles_with_indicators(filtered_df)

        # Prepare the candlestick data for visualization
        candlestick_data = {
            "data": [
                {
                    "type": "candlestick",
                    "x": filtered_df.index.strftime('%Y-%m-%d %H:%M:%S').tolist()
+ [candle["x"] for candle in predicted_candles],
```

```
                "open": filtered_df['Open'].tolist() + [candle["open"] for candle in predicted_candles],
                "high": filtered_df['High'].tolist() + [candle["high"] for candle in predicted_candles],
                "low": filtered_df['Low'].tolist() + [candle["low"] for candle in predicted_candles],
                "close": filtered_df['Close'].tolist() + [candle["close"] for candle in predicted_candles],
            }
        ],
        "layout": {
            "title": f"Stock Price for {stock} ({start_time} - {end_time})",
            "xaxis": {"title": "Time"},
            "yaxis": {"title": "Price (INR)"},
            "showlegend": False,
        }
    }

    # Return the data as JSON for frontend to render the graph
    return JsonResponse({"graph": candlestick_data})

  # For GET requests, render the template
  return render(request, "stockpredapp/stocksearch.html")

def fetch_all_stocks(request):
  stock_symbols = yf.Ticker("^NSEI").symbols  # Adjust with an actual method to fetch symbols
  return JsonResponse(stock_symbols, safe=False)
```

```
#Live NEWS
import feedparser
def stock_market_news_view(request):
    # Yahoo Finance Global Stock Market News RSS Feed
    news_articles = []
    ticker = request.GET.get('ticker','TCS.NS')  # Change to your stock ticker
    rss_url = f"https://feeds.finance.yahoo.com/rss/2.0/headline?s={ticker}&region=IND&lang=en-US"
    news_feed = feedparser.parse(rss_url)

    for entry in news_feed.entries[:10]:  # Get top 10 news
      print(f"Title: {entry.title}")
      print(f"Description: {entry.description}")  # Shows the full news summary
      print(f"Published: {entry.published}\n")

    for entry in news_feed.entries[:10]:  # Get top 10 news
        news_articles.append({
            "title": entry.title,
            "description": entry.description,
            "published": entry.published,
        })
    print(news_articles)

    return render(request, "stockpredapp/stockgeneralnews.html", {"news_articles": news_articles,'ticker':ticker})
def fetch_indian_stock_news():
    # RSS Feeds for Indian stock market news (NSE & BSE)
```

```
    nse_rss =
"https://news.google.com/rss/search?q=NSE+India+stock+market&hl=en-IN&gl=IN&ceid=IN:en"
    bse_rss =
"https://news.google.com/rss/search?q=BSE+India+stock+market&hl=en-IN&gl=IN&ceid=IN:en"

    feeds = [("NSE", nse_rss), ("BSE", bse_rss)]
    news_data = []

    for market, rss_url in feeds:
        feed = feedparser.parse(rss_url)

        articles = []
        for entry in feed.entries[:10]:  # Get top 10 news articles
            articles.append({
                "title": entry.title,
                "link": entry.link,
                "description": entry.summary if 'summary' in entry else "No description available."
            })

        news_data.append({"market": market, "articles": articles})

    return news_data

def stock_news_total_view(request):
    news = fetch_indian_stock_news()
    return render(request, 'stockpredapp/stockspecnews.html', {"news": news})
```