AI/ML Methods in the Stock Market

Teja Vuppu (Pace University MS in CS)

Kadiyala Padma (Pace University Professor)

Abstract: This study delves into an in-depth analysis of Microsoft (MSFT) stock price data extracted from Vayu Financials, focusing on a 30-day span with 1-minute price intervals, totaling 391 intervals per day. Various financial factors such as Effective Bid-Ask Spread, Order Imbalance, Auto Correlation of Stock Returns, Kyle's Lambda, Volatility, and PIN (Probability of Informed Trading) are examined over the selected period. The data undergoes meticulous preprocessing, including addressing missing values and introducing categorical variables to capture different phases of the trading day. Utilizing Python programming language, data visualization techniques, and machine learning algorithms such as Q-Learning, RandomForestRegressor, and LSTM, predictive models are developed to forecast stock prices. The study concludes by providing insights into market dynamics, trade imbalances, and predictive capabilities, contributing to a deeper understanding of stock market behavior and trends.

We extract stock price data for Microsoft (MSFT) from the Vayu Financials website, covering a span of 30 days. Each day's data consists of 1-minute price intervals, totaling 391 intervals per day.

```
[1]: import pandas as pd
     import numpy as np

[2]: stock_price = pd.read_excel("/Users/teja/Desktop/MSFT.xlsx",sheet_name='Stock_Price_MSFT')

[3]: stock_price
```

[3]:

	Date	Time	Stock_Price
0	2023-10-27	09:30	331.12
1	2023-10-27	09:31	331.42
2	2023-10-27	09:32	331.84
3	2023-10-27	09:33	332.15
4	2023-10-27	09:34	331.66
...	...	...	...
11725	2023-09-18	15:56	328.55
11726	2023-09-18	15:57	328.62
11727	2023-09-18	15:58	328.85
11728	2023-09-18	15:59	329.32
11729	2023-09-18	16:00	329.79

11730 rows × 3 columns

We've gathered 30 days of data for the Effective Bid-Ask Spread, which is determined as the difference between the last trade price and the midpoint of the bid-ask spread for buys, as well as the difference between the midpoint of the bid-ask spread and the last trade price. Each day's data comprises 403 values per minute, with duplicate values recorded at intervals of 10:00 am, 10:30 am, 11:00 am, 11:30 am, 12:00 pm, 12:30 pm, 1:00 pm, 1:30 pm, 2:00 pm, 2:30 pm, 3:00 pm, and 3:30 pm from our data source.

```
[62]: # Find duplicates based on 'DateTime' column
      duplicates = my_bid_ask_spread[my_bid_ask_spread.duplicated(subset=['DateTime'], keep=False)]

[63]: duplicates
```

[63]:

	Date	Time	Effective_bid_ask_spread	DateTime
30	2023-10-27	10:00	9.00	2023-10-27 10:00:00
31	2023-10-27	10:00	8.00	2023-10-27 10:00:00
61	2023-10-27	10:30	5.06	2023-10-27 10:30:00
62	2023-10-27	10:30	0.00	2023-10-27 10:30:00
92	2023-10-27	11:00	0.00	2023-10-27 11:00:00
...	...	...	...	...
3131	2023-10-18	14:30	0.00	2023-10-18 14:30:00
3161	2023-10-18	15:00	1.00	2023-10-18 15:00:00
3162	2023-10-18	15:00	0.00	2023-10-18 15:00:00
3192	2023-10-18	15:30	1.98	2023-10-18 15:30:00
3193	2023-10-18	15:30	2.00	2023-10-18 15:30:00

192 rows × 4 columns

We took the mean of these two values to obtain the average bid-ask spread for these datapoints.

```
[64]: # Calculate the average of 'Effective_bid_ask_spread' for each set of duplicates
      averages = duplicates.groupby('DateTime')['Effective_bid_ask_spread'].mean()

[65]: averages

[65]: DateTime
      2023-10-18 10:00:00     1.20
      2023-10-18 10:30:00     2.00
      2023-10-18 11:00:00     0.50
      2023-10-18 11:30:00     1.20
      2023-10-18 12:00:00     3.50
                             ...
      2023-10-27 13:30:00    12.95
      2023-10-27 14:00:00     0.52
      2023-10-27 14:30:00     0.32
      2023-10-27 15:00:00     4.99
      2023-10-27 15:30:00     1.99
      Name: Effective_bid_ask_spread, Length: 96, dtype: float64

[66]: # Replace the original duplicates with the calculated averages
      my_bid_ask_spread.loc[my_bid_ask_spread['DateTime'].isin(averages.index), 'Effective_bid_ask_spread'] = my_bid_ask_spread.loc[my_bid_ask_sprea

[67]: my_bid_ask_spread
```

[67]:

	Date	Time	Effective_bid_ask_spread	DateTime
0	2023-10-27	09:30	6.00	2023-10-27 09:30:00
1	2023-10-27	09:31	11.00	2023-10-27 09:31:00
2	2023-10-27	09:32	11.00	2023-10-27 09:32:00
3	2023-10-27	09:33	13.00	2023-10-27 09:33:00
4	2023-10-27	09:34	6.00	2023-10-27 09:34:00
...	...	...	...	...
3219	2023-10-18	15:56	1.00	2023-10-18 15:56:00
3220	2023-10-18	15:57	1.00	2023-10-18 15:57:00
3221	2023-10-18	15:58	1.00	2023-10-18 15:58:00
3222	2023-10-18	15:59	2.00	2023-10-18 15:59:00
3223	2023-10-18	16:00	4.68	2023-10-18 16:00:00

3224 rows × 4 columns

We've collected 30 days of data for various financial factors:

1. Effective Bid-Ask Spread: The difference between the last trade price and the midpoint of the bid-ask spread for buys, and vice versa. Recorded at 12 intervals throughout the trading day.

2. Order Imbalance: The difference between the bid size and ask size, indicating an imbalance between buyers and sellers. Values were averaged at 12 intervals to estimate the order imbalance.

3. Auto Correlation of Stock Returns: The auto-correlation coefficient of one-minute returns over different time periods (1hr, 45min, 30min, 15min). Some data points were missing for the 45min and 30min intervals.

4. Kyle's Lambda: The price impact of a trade measured as the regression coefficient obtained from regressing returns over signed volumes at 15-minute non-overlapping intervals. Recorded at 27 intervals throughout the trading day.

5. Volatility: The standard deviation of one-minute returns over different time periods (1hr, 45min, 30min, 15min). Some data points were missing for the 45min, 30min, and 15min intervals.

6. PIN (Probability of Informed Trading): The ratio of the elasticity of buy volume to the sum of the elasticities of buy and sell volumes. Values were averaged at 12 intervals from 10:30 am to 3:30 pm to estimate the PIN.

After extracting all the data and conducting data analysis, including determining the number of missing data points, averaging additional data points for specific times, and organizing the data into different Excel files, we utilized Python programming language for further analysis. These factors offer valuable insights into market dynamics, trade imbalances, price impacts, and trading probabilities, facilitating a deeper understanding of market behavior and trends over time.

```
[100]: my_price_data
```

[100]:

	Date	Time	Stock_Price
0	2023-10-27	09:30	331.120
1	2023-10-27	09:31	331.420
2	2023-10-27	09:32	331.840
3	2023-10-27	09:33	332.150
4	2023-10-27	09:34	331.660
...	...	...	...
3123	2023-10-18	15:56	330.135
3124	2023-10-18	15:57	330.275
3125	2023-10-18	15:58	330.340
3126	2023-10-18	15:59	330.560
3127	2023-10-18	16:00	330.110

3128 rows x 3 columns

We opted to focus our analysis on 8 days of data specifically for the 15-minute interval for Microsoft stock. Accordingly, we extracted data from October 18, 2023, to October 27, 2023, using Python and saved it into another Excel file named "MSFT_15min.xlsx".

[55]:
```
# Assuming 'Date' is a datetime column in your DataFrame
stock_price['Date'] = pd.to_datetime(stock_price['Date'])

# Define your date range
start_date = pd.to_datetime('2023-10-18')  # Replace with your start date
end_date = pd.to_datetime('2023-10-27')    # Replace with your end date

# Extract rows within the specified date range
my_price_data = stock_price[(stock_price['Date'] >= start_date) & (stock_price['Date'] <= end_date)]
```

[56]:
```
my_price_data.dtypes
```

[56]:
```
Date           datetime64[ns]
Time                   object
Stock_Price           float64
dtype: object
```

[57]:
```
# 8 days data
my_price_data
```

[57]:

	Date	Time	Stock_Price
0	2023-10-27	09:30	331.120
1	2023-10-27	09:31	331.420
2	2023-10-27	09:32	331.840
3	2023-10-27	09:33	332.150
4	2023-10-27	09:34	331.660
...	...	...	...
3123	2023-10-18	15:56	330.135
3124	2023-10-18	15:57	330.275
3125	2023-10-18	15:58	330.340
3126	2023-10-18	15:59	330.560
3127	2023-10-18	16:00	330.110

3128 rows × 3 columns

[101]:
```
my_price_data_15min = my_price_data.groupby('Date').apply(lambda x: x.iloc[::15]).reset_index(drop=True)
```

[102]:
```
my_price_data_15min
```

[102]:

	Date	Time	Stock_Price
0	2023-10-18	09:30	333.500
1	2023-10-18	09:45	334.080
2	2023-10-18	10:00	334.950
3	2023-10-18	10:15	333.820
4	2023-10-18	10:30	333.580
...	...	...	...
211	2023-10-27	15:00	329.140
212	2023-10-27	15:15	329.785
213	2023-10-27	15:30	329.786
214	2023-10-27	15:45	329.084
215	2023-10-27	16:00	329.920

216 rows × 3 columns

In this file, we've organized the data for the 15-minute interval with the following columns: 'Date', 'Time', 'Stock_Price', 'Effective Bid-Ask Spread', 'Order Imbalance', 'Auto Correlation', 'Kyle's Lambda', 'PIN', and 'Volatility'.

```
[1]: import pandas as pd
     import numpy as np
```

```
[2]: my_15min_data = pd.read_excel("/Users/teja/Desktop/MSFT_15min.xlsx",sheet_name='15min')
```

```
[3]: my_15min_data
```

[3]:

	Date	Time	Stock_Price	Effective_bid_ask_spread	Order_Imbalance	Auto_Correlation	Kyle's_lambda	Pin	Volatility	Time_of_Day
0	2023-10-18	09:30	333.500	10.00	3700	50.950	0.000	39.6720	13.898	A
1	2023-10-18	09:45	334.080	6.98	0	-42.084	0.006	94.2030	11.758	B
2	2023-10-18	10:00	334.950	1.20	50	49.979	0.006	35.4135	8.986	B
3	2023-10-18	10:15	333.820	1.00	-100	30.344	0.004	17.5000	7.595	B
4	2023-10-18	10:30	333.580	2.00	-50	-7.225	0.004	15.2175	6.313	B
...	...	...	...	...	...	...	...	...	...	...
211	2023-10-27	15:00	329.140	4.99	-50	-35.999	0.003	70.0000	5.045	B
212	2023-10-27	15:15	329.785	4.00	-100	5.925	0.002	50.0000	4.549	B
213	2023-10-27	15:30	329.786	1.99	-100	-14.224	0.002	6.3490	7.374	B
214	2023-10-27	15:45	329.084	2.98	100	-30.680	0.004	-2.3200	5.481	B
215	2023-10-27	16:00	329.920	6.00	0	-54.243	0.003	0.0000	4.105	C

216 rows × 10 columns

We find the missing percentages of each column using python, and there are no missing values.

```
[7]: missing_percentages = data.isnull().mean() * 100
     print(missing_percentages)
```

```
Date                       0.0
Time                       0.0
Stock_Price                0.0
Effective_bid_ask_spread   0.0
Order_Imbalance            0.0
Auto_Correlation           0.0
Kyle's_lambda              0.0
Pin                        0.0
Volatility                 0.0
Time_of_Day                0.0
dtype: float64
```

Recognizing that all existing columns are numerical, we aimed to introduce a new categorical column named 'Time_of_Day'. This column distinguishes between different phases of the trading day: 'A' for the start (9:30), 'B' for the trading hours (9:45-15:45), and 'C' for the end (16:00). To achieve this, we utilized StandardScaler for normalization of numerical columns and OneHotEncoder for encoding categorical columns.

[8]:
```
categorical_columns = ['Time_of_Day']
numeric_columns = ['Effective_bid_ask_spread', 'Order_Imbalance', 'Auto_Correlation', "Kyle's_lambda", 'Pin', 'Volatility']
```

[9]:
```
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Normalize numeric columns
numeric_scaler = StandardScaler()
data[numeric_columns] = numeric_scaler.fit_transform(data[numeric_columns])

# Encoding categorical columns
categorical_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
encoded_time_of_day = pd.DataFrame(categorical_encoder.fit_transform(data[categorical_columns]),
                                   columns=categorical_encoder.get_feature_names_out(categorical_columns))
data = pd.concat([data, encoded_time_of_day], axis=1)
```

[10]:
```
# Drop the original categorical column
data.drop(['Time_of_Day'], axis=1, inplace=True)

# Drop the 'Date' and 'Time' column
data = data.drop(columns=['Date', 'Time'])
```

[11]:
```
data
```

[11]:

Stock_Price	Effective_bid_ask_spread	Order_Imbalance	Auto_Correlation	Kyle's_lambda	Pin	Volatility	Time_of_Day_A	Time_of_Day_B	Time_of_Day_C
333.500	1.083581	9.405514	2.279085	-2.113473	-0.050553	1.049544	1.0	0.0	0.0
334.080	0.623197	0.018206	-1.379706	1.914558	0.814298	0.721615	0.0	1.0	0.0
334.950	-0.257936	0.145062	2.240898	1.914558	-0.118092	0.296840	0.0	1.0	0.0
333.820	-0.288425	-0.235505	1.468703	0.571881	-0.402196	0.083686	0.0	1.0	0.0
333.580	-0.135980	-0.108649	-0.008790	0.571881	-0.438396	-0.112765	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...
329.140	0.319831	-0.108649	-1.140399	-0.099458	0.430443	-0.307070	0.0	1.0	0.0
329.785	0.168911	-0.235505	0.508366	-0.770796	0.113247	-0.383076	0.0	1.0	0.0
329.786	-0.137504	-0.235505	-0.284043	-0.770796	-0.579049	0.049821	0.0	1.0	0.0
329.084	0.013417	0.271917	-0.931216	0.571881	-0.716538	-0.240258	0.0	1.0	0.0
329.920	0.473801	0.018206	-1.857889	-0.099458	-0.679743	-0.451113	0.0	0.0	1.0

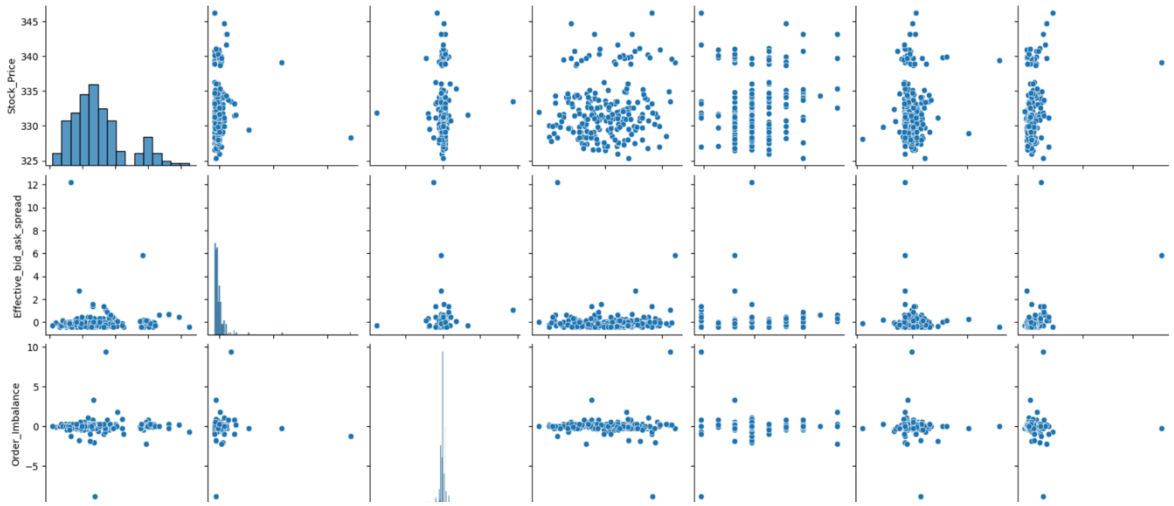
ws × 10 columns

We utilized Seaborn and Matplotlib libraries to plot the data and visualize trends

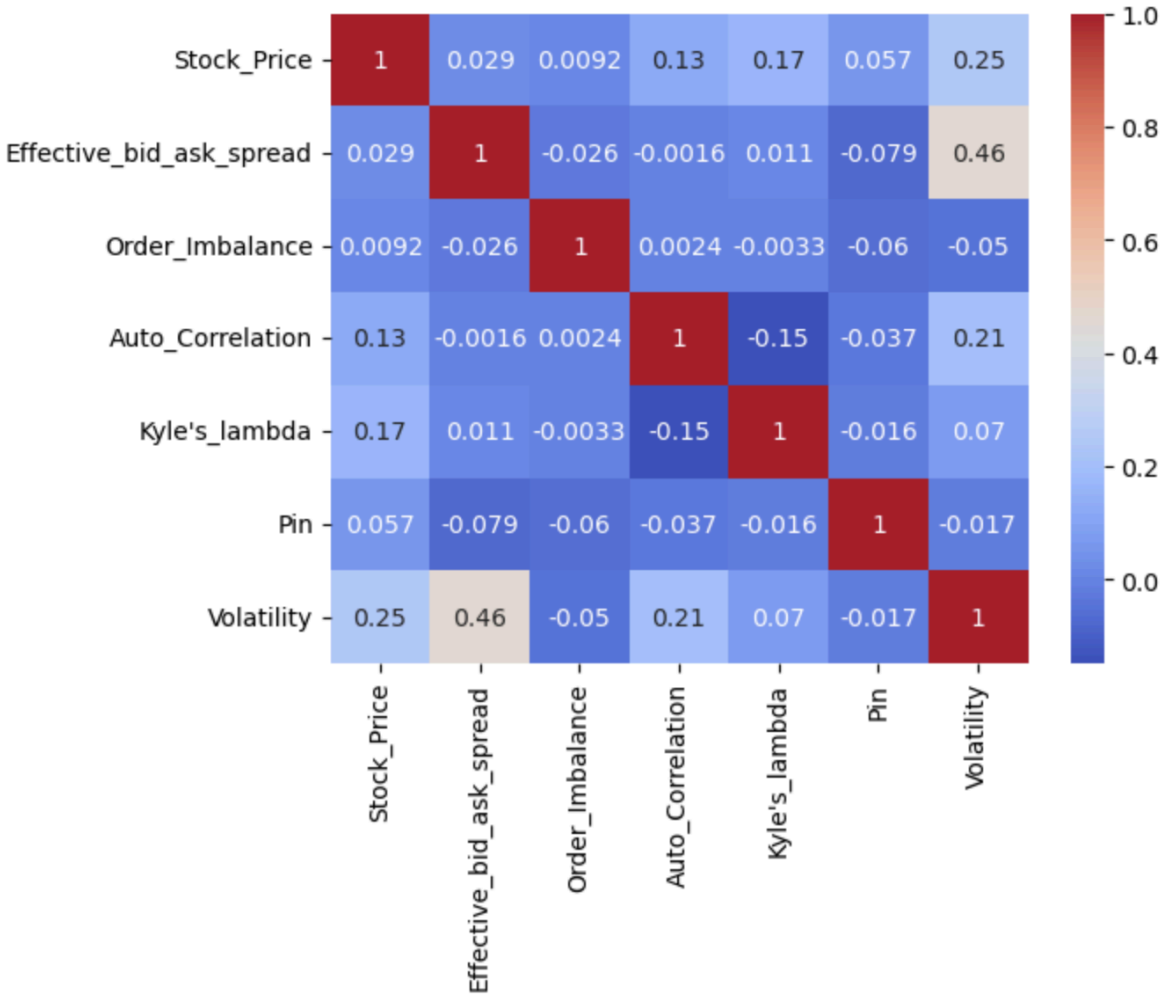
[12]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# EDA
sns.pairplot(data[['Stock_Price', 'Effective_bid_ask_spread', 'Order_Imbalance',
                   'Auto_Correlation', "Kyle's_lambda", 'Pin', 'Volatility']])
plt.show()

# Correlation matrix
correlation_matrix = data[['Stock_Price', 'Effective_bid_ask_spread', 'Order_Imbalance',
                           'Auto_Correlation', "Kyle's_lambda", 'Pin', 'Volatility']].corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.show()
```



Additionally, we generated a correlation matrix to examine the relationships between different variables



Employing various machine learning techniques, we conducted further analysis of the data to gain insights and make predictions.

We train the model using 75% of the data for training and 25% for testing. We used Q-Learning, a Reinforcement Learning algorithm, RandomForestRegressor and LSTM.

Q-Learning

This code implements a Q-learning algorithm for stock price prediction using reinforcement learning. Here's a brief explanation of the key components:

1. Initialization: The code initializes the Q-table with zeros, which stores the expected cumulative rewards for each state-action pair.

2. Training: The algorithm iterates through a fixed number of episodes and updates the Q-values based on observed rewards. Within each episode, it iterates through each state and selects actions according to an epsilon-greedy policy, balancing exploration and exploitation. The reward is calculated based on the difference between the predicted and actual stock price changes. The Q-values are updated using the Bellman equation.

3. Testing: After training, the algorithm is tested on unseen data. For each state, it selects the action with the highest Q-value and predicts the next stock price change. The predicted stock prices are compared with the actual stock prices to evaluate the performance of the model.

4. Results: The predicted and actual stock prices are stored in a DataFrame for comparison and analysis.

Overall, this code demonstrates a basic implementation of Q-learning for stock price prediction, where the agent learns to make decisions based on historical stock price data and updates its strategy over time to maximize cumulative rewards.

```
# Initialize Q-table with zeros
Q = np.zeros((num_states, num_actions), dtype=float)

learning_rate = 1.0
discount_factor = 0.2
epsilon = 0.2
num_episodes = 1000

def epsilon_greedy_policy(Q, state_index, epsilon):
    if np.random.rand() < epsilon:
        # Exploration: Choose a random action
        return np.random.randint(len(Q[state_index, :]))
    else:
        # Exploitation: Choose the action with the highest Q-value
        return np.argmax(Q[state_index, :])

# Training
for episode in range(num_episodes):
    for state_index in range(num_states - 1):
        state = state_index  # Use the index as the state
        action = epsilon_greedy_policy(Q, state, epsilon)
        next_state = state_index + 1
        stock_price_2 = data.iloc[next_state]['Stock_Price']
        stock_price_1 = data.iloc[state_index]['Stock_Price']
        actual_stock_price_change = stock_price_2 - stock_price_1
        percent_stock_price_change = (actual_stock_price_change / stock_price_1) * 100

        # Calculate reward based on the difference between predicted and actual stock price change
        predicted_stock_price_change = (action - 1)  # Assuming actions are -1, 0, 1
        predicted_stock_price = stock_price_1 + predicted_stock_price_change

        # Calculate reward
        if predicted_stock_price <= stock_price_2:
            reward = actual_stock_price_change - abs(predicted_stock_price_change)
        else:
            reward = -actual_stock_price_change

        # Update Q-value using the Bellman equation
        Q[state_index, action] = (1 - learning_rate) * Q[state_index, action] + \
                                 learning_rate * (reward + discount_factor * np.max(Q[next_state, :]))
```

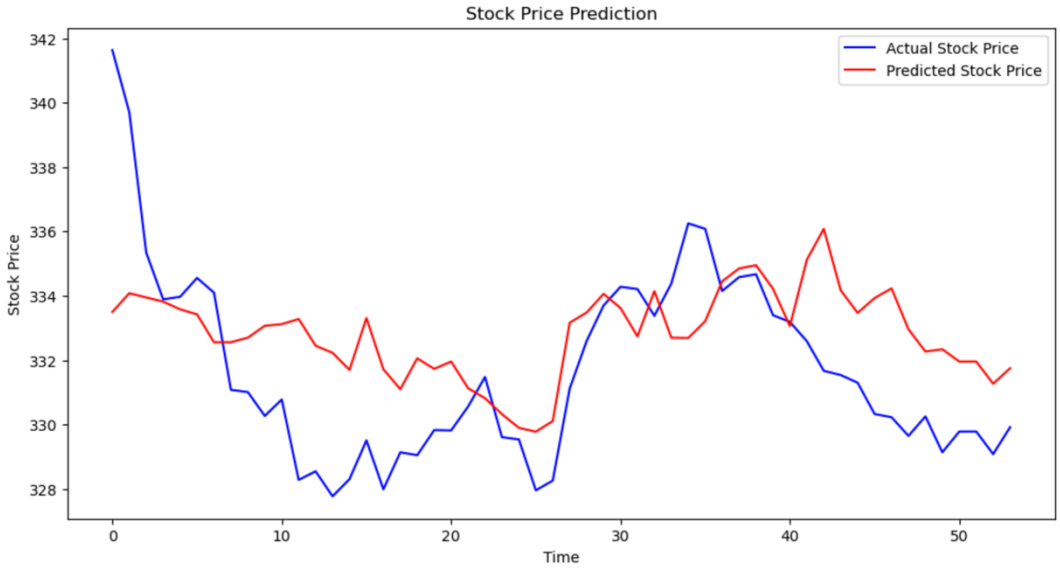
[15]:
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Assuming 'actual_values' and 'predicted_stock_prices' are your actual and predicted values
actual_values = result_df['Stock_Price'].values
predicted_stock_prices = result_df['Predicted_Stock_Price'].values
mae = mean_absolute_error(actual_values, predicted_stock_prices)
mse = mean_squared_error(actual_values, predicted_stock_prices)
rmse = np.sqrt(mse)
r2 = r2_score(actual_values, predicted_stock_prices)

print(f'MAE: {mae}')
print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
print(f'R2: {r2}')
```

```
MAE: 2.188300000000002
MSE: 7.280401490000006
RMSE: 2.698221912667675
R2: 0.14302324855540582
```

[14]:
```
# Visualize the results
plt.figure(figsize=(12, 6))
plt.plot(data.iloc[162:]['Stock_Price'].values, label='Actual Stock Price', color='blue')
plt.plot(predicted_stock_prices, label='Predicted Stock Price', color='red')
plt.title('Stock Price Prediction')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.legend()
plt.show()
```



The provided code trains a RandomForestRegressor model for predicting stock prices and evaluates its performance using the R-squared (R2) score. Here's a brief explanation of each section:

Without Grid Search:

1. Train RandomForestRegressor: The RandomForestRegressor model is instantiated and trained using the training data (`X_train_rf`, `y_train_rf`), where `X_train_rf` contains the feature columns and `y_train_rf` contains the target variable (stock prices).

2. Testing RandomForestRegressor: The trained model is used to make predictions on the test data (`X_test_rf`). The actual stock prices from the test data are compared with the predicted prices (`y_predicted_rf`).

3. Evaluate the Model: The R2 score is calculated to evaluate the performance of the RandomForestRegressor model. The R2 score measures the proportion of the variance in the dependent variable (stock prices) that is predictable from the independent variables (features).

```
# Split the data into training and testing sets
train_data, test_data = train_test_split(data, test_size=0.25, random_state=42)

# Step 1: Train RandomForestRegressor for initial prediction
rf_model = RandomForestRegressor()
X_train_rf = train_data[feature_columns]
y_train_rf = train_data['Stock_Price']
rf_model.fit(X_train_rf, y_train_rf)

# Testing RandomForestRegressor
X_test_rf = test_data[feature_columns]
y_test_actual = test_data['Stock_Price']
y_predicted_rf = rf_model.predict(X_test_rf)

# Evaluate the RandomForestRegressor model
r2_rf = r2_score(y_test_actual, y_predicted_rf)
print(f'R2 Score (Random Forest): {r2_rf}')
```

With Grid Search (Second Part):

1. Split Data: The data is split into training and testing sets using the `train_test_split` function from `sklearn.model_selection`. This split allows for model evaluation on unseen data.

2. Train RandomForestRegressor with GridSearchCV: The RandomForestRegressor model is trained with hyperparameter tuning using GridSearchCV. Hyperparameters like the number of estimators, maximum depth, minimum samples split, minimum samples leaf, and maximum features are tuned to optimize the model's performance.

3. Testing RandomForestRegressor: The best model obtained from GridSearchCV is used to make predictions on the test data (`X_test_rf`). The actual stock prices from the test data are compared with the predicted prices.

4. Evaluate the Model: The predictions are stored in a DataFrame (`result_df`) along with the actual values. This DataFrame allows for comparison between the predicted and actual stock prices.In summary, the code demonstrates how to train and evaluate RandomForestRegressor models for stock price prediction, both with and without hyperparameter tuning using GridSearchCV.

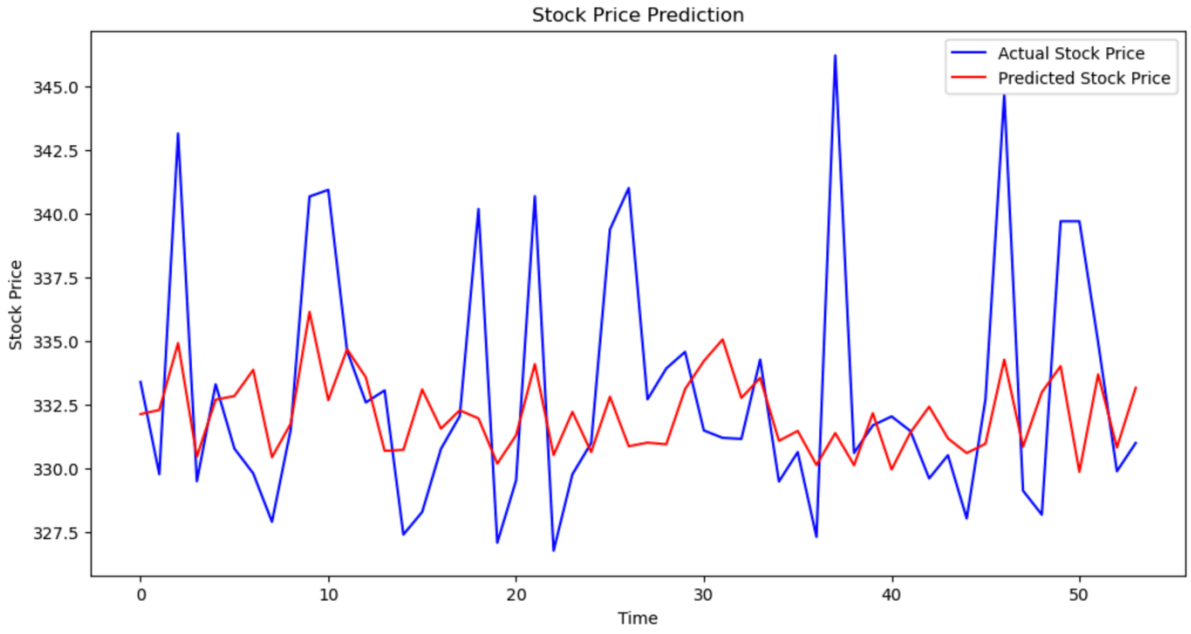
```python
# Step 1: Train RandomForestRegressor with fine-tuning using GridSearchCV
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [1.0]  # Explicitly set max_features to 1.0
}

rf_model = RandomForestRegressor(random_state=42)
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid, cv=5, scoring='r2', n_jobs=-1)
grid_search.fit(train_data[feature_columns], train_data['Stock_Price'])

# Get the best RandomForestRegressor model
best_rf_model = grid_search.best_estimator_

# Step 2: Testing RandomForestRegressor
y_test_actual = test_data['Stock_Price']
y_predicted_rf = best_rf_model.predict(test_data[feature_columns])

# Create a DataFrame with actual and predicted values
result_df = pd.DataFrame({'Actual': y_test_actual, 'Predicted': y_predicted_rf})
```



This code implements a Long Short-Term Memory (LSTM) neural network model for predicting stock prices using historical price data and additional features. Here's a summary of the key steps:

1. Data Preprocessing:

- Feature columns and the 'Stock_Price' column are extracted from the DataFrame 'data'.

- The data is normalized using Min-Max scaling to ensure all features and stock prices are within the range [0, 1].

2. Prepare Data for LSTM:

- The data is transformed into sequences of features and corresponding stock prices, with a defined look-back window (number of previous time steps to consider).

- Each sequence is flattened and concatenated with the corresponding stock prices to create the input data for the LSTM model.

3. Model Building:

- A Sequential model is initialized.

- An LSTM layer with 50 units is added as the input layer, specifying the input shape.

- A Dense layer with 1 unit (output layer) is added to predict the next stock price.

- The model is compiled using the Adam optimizer and Mean Squared Error (MSE) loss function.

4. Model Training:

- The model is trained on the training data (X_train, y_train) for 50 epochs with a batch size of 32.

5. Prediction:

- The trained model is used to make predictions on the test data (X_test).

- The predicted stock prices are transformed back to the original scale using the inverse Min-Max scaler.

6. Evaluation:

- Mean Squared Error (MSE) could be calculated to evaluate the performance of the model by comparing the predicted stock prices with the actual stock prices.

Overall, this code demonstrates the implementation of an LSTM neural network for stock price prediction, incorporating additional features alongside historical stock prices. The model learns patterns in the data to make predictions about future stock prices.

```
# Split the data into training and testing sets
train_size = int(len(X) * 0.75)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Build the LSTM model
model = Sequential()
model.add(LSTM(units=50, input_shape=(X_train.shape[1], X_train.shape[2])))
model.add(Dense(units=1))
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=32)

# Make predictions on the test data
y_pred = model.predict(X_test)
```