Machine Learning Trading Bot

In this Challenge, you'll assume the role of a financial advisor at one of the top five financial advisory firms in the world. Your firm constantly competes with the other major firms to manage and automatically trade assets in a highly dynamic environment. In recent years, your firm has heavily profited by using computer algorithms that can buy and sell faster than human traders.

The speed of these transactions gave your firm a competitive advantage early on. But, people still need to specifically program these systems, which limits their ability to adapt to new data. You're thus planning to improve the existing algorithmic trading systems and maintain the firm's competitive advantage in the market. To do so, you'll enhance the existing trading signals with machine learning algorithms that can adapt to new data.

Instructions:

Use the starter code file to complete the steps that the instructions outline. The steps for this Challenge are divided into the following sections:

- Establish a Baseline Performance
- Tune the Baseline Trading Algorithm
- Evaluate a New Machine Learning Classifier
- Create an Evaluation Report

Establish a Baseline Performance

In this section, you'll run the provided starter code to establish a baseline performance for the trading algorithm. To do so, complete the following steps.

Open the Jupyter notebook. Restart the kernel, run the provided cells that correspond with the first three steps, and then proceed to step four.

1. Import the OHLCV dataset into a Pandas DataFrame.

2. Generate trading signals using short- and long-window SMA values.

3. Split the data into training and testing datasets.

4. Use the `SVC` classifier model from SKLearn's support vector machine (SVM) learning method to fit the training data and make predictions based on the testing data. Review the predictions.

5. Review the classification report associated with the `SVC` model predictions.

6. Create a predictions DataFrame that contains columns for “Predicted” values, “Actual Returns”, and “Strategy Returns”.

7. Create a cumulative return plot that shows the actual returns vs. the strategy returns. Save a PNG image of this plot. This will serve as a baseline against which to compare the effects of tuning the trading algorithm.

8. Write your conclusions about the performance of the baseline trading algorithm in the `README.md` file that’s associated with your GitHub repository. Support your findings by using the PNG image that you saved in the previous step.

Tune the Baseline Trading Algorithm

In this section, you’ll tune, or adjust, the model’s input features to find the parameters that result in the best trading outcomes. (You’ll choose the best by comparing the cumulative products of the strategy returns.) To do so, complete the following steps:

1. Tune the training algorithm by adjusting the size of the training dataset. To do so, slice your data into different periods. Rerun the notebook with the updated parameters, and record the results in your `README.md` file. Answer the following question: What impact resulted from increasing or decreasing the training window?

> **Hint** To adjust the size of the training dataset, you can use a different `DateOffset` value—for example, six months. Be aware that changing the size of the training dataset also affects the size of the testing dataset.

1. Tune the trading algorithm by adjusting the SMA input features. Adjust one or both of the windows for the algorithm. Rerun the notebook with the updated parameters, and record the results in your `README.md` file. Answer the following question: What impact resulted from increasing or decreasing either or both of the SMA windows?

2. Choose the set of parameters that best improved the trading algorithm returns. Save a PNG image of the cumulative product of the actual returns vs. the strategy returns, and document your conclusion in your `README.md` file.

Evaluate a New Machine Learning Classifier

In this section, you'll use the original parameters that the starter code provided. But, you'll apply them to the performance of a second machine learning model. To do so, complete the following steps:

1. Import a new classifier, such as `AdaBoost`, `DecisionTreeClassifier`, or `LogisticRegression`. (For the full list of classifiers, refer to the Supervised learning page in the scikit-learn documentation.)

2. Using the original training data as the baseline model, fit another model with the new classifier.

3. Backtest the new model to evaluate its performance. Save a PNG image of the cumulative product of the actual returns vs. the strategy returns for this updated trading algorithm, and write your conclusions in your `README.md` file. Answer the following questions: Did this new model perform better or worse than the provided baseline model? Did this new model perform better or worse than your tuned trading algorithm?

Create an Evaluation Report

In the previous sections, you updated your `README.md` file with your conclusions. To accomplish this section, you need to add a summary evaluation report at the end of the `README.md` file. For this report, express your final conclusions and analysis. Support your findings by using the PNG images that you created.

In [1]:
```
# Imports
import pandas as pd
import numpy as np
from pathlib import Path
import hvplot.pandas
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.preprocessing import StandardScaler
from pandas.tseries.offsets import DateOffset
from sklearn.metrics import classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

Establish a Baseline Performance

In this section, you'll run the provided starter code to establish a baseline performance for the trading algorithm. To do so, complete the following steps.

Open the Jupyter notebook. Restart the kernel, run the provided cells that correspond with the first three steps, and then proceed to step four.

Step 1: Import the OHLCV dataset into a Pandas DataFrame.

In [2]:
```
# Import the OHLCV dataset into a Pandas Dataframe
ohlcv_df = pd.read_csv(
    Path("./Resources/emerging_markets_ohlcv.csv"),
    index_col='date',
    infer_datetime_format=True,
    parse_dates=True
)

# Review the DataFrame
ohlcv_df.head()
```

Out[2]:

date	open	high	low	close	volume
2015-01-21 09:30:00	23.83	23.83	23.83	23.83	100
2015-01-21 11:00:00	23.98	23.98	23.98	23.98	100
2015-01-22 15:00:00	24.42	24.42	24.42	24.42	100
2015-01-22 15:15:00	24.42	24.44	24.42	24.44	200
2015-01-22 15:30:00	24.46	24.46	24.46	24.46	200

In [3]:
```
# Filter the date index and close columns
signals_df = ohlcv_df.loc[:, ["close"]]

# Use the pct_change function to generate returns from close prices
signals_df["Actual Returns"] = signals_df["close"].pct_change()

# Drop all NaN values from the DataFrame
signals_df = signals_df.dropna()
signals_df
```

```
# Review the DataFrame
display(signals_df.head())
display(signals_df.tail())
```

	close	Actual Returns
date		
2015-01-21 11:00:00	23.98	0.006295
2015-01-22 15:00:00	24.42	0.018349
2015-01-22 15:15:00	24.44	0.000819
2015-01-22 15:30:00	24.46	0.000818
2015-01-26 12:30:00	24.33	-0.005315

	close	Actual Returns
date		
2021-01-22 09:30:00	33.27	-0.006866
2021-01-22 11:30:00	33.35	0.002405
2021-01-22 13:45:00	33.42	0.002099
2021-01-22 14:30:00	33.47	0.001496
2021-01-22 15:45:00	33.44	-0.000896

Step 2: Generate trading signals using short- and long-window SMA values.

In [4]:
```
# Set the short window and long window
short_window = 4
long_window = 100

# Generate the fast and slow simple moving averages (4 and 100 days, respectively)
signals_df['SMA_Fast'] = signals_df['close'].rolling(window=short_window).mean()
signals_df['SMA_Slow'] = signals_df['close'].rolling(window=long_window).mean()

signals_df = signals_df.dropna()
```

```
# Review the DataFrame
display(signals_df.head())
display(signals_df.tail())
```

date	close	Actual Returns	SMA_Fast	SMA_Slow
2015-04-02 14:45:00	24.92	0.000000	24.9175	24.3214
2015-04-02 15:00:00	24.92	0.000000	24.9200	24.3308
2015-04-02 15:15:00	24.94	0.000803	24.9250	24.3360
2015-04-02 15:30:00	24.95	0.000401	24.9325	24.3411
2015-04-02 15:45:00	24.98	0.001202	24.9475	24.3463

date	close	Actual Returns	SMA_Fast	SMA_Slow
2021-01-22 09:30:00	33.27	-0.006866	33.2025	30.40215
2021-01-22 11:30:00	33.35	0.002405	33.2725	30.44445
2021-01-22 13:45:00	33.42	0.002099	33.3850	30.48745
2021-01-22 14:30:00	33.47	0.001496	33.3775	30.53085
2021-01-22 15:45:00	33.44	-0.000896	33.4200	30.57495

In [5]:

```
# Initialize the new Signal column
signals_df['Signal'] = 0.0

# When Actual Returns are greater than or equal to 0, generate signal to buy stock long
signals_df.loc[(signals_df['Actual Returns'] >= 0), 'Signal'] = 1

# When Actual Returns are less than 0, generate signal to sell stock short
signals_df.loc[(signals_df['Actual Returns'] < 0), 'Signal'] = -1

# Review the DataFrame
display(signals_df.head())
display(signals_df.tail())
```

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal
2015-04-02 14:45:00	24.92	0.000000	24.9175	24.3214	1.0
2015-04-02 15:00:00	24.92	0.000000	24.9200	24.3308	1.0
2015-04-02 15:15:00	24.94	0.000803	24.9250	24.3360	1.0
2015-04-02 15:30:00	24.95	0.000401	24.9325	24.3411	1.0
2015-04-02 15:45:00	24.98	0.001202	24.9475	24.3463	1.0

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal
2021-01-22 09:30:00	33.27	-0.006866	33.2025	30.40215	-1.0
2021-01-22 11:30:00	33.35	0.002405	33.2725	30.44445	1.0
2021-01-22 13:45:00	33.42	0.002099	33.3850	30.48745	1.0
2021-01-22 14:30:00	33.47	0.001496	33.3775	30.53085	1.0
2021-01-22 15:45:00	33.44	-0.000896	33.4200	30.57495	-1.0

In [6]:
```
signals_df['Signal'].value_counts()
```

Out[6]:
```
 1.0    2368
-1.0    1855
Name: Signal, dtype: int64
```

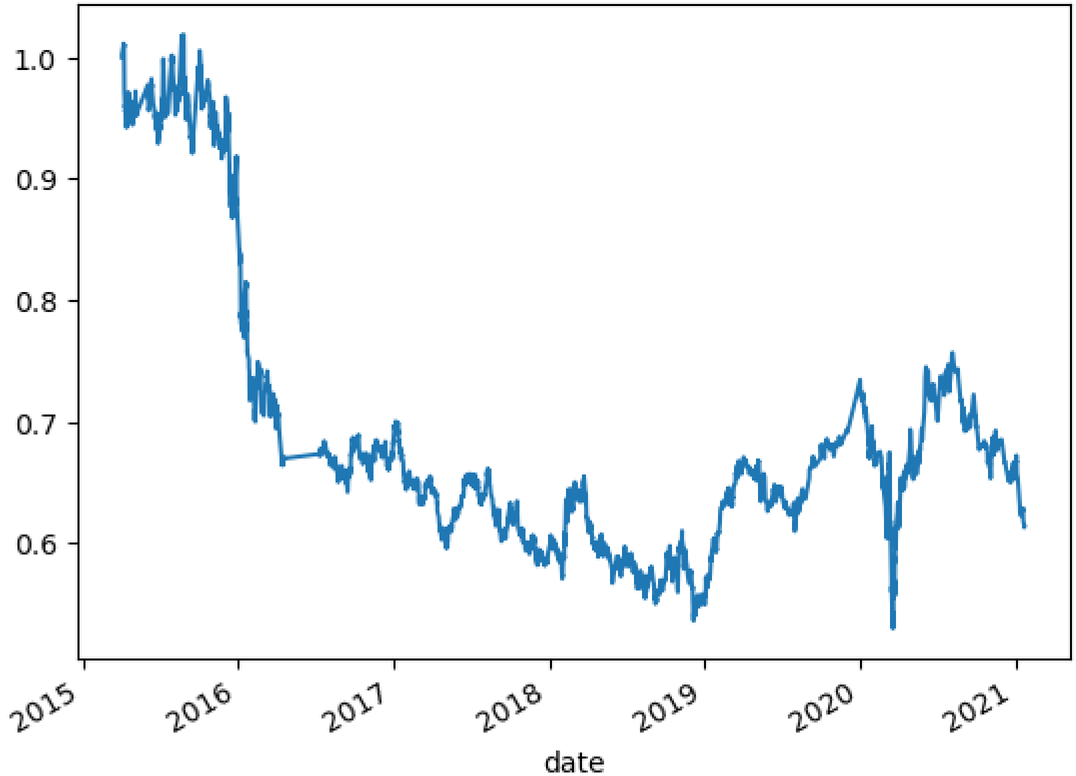
In [7]:
```
# Calculate the strategy returns and add them to the signals_df DataFrame
signals_df['Strategy Returns'] = signals_df['Actual Returns'] * signals_df['Signal'].shift()

# Review the DataFrame
display(signals_df.head())
display(signals_df.tail())
```

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns
2015-04-02 14:45:00	24.92	0.000000	24.9175	24.3214	1.0	NaN
2015-04-02 15:00:00	24.92	0.000000	24.9200	24.3308	1.0	0.000000
2015-04-02 15:15:00	24.94	0.000803	24.9250	24.3360	1.0	0.000803
2015-04-02 15:30:00	24.95	0.000401	24.9325	24.3411	1.0	0.000401
2015-04-02 15:45:00	24.98	0.001202	24.9475	24.3463	1.0	0.001202

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns
2021-01-22 09:30:00	33.27	-0.006866	33.2025	30.40215	-1.0	-0.006866
2021-01-22 11:30:00	33.35	0.002405	33.2725	30.44445	1.0	-0.002405
2021-01-22 13:45:00	33.42	0.002099	33.3850	30.48745	1.0	0.002099
2021-01-22 14:30:00	33.47	0.001496	33.3775	30.53085	1.0	0.001496
2021-01-22 15:45:00	33.44	-0.000896	33.4200	30.57495	-1.0	-0.000896

In [8]:
```
# Plot Strategy Returns to examine performance
(1 + signals_df['Strategy Returns']).cumprod().plot()
```

Out[8]:
```
<AxesSubplot:xlabel='date'>
```



Step 3: Split the data into training and testing datasets.

In [9]:
```
# Assign a copy of the sma_fast and sma_slow columns to a features DataFrame called X
X = signals_df[['SMA_Fast', 'SMA_Slow']].shift().dropna()

# Review the DataFrame
X.head()
```

Out[9]:

date	SMA_Fast	SMA_Slow
2015-04-02 15:00:00	24.9175	24.3214
2015-04-02 15:15:00	24.9200	24.3308
2015-04-02 15:30:00	24.9250	24.3360
2015-04-02 15:45:00	24.9325	24.3411
2015-04-06 09:30:00	24.9475	24.3463

In [10]:

```
# Create the target set selecting the Signal column and assiging it to y
y = signals_df['Signal']

# Review the value counts
y.value_counts()
```

Out[10]:

```
 1.0    2368
-1.0    1855
Name: Signal, dtype: int64
```

In [11]:

```
# Select the start of the training period
training_begin = X.index.min()

# Display the training begin date
print(training_begin)
```

```
2015-04-02 15:00:00
```

In [12]:

```
# Select the ending period for the training data with an offset of 3 months
training_end = X.index.min() + DateOffset(months=3)

# Display the training end date
print(training_end)
```

```
2015-07-02 15:00:00
```

In [13]:

```
# Generate the X_train and y_train DataFrames
X_train = X.loc[training_begin:training_end]
y_train = y.loc[training_begin:training_end]

# Review the X_train DataFrame
X_train.head()
```

Out[13]:

date	SMA_Fast	SMA_Slow
2015-04-02 15:00:00	24.9175	24.3214
2015-04-02 15:15:00	24.9200	24.3308
2015-04-02 15:30:00	24.9250	24.3360
2015-04-02 15:45:00	24.9325	24.3411
2015-04-06 09:30:00	24.9475	24.3463

In [14]:
```
X_train.shape
```
Out[14]:
```
(128, 2)
```

In [15]:
```
# Generate the X_test and y_test DataFrames
X_test = X.loc[training_end+DateOffset(hours=1):]
y_test = y.loc[training_end+DateOffset(hours=1):]

# Review the X_test DataFrame
X_test.head()
```

Out[15]:

date	SMA_Fast	SMA_Slow
2015-07-06 10:00:00	24.1250	25.0919
2015-07-06 10:45:00	23.9700	25.0682
2015-07-06 14:15:00	23.8475	25.0458
2015-07-06 14:30:00	23.6725	25.0206
2015-07-07 11:30:00	23.4800	24.9951

In [16]:
```
X_test.shape
```
Out[16]:
```
(4092, 2)
```

In [17]:
```
# Scale the features DataFrames
# Create a StandardScaler instance
```

```python
scaler = StandardScaler()

# Apply the scaler model to fit the X-train data
X_scaler = scaler.fit(X_train)

# Transform the X_train and X_test DataFrames using the X_scaler
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

In [18]:
```python
X_train_scaled.shape
```

Out[18]:
```
(128, 2)
```

In [19]:
```python
X_test_scaled.shape
```

Out[19]:
```
(4092, 2)
```

Step 4: Use the `SVC` classifier model from SKLearn's support vector machine (SVM) learning method to fit the training data and make predictions based on the testing data. Review the predictions.

In [20]:
```python
# From SVM, instantiate SVC classifier model instance
svm_model = svm.SVC()

# Fit the model to the data using the training data
svm_model = svm_model.fit(X_train_scaled, y_train)

# Use the testing data to make the model predictions
svm_pred =  svm_model.predict(X_test_scaled)

# Review the model's predicted values
svm_pred[:10]
```

Out[20]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [21]:
```python
svm_pred.shape
```

Out[21]:
```
(4092,)
```

Step 5: Review the classification report associated with the `SVC` model predictions.

In [22]:
```
# Use a classification report to evaluate the model using the predictions and testing data
svm_testing_report = classification_report(y_test, svm_pred)

# Print the classification report
print(svm_testing_report)
```

```
              precision    recall  f1-score   support

        -1.0       0.43      0.04      0.07      1804
         1.0       0.56      0.96      0.71      2288

    accuracy                           0.55      4092
   macro avg       0.49      0.50      0.39      4092
weighted avg       0.50      0.55      0.43      4092
```

Step 6: Create a predictions DataFrame that contains columns for “Predicted” values, “Actual Returns”, and “Strategy Returns”.

In [23]:
```
# Create a predictions DataFrame
predictions_df = pd.DataFrame(index=X_test.index)
```

In [24]:
```
print(len(signals_df))
print(len(predictions_df))
```

```
4223
4092
```

In [25]:
```
print(signals_df.index.equals(predictions_df.index))
print(len(signals_df) == len(predictions_df))
```

```
False
False
```

In [26]:
```
print(signals_df.index)
print(predictions_df.index)
```

```
DatetimeIndex(['2015-04-02 14:45:00', '2015-04-02 15:00:00',
               '2015-04-02 15:15:00', '2015-04-02 15:30:00',
               '2015-04-02 15:45:00', '2015-04-06 09:30:00',
               '2015-04-06 09:45:00', '2015-04-06 10:15:00',
               '2015-04-06 11:45:00', '2015-04-06 12:00:00',
               ...
               '2021-01-14 15:45:00', '2021-01-19 09:30:00',
               '2021-01-19 11:15:00', '2021-01-19 12:30:00',
               '2021-01-20 09:45:00', '2021-01-22 09:30:00',
               '2021-01-22 11:30:00', '2021-01-22 13:45:00',
               '2021-01-22 14:30:00', '2021-01-22 15:45:00'],
              dtype='datetime64[ns]', name='date', length=4223, freq=None)
DatetimeIndex(['2015-07-06 10:00:00', '2015-07-06 10:45:00',
               '2015-07-06 14:15:00', '2015-07-06 14:30:00',
               '2015-07-07 11:30:00', '2015-07-07 13:45:00',
               '2015-07-07 15:00:00', '2015-07-07 15:45:00',
               '2015-07-08 10:00:00', '2015-07-08 12:00:00',
               ...
               '2021-01-14 15:45:00', '2021-01-19 09:30:00',
               '2021-01-19 11:15:00', '2021-01-19 12:30:00',
               '2021-01-20 09:45:00', '2021-01-22 09:30:00',
               '2021-01-22 11:30:00', '2021-01-22 13:45:00',
               '2021-01-22 14:30:00', '2021-01-22 15:45:00'],
              dtype='datetime64[ns]', name='date', length=4092, freq=None)
```

In [27]:

```
print(signals_df.index.equals(predictions_df.index))
print(len(signals_df) == len(predictions_df))
```

```
False
False
```

In [28]:

```
predictions_df["predicted_signal"] = svm_pred
# Add the actual returns to the DataFrame
predictions_df["Actual Returns"] = signals_df["Actual Returns"]
# Add the strategy returns to the DataFrame
predictions_df["Strategy Returns"] = (signals_df["Actual Returns"] * predictions_df["predicted_signal"])

# Review the DataFrame
predictions_df.head()
```

Out[28]:

date	predicted_signal	Actual Returns	Strategy Returns
2015-07-06 10:00:00	1.0	-0.025715	-0.025715
2015-07-06 10:45:00	1.0	0.007237	0.007237
2015-07-06 14:15:00	1.0	-0.009721	-0.009721
2015-07-06 14:30:00	1.0	-0.003841	-0.003841
2015-07-07 11:30:00	1.0	-0.018423	-0.018423

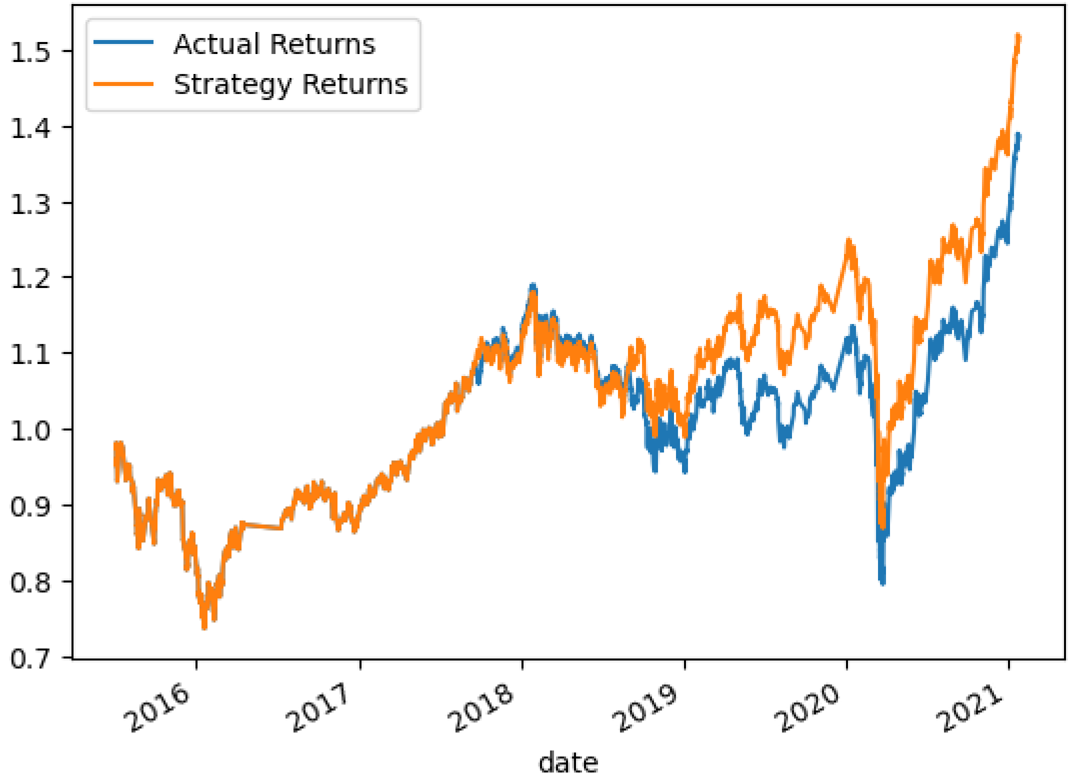
Step 7: Create a cumulative return plot that shows the actual returns vs. the strategy returns. Save a PNG image of this plot. This will serve as a baseline against which to compare the effects of tuning the trading algorithm.

In [29]:

```
# Plot the actual returns versus the strategy returns
(1 + predictions_df[["Actual Returns", "Strategy Returns"]]).cumprod().plot()
```

Out[29]:

```
<AxesSubplot:xlabel='date'>
```



Tune the Baseline Trading Algorithm

In this section, you'll tune, or adjust, the model's input features to find the parameters that result in the best trading outcomes. You'll choose the best by comparing the cumulative products of the strategy returns.

Step 1: Tune the training algorithm by adjusting the size of the training dataset.

To do so, slice your data into different periods. Rerun the notebook with the updated parameters, and record the results in your `README.md` file.

Answer the following question: What impact resulted from increasing or decreasing the training window?

In [30]:
```
# Select the ending period for the training data with an offset of 6 months
training_endnew = X.index.min() + DateOffset(months=6)

# Display the training end date
print(training_endnew)
```

```
2015-10-02 15:00:00
```

In [31]:
```
# Generate the X_train and y_train DataFrames
X_trainn = X.loc[training_begin:training_endnew]
y_trainn = y.loc[training_begin:training_endnew]

# Review the X_train DataFrame
X_trainn.head()
X_trainn.tail()
```

Out[31]:

date	SMA_Fast	SMA_Slow
2015-09-30 14:45:00	21.4800	21.6888
2015-10-02 09:30:00	21.2325	21.6672
2015-10-02 10:30:00	20.9875	21.6465
2015-10-02 11:30:00	20.7775	21.6293
2015-10-02 14:45:00	20.9450	21.6144

In [32]:
```
X_trainn.shape
```

Out[32]:
```
(276, 2)
```

In [33]:
```
# Generate the X_test and y_test DataFrames
X_testn = X.loc[training_endnew+DateOffset(hours=1):]
y_testn = y.loc[training_endnew+DateOffset(hours=1):]

# Review the X_test DataFrame
X_testn.head()
```

Out[33]:

date	SMA_Fast	SMA_Slow
2015-10-05 09:45:00	21.42725	21.56409
2015-10-05 11:30:00	21.53225	21.55469
2015-10-05 13:15:00	21.60250	21.54289
2015-10-05 14:30:00	21.66750	21.53089
2015-10-05 14:45:00	21.75250	21.51939

In [34]:

```
X_testn.shape
```

Out[34]:

```
(3943, 2)
```

In [35]:

```
# Scale the features DataFrames
# Create a StandardScaler instance
scaler = StandardScaler()

# Apply the scaler model to fit the X-train data
X_scalern = scaler.fit(X_trainn)

# Transform the X_train and X_test DataFrames using the X_scaler
X_train_scaledn = X_scalern.transform(X_trainn)
X_test_scaledn = X_scalern.transform(X_testn)
```

In [36]:

```
# From SVM, instantiate SVC classifier model instance
svm_model = svm.SVC()

# Fit the model to the data using the training data
svm_model = svm_model.fit(X_train_scaledn, y_trainn)

# Use the testing data to make the model predictions
svm_predn =  svm_model.predict(X_test_scaledn)

# Review the model's predicted values
svm_predn[:10]
```

Out[36]:

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

In [37]:
```python
# Use a classification report to evaluate the model using the predictions and testing data
svm_testing_reportn = classification_report(y_testn, svm_predn)

# Print the classification report
print(svm_testing_reportn)
```

```
              precision    recall  f1-score   support

        -1.0       0.44      0.02      0.04      1732
         1.0       0.56      0.98      0.71      2211

    accuracy                           0.56      3943
   macro avg       0.50      0.50      0.38      3943
weighted avg       0.51      0.56      0.42      3943
```

In [38]:
```python
# Create a predictions DataFrame
predictions_dfn = pd.DataFrame(index=X_testn.index)
```

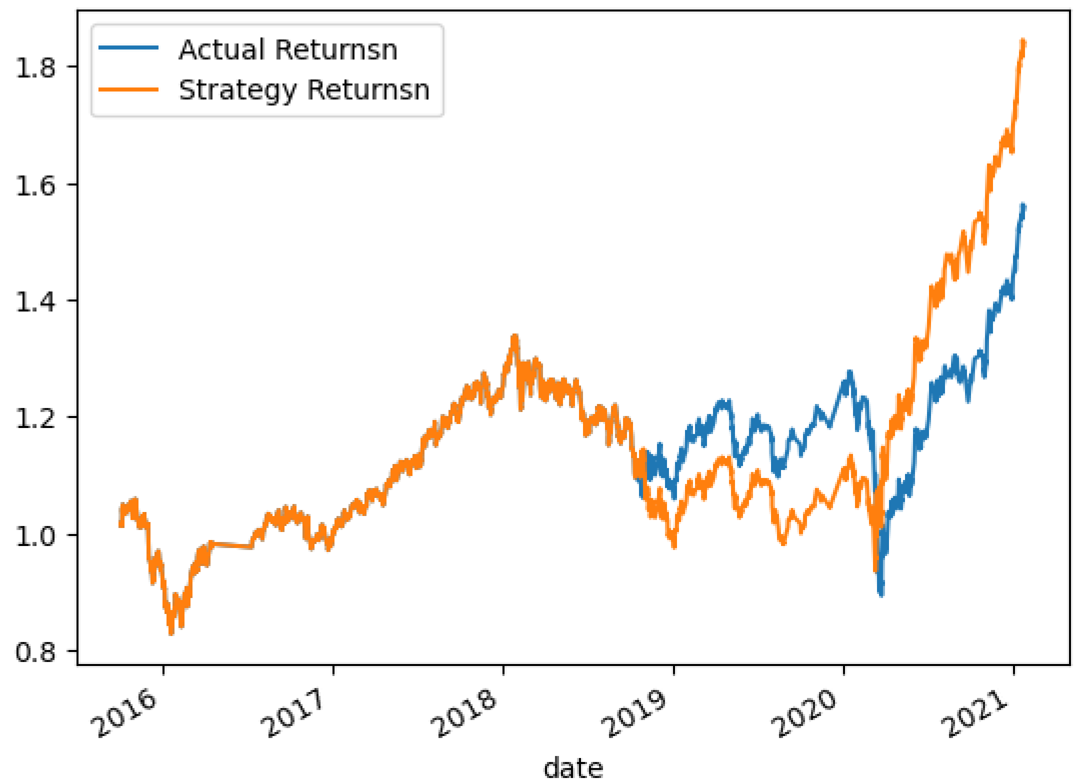
In [39]:
```python
predictions_dfn["predicted_signaln"] = svm_predn
# Add the actual returns to the DataFrame
predictions_dfn["Actual Returnsn"] = signals_df["Actual Returns"]
# Add the strategy returns to the DataFrame
predictions_dfn["Strategy Returnsn"] = (signals_df["Actual Returns"] * predictions_dfn["predicted_signaln"])

# Review the DataFrame
predictions_dfn.head()
```

Out[39]:

date	predicted_signaln	Actual Returnsn	Strategy Returnsn
2015-10-05 09:45:00	1.0	0.013532	0.013532
2015-10-05 11:30:00	1.0	0.002302	0.002302
2015-10-05 13:15:00	1.0	-0.000919	-0.000919
2015-10-05 14:30:00	1.0	0.000920	0.000920
2015-10-05 14:45:00	1.0	0.002756	0.002756

In [40]:
```python
# Plot the actual returns versus the strategy returns
(1 + predictions_dfn[["Actual Returnsn", "Strategy Returnsn"]]).cumprod().plot()
```

Out[40]:
```
<AxesSubplot:xlabel='date'>
```



Step 2: Tune the trading algorithm by adjusting the SMA input features.

Adjust one or both of the windows for the algorithm. Rerun the notebook with the updated parameters, and record the results in your `README.md` file.

Answer the following question: What impact resulted from increasing or decreasing either or both of the SMA windows?

In [81]:
```
# Set the short window and long window
short_window = 10
long_window = 200

# Generate the fast and slow simple moving averages (4 and 100 days, respectively)
signals_df['SMA_Fast'] = signals_df['close'].rolling(window=short_window).mean()
signals_df['SMA_Slow'] = signals_df['close'].rolling(window=long_window).mean()
```

```
signals_df = signals_df.dropna()

# Review the DataFrame
display(signals_df.head())
display(signals_df.tail())
```

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns
2016-01-20 12:45:00	17.74	-0.005048	18.040	19.79235	-1.0	0.005048
2016-01-20 13:00:00	17.83	0.005073	17.986	19.77195	1.0	-0.005073
2016-01-20 14:30:00	17.95	0.006730	17.952	19.75335	1.0	0.006730
2016-01-20 14:45:00	17.93	-0.001114	17.910	19.73470	-1.0	-0.001114
2016-01-20 15:00:00	17.97	0.002231	17.870	19.71575	1.0	-0.002231

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns
2021-01-22 09:30:00	33.27	-0.006866	32.930	29.05610	-1.0	-0.006866
2021-01-22 11:30:00	33.35	0.002405	33.014	29.08625	1.0	-0.002405
2021-01-22 13:45:00	33.42	0.002099	33.098	29.11665	1.0	0.002099
2021-01-22 14:30:00	33.47	0.001496	33.170	29.14690	1.0	0.001496
2021-01-22 15:45:00	33.44	-0.000896	33.249	29.17835	-1.0	-0.000896

In [82]:

```
# Initialize the new Signal column
signals_df['Signal'] = 0.0

# When Actual Returns are greater than or equal to 0, generate signal to buy stock long
signals_df.loc[(signals_df['Actual Returns'] >= 0), 'Signal'] = 1

# When Actual Returns are less than 0, generate signal to sell stock short
signals_df.loc[(signals_df['Actual Returns'] < 0), 'Signal'] = -1

# Review the DataFrame
display(signals_df.head())
display(signals_df.tail())
```

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns
2016-01-20 12:45:00	17.74	-0.005048	18.040	19.79235	-1.0	0.005048
2016-01-20 13:00:00	17.83	0.005073	17.986	19.77195	1.0	-0.005073
2016-01-20 14:30:00	17.95	0.006730	17.952	19.75335	1.0	0.006730
2016-01-20 14:45:00	17.93	-0.001114	17.910	19.73470	-1.0	-0.001114
2016-01-20 15:00:00	17.97	0.002231	17.870	19.71575	1.0	-0.002231

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns
2021-01-22 09:30:00	33.27	-0.006866	32.930	29.05610	-1.0	-0.006866
2021-01-22 11:30:00	33.35	0.002405	33.014	29.08625	1.0	-0.002405
2021-01-22 13:45:00	33.42	0.002099	33.098	29.11665	1.0	0.002099
2021-01-22 14:30:00	33.47	0.001496	33.170	29.14690	1.0	0.001496
2021-01-22 15:45:00	33.44	-0.000896	33.249	29.17835	-1.0	-0.000896

In [83]:
```
signals_df['Signal'].value_counts()
```

Out[83]:
```
 1.0    2052
-1.0    1574
Name: Signal, dtype: int64
```

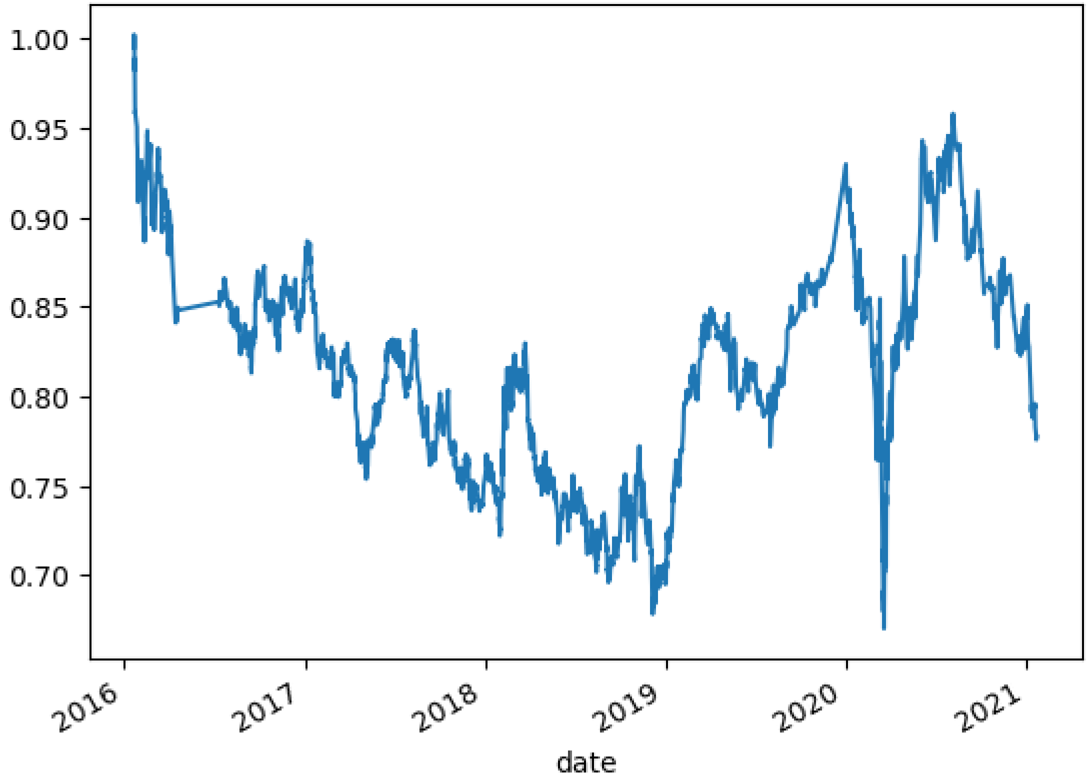
In [86]:
```
# Calculate the strategy returns and add them to the signals_df DataFrame
signals_df['Strategy Returnsf'] = signals_df['Actual Returns'] * signals_df['Signal'].shift()

# Review the DataFrame
display(signals_df.head())
display(signals_df.tail())
```

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns	Strategy Returnsf
2016-01-20 12:45:00	17.74	-0.005048	18.040	19.79235	-1.0	0.005048	NaN
2016-01-20 13:00:00	17.83	0.005073	17.986	19.77195	1.0	-0.005073	-0.005073
2016-01-20 14:30:00	17.95	0.006730	17.952	19.75335	1.0	0.006730	0.006730
2016-01-20 14:45:00	17.93	-0.001114	17.910	19.73470	-1.0	-0.001114	-0.001114
2016-01-20 15:00:00	17.97	0.002231	17.870	19.71575	1.0	-0.002231	-0.002231

date	close	Actual Returns	SMA_Fast	SMA_Slow	Signal	Strategy Returns	Strategy Returnsf
2021-01-22 09:30:00	33.27	-0.006866	32.930	29.05610	-1.0	-0.006866	-0.006866
2021-01-22 11:30:00	33.35	0.002405	33.014	29.08625	1.0	-0.002405	-0.002405
2021-01-22 13:45:00	33.42	0.002099	33.098	29.11665	1.0	0.002099	0.002099
2021-01-22 14:30:00	33.47	0.001496	33.170	29.14690	1.0	0.001496	0.001496
2021-01-22 15:45:00	33.44	-0.000896	33.249	29.17835	-1.0	-0.000896	-0.000896

In [87]:
```
# Plot Strategy Returns to examine performance
(1 + signals_df['Strategy Returnsf']).cumprod().plot()
```

Out[87]:
```
<AxesSubplot:xlabel='date'>
```



In [88]:

```
# Assign a copy of the sma_fast and sma_slow columns to a features DataFrame called X
X = signals_df[['SMA_Fast', 'SMA_Slow']].shift().dropna()

# Review the DataFrame
X.head()
```

Out[88]:

date	SMA_Fast	SMA_Slow
2016-01-20 13:00:00	18.040	19.79235
2016-01-20 14:30:00	17.986	19.77195
2016-01-20 14:45:00	17.952	19.75335
2016-01-20 15:00:00	17.910	19.73470
2016-01-20 15:30:00	17.870	19.71575

In [91]:
```
# Create the target set selecting the Signal column and assiging it to y
y = signals_df['Signal']

# Review the value counts
y.value_counts()
```

Out[91]:
```
 1.0    2052
-1.0    1574
Name: Signal, dtype: int64
```

In [93]:
```
# Select the start of the training period
training_begin = X.index.min()

# Display the training begin date
print(training_begin)
```

```
2016-01-20 13:00:00
```

In [94]:
```
# Select the ending period for the training data with an offset of 3 months
training_end = X.index.min() + DateOffset(months=3)

# Display the training end date
print(training_end)
```

```
2016-04-20 13:00:00
```

In [95]:
```
# Generate the X_train and y_train DataFrames
X_train = X.loc[training_begin:training_end]
y_train = y.loc[training_begin:training_end]

# Review the X_train DataFrame
X_train.head()
```

Out[95]:

date	SMA_Fast	SMA_Slow
2016-01-20 13:00:00	18.040	19.79235
2016-01-20 14:30:00	17.986	19.77195
2016-01-20 14:45:00	17.952	19.75335
2016-01-20 15:00:00	17.910	19.73470
2016-01-20 15:30:00	17.870	19.71575

In [96]:

```
# Generate the X_test and y_test DataFrames
X_test = X.loc[training_end+DateOffset(hours=1):]
y_test = y.loc[training_end+DateOffset(hours=1):]

# Review the X_test DataFrame
X_test.head()
```

Out[96]:

date	SMA_Fast	SMA_Slow
2016-07-11 12:15:00	20.995	19.671220
2016-07-11 13:00:00	20.990	19.683820
2016-07-12 15:00:00	20.992	19.696345
2016-07-12 15:15:00	21.015	19.710095
2016-07-12 15:45:00	21.043	19.723795

In [97]:

```
# Scale the features DataFrames
# Create a StandardScaler instance
scaler = StandardScaler()

# Apply the scaler model to fit the X-train data
X_scaler = scaler.fit(X_train)

# Transform the X_train and X_test DataFrames using the X_scaler
X_train_scaled = X_scaler.transform(X_train)
X_test_scaled = X_scaler.transform(X_test)
```

In [98]:
```
# From SVM, instantiate SVC classifier model instance
svm_model = svm.SVC()

# Fit the model to the data using the training data
svm_model = svm_model.fit(X_train_scaled, y_train)

# Use the testing data to make the model predictions
svm_predf =  svm_model.predict(X_test_scaled)

# Review the model's predicted values
svm_predf[:10]
```

Out[98]:
```
array([-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.])
```

In [99]:
```
# Use a classification report to evaluate the model using the predictions and testing data
svm_testing_reportf = classification_report(y_test, svm_predf)

# Print the classification report
print(svm_testing_reportf)
```

```
              precision    recall  f1-score   support

        -1.0       0.29      0.00      0.01      1484
         1.0       0.56      0.99      0.72      1921

    accuracy                           0.56      3405
   macro avg       0.43      0.50      0.36      3405
weighted avg       0.45      0.56      0.41      3405
```

In [100…
```
# Create a predictions DataFrame
predictions_dff = pd.DataFrame(index=X_test.index)
```

In [102…
```
predictions_dff["predicted_signalf"] = svm_predf
# Add the actual returns to the DataFrame
predictions_dff["Actual Returnsf"] = signals_df["Actual Returns"]
# Add the strategy returns to the DataFrame
predictions_dff["Strategy Returnsf"] = (signals_df["Actual Returns"] * predictions_dff["predicted_signalf"])

# Review the DataFrame
predictions_dff.head()
```

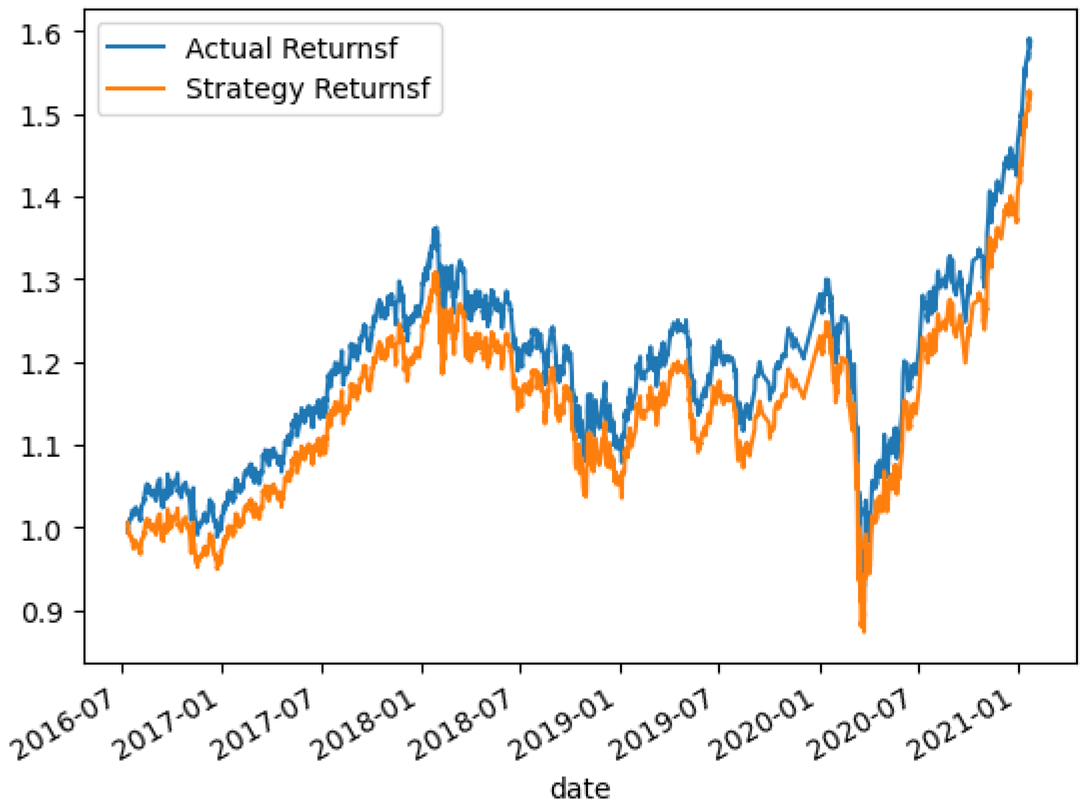
Out[102]:

date	predicted_signalf	Actual Returnsf	Strategy Returnsf
2016-07-11 12:15:00	-1.0	-0.005701	0.005701
2016-07-11 13:00:00	-1.0	0.002867	-0.002867
2016-07-12 15:00:00	-1.0	0.009528	-0.009528
2016-07-12 15:15:00	-1.0	0.000000	-0.000000
2016-07-12 15:45:00	-1.0	-0.000472	0.000472

In [103…

```
# Plot the actual returns versus the strategy returns
(1 + predictions_dff[["Actual Returnsf", "Strategy Returnsf"]]).cumprod().plot()
```

Out[103]: <AxesSubplot:xlabel='date'>



Step 3: Choose the set of parameters that best improved the trading algorithm returns.

Save a PNG image of the cumulative product of the actual returns vs. the strategy returns, and document your conclusion in your `README.md` file.

Evaluate a New Machine Learning Classifier

In this section, you'll use the original parameters that the starter code provided. But, you'll apply them to the performance of a second machine learning model.

Step 1: Import a new classifier, such as `AdaBoost`, `DecisionTreeClassifier`, or `LogisticRegression`. (For the full list of classifiers, refer to the Supervised learning page in the scikit-learn documentation.)

In [57]:
```
# Select the ending period for the training data with an offset of 3 months
training_end2 = X.index.min() + DateOffset(months=3)

# Display the training end date
print(training_end2)
```

```
2015-07-02 15:00:00
```

In [58]:
```
# Generate the X_train and y_train DataFrames
X_train2 = X.loc[training_begin:training_end2]
y_train2 = y.loc[training_begin:training_end2]

# Review the X_train DataFrame
X_train2.head()
X_train2.tail()
```

Out[58]:

date	SMA_Fast	SMA_Slow
2015-06-30 12:15:00	24.2150	25.2106
2015-06-30 14:00:00	24.1050	25.1930
2015-06-30 14:15:00	24.0775	25.1767
2015-06-30 15:00:00	24.1000	25.1597
2015-07-02 10:45:00	24.1175	25.1427

In [59]:
```
X_train2.shape
```

Out[59]:
```
(128, 2)
```

In [60]:
```
# Generate the X_test and y_test DataFrames
X_test2 = X.loc[training_end2+DateOffset(hours=1):]
y_test2 = y.loc[training_end2+DateOffset(hours=1):]

# Review the X_test DataFrame
X_test2.head()
```

Out[60]:

date	SMA_Fast	SMA_Slow
2015-07-06 10:00:00	24.1250	25.0919
2015-07-06 10:45:00	23.9700	25.0682
2015-07-06 14:15:00	23.8475	25.0458
2015-07-06 14:30:00	23.6725	25.0206
2015-07-07 11:30:00	23.4800	24.9951

In [61]:
```
X_test2.shape
```

Out[61]:
```
(4092, 2)
```

In [62]:
```
# Import a new classifier from SKLearn
from sklearn.linear_model import LogisticRegression
```

```
# Initiate the model instance
logistic_regression_model = LogisticRegression()
```

Step 2: Using the original training data as the baseline model, fit another model with the new classifier.

In [63]:
```
# Fit the model using the training data
model = logistic_regression_model.fit(X_train2, y_train2)

# Use the testing dataset to generate the predictions for the new model
lr_pred  = logistic_regression_model.predict(X_test2)

# Display the predictions
lr_pred
```

Out[63]:
```
array([ 1.,  1.,  1., ..., -1., -1., -1.])
```

Step 3: Backtest the new model to evaluate its performance.

Save a PNG image of the cumulative product of the actual returns vs. the strategy returns for this updated trading algorithm, and write your conclusions in your `README.md` file.

Answer the following questions: Did this new model perform better or worse than the provided baseline model? Did this new model perform better or worse than your tuned trading algorithm?

In [64]:
```
# Use a classification report to evaluate the model using the predictions and testing data
lr_training_report = classification_report(y_test2, lr_pred)

# Print the classification report
print(lr_training_report)
```

```
              precision    recall  f1-score   support

        -1.0       0.44      0.32      0.37      1804
         1.0       0.56      0.68      0.61      2288

    accuracy                           0.52      4092
   macro avg       0.50      0.50      0.49      4092
weighted avg       0.51      0.52      0.51      4092
```

In [65]:

```python
# Create a new empty predictions DataFrame:

# Create a predictions DataFrame
predictions_df2 = pd.DataFrame(index=X_test2.index)

# Add the SVM model predictions to the DataFrame
predictions_df2["predicted_signal2"] = lr_pred

# Add the actual returns to the DataFrame
predictions_df2["LRActual Returns2"] = signals_df["Actual Returns"]

# Add the strategy returns to the DataFrame
predictions_df2["LRStrategy Returns2"] = (
    signals_df["Actual Returns"] * predictions_df2["predicted_signal2"]
)

# Review the DataFrame
predictions_df2
```

Out[65]:

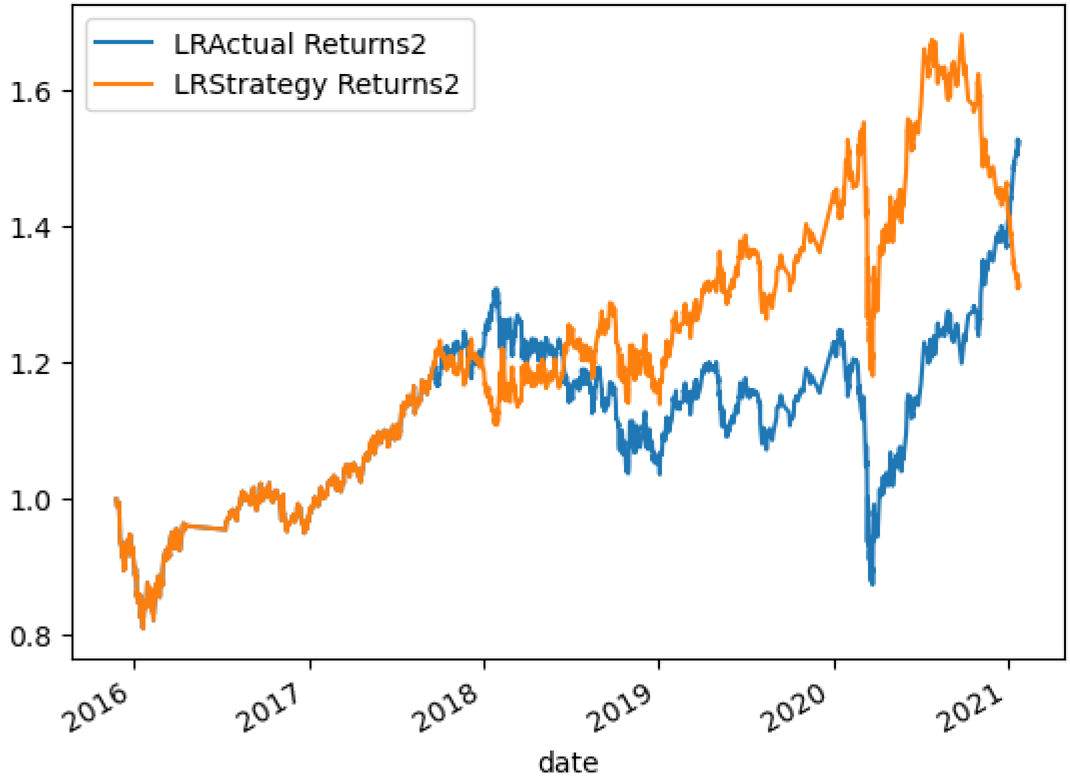
date	predicted_signal2	LRActual Returns2	LRStrategy Returns2
2015-07-06 10:00:00	1.0	NaN	NaN
2015-07-06 10:45:00	1.0	NaN	NaN
2015-07-06 14:15:00	1.0	NaN	NaN
2015-07-06 14:30:00	1.0	NaN	NaN
2015-07-07 11:30:00	1.0	NaN	NaN
...	...	...	...
2021-01-22 09:30:00	-1.0	-0.006866	0.006866
2021-01-22 11:30:00	-1.0	0.002405	-0.002405
2021-01-22 13:45:00	-1.0	0.002099	-0.002099
2021-01-22 14:30:00	-1.0	0.001496	-0.001496
2021-01-22 15:45:00	-1.0	-0.000896	0.000896

4092 rows × 3 columns

In [66]:

```
# Plot the actual returns versus the strategy returns
(1 + predictions_df2[["LRActual Returns2", "LRStrategy Returns2"]]).cumprod().plot()
```

Out[66]: <AxesSubplot:xlabel='date'>



In [67]:
```
accuracy = accuracy_score(y_test2, lr_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 0.5195503421309873
```

In [68]:
```
### Step 1:  Import a new classifier, such as `AdaBoost`, `DecisionTreeClassifier`, or `LogisticRegression`. (For the f
```

In [69]:
```
# Select the ending period for the training data with an offset of 3 months
training_end3 = X.index.min() + DateOffset(months=3)

# Display the training end date
print(training_end3)
```

```
2015-07-02 15:00:00
```

In [70]:
```
# Generate the X_train and y_train DataFrames
X_train3 = X.loc[training_begin:training_end3]
y_train3 = y.loc[training_begin:training_end3]
```

```
# Review the X_train DataFrame
X_train3.head()
X_train3.tail()
```

Out[70]:

date	SMA_Fast	SMA_Slow
2015-06-30 12:15:00	24.2150	25.2106
2015-06-30 14:00:00	24.1050	25.1930
2015-06-30 14:15:00	24.0775	25.1767
2015-06-30 15:00:00	24.1000	25.1597
2015-07-02 10:45:00	24.1175	25.1427

In [71]:

```
X_train3.shape
```

Out[71]:

```
(128, 2)
```

In [72]:

```
# Generate the X_test and y_test DataFrames
X_test3 = X.loc[training_end3+DateOffset(hours=1):]
y_test3 = y.loc[training_end3+DateOffset(hours=1):]

# Review the X_test DataFrame
X_test3.head()
```

Out[72]:

date	SMA_Fast	SMA_Slow
2015-07-06 10:00:00	24.1250	25.0919
2015-07-06 10:45:00	23.9700	25.0682
2015-07-06 14:15:00	23.8475	25.0458
2015-07-06 14:30:00	23.6725	25.0206
2015-07-07 11:30:00	23.4800	24.9951

In [73]:

```
X_test3.shape
```

Out[73]:
```
(4092, 2)
```

In [74]:
```
from sklearn.ensemble import AdaBoostClassifier
```

In [75]:
```
ada = AdaBoostClassifier()
```

In [76]:
```
# Fit the classifier to the training data
ada.fit(X_train3, y_train3)

# Make predictions on the testing data
ada_predictions = ada.predict(X_test3)

# Print the classification report for the AdaBoost classifier
print(classification_report(y_test3, ada_predictions))
```

```
              precision    recall  f1-score   support

        -1.0       0.44      0.08      0.13      1804
         1.0       0.56      0.92      0.70      2288

    accuracy                           0.55      4092
   macro avg       0.50      0.50      0.41      4092
weighted avg       0.51      0.55      0.45      4092
```

In [77]:
```
# Create a new empty predictions DataFrame:
ada_df = pd.DataFrame(index=X_test3.index)
```

In [78]:
```
ada_df['AdaBoost Predicted'] = ada_predictions
ada_df['AdaBoost Actual Returns'] = signals_df["Actual Returns"]
ada_df['AdaBoost Strategy Returns'] = ada_df['AdaBoost Actual Returns'] * ada_df['AdaBoost Predicted']
ada_df
```

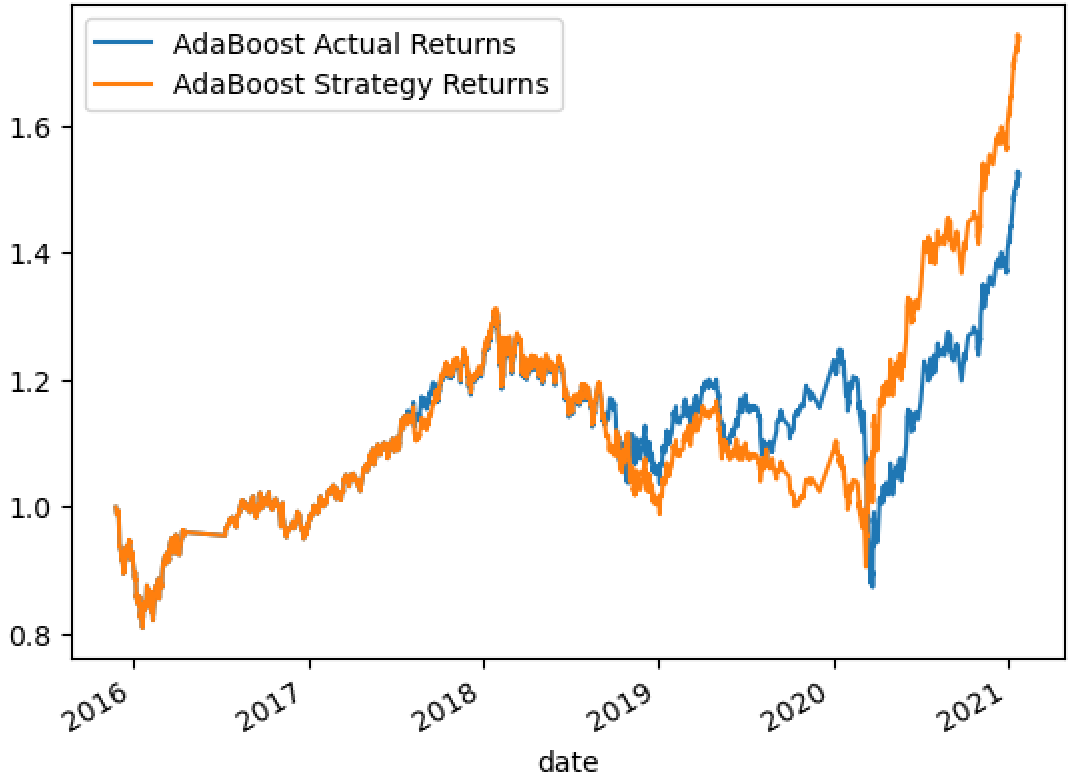
Out[78]:

date	AdaBoost Predicted	AdaBoost Actual Returns	AdaBoost Strategy Returns
2015-07-06 10:00:00	1.0	NaN	NaN
2015-07-06 10:45:00	-1.0	NaN	NaN
2015-07-06 14:15:00	-1.0	NaN	NaN
2015-07-06 14:30:00	-1.0	NaN	NaN
2015-07-07 11:30:00	-1.0	NaN	NaN
...	...	...	...
2021-01-22 09:30:00	1.0	-0.006866	-0.006866
2021-01-22 11:30:00	1.0	0.002405	0.002405
2021-01-22 13:45:00	1.0	0.002099	0.002099
2021-01-22 14:30:00	1.0	0.001496	0.001496
2021-01-22 15:45:00	1.0	-0.000896	-0.000896

4092 rows × 3 columns

In [79]:
```
(1 + ada_df[["AdaBoost Actual Returns", "AdaBoost Strategy Returns"]]).cumprod().plot()
```

Out[79]:
```
<AxesSubplot:xlabel='date'>
```



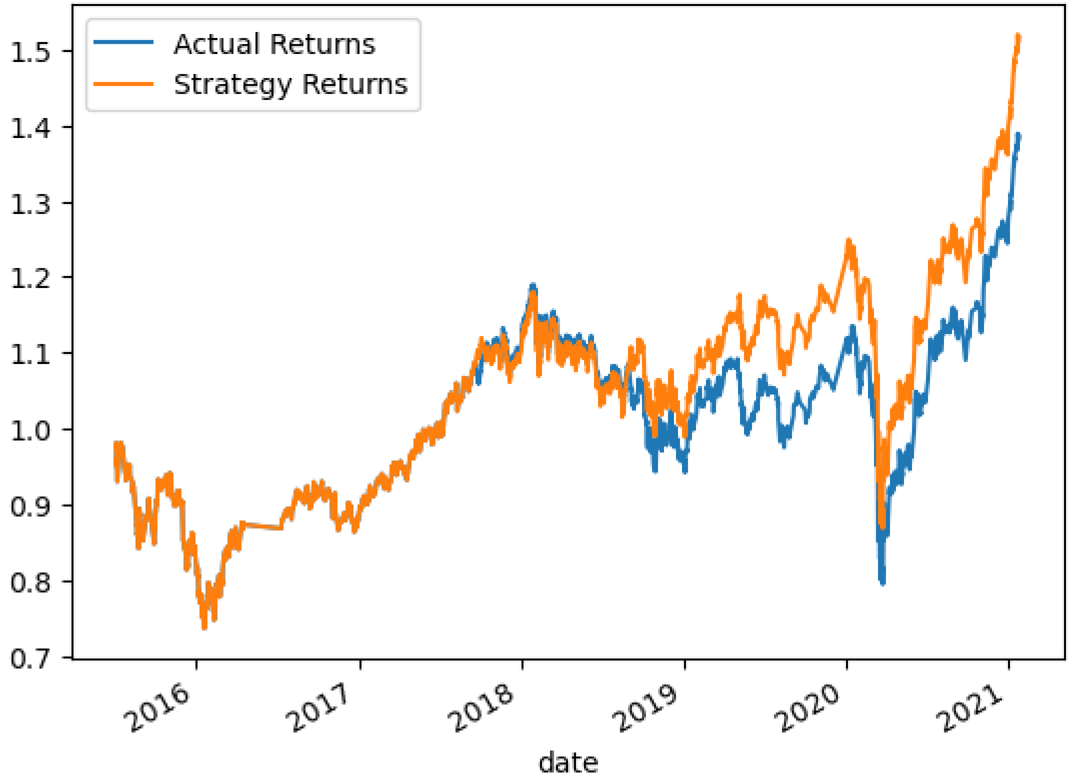
#All the predictions

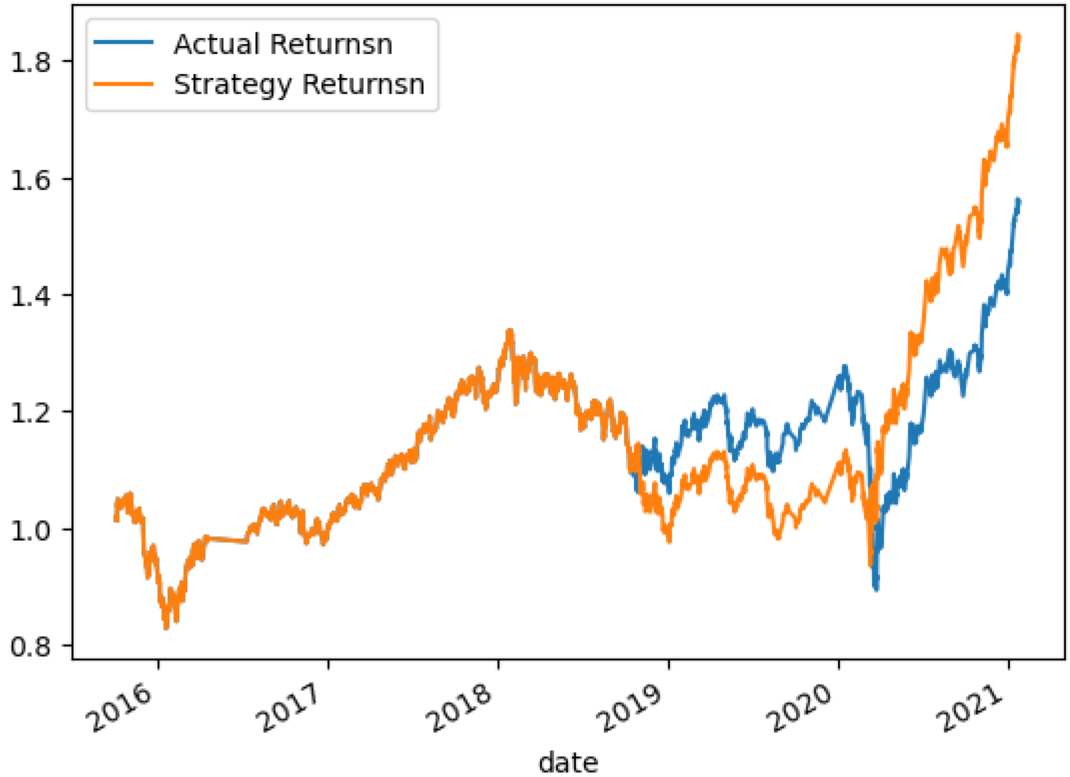
In [104…

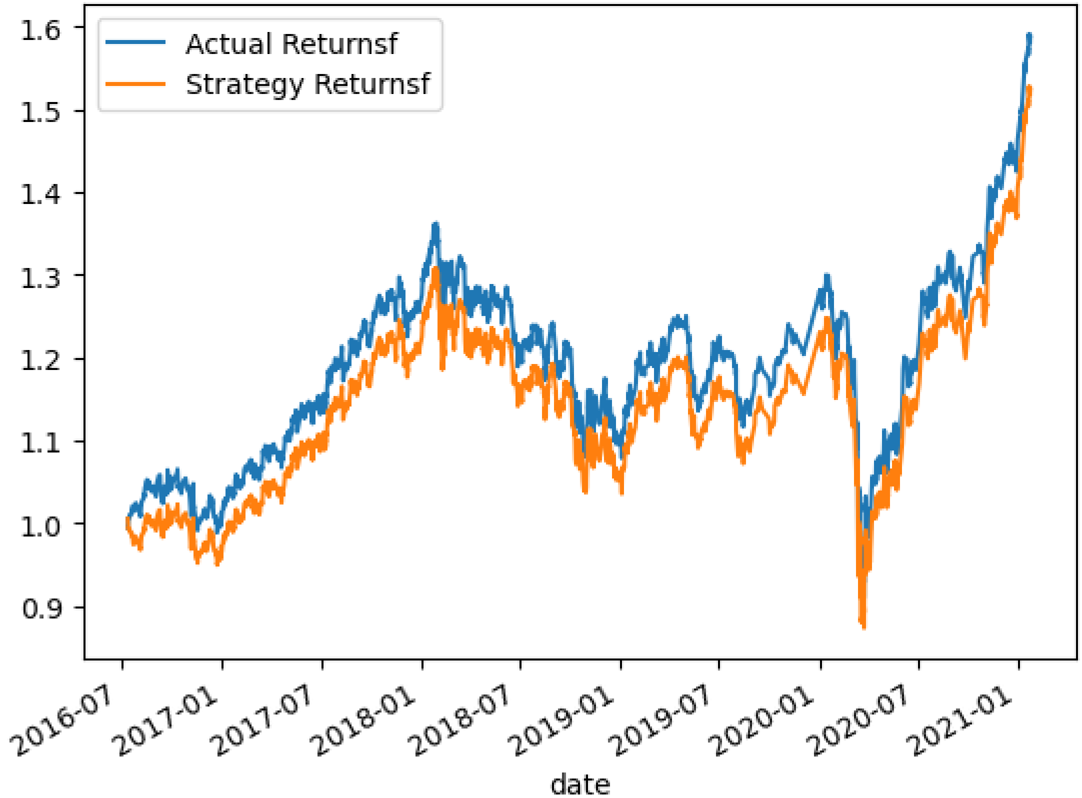
```
(1 + predictions_df[["Actual Returns", "Strategy Returns"]]).cumprod().plot()
(1 + predictions_dfn[["Actual Returnsn", "Strategy Returnsn"]]).cumprod().plot()
(1 + predictions_dff[["Actual Returnsf", "Strategy Returnsf"]]).cumprod().plot()
(1 + predictions_df2[["LRActual Returns2", "LRStrategy Returns2"]]).cumprod().plot()
(1 + ada_df[["AdaBoost Actual Returns", "AdaBoost Strategy Returns"]]).cumprod().plot()
```

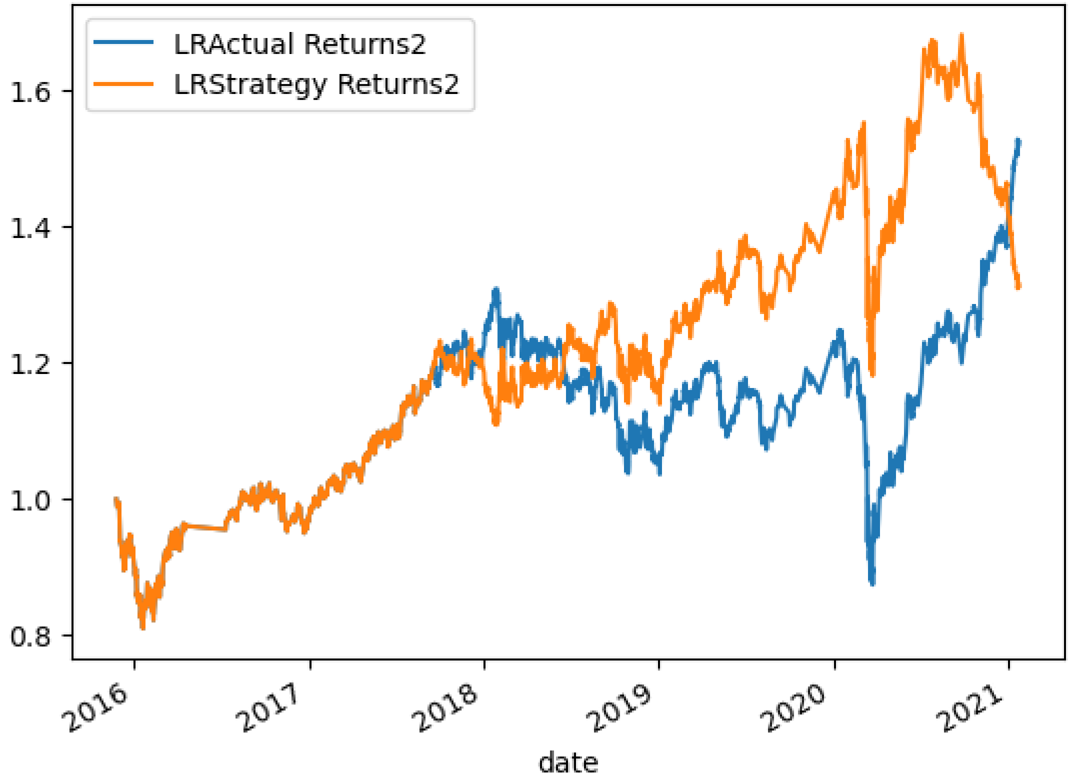
Out[104]:

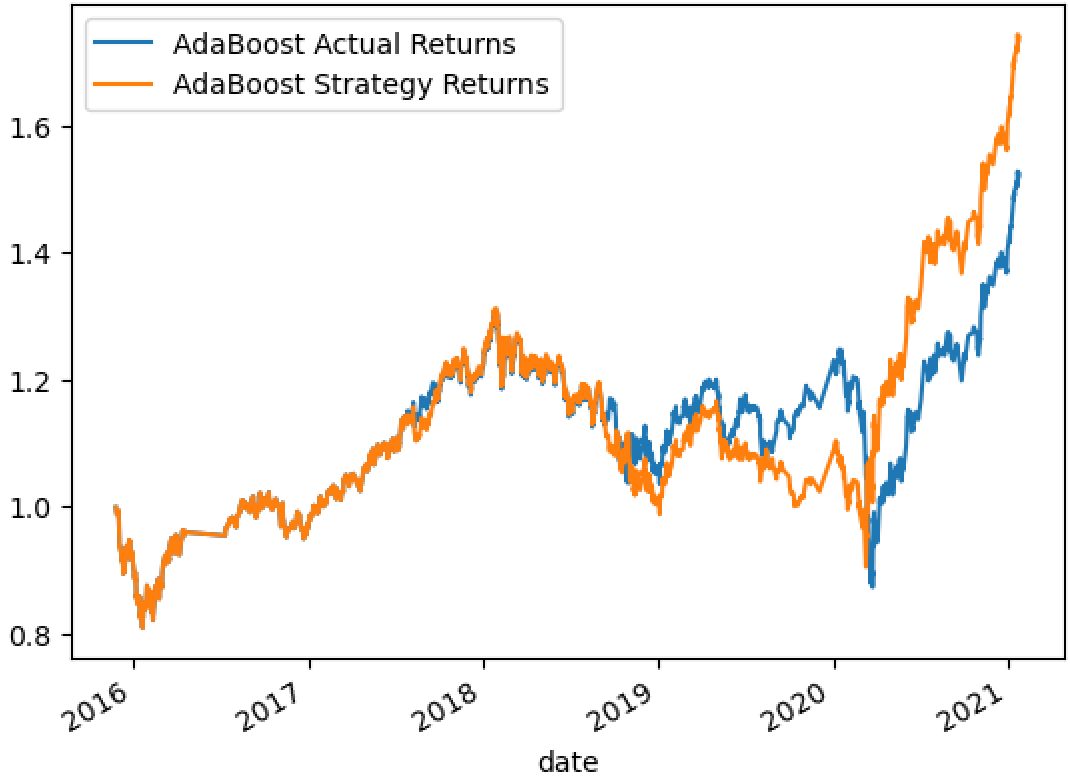
```
<AxesSubplot:xlabel='date'>
```


Actual Returns
Strategy Returns
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
2016
2017
2018
2019
2020
2021
date


Actual Returnsn
Strategy Returnsn
1.8
1.6
1.4
1.2
1.0
0.8
2016
2017
2018
2019
2020
2021
date


Actual Returnsf
Strategy Returnsf
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
2016-07
2017-01
2017-07
2018-01
2018-07
2019-01
2019-07
2020-01
2020-07
2021-01
date


LRActual Returns2
LRStrategy Returns2
1.6
1.4
1.2
1.0
0.8
2016
2017
2018
2019
2020
2021
date


AdaBoost Actual Returns
AdaBoost Strategy Returns
1.6
1.4
1.2
1.0
0.8
2016
2017
2018
2019
2020
2021
date

In [ ]:

In [ ]:

In [ ]:

In [ ]: