International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

# Random Forest Classifier to Predict Financial Data

**Aumkar Wagle**

Random Forest Classifier is a powerful machine learning algorithm widely utilized in the field of finance to predict uptrends and downtrends in financial data. By leveraging the collective wisdom of multiple decision trees, the Random Forest Classifier excels in handling complex datasets with numerous features and variables. This ensemble learning technique works by aggregating the predictions of individual decision trees to provide accurate and reliable classifications, making it a popular choice for financial analysts and traders seeking to forecast market movements and make informed investment decisions.

I will be using this algorithm to predict positive moves (up trend).

We will first import all the libraries related to this exercise. The use cases for these libraries ranges from being able to store and manipulate data via dataframes to using learning algorithms on our dataset.

```
import warnings
warnings.filterwarnings('ignore')
warnings.simplefilter(action='ignore',
category=FutureWarning)
# Data manipulation
import pandas as pd
import numpy as np
# Plotting
import matplotlib.pyplot as plt
import seaborn as sns
# Preprocessing
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import (
train_test_split,
RandomizedSearchCV,
TimeSeriesSplit,
cross_val_score
)
# metrics
from sklearn.metrics import (precision_recall_curve,
roc_curve,
RocCurveDisplay,
ConfusionMatrixDisplay
)
from sklearn.metrics import (accuracy_score,
f1_score,
recall_score,
precision_score,
roc_auc_score,
auc)
from sklearn.metrics import (classification_report,
confusion_matrix
)
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV,
RandomizedSearchCV
```

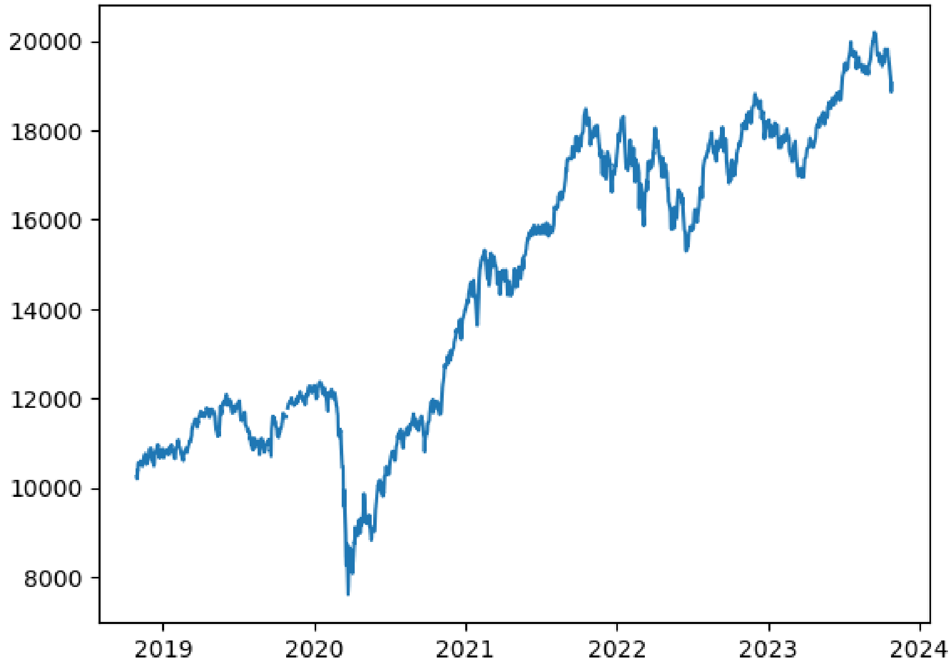
I am using a file that has been downloaded from Yahoo Finance onto my work computer. Not pinging the Yahoo Finance API directly to retrieve the data since I believe it is constantly pinged by my company server and hence pulling the data is not easy. Therefore, I downloaded the file directly from the website and will be using Pandas to access it.

```
#import os to know which directory we are in
import os
os.getcwd()
'C:\\Users\\waglaum'

#change the directory to the one where the data file from Yahoo Finance has been downloaded
os.chdir('C:\\Users\\waglaum\\Downloads')

#confirming the directory has changed successfully
os.getcwd()

'C:\\Users\\waglaum\\Downloads'
```

The dataset that we are working on is that of an Indian index Nifty 50. Data worth 5 years has been drawn on a daily basis.

```
#reading and plotting the downloaded data
df = pd.read_csv('Nifty50.csv', index_col=0, parse_dates=True)[['Open', 'High', 'Low', 'Close','Adj Close','Volume']]
df.shape
plt.plot(df['Adj Close']);
df.head()
```

| Date | Open | High | Low | Close | Adj Close | Volume |
|---|---|---|---|---|---|---|
| 29-10-2018 | 10078.0996 | 10275.2998 | 10020.3496 | 10250.8496 | 10250.8496 | 364400 |
| 30-10-2018 | 10239.4004 | 10285.0996 | 10175.3496 | 10198.4004 | 10198.4004 | 289800 |
| 31-10-2018 | 10209.5498 | 10396 | 10105.0996 | 10386.5996 | 10386.5996 | 375000 |
| 01-11-2018 | 10441.7002 | 10441.9004 | 10341.9004 | 10380.4502 | 10380.4502 | 348500 |
| 02-11-2018 | 10462.2998 | 10606.9502 | 10457.7002 | 10553 | 10553 | 421200 |




As we can see from the above data, the trend is mainly upwards which is coherent with the idea that over a period of time markets usually move upwards. The data also includes the massive downturn that was seen in the market due to the COVID-19 pandemic so it also includes negative territory for returns. This is good for our model building exercise as it encompasses the general idea of pregoressive markets along with a downturn to capture negative returns as well.

```
df.describe()
```

| | Open | High | Low | Close | Adj Close | Volume |
|---|---|---|---|---|---|---|
| count | 1233.00 | 1233.00 | 1233.00 | 1233.00 | 1233.00 | 1.23E+03 |
| mean | 14634.85544 | 14706.97444 | 14535.58445 | 14624.23452 | 14624.23452 | 4.21E+05 |
| std | 3228.140787 | 3228.071332 | 3225.610422 | 3228.117879 | 3228.117879 | 2.18E+05 |
| min | 7735.149902 | 8036.950195 | 7511.100098 | 7610.25 | 7610.25 | 0.00E+00 |
| 25% | 11542.7002 | 11588.5 | 11461.84961 | 11527.4502 | 11527.4502 | 2.61E+05 |
| 50% | 15073.25 | 15188.5 | 15008.84961 | 15108.09961 | 15108.09961 | 3.54E+05 |
| 75% | 17599.90039 | 17683.15039 | 17485.84961 | 17599.15039 | 17599.15039 | 5.55E+05 |
| max | 20156.44922 | 20222.44922 | 20129.69922 | 20192.34961 | 20192.34961 | 1.81E+06 |

```
#checking for null values in the dataset
df.isnull().sum()

Open        3
High        3
Low         3
Close       3
Adj Close   3
Volume      3
dtype: int64

df['return'] = np.log(df['Adj Close'] / df['Adj Close'].shift(1))
# create logarithmic returns
df['return_sign'] = np.sign(df['return']) # create a variable to check the sign depending on the above define return

#create function to compute exponential moving average

def EMAcreate(price, period):
    modifiedPrice = price.copy()
    sma_period = price.rolling(period).mean()
    modifiedPrice.iloc[0:period] = sma_period[0:period]
    ema_period = modifiedPrice.ewm(span=period, adjust=False).mean()
    return ema_period

df['Adj Close Lagged'] = df['Adj Close'].shift(1) # lagged adjusted close price
df['Open Lagged'] = df.Open.shift(1) # lagged open price
df['Close Lagged'] = df.Close.shift(1) # lagged close price
df['High Lagged'] = df.High.shift(1) # lagged high price
df['Low Lagged'] = df.Low.shift(1) # lagged low price
df['Volume Lagged'] = df.Volume.shift(1) # lagged Volume

# creating lagged returns

lags = 8
cols = []
for lag in range(1, lags+1):
    col_ret = 'ret_%d' % lag
    df[col_ret] = df['return'].shift(lag)
    cols.append(col_ret)

#creating a list of features that includes rolling and lagged returns
features_list = []
for r in range(10, 65, 5):
    df['Ret_'+str(r)] = df['return'].rolling(r).sum()
```

International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942

```
    df['Std_'+str(r)] = df['return'].rolling(r).std()
    features_list.append('Ret_'+str(r))
    features_list.append('Std_'+str(r))
features_list.append('ret_over_21d')
features_list.append('MOM_1d')
features_list.append('MOM_5d')
features_list.append('MA_5d')
features_list.append('EMA_7d')
df.dropna(inplace = True)

df['ret_over_5d'] = np.log(df['Adj Close Lagged'] / df['Adj Close Lagged'].shift(5)) # lagged 5-day return
df['ret_over_21d'] = np.log(df['Adj Close Lagged'] / df['Adj Close Lagged'].shift(21)) # lagged 21-day return
df['MOM_1d'] = df['Adj Close Lagged'].diff(1) # lagged 1-day adjusted close price difference
df['MOM_5d'] = df['Adj Close Lagged'].diff(5) # lagged 5-day adjusted close price difference
df['MA_5d'] = df['Adj Close Lagged'].rolling(1).mean() # lagged 5-day adjusted close price moving average
df['EMA_7d'] = EMAcreate(df['Adj Close Lagged'], 1) # lagged 7-day adjusted close price exponential moving average
df.dropna(inplace = True)
```

All the 50 reated features are still part of our original dataframe 'df' so we will now create a copy of our dataframe to store the feature set as a new datafram 'features_df'.

```
# create a copy of our dataframe
features_df = df.copy()

features_df.head(5)
```

| Date | Open | High | Low | Close | Adj Close | Volume | return | return_sign | dj Close Lagge | Open Lagged | ... | Ret_55 | Std_55 | Ret_60 | Std_60 | ret_over_5d | ret_over_21d | MOM_1d | MOM_5d | MA_5d | EMA_7d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2019-05-07 | 11651.500000 | 11657.049805 | 11484.450195 | 11497.900391 | 11497.900391 | 337500.0 | -0.008690 | -1.0 | 11598.250000 | 11605.799805 | ... | 0.037980 | 0.006964 | 0.059757 | 0.006842 | -0.003748 | -0.006094 | -114.000000 | -43.549805 | 11598.250000 | 11598.250000 |
| 2019-05-08 | 11478.700195 | 11479.099609 | 11346.950195 | 11359.450195 | 11359.450195 | 372800.0 | -0.012114 | -1.0 | 11497.900391 | 11651.500000 | ... | 0.037295 | 0.006987 | 0.041870 | 0.007018 | -0.022084 | -0.018552 | -100.349609 | -256.750000 | 11497.900391 | 11497.900391 |
| 2019-05-09 | 11322.400391 | 11357.599609 | 11255.049805 | 11301.799805 | 11301.799805 | 373000.0 | -0.005088 | -1.0 | 11359.450195 | 11478.700195 | ... | 0.037227 | 0.006988 | 0.035076 | 0.007056 | -0.033646 | -0.024737 | -138.450196 | -388.700196 | 11359.450195 | 11359.450195 |
| 2019-05-10 | 11314.150391 | 11345.799805 | 11251.049805 | 11278.900391 | 11278.900391 | 387300.0 | -0.002028 | -1.0 | 11301.799805 | 11322.400391 | ... | 0.040484 | 0.006950 | 0.031025 | 0.007061 | -0.036740 | -0.025871 | -57.650390 | -422.950195 | 11301.799805 | 11301.799805 |
| 2019-05-13 | 11258.700195 | 11300.200195 | 11125.599609 | 11148.200195 | 11148.200195 | 357600.0 | -0.011656 | -1.0 | 11278.900391 | 11314.150391 | ... | 0.036740 | 0.007054 | 0.007722 | 0.007080 | -0.037702 | -0.033741 | -22.899414 | -433.349609 | 11278.900391 | 11278.900391 |

5 rows × 50 columns

```
features_df.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 930 entries, 2019-07-30 to 2023-10-27
Data columns (total 50 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Open              930 non-null    float64
 1   High              930 non-null    float64
 2   Low               930 non-null    float64
 3   Close             930 non-null    float64
 4   Adj Close         930 non-null    float64
 5   Volume            930 non-null    float64
 6   return            930 non-null    float64
```

```
7   return_sign      930 non-null    float64
8   Ret_10           930 non-null    float64
9   Std_10           930 non-null    float64
10  Ret_15           930 non-null    float64
11  Std_15           930 non-null    float64
12  Ret_20           930 non-null    float64
13  Std_20           930 non-null    float64
14  Ret_25           930 non-null    float64
15  Std_25           930 non-null    float64
16  Ret_30           930 non-null    float64
17  Std_30           930 non-null    float64
18  Ret_35           930 non-null    float64
19  Std_35           930 non-null    float64
20  Ret_40           930 non-null    float64
21  Std_40           930 non-null    float64
22  Ret_45           930 non-null    float64
23  Std_45           930 non-null    float64
24  Ret_50           930 non-null    float64
25  Std_50           930 non-null    float64
26  Ret_55           930 non-null    float64
27  Std_55           930 non-null    float64
28  Ret_60           930 non-null    float64
29  Std_60           930 non-null    float64
30  Adj Close Lagged 930 non-null    float64
31  Open Lagged      930 non-null    float64
32  Close Lagged     930 non-null    float64
33  High Lagged      930 non-null    float64
34  Low Lagged       930 non-null    float64
35  Volume Lagged    930 non-null    float64
36  ret_1            930 non-null    float64
37  ret_2            930 non-null    float64
38  ret_3            930 non-null    float64
39  ret_4            930 non-null    float64
40  ret_5            930 non-null    float64
41  ret_6            930 non-null    float64
42  ret_7            930 non-null    float64
43  ret_8            930 non-null    float64
44  ret_over_5d      930 non-null    float64
45  ret_over_21d     930 non-null    float64
46  MOM_1d           930 non-null    float64
47  MOM_5d           930 non-null    float64
48  MA_5d            930 non-null    float64
49  EMA_7d           930 non-null    float64
dtypes: float64(50)
memory usage: 370.5 KB
```

**Target or Label Definition**

Label or the target variable is the variable we are trying to predict. Here, the target variable is whether Nifty Index price will close up or down on the next trading day. If the tomorrow's closing price is greater than the 0.99995 of today's closing price, then we will buy the Nifty Index, else we will sell the index.

We assign a value of +1 for the buy signal and 0 for the sell signal to target variable. The target can be described as :

Target = 1, if pt+1 > 0.99995 ∗ pt

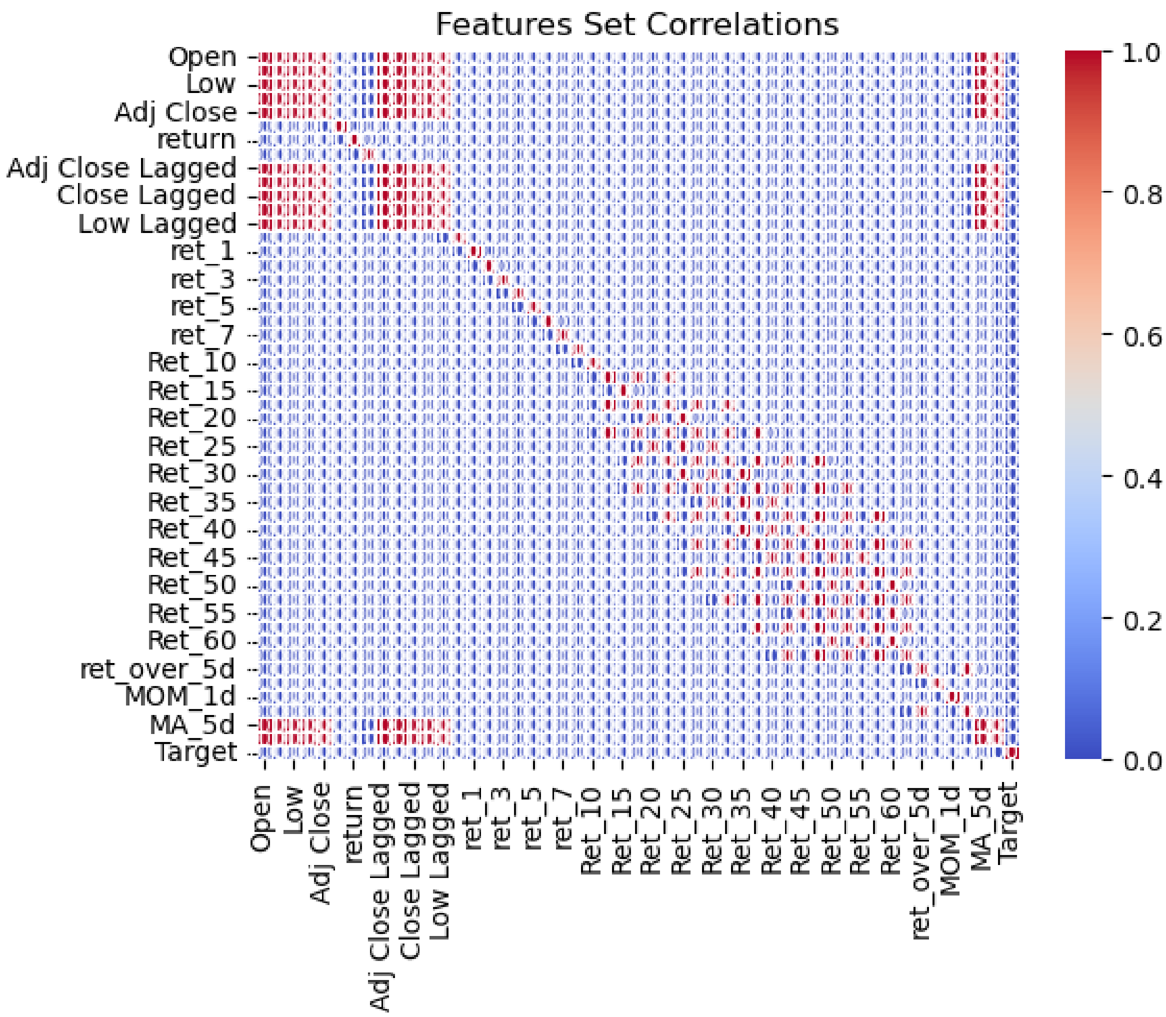
0, if pt+1 Otherwise

where, pt is the Adjusted closing Price of Nifty Index and pt+1 is the 1-day forward Adjusted Closing Price of the index.

```
features_df['Target'] = np.where(features_df['Adj Close'].shift(-1)> 0.99995 * features_df['Adj Close'],1,0)

#creating set for our explained variable
y = features_df['Target']
```

Now that we have our list of features, we will try and find which features are correlated above a threshold of 0.9. We do this to remove any redundant features present in our features list as they would not provide a value add to our model.

```
#create a visualisation of the correlation matrix of features to know which features are highly correlated
sns.heatmap(features_df.corr()>0.9,
   annot=True,
   annot_kws={"size": 8},
   fmt=".2f",
   linewidth=.5,
   cmap="coolwarm",
   cbar=True); #cmap="crest", virids, magma

plt.title('Features Set Correlations');
```




```
# remove the first feature that is correlated with any other feature
def correlated_features(data, threshold=0.9):
   col_corr = set()
   corr_matrix = features_df.corr()
   for i in range(len(corr_matrix.columns)):
      for j in range(i):
         if abs(corr_matrix.iloc[i, j]) > threshold:
            colname = corr_matrix.columns[i]
            col_corr.add(colname)
   return col_corr

# total correlated features
drop_correlated_features = correlated_features(features_df)
```

```
# drop the highly correlated features
New_features_df = features_df.drop(drop_correlated_features, axis=1)

New_features_df

#creating our explanatory variables set without the explained variable as a part of it
A = New_features_df.drop(['return_sign'],axis=1) #dropping return sign as I'm not finding it extremely important to be added at this point.
X = A.drop(['Target'],axis=1)
X
```

| Date | 2019-05-07 | 2019-05-08 | 2019-05-09 | 2019-05-10 | 2019-05-13 | ... | 2023-10-20 | 2023-10-23 | 2023-10-25 |
|---|---|---|---|---|---|---|---|---|---|
| **Open** | 11651.500000 | 11478.700195 | 11322.400391 | 11314.150391 | 11258.700195 | ... | 19542.150391 | 19521.599609 | 19286.449219 |
| **Volume** | 337500.0 | 372800.0 | 373000.0 | 387300.0 | 357600.0 | ... | 198300.0 | 176000.0 | 225300.0 |
| **return** | -0.008690 | -0.012114 | -0.005088 | -0.002028 | -0.011656 | ... | -0.004190 | -0.013440 | -0.008312 |
| **Volume Lagged** | 299000.0 | 337500.0 | 372800.0 | 373000.0 | 387300.0 | ... | 230300.0 | 198300.0 | 176000.0 |
| **ret_1** | -0.009781 | -0.008690 | -0.012114 | -0.005088 | -0.002028 | ... | -0.002362 | -0.004190 | -0.013440 |
| **ret_2** | -0.001067 | -0.009781 | -0.008690 | -0.012114 | -0.005088 | ... | -0.007112 | -0.002362 | -0.004190 |
| **ret_3** | -0.001994 | -0.001067 | -0.009781 | -0.008690 | -0.012114 | ... | 0.004034 | -0.007112 | -0.002362 |
| **ret_4** | -0.000553 | -0.001994 | -0.001067 | -0.009781 | -0.008690 | ... | -0.000978 | 0.004034 | -0.007112 |
| **ret_5** | 0.009647 | -0.000553 | -0.001994 | -0.001067 | -0.009781 | ... | -0.002172 | -0.000978 | 0.004034 |
| **ret_6** | -0.007219 | 0.009647 | -0.000553 | -0.001994 | -0.001067 | ... | -0.000876 | -0.002172 | -0.000978 |
| **ret_7** | 0.012892 | -0.007219 | 0.009647 | -0.000553 | -0.001994 | ... | 0.006152 | -0.000876 | -0.002172 |
| **ret_8** | -0.001597 | 0.012892 | -0.007219 | 0.009647 | -0.000553 | ... | 0.009056 | 0.006152 | -0.000876 |
| **Ret_10** | -0.021927 | -0.020477 | -0.023968 | -0.038888 | -0.043324 | ... | -0.005656 | -0.011889 | -0.029256 |
| **Std_10** | 0.008343 | 0.008133 | 0.008187 | 0.006212 | 0.006621 | ... | 0.005427 | 0.006522 | 0.005758 |
| **Ret_15** | -0.007486 | -0.020671 | -0.029782 | -0.035830 | -0.055732 | ... | 0.000978 | -0.018323 | -0.021040 |
| **Ret_20** | -0.012622 | -0.020783 | -0.031712 | -0.028459 | -0.045910 | ... | -0.010167 | -0.020152 | -0.028479 |
| **ret_over_5d** | -0.003748 | -0.022084 | -0.033646 | -0.036740 | -0.037702 | ... | -0.008590 | -0.010607 | -0.023070 |
| **ret_over_21d** | -0.006094 | -0.018552 | -0.024737 | -0.025871 | -0.033741 | ... | -0.025586 | -0.018191 | -0.023607 |
| **MOM_1d** | -114.000000 | -100.349609 | -138.450196 | -57.650390 | -22.899414 | ... | -46.400390 | -82.048828 | -260.900391 |


Paper ID: SR24418155701 DOI: https://dx.doi.org/10.21275/SR24418155701

| Date | Open | Volume | return | Volume Lagged | ret_1 | ret_2 | ret_3 | ret_4 | ret_5 | ret_6 | ret_7 | ret_8 | Ret_10 | Std_10 | Ret_15 | Ret_20 | ret_over_5d | ret_over_21d | MOM_1d |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2023-10-26 | 19027.250000 | 300400.0 | -0.013950 | 225300.0 | -0.008312 | -0.013440 | -0.004190 | -0.002362 | -0.007112 | 0.004034 | -0.000978 | -0.002172 | -0.049358 | 0.005746 | -0.030235 | -0.041928 | -0.035415 | -0.028463 | -159.599609 |
| 2023-10-27 | 18928.750000 | 205200.0 | 0.010025 | 300400.0 | -0.013950 | -0.008312 | -0.013440 | -0.004190 | -0.002362 | -0.007112 | 0.004034 | -0.000978 | -0.038456 | 0.007398 | -0.025835 | -0.034531 | -0.042253 | -0.042429 | -264.900391 |

989 rows × 19 columns

```
X.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 989 entries, 2019-05-07 to 2023-10-27
Data columns (total 19 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Open          989 non-null    float64
 1   Volume        989 non-null    float64
 2   return        989 non-null    float64
 3   Volume Lagged  989 non-null   float64
 4   ret_1         989 non-null    float64
 5   ret_2         989 non-null    float64
 6   ret_3         989 non-null    float64
 7   ret_4         989 non-null    float64
 8   ret_5         989 non-null    float64
 9   ret_6         989 non-null    float64
 10  ret_7         989 non-null    float64
 11  ret_8         989 non-null    float64
 12  Ret_10        989 non-null    float64
 13  Std_10        989 non-null    float64
 14  Ret_15        989 non-null    float64
 15  Ret_20        989 non-null    float64
 16  ret_over_5d   989 non-null    float64
 17  ret_over_21d  989 non-null    float64
 18  MOM_1d        989 non-null    float64
dtypes: float64(19)
memory usage: 154.5 KB
```

```
# Value counts for class 1 and 0
pd.Series(y).value_counts()
1    533
0    456
Name: Target, dtype: int64
```

Above we see that the two classes are not perfectly balanced i.e. there are more values for class 1 as compared to class 0. Although this could be addressed by changing our earlier threshold for what get's classifed as 1 or 0, it has been kept as is. This is because we see that the general trend in the data is upwards i.e. positive return and that is also the characteristic of the equity market since the economy of a developing country is expected to grow over time (unless it is part of a depression cycle).

That being said, there isnt too severe of a class imbalance in this case.

```
# Splitting the datasets into training and testing data.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shuffle=False)
# Output the train and test data size
print(f"Train and Test Size {len(X_train)}, {len(X_test)}")
Train and Test Size 791, 198
```

Above we have split the data into training and test data with a 80-20 ratio and shuffle has been set to False because Time Series data is sequential in nature.

The training data is used to fit the model. The algorithm that we are going to use will use the training data to learn the relationship between the features and the target. The test data will then be used to evaluate the performance of the model that we have built.

**Base Model**

We now build a base model with the default parameters. Our base model will be built using the Random Forest Classifier which is a machine learning algorithm that creates a forest of decision trees and combines their predictions to make a final prediction i.e it is an ensemble learning technique combining numerous classifiers to enhance a model's performance.

```
model = RandomForestClassifier() #default parameters
model.fit(X_train,y_train)
# Predicting the test dataset
y_pred = model.predict(X_test)
# Predict Probabilities
y_proba = model.predict_proba(X_test)
```

Now that we have fitted our model and added the code to predict as well, we will need to know if the model is any good at making these predictions.

```
acc_train = accuracy_score(y_train, model.predict(X_train))
acc_test = accuracy_score(y_test, y_pred)
```

```
print(f'Train Accuracy: {acc_train:0.4}, Test Accuracy: {acc_test:0.4}')
```

Train Accuracy: 1.0, Test Accuracy: 0.5101

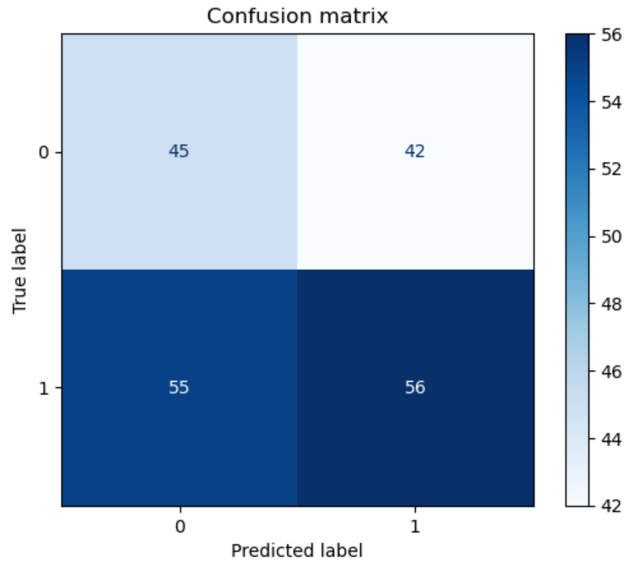
Comparing the Train Accuracy to the Test Accuracy, we can see that clearly the data is overffiting.

Taking a step back, Accuracy is essentially checking the predictions that the model made against the actual values in the set. In other words we are comparing the prediction that was made with the actual values from our dataset. Keeping this in mind, we see that the Train Accuracy is 1.0 and Test Accuracy is 0.5101. This means that our model is memorizing the training data and is not able to generalize on a test dataset. This occurs when the model is too complex and is thus unable to generalize well on data points outside of what is learnt from the training data. It also indicates that the model has low bias meaning that it is overly expressive. In the Bias - Variance tradeoff, this model has high Variance and low Bias.

Thus, in the following part of this exercise we look to reduce the complexity of this model i.e. reduce the variance problem so that it is able to generalize on a test dataset. This can be achieved through hyperparameter tuning.

But let us first look at some more interesting observations about how the algorithm has performed.

```
disp = ConfusionMatrixDisplay.from_estimator(
model,
X_test,
y_test,
display_labels=model.classes_,
cmap=plt.cm.Blues
)
disp.ax_.set_title('Confusion matrix')
plt.show()
```




From the above confusion matrix we can see that the true positive values are greater than the false positive i.e. the model is predicting the uptrend (class 1) correctly more times than it is misclassifying it. The same is with the case with the downtrend (class 0).

This is another indication that we would need to fine tune our model some more.

We will now take a look at the Classification Report which will give us a table of addtional metrics we can use to guage the performance of the model.

```
print(classification_report(y_test, y_pred))
```

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.45 | 0.52 | 0.48 | 87 |
| 1 | 0.57 | 0.50 | 0.54 | 111 |
| accuracy | | | 0.51 | 198 |
| macro avg | 0.51 | 0.51 | 0.51 | 198 |
| weighted avg | 0.52 | 0.51 | 0.51 | 198 |

Building from the comments about the confusion matrix wherein the true positives were predicted greater than the true negatives is also visible in the accuracy report when we take a look at the 'precision' column. Precision tells us how many selected items are relevant whereas 'recall' tells us how many relevant items are selected.

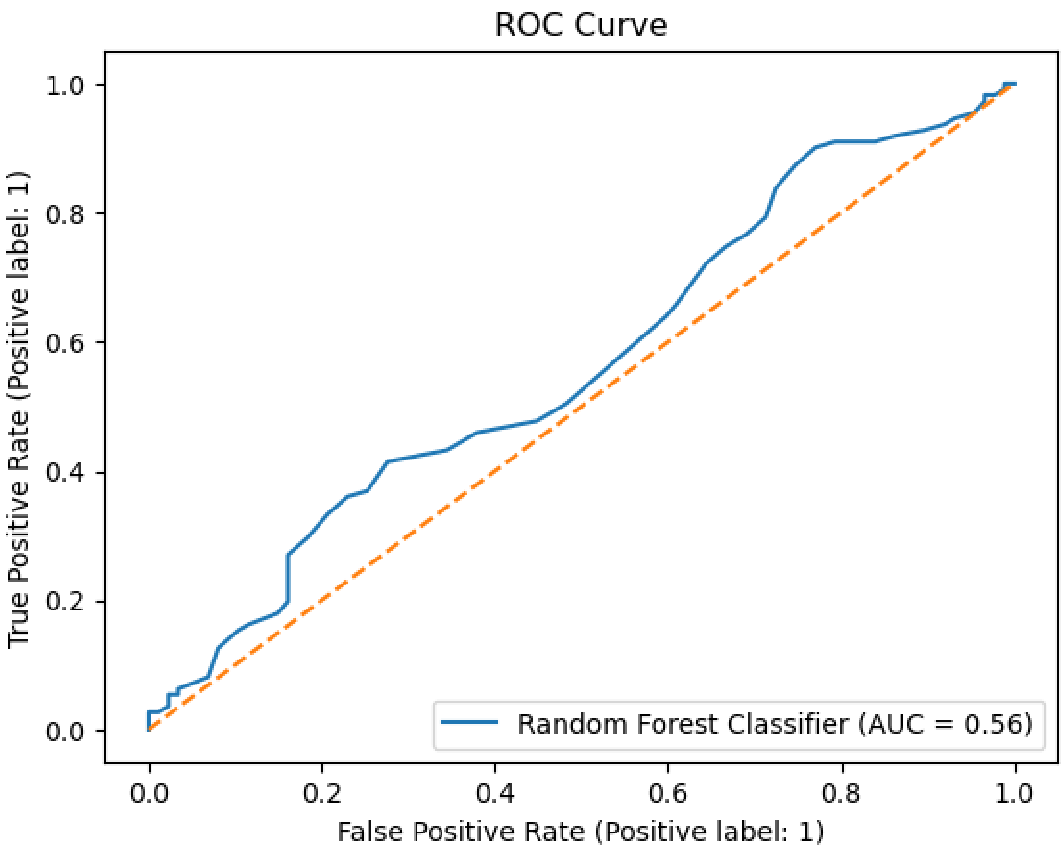
We will now look at another metric called the ROC (Receiver Operating Characteristic) Curve below.

```
# Display ROCCurve
disp_roc = RocCurveDisplay.from_estimator(
model,
X_test,
y_test,
name='Random Forest Classifier')
disp_roc.ax_.set_title('ROC Curve')
plt.plot([0,1], [0,1], linestyle='--')
plt.show()
```



Paper ID: SR24418155701 DOI: https://dx.doi.org/10.21275/SR24418155701

International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2022): 7.942




The ROC Curve tells us the tradeoff netween the True Positive Rate and the False Positive Rate and hence we check for the steepness of the curve. The graph shows us the performance of the model at all classification thresholds.
The objective is to defeat randomness with our model so an ROC curve greater than 0.5 would mean that it is working better than a fair coin toss. In our case we see that the value is greater than 0.5. Although the value isnt extremely high and signifies a week learning algorithm, it will perform better than a coint toss for predictions.

Now that we have seen how our base model is performing, we will try and enhance it's performance. This is done through hyperparameter tuning. Hyperparameters are parameters that are not directly learnt within estimators. It is possible and recommended to search the hyperparameter space for the best cross validation score. Any parameter provided when constructing an estimator may be optimized in this manner. Hyperparameter tuning is a method to choose the best loss minimizing function to maximize Accuracy or whatever function we are scoring for (example F1 score, etc.)

First we will get a list of parameters used in our model, then we will tune the hyperparameters to select the best score by TimeSeriesSplit cross-validation. Once we get a list of the best parameters and best score, we will tune our base model to use these parameters. Once we have fitted the model with the best parameters, we will go through all the above metrics again to see if the model has improved or not.

```
# Get params list
model.get_params()

{'bootstrap': True,
'ccp_alpha': 0.0,
'class_weight': None,
'criterion': 'gini',
'max_depth': None,
'max_features': 'auto',
'max_leaf_nodes': None,
'max_samples': None,
'min_impurity_decrease': 0.0,
'min_samples_leaf': 1,
'min_samples_split': 2,
'min_weight_fraction_leaf': 0.0,
'n_estimators': 100,
'n_jobs': None,
'oob_score': False,
'random_state': None,
'verbose': 0,
'warm_start': False}

# Timeseries CV 3-split
tscv = TimeSeriesSplit(n_splits=5, gap=1)

# Hyper parameter optimization
param_grid = {
   'n_estimators': [25, 50, 100, 150],
   'max_features': ['sqrt', 'log2', None],
   'max_depth': [3, 6, 9],
   'max_leaf_nodes': [3, 6, 9],
}
```

The RandomizedSearchCV implements a “fit” and a “score” method and perform randomized search on hyperparameters. The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings. Not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions.

```
# perform random search
rs = RandomizedSearchCV(model, param_grid, n_iter=100, scoring='f1', cv=tscv,verbose=0)
rs.fit(X_train, y_train)

RandomizedSearchCV(cv=TimeSeriesSplit(gap=1, max_train_size=None, n_splits=5, test_size=None),
                   estimator=RandomForestClassifier(), n_iter=100
,
                   param_distributions={'max_depth': [3, 6, 9],
                                        'max_features': ['sqrt', 'log2', None],
                                        'max_leaf_nodes': [3, 6, 9],
                                        'n_estimators': [25, 50, 100, 150]},
                   scoring='f1')

# best parameters

rs.best_params_

{'n_estimators': 25,
 'max_leaf_nodes': 3,
 'max_features': 'sqrt',
 'max_depth': 9}

# best score
rs.best_score_

0.6387682740908026
```

**Tuned Model**

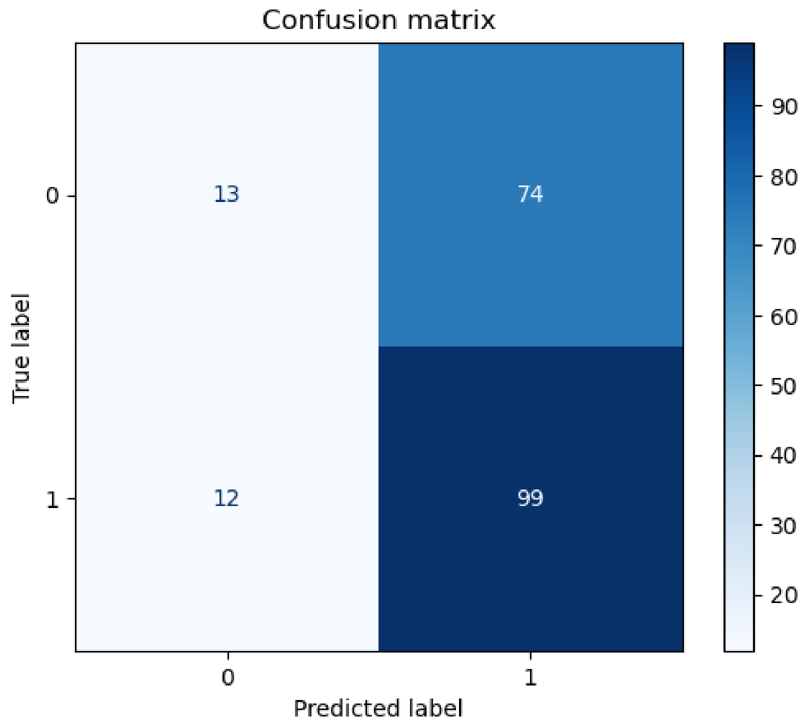
Now that we have our best parameters and score, we will refit our base modeel to use these parameters to check for improvements in the performance of our model.

```
# Refit the Random Forest Classifier with the best params
cls = RandomForestClassifier(**rs.best_params_)
cls.fit(X_train, y_train)
eval_set=[(X_train, y_train), (X_test, y_test)]

score = cross_val_score(cls,X_train,y_train,cv=tscv)
print(f'Mean CV Score : {score.mean():0.4}')
Mean CV Score : 0.5145

# Predicting the test dataset
y_pred = cls.predict(X_test)
# Measure Accuracy
acc_train = accuracy_score(y_train, cls.predict(X_train))
acc_test = accuracy_score(y_test, y_pred)
# Print Accuracy
print(f'\n Training Accuracy \t: {acc_train :0.4} \n Test Accuracy \t\t:{acc_test :0.4}')
 Training Accuracy      : 0.6106
 Test Accuracy                          :0.5657
```

We can now see that the Training and Test Accuracy scores have changed as compared to our base model. The Training score has drastically reduced meaning that our algorithm is not memorizing the dataset anymore. It has come down severly from a perfect score of 1 which indicates that the algorithm has been simplified and isnt as complex as before. The score is also extremely close to the Test Accuracy Score, although the Test Accuracy score is still a little lower than Training Accuracy score, there isnt as much of a gap between the two anymore. This means that although there is slight overfitting taking place, the tuned model is performaing better than the base model which is the purpose of hyperparameter tuning. This is also visible in the fact that the Test Accuracy score of the tuned model is greater than the base model.

```
# Display confussion matrix
disp = ConfusionMatrixDisplay.from_estimator(
cls,
X_test,
y_test,
display_labels=model.classes_,
cmap=plt.cm.Blues
)
disp.ax_.set_title('Confusion matrix')
plt.show()
```

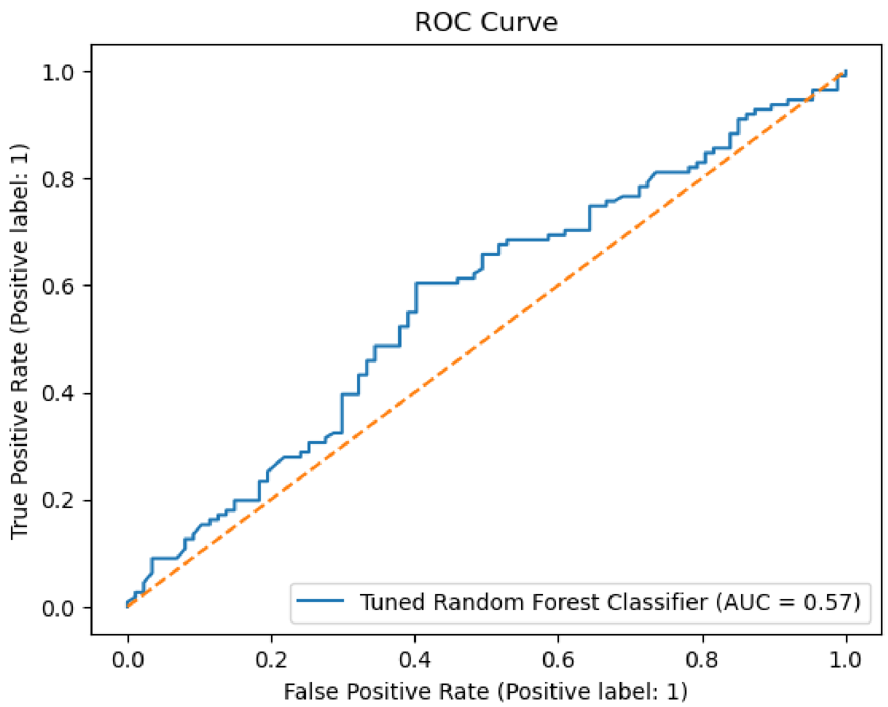

Paper ID: SR24418155701 DOI: https://dx.doi.org/10.21275/SR24418155701




We can see that the True Positives VS False Positive ratio has increased as compared to that from the base model. This is one of the indicators that the model is performing better as it is able to classify the positive class better.

```
# Classification Report
print(classification_report(y_test, y_pred))
              precision    recall  f1-score   support

           0       0.52      0.15      0.23        87
           1       0.57      0.89      0.70       111

    accuracy                           0.57       198
   macro avg       0.55      0.52      0.46       198
weighted avg       0.55      0.57      0.49       198
```

```
# Display ROCCurve
disp_roc = RocCurveDisplay.from_estimator(
cls,
X_test,
y_test,
name='Tuned Random Forest Classifier')
disp_roc.ax_.set_title('ROC Curve')
plt.plot([0,1], [0,1], linestyle='--')
plt.show()
```



The ROC Curve metric now has a greater score of 0.57 compared to the score of 0.56 of the base model. Although it is only slightly better and the results arent extremely promising, it still shows an improvement in our model.

## Concluding Remarks

We can see from our Training and Test Accuracy Scores for the base and tuned model that this algorithm is indeed a week learner. However, if the features are further refined and selected, then the accuracy score would improve. It takes quite a lot of time (can take 6 months or so ) to try and get the correct set of features to improve the model.

No random seed parameter has been applied through this exercise so rerunning the code could lead to different results since the data would be split randomly on the rerun. This would give different training and test datasets on every run.