CQF INSTITUTE

CQF Exam Three

Machine Learning

Bach Van Hoang Bao
June 2023 cohort

FitchLearning

November 9, 2023

1 What are voting classifiers in ensemble learning?

Voting classifiers are based on the idea of aggregating the predictions of multiple classifiers to make a final decision [1]. There are two main types of voting classifiers:

1. Majority Class Labels (Majority/Hard Voting): In majority voting, the predicted class label for a particular sample is the class label that represents the majority (mode) of the class labels predicted by each individual classifier.

   E.g., if the prediction for a given sample is

   - classifier 1 $\rightarrow$ class 1
   - classifier 2 $\rightarrow$ class 1
   - classifier 3 $\rightarrow$ class 2

   The VotingClassifier (with voting = 'hard') would classify the sample as "class 1" based on the majority class label.

   In the cases of a tie, the VotingClassifier will select the class based on the ascending sort order. E.g., in the following scenario

   - classifier 1 $\rightarrow$ class 2
   - classifier 2 $\rightarrow$ class 1

   The class label 1 will be assigned to the sample.

2. Soft Voting Classifier: In contrast to majority voting (hard voting), soft voting returns the class label as argmax of the sum of predicted probabilities.

   Specific weights can be assigned to each classifier via the weights parameter. When weights are provided, the predicted class probabilities for each classifier are collected, multiplied by the classifier weight, and averaged. The final class label is then derived from the class label with the highest average probability.

   To illustrate this with a simple example, let's assume we have 3 classifiers and a 3-class classification problems where we assign equal weights to all classifiers: $w_1 = 1, w_2 = 1, w_3 = 1$.

   The weighted average probabilities for a sample would then be calculated as follows:

Classifier	Class 1	Class 2	Class 3
Classifier 1	$w_1 \times 0.2$	$w_1 \times 0.5$	$w_1 \times 0.3$
Classifier 2	$w_2 \times 0.6$	$w_2 \times 0.3$	$w_2 \times 0.1$
Classifier 3	$w_3 \times 0.3$	$w_3 \times 0.4$	$w_3 \times 0.3$
Weighted Avg	0.37	0.4	0.23

   Here, the predicted class label is 2, since it has the highest average probability.

2 Explain the role of the regularization parameter C in a Support Vector Machine (SVM) model. How does varying C affect the model's bias and variance trade-off?

Consider the mathematical equation for the solf margin of non linearly separable data [2] $\mathbf{x}_n$, y_n:

$$
\begin{aligned}
\min_{\boldsymbol{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{n=1}^{N}\xi_n \\
\text{subject to} \quad & y_n(\langle \boldsymbol{w}, \boldsymbol{x}_n\rangle + b) \geqslant 1 - \xi_n \\
& \xi_n \geqslant 0
\end{aligned}
$$

Where:

- $\boldsymbol{w}$ is the normal vector of the hyper plane
- $\|\boldsymbol{w}\|^2$ is the regularizer
- $\boldsymbol{x}_n$ is the feature vector n^{th}
- y_n is the label n^{th}
- ξ_n is the slack term that measures the distance of a positive example x_+ to the positive margin hyperplane ($\langle \mathbf{w}, \mathbf{x}\rangle + b = 1$) when x_+ is on the wrong side.

The parameter $C > 0$ trades off the size of the margin and the total amount of slack that we have. A large value of C implies low regularization, as we give the slack variables larger weight, hence giving more priority to examples that do not lie on the correct side of the margin.

Let take a look at the impact of C in the model using the breast cancer data from `Scikit-Learn` library.

[1]:
```python
import pandas as pd
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.datasets import load_breast_cancer
from sklearn.svm import SVC
import warnings
warnings.filterwarnings("ignore")

cancer = load_breast_cancer()

# The data set is presented in a dictionary form:
print(cancer.keys())
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
'filename', 'data_module'])
```

Now we will extract all features into the new data frame and our target features into separate data frames.

[2]:
```
df_feat = pd.DataFrame(cancer['data'], columns = cancer['feature_names'])
# cancer column is our target
df_target = pd.DataFrame(cancer['target'], columns =['Cancer'])
print("Feature Variables: ")
print(df_feat.info())
```

```
Feature Variables:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 569 entries, 0 to 568
Data columns (total 30 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   mean radius              569 non-null    float64
 1   mean texture             569 non-null    float64
 2   mean perimeter           569 non-null    float64
 3   mean area                569 non-null    float64
 4   mean smoothness          569 non-null    float64
 5   mean compactness         569 non-null    float64
 6   mean concavity           569 non-null    float64
 7   mean concave points      569 non-null    float64
 8   mean symmetry            569 non-null    float64
 9   mean fractal dimension   569 non-null    float64
 10  radius error             569 non-null    float64
 11  texture error            569 non-null    float64
 12  perimeter error          569 non-null    float64
 13  area error               569 non-null    float64
 14  smoothness error         569 non-null    float64
 15  compactness error        569 non-null    float64
 16  concavity error          569 non-null    float64
 17  concave points error     569 non-null    float64
 18  symmetry error           569 non-null    float64
 19  fractal dimension error  569 non-null    float64
 20  worst radius             569 non-null    float64
 21  worst texture            569 non-null    float64
 22  worst perimeter          569 non-null    float64
 23  worst area               569 non-null    float64
 24  worst smoothness         569 non-null    float64
 25  worst compactness        569 non-null    float64
 26  worst concavity          569 non-null    float64
 27  worst concave points     569 non-null    float64
 28  worst symmetry           569 non-null    float64
 29  worst fractal dimension  569 non-null    float64
dtypes: float64(30)
memory usage: 133.5 KB
None
```

We will split the training data and the test data using 70:30 ratio

[3]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split( df_feat, np.
 ↪ravel(df_target),
                                                     test_size = 0.30, random_state = 101)
```

And fit the data to our SVC model. After that we can see the hyperparameters of our model using `get_param()` method.

[4]:
```
# train the model on train set
model = SVC()
model.fit(X_train, y_train)
```

[5]:
```
model.get_params()
```

[5]:
```
{'C': 1.0,
 'break_ties': False,
 'cache_size': 200,
 'class_weight': None,
 'coef0': 0.0,
 'decision_function_shape': 'ovr',
 'degree': 3,
 'gamma': 'scale',
 'kernel': 'rbf',
 'max_iter': -1,
 'probability': False,
 'random_state': None,
 'shrinking': True,
 'tol': 0.001,
 'verbose': False}
```

The default value of C is 1. Now we can observe the model performace using the confusion matrix.

[6]:
```
# print prediction results
predictions = model.predict(X_test)
print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.95      0.85      0.90        66
           1       0.91      0.97      0.94       105

    accuracy                           0.92       171
   macro avg       0.93      0.91      0.92       171
weighted avg       0.93      0.92      0.92       171
```

Let's change the value of $C = 0.001$ and see the result.

[7]:
```
# train the model on train set
model = SVC(C=0.001)
model.fit(X_train, y_train)
# print prediction results
predictions = model.predict(X_test)
print(classification_report(y_test, predictions))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        66
           1       0.61      1.00      0.76       105

    accuracy                           0.61       171
   macro avg       0.31      0.50      0.38       171
weighted avg       0.38      0.61      0.47       171
```

2.1 Conclusion

When C is set to a small value (e.g., close to 0), the SVM places a higher emphasis on maximizing the margin and finding the hyperplane that separates the data points with as few errors as possible. In this case, the model is more tolerant of misclassifications (training errors) and is willing to accept a wider margin with a few support vectors. The model's bias is higher, as it tends to underfit the training data by allowing more training errors, but the variance is lower because it maintains a simpler decision boundary.

When C is set to a large value, the SVM imposes a stronger penalty on misclassified points and strives to minimize training errors, even if it means having a narrower margin and more support vectors. The model's bias is lower because it tries to fit the training data as closely as possible, potentially leading to a more complex decision boundary. However, the variance is higher because the model is sensitive to individual data points, which can result in overfitting.

3 Produce a model to predict positive moves (up trend) using machine learning model.

In this section, we will create a Machine Learning (ML) model to predict positive movements using raw financial data from Yahoo Finance. The goal is to emphasize the work flow structure and develop a comprehensive framework from ideation to model evaluation, applying techniques and knowledge from the CQF module 4.

The 7 steps of ML work flow [3]:

Step	Workflow	Remark
1	Ideation	Predict positive moves from the given dataset
2	Data Collection	Download the data from Yahoo Finance and store the data set
3	EDA	Study summary statistics
4	Cleaning Dataset	Trying to resolve the missing data
5	Transformation	Perform feature scaling based on EDA
6	Modelling	Building and training classification model
7	Metrics	Validating the model performance

3.1 STEP 1: Ideation

The objective of the exam is to create a model for predicting upward and downward movements of the underlying asset. This is a classification problem, so we will approach it by assigning a classification label `[0]` for a downward trend and `[1]` for an upward trend. We will use the `SPY` ticker as an example and utilize data from `2008-10-16` to `2023-10-16`. The reason for choosing this date range is that it encompasses various financial regimes, including the 2008 financial crash and the 2020 COVID-19 recession.

Given that we are working with financial time series data, the work flow is relatively straightforward. After downloading and storing the data, we will begin by exploring the data to identify any meaningful structures or trends. Next, we'll create the classification labels and address class imbalance. We will apply feature engineering techniques to generate new features from the original data and select the most crucial features for our model. The data will be saved under `../SPY1D.csv`.

Subsequently, we will split the data into training and testing sets, fitting and transforming the training set, and then transforming the test set. This ensures that we avoid any data leakage issues. We will explore the training set to identify trends and significant structures in the data. If necessary, we will scale the data and use the transformed data to train our model.

During the model training process, we will compare the cross-validation accuracy of multiple classification algorithms with default parameters and select the most promising candidate for further tuning. Hyper parameters will be tuned to optimize the selected candidate, leading to the creation of a final model. This final model will be saved as `final_model.joblib` for future use.

In terms of performance measurement, we will employ the confusion matrix and AUC-ROC curve to evaluate our model's performance and document the entire process.

3.2 STEP 2: Data Collection

We will use the `yfinance` package to download daily trading data from Yahoo Finance. The recommended data should span a 5-year period, which is considered sufficient. The downloaded data will be saved in the `.csv` format and can be accessed later using the file name `SPY1D.csv`.

[9]:
```
# Download data for TSLA and store as csv file
spy = yf.download("SPY", start = '2008-10-16', end = '2023-10-16' ,␣
↪interval='1D')
spy.to_csv('SPY1D.csv')
```

```
[*********************100%%***********************]  1 of 1 completed
```

[10]:
```
spy = pd.read_csv('../module_4/SPY1D.csv')
```

[11]:
```
# Verify the downloaded data
spy.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3774 entries, 0 to 3773
Data columns (total 7 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   Date       3774 non-null   object
 1   Open       3774 non-null   float64
 2   High       3774 non-null   float64
 3   Low        3774 non-null   float64
 4   Close      3774 non-null   float64
 5   Adj Close  3774 non-null   float64
 6   Volume     3774 non-null   int64
dtypes: float64(5), int64(1), object(1)
memory usage: 206.5+ KB
```

3.3 STEP 3: EDA

Visualize asset path:

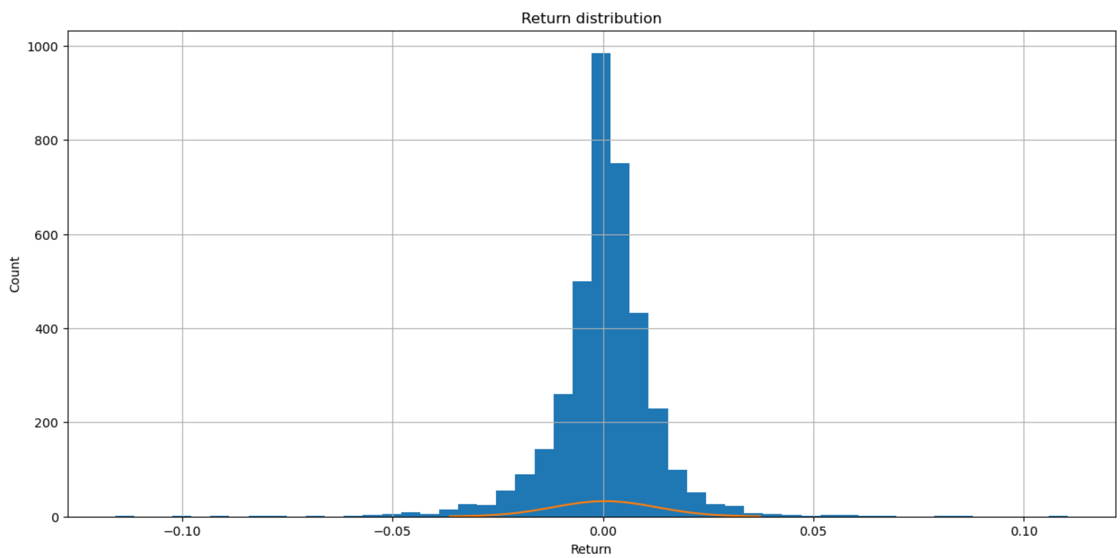
[12]:
```
import plotly.express as px

fig = px.line(spy, x= 'Date', y='Adj Close', labels = {'Adj Close': 'Close Price␣
↪(USD)'}, title = 'S&P 500 ETF Trust (SPY) Daily')
fig.show();
```

3.3.1 Calculate returns

We can plot the distribution of returns and the closing price movement to identify any trends or significant information regarding the returns that could be useful.

[13]:
```
spy['Returns'] = np.log(spy['Adj Close']).diff()
```

[14]:
```
from scipy.stats import norm

# Plot the return histogram
fig = plt.figure(figsize=(15, 7))
ax1 = fig.add_subplot(1, 1, 1)
spy['Returns'].hist(bins=50, ax=ax1)
ax1.set_xlabel('Return')
ax1.set_ylabel('Count')
ax1.set_title('Return distribution')

# Plot the normal distribution
mu = spy['Returns'].mean()
sigma = spy['Returns'].std()
x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
plt.plot(x, norm.pdf(x, mu, sigma))
plt.show()
```



The return is definately not normally distributed. There is a high peak and very fat tails.

3.3.2 Feature Specify

Using the feature list table from the exam, we will generate features based on the historical data we have acquired. Additionally, I've included 10 lagged prices in the feature list, operating on the assumption that historical data may possess predictive capabilities.

[15]:
```
# Create features (predictors) list
features_list = []
# Intraday price range
spy['OC'] = spy['Open'] - spy['Close']
```

```
spy['HL'] = spy['High'] - spy['Low']
# Sign of return or momentum
spy['Sign'] = np.sign(spy.Returns)

# Append feature list
features_list.append('OC')
features_list.append('HL')
features_list.append('Sign')

# Pass Returns, Volatility
for r in range(10, 65, 5):
    spy['Ret_'+str(r)] = spy.Returns.rolling(r).sum()
    spy['Std_'+str(r)] = spy.Returns.rolling(r).std()
    features_list.append('Ret_'+str(r))
    features_list.append('Std_'+str(r))

# SMA and EMA
for a in range(20, 200, 10):
    spy['SMA_'+str(r)] = spy['Adj Close'].rolling(r).mean()
    spy['EMA_'+str(a)] = spy['Adj Close'].ewm(span = a).mean()
    features_list.append('SMA_'+str(r))
    features_list.append('EMA_'+str(r))

# Lag price
for lag in range(1, 10):
    spy['lag_' + str(lag)] = spy['Adj Close'].shift(lag)

# Drop NaN values
spy.dropna(inplace=True)
```

3.3.3 Define target

We define the target variable to be whether the ‘SPY’ price will close up or down on the next trading day. If tomorrow’s closing price is greater than today’s closing price by at least 5%, we consider the asset to be “up”; otherwise, it is considered “down.”

We assign a value of `1` to denote an “up” move and `0` to represent a “down” move for the target variable. This target variable can be described as follows:

$$y_t = \begin{cases} 1, & \text{If} \quad p_t < 0.995 \times p_{t+1} \\ 0, & \text{Otherwise} \end{cases}$$

[16]:
```
# Define Target
spy['Target'] = np.where(spy['Adj Close'].shift(-1) > 0.995 * spy['Adj␣
↪Close'],1,0)
# Check output
spy.head(10)
```

I am going to split the data into the `train_set` and `test_set` and perform exploratory data analysis (EDA) and data cleaning exclusively on the `train_set` to prevent any potential data leakage from the EDA process.

[17]:
```
# Copy the original data
data = spy.copy().set_index('Date')
```

[18]:
```
# Specify the features matrix `X`
X = data.drop(['Open', 'Close', 'High', 'Low', 'Adj Close', 'Returns', 'Volume',␣
↪'Target'],axis=1)
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 3714 entries, 2009-01-13 to 2023-10-13
Data columns (total 53 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   OC       3714 non-null   float64
 1   HL       3714 non-null   float64
 2   Sign     3714 non-null   float64
 3   Ret_10   3714 non-null   float64
 4   Std_10   3714 non-null   float64
 5   Ret_15   3714 non-null   float64
 6   Std_15   3714 non-null   float64
 7   Ret_20   3714 non-null   float64
 8   Std_20   3714 non-null   float64
 9   Ret_25   3714 non-null   float64
 10  Std_25   3714 non-null   float64
 11  Ret_30   3714 non-null   float64
 12  Std_30   3714 non-null   float64
 13  Ret_35   3714 non-null   float64
 14  Std_35   3714 non-null   float64
 15  Ret_40   3714 non-null   float64
 16  Std_40   3714 non-null   float64
 17  Ret_45   3714 non-null   float64
 18  Std_45   3714 non-null   float64
 19  Ret_50   3714 non-null   float64
 20  Std_50   3714 non-null   float64
 21  Ret_55   3714 non-null   float64
 22  Std_55   3714 non-null   float64
 23  Ret_60   3714 non-null   float64
 24  Std_60   3714 non-null   float64
 25  SMA_60   3714 non-null   float64
 26  EMA_20   3714 non-null   float64
 27  EMA_30   3714 non-null   float64
 28  EMA_40   3714 non-null   float64
 29  EMA_50   3714 non-null   float64
 30  EMA_60   3714 non-null   float64
```

```
 31  EMA_70   3714 non-null   float64
 32  EMA_80   3714 non-null   float64
 33  EMA_90   3714 non-null   float64
 34  EMA_100  3714 non-null   float64
 35  EMA_110  3714 non-null   float64
 36  EMA_120  3714 non-null   float64
 37  EMA_130  3714 non-null   float64
 38  EMA_140  3714 non-null   float64
 39  EMA_150  3714 non-null   float64
 40  EMA_160  3714 non-null   float64
 41  EMA_170  3714 non-null   float64
 42  EMA_180  3714 non-null   float64
 43  EMA_190  3714 non-null   float64
 44  lag_1    3714 non-null   float64
 45  lag_2    3714 non-null   float64
 46  lag_3    3714 non-null   float64
 47  lag_4    3714 non-null   float64
 48  lag_5    3714 non-null   float64
 49  lag_6    3714 non-null   float64
 50  lag_7    3714 non-null   float64
 51  lag_8    3714 non-null   float64
 52  lag_9    3714 non-null   float64
dtypes: float64(53)
memory usage: 1.5+ MB
```

[19]:
```python
# Define label or target vector `y`
y = data['Target']
y
```

[19]:
```
Date
2009-01-13    0
2009-01-14    1
2009-01-15    1
2009-01-16    0
2009-01-20    1
             ..
2023-10-09    1
2023-10-10    1
2023-10-11    0
2023-10-12    1
2023-10-13    0
Name: Target, Length: 3714, dtype: int64
```

[20]:
```python
# 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 2971, 743
```

3.3.4 Imbalance class

Since this is a classification problem, it's important to check for any imbalances in our labels.

[21]:
```
# class frequency
c = y_train.value_counts()
c
```

[21]:
```
1    2361
0     610
Name: Target, dtype: int64
```

The label is imbalanced. We will create a weight function and subsequently use it to address our problem when building a model.

[22]:
```
# class weight function
def cwts(label):
    c0, c1 = np.bincount(label)
    w0=(1/c0)*(len(label))/2
    w1=(1/c1)*(len(label))/2
    return {0: w0, 1: w1}
```

[23]:
```
# check class weights
class_weight = cwts(y_train)
class_weight
```

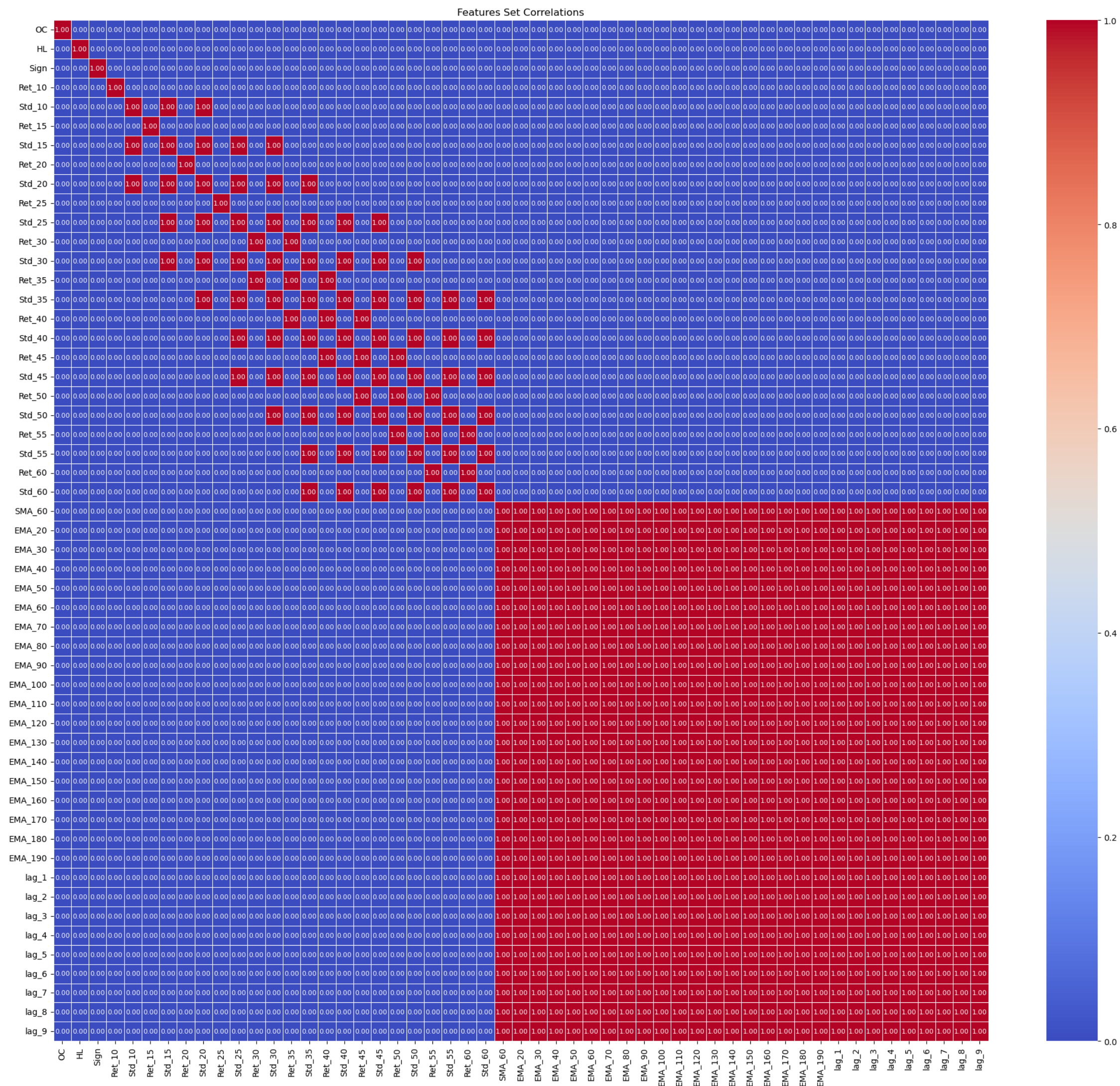
[23]:
```
{0: 2.435245901639344, 1: 0.6291825497670478}
```

3.3.5 Multi collinearity features

Collinear features can adversely affect our model's performance. We will create a function to help us identify and drop these features, and then apply it to our test dataset. Let's also visualize our correlation matrix using the `sns.heatmap()` method.

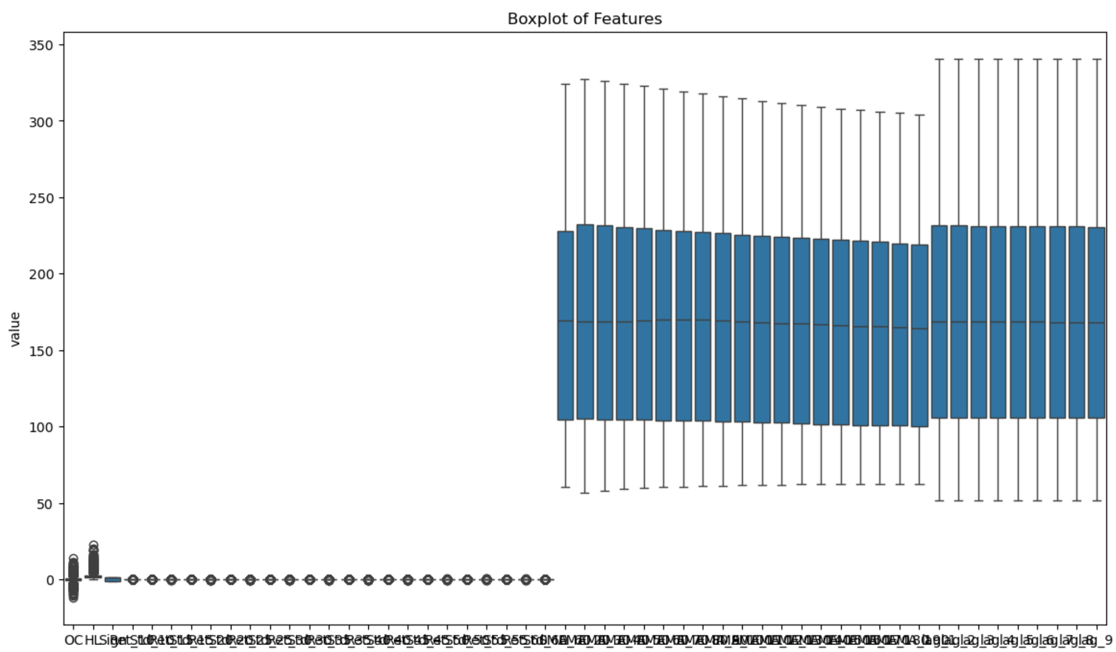
[24]:

```
plt.figure(figsize=(25, 22))

# Identify features that are highly correlated
sns.heatmap(X_train.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');
```



Feature scaling is also a crucial factor in our model's accuracy. We need to scale the data before inputting it into our learning algorithm. We can easily identify features that require scaling by using the `sns.boxplot()` method.

[25]:
```
# study the distribution
fig, ax = plt.subplots(figsize=(14,8))
sns.boxplot(x='variable', y='value', data=pd.melt(X_train))
plt.xlabel(' ')
plt.title('Boxplot of Features');
```



Alternatively, we can identify features that require scaling by using the `pd.describe()` method.

[26]:
```
X_train.describe()
```

[26]:
```
                OC           HL         Sign       Ret_10       Std_10  \
count  2971.000000  2971.000000  2971.000000  2971.000000  2971.000000
mean     -0.032289     2.180993     0.119488     0.005261     0.009328
std       1.653484     1.984382     0.991306     0.033085     0.006957
min     -11.680008     0.299995    -1.000000    -0.265117     0.001264
25%      -0.750000     1.129997    -1.000000    -0.007946     0.005041
50%      -0.110001     1.619995     1.000000     0.008294     0.007418
75%       0.580017     2.490005     1.000000     0.022695     0.011363
max      13.729996    22.960007     1.000000     0.195407     0.071223

            Ret_15       Std_15       Ret_20       Std_20       Ret_25  ...  \
count  2971.000000  2971.000000  2971.000000  2971.000000  2971.000000  ...
```

```
mean       0.007996     0.009473     0.010643     0.009577     0.013273  ...
std        0.040366     0.006703     0.045877     0.006543     0.050274  ...
min       -0.320822     0.001494    -0.370872     0.002007    -0.409051  ...
25%       -0.007578     0.005288    -0.007721     0.005570    -0.006560  ...
50%        0.013048     0.007591     0.016520     0.007800     0.019930  ...
75%        0.028728     0.011410     0.034884     0.011496     0.040088  ...
max        0.241287     0.065627     0.212051     0.059167     0.248100  ...

            EMA_190        lag_1        lag_2        lag_3        lag_4  \
count   2971.000000  2971.000000  2971.000000  2971.000000  2971.000000
mean     163.806877   171.258269   171.175743   171.089990   171.003766
std       68.167202    71.091955    71.070362    71.040253    71.009352
min       62.545531    51.386787    51.386787    51.386787    51.386787
25%      100.199986   105.710148   105.662731   105.651165   105.642242
50%      164.247036   168.587280   168.572571   168.515594   168.472672
75%      219.259284   231.435677   231.315384   231.196716   231.115906
max      304.055049   340.724152   340.724152   340.724152   340.724152

              lag_5        lag_6        lag_7        lag_8        lag_9
count   2971.000000  2971.000000  2971.000000  2971.000000  2971.000000
mean     170.916197   170.828847   170.742130   170.654513   170.567015
std       70.972773    70.937136    70.902683    70.868631    70.835903
min       51.386787    51.386787    51.386787    51.386787    51.386787
25%      105.641220   105.630589   105.619759   105.619560   105.594227
50%      168.462601   168.411850   168.332382   168.284988   168.247406
75%      231.058403   230.981613   230.868690   230.719635   230.638336
max      340.724152   340.724152   340.724152   340.724152   340.724152

[8 rows x 53 columns]
```

Some features exhibit significantly higher absolute values compared to the others. For these features, we will use the `MinMaxScaler()` method to scale them appropriately.

3.4 STEP 4: Cleaning Data

From our exploratory data analysis (EDA) process, we have identified multicollinear features. We will develop a function to eliminate these features and then implement it on our training data. Subsequently, we will apply the same function to our test data.

[27]:
```
# remove the first feature that is correlated with any other feature
def correlated_features(data, threshold=0.9):
    col_corr = set()
    corr_matrix = X_train.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
```

[28]:
```
# Get the list of remaining features
drop_correlated_features = correlated_features(X_train, threshold=0.9)
```

[29]:
```
# drop the highly correlated features
X_train_drop = X_train.drop(drop_correlated_features, axis=1)
X_train_drop.describe()
```

[29]:
```
                OC           HL         Sign       Ret_10       Std_10  \
count  2971.000000  2971.000000  2971.000000  2971.000000  2971.000000
mean     -0.032289     2.180993     0.119488     0.005261     0.009328
std       1.653484     1.984382     0.991306     0.033085     0.006957
min     -11.680008     0.299995    -1.000000    -0.265117     0.001264
25%      -0.750000     1.129997    -1.000000    -0.007946     0.005041
50%      -0.110001     1.619995     1.000000     0.008294     0.007418
75%       0.580017     2.490005     1.000000     0.022695     0.011363
max      13.729996    22.960007     1.000000     0.195407     0.071223

            Ret_15       Ret_20       Ret_25       Ret_30       SMA_60
count  2971.000000  2971.000000  2971.000000  2971.000000  2971.000000
mean      0.007996     0.010643     0.013273     0.015933   168.784765
std       0.040366     0.045877     0.050274     0.053629    69.918583
min      -0.320822    -0.370872    -0.409051    -0.392927    60.405024
25%      -0.007578    -0.007721    -0.006560    -0.005451   104.476129
50%       0.013048     0.016520     0.019930     0.021935   169.387304
75%       0.028728     0.034884     0.040088     0.044824   227.957799
max       0.241287     0.212051     0.248100     0.249708   323.832030
```

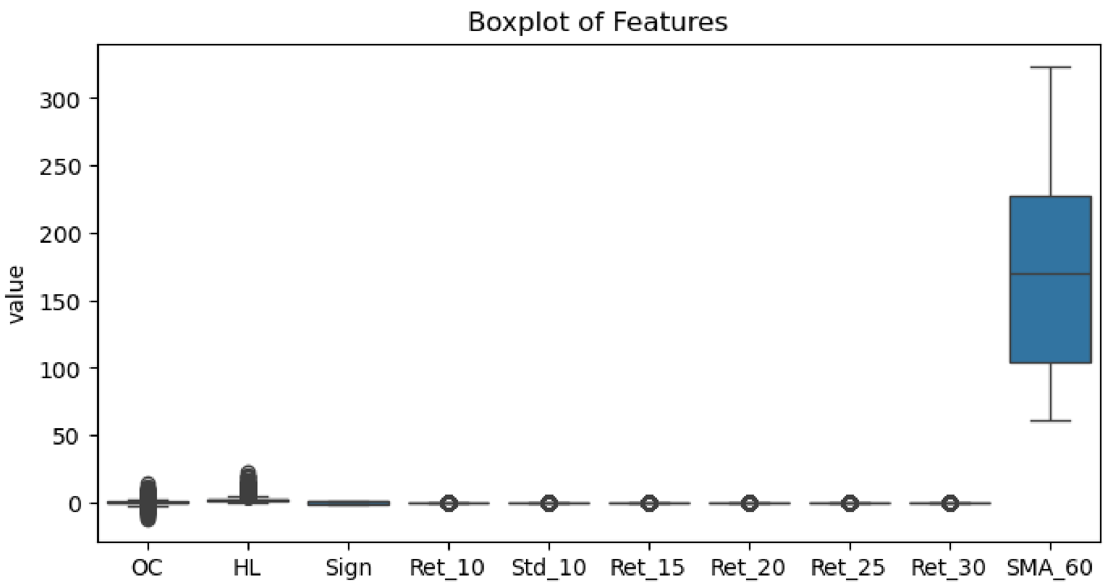
After removing most of the highly correlated features, it appears that past returns, past volatility, SMA (Simple Moving Average), OC (Open-Close), HL (High-Low), and Sign have significant predictive power.

[30]:
```
X_test_drop = X_test.drop(drop_correlated_features, axis=1)
```

3.5 STEP 5: Transformation

We will visualize the scale of our data once more before proceeding with feature transformation.

[31]:
```
# study the distribution
plt.figure(figsize=(8, 4))
sns.boxplot(x='variable', y='value', data=pd.melt(X_train_drop))
plt.xlabel(' ')
plt.title('Boxplot of Features');
```



The only remaining feature with a high value is `SMA_60`. We will scale this feature using `MinMaxScaler()`.

[32]:
```
minmax = ColumnTransformer([
    ('scaled', MinMaxScaler(), ['SMA_60'])
],remainder = 'passthrough')
```

[33]:
```
# Fit and transform the data
sma_60 = minmax.fit_transform(X_train_drop)
```

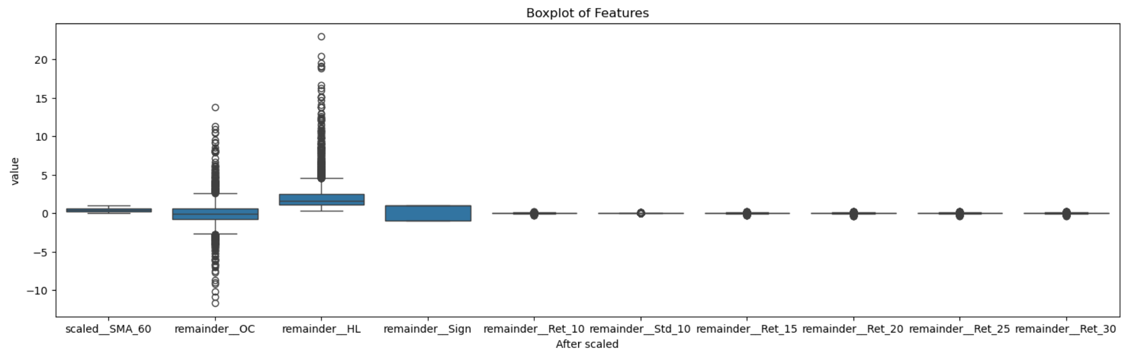
[34]:
```
X_train_dropped_scaled = pd.DataFrame(
    sma_60, columns=minmax.get_feature_names_out(),
    index=X_train_drop.index)
```

[35]:
```
X_train_dropped_scaled.describe()
```

[35]:
```
       scaled__SMA_60  remainder__OC  remainder__HL  remainder__Sign  \
count     2971.000000    2971.000000    2971.000000      2971.000000
mean         0.411422      -0.032289       2.180993         0.119488
std          0.265419       1.653484       1.984382         0.991306
min          0.000000     -11.680008       0.299995        -1.000000
25%          0.167299      -0.750000       1.129997        -1.000000
50%          0.413710      -0.110001       1.619995         1.000000
75%          0.636050       0.580017       2.490005         1.000000
max          1.000000      13.729996      22.960007         1.000000

       remainder__Ret_10  remainder__Std_10  remainder__Ret_15  \
count        2971.000000        2971.000000        2971.000000
mean            0.005261           0.009328           0.007996
std             0.033085           0.006957           0.040366
min            -0.265117           0.001264          -0.320822
25%            -0.007946           0.005041          -0.007578
50%             0.008294           0.007418           0.013048
75%             0.022695           0.011363           0.028728
max             0.195407           0.071223           0.241287

       remainder__Ret_20  remainder__Ret_25  remainder__Ret_30
count        2971.000000        2971.000000        2971.000000
mean            0.010643           0.013273           0.015933
std             0.045877           0.050274           0.053629
min            -0.370872          -0.409051          -0.392927
25%            -0.007721          -0.006560          -0.005451
50%             0.016520           0.019930           0.021935
75%             0.034884           0.040088           0.044824
max             0.212051           0.248100           0.249708
```

Let's visualize our scaled data once more.

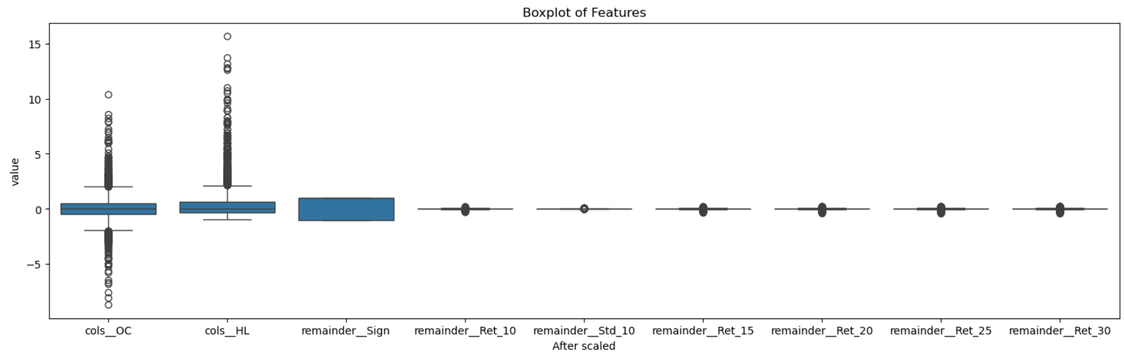
[36]:
```
fig, ax = plt.subplots(figsize=(18,5))
sns.boxplot(x='variable', y='value', data=pd.melt(X_train_dropped_scaled))
plt.xlabel('After scaled')
plt.title('Boxplot of Features');
```



It appears that the `OC` and `HL` columns contain a substantial number of outliers. We will employ the `RobustScaler` to transform these features.

[37]:
```
robust = ColumnTransformer([
        ('cols', RobustScaler(), ['OC','HL'])
    ],remainder = 'passthrough')

oc_hl = robust.fit_transform(X_train_drop)

X_train_dropped_scaled = pd.DataFrame(
    oc_hl, columns=robust.get_feature_names_out(),
    index=X_train_drop.index)
```

[38]:
```
fig, ax = plt.subplots(figsize=(18,5))
sns.boxplot(x='variable', y='value', data=pd.melt(X_train_dropped_scaled.
 ↪drop(['remainder__SMA_60'],axis=1)))
plt.xlabel('After scaled')
plt.title('Boxplot of Features');
```



Now we can construct a preprocessing transformer that applies the specified transformations to particular columns. We will fit and transform the training data and subsequently transform the test data.

[39]:
```
# Instantiate transformer
preprocessing = ColumnTransformer([
        ('MinMax', MinMaxScaler(), ['SMA_60']),
        ['Robust', RobustScaler(),['OC','HL'] ]
    ],remainder = 'passthrough')
```

[40]:
```
# Fit and transform train set
train_transformed = preprocessing.fit_transform(X_train_drop)
X_train_transformed = pd.DataFrame(
    train_transformed, columns=preprocessing.get_feature_names_out(),
    index=X_train_drop.index)

# Transform test set
test_transformed = preprocessing.transform(X_test_drop)
X_test_transformed = pd.DataFrame(
    test_transformed, columns=preprocessing.get_feature_names_out(),
    index=X_test_drop.index)
```

3.6 STEP 6: Modelling

We will compare the default settings of some classifiers using the cross-validation technique to identify potential candidates for our final model. Additionally, we will use the `class_weight` parameter to address the previously identified class imbalance problem.

```
[41]: # cross-validation
tscv = TimeSeriesSplit(n_splits=5)
```

```
[42]: # specify estimators
random_state = 42
dtc = DecisionTreeClassifier(class_weight=class_weight)
rfc = RandomForestClassifier(max_depth = 5 ,class_weight=class_weight,␣
 ↪random_state=random_state)
knn = KNeighborsClassifier()
gbc = GradientBoostingClassifier(random_state=random_state)
svc = SVC(class_weight=class_weight, random_state=random_state)
```

```
[43]: # get cv scores
clf = [dtc, rfc, knn, gbc, svc]
for estimator in clf:
    score = cross_val_score(estimator, X_train_transformed, y_train, scoring =␣
 ↪'accuracy', cv=tscv, n_jobs=-1)
    print(f"The accuracy score of {estimator} is: {score.mean():0.4}")
```

```
The accuracy score of DecisionTreeClassifier(class_weight={0: 2.435245901639344,
                                                       1: 0.6291825497670478}) is: 0.6505
The accuracy score of RandomForestClassifier(class_weight={0: 2.435245901639344,
                                                       1: 0.6291825497670478},
                       max_depth=5, random_state=42) is: 0.6949
The accuracy score of KNeighborsClassifier() is: 0.7483
The accuracy score of GradientBoostingClassifier(random_state=42) is: 0.5192
The accuracy score of SVC(class_weight={0: 2.435245901639344, 1:
0.6291825497670478}, random_state=42) is: 0.5107
```

It appears that the `RandomForestClassifier()` and `KNeighborsClassifier()` have the highest scores. Given that the `KNeighborsClassifier()` may not perform well with imbalanced classes, we will concentrate on building the model using the `RandomForestClassifier()`.

3.6.1 Base Model

The default values for the parameters that determine the size of the trees (e.g., `max_depth`, `min_samples_leaf`, etc.) result in fully grown and unpruned trees, which have the potential to overfit our model. To address this, I will set `max_depth` to 5 and then fine-tune this hyperparameter later.

```
[44]: base_model = RandomForestClassifier(max_depth = 5, class_weight=class_weight,␣
 ↪random_state=random_state)
base_model.fit(X_train_transformed,y_train)
```

```
print (classification_report(y_train[-252:], base_model.
↪predict(X_train_transformed[-252:])))
```

```
              precision    recall  f1-score   support

           0       0.48      0.88      0.62        67
           1       0.94      0.66      0.77       185

    accuracy                           0.72       252
   macro avg       0.71      0.77      0.70       252
weighted avg       0.82      0.72      0.73       252
```

3.6.2 Tuning Hyper-params

We will obtain all the parameters and define our hyperparameter grid.

[45]:
```
model = RandomForestClassifier(class_weight=class_weight,␣
↪random_state=random_state, n_jobs=-1)
```

[46]:
```
model.get_params()
```

[46]:
```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': {0: 2.435245901639344, 1: 0.6291825497670478},
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'sqrt',
 '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': -1,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

As mentioned earlier, we will include `max_depth`, `max_leaf_nodes`, and `n_estimators` in our hyperparameter grid for tuning to prevent overfitting. Additionally, since we are dealing with an imbalanced classification problem, we will experiment with different loss functions to determine their impact on model performance during the hyperparameter search.

[47]:
```
# Hyper parameter optimization
param_grid = {  'criterion': ['gini', 'entropy', 'log_loss'],
                'max_depth': [80, 90, 100, 110],
                'max_features': [2, 3],
                'min_samples_leaf': [3, 4, 5],
                'min_samples_split': [8, 10, 12],
                'n_estimators': [100, 200, 300, 1000]
            }
```

[48]:
```
# perform random search
gs = GridSearchCV(model, param_grid, scoring='f1', cv=tscv, verbose=0, n_jobs=-1)
gs.fit(X_train_transformed, y_train)
```

[48]:
```
GridSearchCV(cv=TimeSeriesSplit(gap=0, max_train_size=None, n_splits=5,
test_size=None),
             estimator=RandomForestClassifier(class_weight={0:
2.435245901639344,
                                                            1:
0.6291825497670478},
                                              n_jobs=-1, random_state=42),
             n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy', 'log_loss'],
                         'max_depth': [80, 90, 100, 110],
                         'max_features': [2, 3], 'min_samples_leaf': [3, 4, 5],
                         'min_samples_split': [8, 10, 12],
                         'n_estimators': [100, 200, 300, 1000]},
             scoring='f1')
```

[49]:
```
# best parameters
gs.best_params_
```

[49]:
```
{'criterion': 'entropy',
 'max_depth': 80,
 'max_features': 2,
 'min_samples_leaf': 3,
 'min_samples_split': 8,
 'n_estimators': 1000}
```

[50]:
```
# best score
gs.best_score_
```

[50]:
```
0.8830577734916052
```

3.7 STEP 7: Metrics

After fine-tuning our model and conducting a search for the best hyperparameters, we will evaluate our model's performance and compare it to our base model.

[51]:
```
# Refit the XGB Classifier with the best params
final_model = RandomForestClassifier(class_weight=class_weight,␣
↪random_state=random_state, n_jobs=-1, **gs.best_params_)
final_model.fit(X_train_transformed, y_train)
```

[51]:
```
RandomForestClassifier(class_weight={0: 2.435245901639344,
                                     1: 0.6291825497670478},
                       criterion='entropy', max_depth=80, max_features=2,
                       min_samples_leaf=3, min_samples_split=8,
                       n_estimators=1000, n_jobs=-1, random_state=42)
```

[52]:
```
# Predicting the test dataset
y_pred = final_model.predict(X_test_transformed)
# Measure Accuracy
acc_train = accuracy_score(y_train, final_model.predict(X_train_transformed))
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}')
```

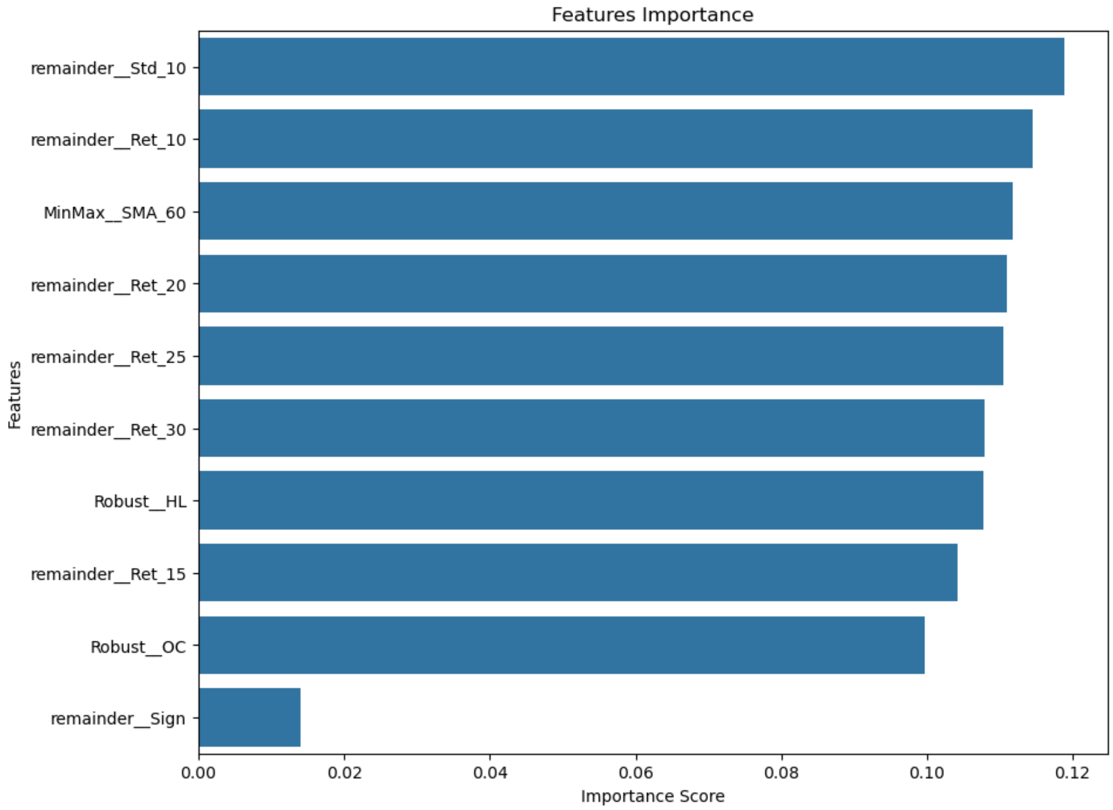
```
 Training Accuracy      : 0.9973
 Test Accuracy          : 0.6824
```

Our final model outperforms the base model, but it appears to suffer from severe overfitting.

[53]:
```
# Cross validation score
score = cross_val_score(final_model,X_train_transformed,y_train,cv=tscv)
print(f'Mean CV Score : {score.mean():0.4}')
```

```
Mean CV Score : 0.7947
```

[54]:
```
# Plot feature importance
fig, ax = plt.subplots(figsize=(10,8))
feature_imp = pd.DataFrame({'Importance Score': final_model.
↪feature_importances_,'Features': X_train_transformed.columns}).
↪sort_values(by='Importance Score', ascending=False)
sns.barplot(x=feature_imp['Importance Score'], y=feature_imp['Features'])
ax.set_title('Features Importance');
```



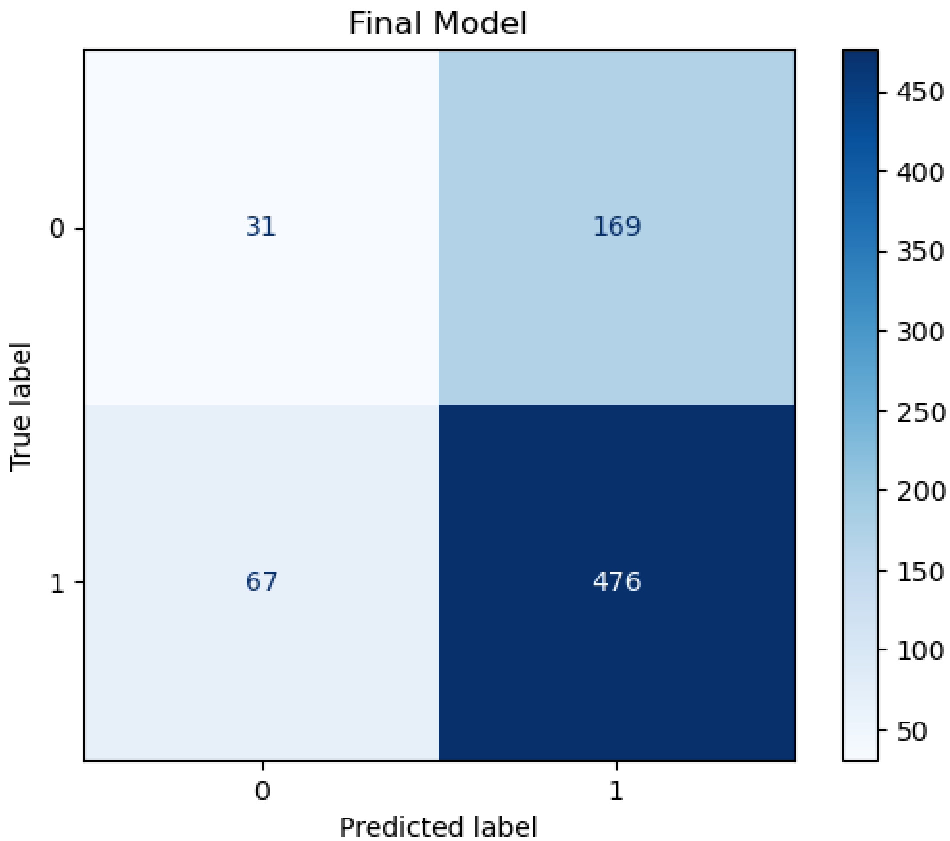
It appears that the most important features are the asset returns, volatilities, and H-L (High-Low) values. There is some predictive power in the `SMA_60` feature, while the `Sign` feature contributes almost no predictive power.

[55]:
```
# Classification Report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.32      0.15      0.21       200
           1       0.74      0.88      0.80       543

    accuracy                           0.68       743
   macro avg       0.53      0.52      0.50       743
weighted avg       0.62      0.68      0.64       743
```

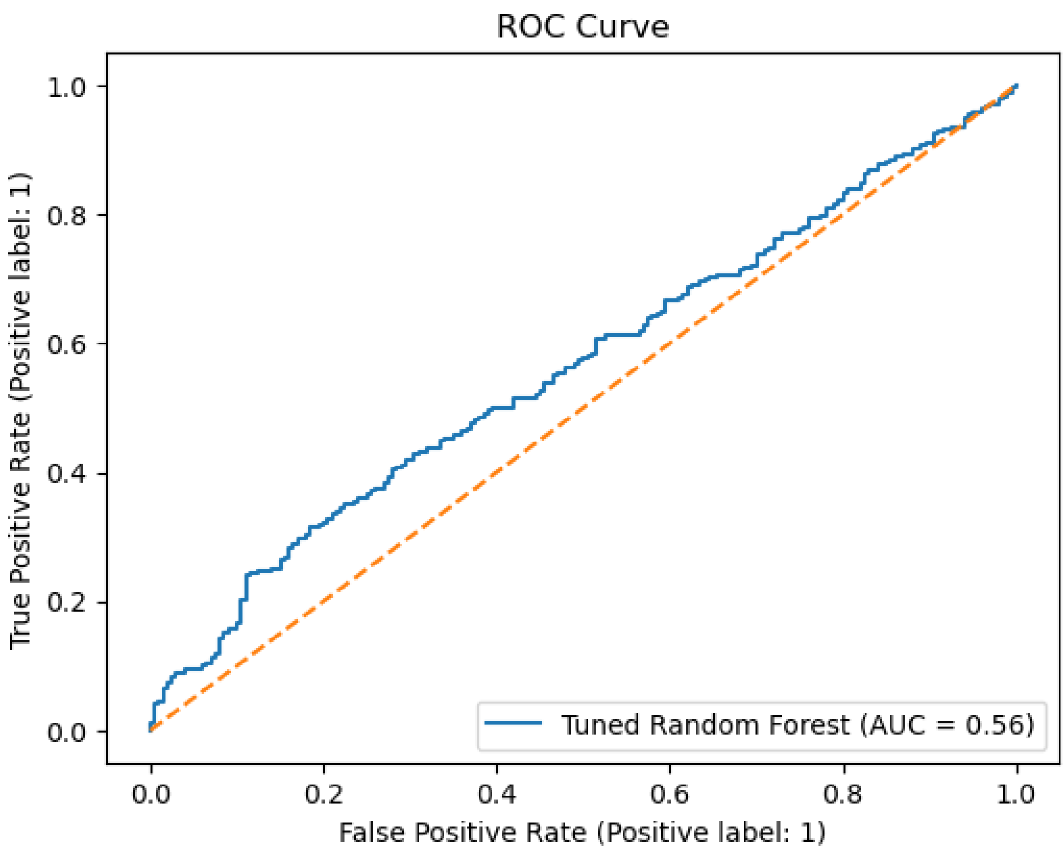
[56]:
```
# Display confussion matrix
disp = ConfusionMatrixDisplay.from_estimator(
        final_model,
        X_test_transformed,
        y_test,
```

```
        display_labels=final_model.classes_,
        cmap=plt.cm.Blues
    )
disp.ax_.set_title('Final Model')
plt.show()
```



Our model is performing well when predicting the majority class but struggles when predicting the minority class.

[57]:
```
# Display ROCCurve
disp_roc = RocCurveDisplay.from_estimator(
        final_model,
        X_test_transformed,
        y_test,
        name='Tuned Random Forest')
disp_roc.ax_.set_title('ROC Curve')
plt.plot([0,1], [0,1], linestyle='--')
plt.show()
```



When analyzing the ROC curve, it becomes evident that our model's performance is only marginally better than random chance. This suggests that the model may not effectively discriminate between positive and negative outcomes. To enhance its predictive power and better address the minority class, we may need to further refine our model or consider additional strategies such as resampling techniques or employing different algorithms.

[58]:
```
# Saving final model
from joblib import dump, load
dump(clf, 'final_model.joblib')
```

[58]: ['final_model.joblib']

3.8 Conclusion

We have successfully created a machine learning model capable of predicting upward and downward movements of the underlying asset using raw data obtained from Yahoo Finance APIs. Our process involved feature engineering to create a relevant feature set, feature selection to determine the most important features, and data transformation for our chosen set of features. The Random Forest Classifier is used as a candidate, fine-tuned, and optimize as our final model.

However, there are certain limitations to our model. The imbalanced nature of our labels, particularly in the context of financial time series, poses a challenge, which we've partly addressed using the `class_weight` function. The daily price data is limited, providing relatively weak data structure for our algorithm; exploring higher frequency data may lead to a more robust training set. Additionally, our model exhibits overfitting, suggesting the potential need for a more complex model or better feature enginerring techniques.

4 References

[1] Scikit-learn 1.3.2 - Voting Classifier

[2] Mathematics for Machine Learning - Marc Peter Deisenroth - p.380

[3] Introduction to ML using Scikit-learn - Pythonlab 09