Technical Report for Movement Prediction of the Stock Market

1. Introduction

In this project we are going to predict the movement directions of a specific Stock (Google) of the Stock Market. Our dataset is imported from yahoo servers for the time period from 2010 until now. With the use of the classification ensemble method of Random Forest, our purpose is to extract knowledge from the historical data and make predictions on the stock price movements. In order to do that, we approach the problem on two sides. Firstly, we are trying to make predictions using fitting the raw data on a randomized search of various Random Forest classifiers. Afterwards, we are trying to improve the predictions, by manufacturing new features and following the same strategy of randomized search. Also, we provide several visualizations for interpreting the models' performances.

2. Technical Part

2.1. Import Packages

We begin by importing the necessary packages, that we used to perform the task we were assigned to. Some of the packages that we used are Numpy, Pandas for creating Arrays and Dataframes. Also, we imported data from pandas_datareader library for loading our dataset, matplotlib library for visualizations and we use sklearn library for feature creation, modelization, prediction making and evaluating our results.

```
[1]  import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     from datetime import datetime
     from pandas_datareader import data as wb
     from sklearn.ensemble import RandomForestClassifier
     from sklearn.model_selection import train_test_split
     from sklearn.model_selection import RandomizedSearchCV
     from sklearn.metrics import plot_roc_curve
     from sklearn.metrics import accuracy_score

     plt.style.use('ggplot')
```

Figure 1.

2.2. Import Dataset

We import our data which contains Google's Stock prices from yahoo server. The datetime we use is from "1/1/2020" until now "24/5/2021". Also, we are visualizing Google's stock performance to have an overview on our dataset information. As we can see our dataset apart from 6 columns and 2867 rows (one row per working day). The columns are, High (higher price of the day), Low (lower price of the day), Open (open price of the day), Close (close price of day), Volume (Volume measures the number of shares traded in a stock) and Adj Close (adjusted closing price amends a stock's closing price to reflect that stock's value after accounting for any corporate actions).

```
[2] start_date = datetime(2010,1,1)
    end_date = datetime.now()
```

```
[3] df = wb.DataReader('GOOGL', 'yahoo', start_date, end_date)
    df.head(5)
```

Date	High	Low	Open	Close	Volume	Adj Close
2010-01-04	315.070068	312.432434	313.788788	313.688690	3908488.0	313.688690
2010-01-05	314.234222	311.081085	313.903900	312.307312	6003391.0	312.307312
2010-01-06	313.243256	303.483490	313.243256	304.434448	7949443.0	304.434448
2010-01-07	305.305298	296.621613	305.005005	297.347351	12815771.0	297.347351
2010-01-08	301.926941	294.849854	296.296295	301.311310	9439151.0	301.311310

```
[4] df.shape

    (2867, 6)
```

```
[5] plt.plot(df['Adj Close'])

    [<matplotlib.lines.Line2D at 0x7f417bede510>]
```

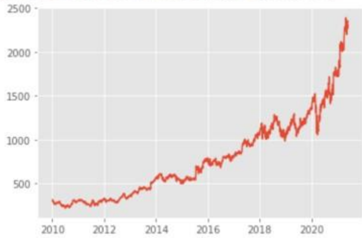

Figure 2.

2.3 Stock Price Predictions with raw data

2.3.1 Creating the target values for Classification and splitting our dataset.

As classes for the classification, we are differencing each day with the previous one to get the change of price and we save them to a new column called “Movements”, which we wish to predict. For negative and zero values we assign -1 as the first class, and for positive values we assign 1.

As features of our dataset, we are using the raw data imported from yahoo (High, Low, Open, Close, Volume) and split our dataset into a train and a test set with ratio 80:20.

```
[6] df['Change_of_price'] = df['Adj Close'].diff() # Create a dataframe with the change in price.

    # Create a column we wish to predict
    df['Movements'] = np.where(df['Change_of_price'] > 0, 1, -1)# Assign 1 when change in price is more than 0 and -1 when it is less that or equal to 0.
```

```
[7] # Grab our features & classes and split into train and test sets of the ratio 80:20
    features = df[['High','Low','Open','Close','Volume']]
    classes = df['Movements']

    X_train, X_test, y_train, y_test = train_test_split(features, classes, random_state = None)
```

Figure 3.

2.3.2 Optimizing Random Forests algorithm

Our next goal is to create the “random grid" which will search random combinations of the hyperparameters to find the best solution for tuning our Random Forests model.

```python
[8] n_estimators = list(range(200, 2000, 200))  #Setting number of trees.

    max_features = ['auto', 'sqrt', None, 'log2']  # Number of features to consider at every split

    max_depth = list(range(10, 110, 10)) # Maximum number of levels in tree
    max_depth.append(None)

    min_samples_split = [2, 5, 10, 20, 30, 40] # Minimum number of samples required to split a node

    min_samples_leaf = [1, 2, 7, 12, 14, 16 ,20] # Minimum number of samples required at each leaf node

    bootstrap = [True, False] # Method of selecting samples for training each tree

    # Create the random grid
    random_grid = {'n_estimators': n_estimators,
                   'max_features': max_features,
                   'max_depth': max_depth,
                   'min_samples_split': min_samples_split,
                   'min_samples_leaf': min_samples_leaf,
                   'bootstrap': bootstrap}
```

Figure 4.

2.3.3 Building Random Forests model

Now we will create our Random Forests ensemble model and tune its parameters properly with the use of "random grid". Also, we will set 100 iterations, cv = 3 for our randomized search. Next, we will fit our model and finally use the best estimator. As a criterion for splitting the Decision Trees, we use the GINI impurity.

```python
# Random Forest Classifier
rf = RandomForestClassifier()

# Specify the details of our Randomized Search
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid, n_iter = 100, cv = 3, verbose=2, random_state=0, n_jobs = -1)

# Fit the random search model
rf_random.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 100 candidates, totalling 300 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed:  2.8min
[Parallel(n_jobs=-1)]: Done 158 tasks      | elapsed: 16.0min
[Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed: 27.0min finished
RandomizedSearchCV(cv=3, error_score=nan,
                   estimator=RandomForestClassifier(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_impurity_split=None,
                                                    min_samples_leaf=1,
                                                    min_samples_split=2,
                                                    min_weight_fraction_leaf=0.0,
                                                    n_estimators=100,
                                                    n_jobs...
                   param_distributions={'bootstrap': [True, False],
                                        'max_depth': [10, 20, 30, 40, 50, 60,
                                                      70, 80, 90, 100, None],
                                        'max_features': ['auto', 'sqrt', None,
                                                         'log2'],
                                        'min_samples_leaf': [1, 2, 7, 12, 14,
                                                             16, 20],
                                        'min_samples_split': [2, 5, 10, 20, 30,
                                                              40],
                                        'n_estimators': [200, 400, 600, 800,
                                                         1000, 1200, 1400, 1600,
                                                         1800]},
                   pre_dispatch='2*n_jobs', random_state=0, refit=True,
                   return_train_score=False, scoring=None, verbose=2)
```

```python
best_estimator = rf_random.best_estimator_
best_estimator
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=100, max_features=None,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=1000,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Figure 5.

2.3.4 First Prediction Result

Finally, we are making predictions using the test data to evaluate our model. As we can see, we achieve a 67.5% accuracy with the raw data.

```
# Make predictions
y_pred = best_estimator.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred, normalize = True) * 100.0)
```

```
Accuracy: 67.50348675034867
```

Figure 6.

2.3.5 Visualizations

2.3.5.1 Confusion Matrix

In the following Confusion Matrix, we can see that the model correctly predicted 62% of the downward movement directions and the 73% of the upward movement directions of Google's stock prices.

```
from sklearn.metrics import confusion_matrix, plot_confusion_matrix

rf_matrix = confusion_matrix(y_test, y_pred)

board = plot_confusion_matrix(rf_random, X_test, y_test,
                              display_labels = ['Down Movements', 'Up Movements'],
                              normalize = 'true',
                              cmap=plt.cm.Greens)
board.ax_.set_title('Confusion Matrix')
plt.show()
```

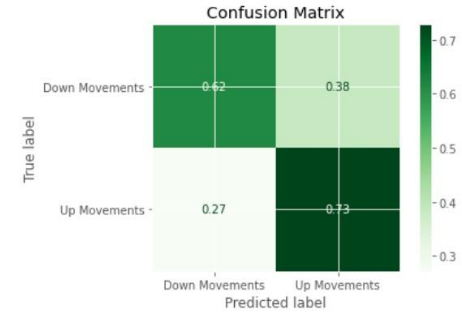

Figure 7.

2.3.5.2 Feature Importance

Here, we see that the features that contribute the most at the Random Forest's performance are Open price, Close price and Volume with 28.1%, 27.8% and 23.6% each. We provide a line graph of the cumulative importance of the features for further understanding.

```
# Calculate feature importance and store in pandas series
feature_imp = pd.Series(best_estimator.feature_importances_, index= features.columns).sort_values(ascending=False)
feature_imp
```

```
Open      0.281437
Close     0.278449
Volume    0.236998
High      0.102284
Low       0.100832
dtype: float64
```

```
# store the values in a list to plot.
x_values = list(range(len(best_estimator.feature_importances_)))

# Cumulative importances
cumulative_importances = np.cumsum(feature_imp.values)

# Make a line graph
plt.plot(x_values, cumulative_importances, 'g-')

# Draw line at 95% of importance retained
plt.hlines(y = 0.95, xmin = 0, xmax = len(feature_imp), color = 'r', linestyles = 'dashed')

# Format x ticks and labels
plt.xticks(x_values, feature_imp.index, rotation = 'vertical')

# Axis labels and title
plt.xlabel('Variable')
plt.ylabel('Cumulative Importance')
plt.title('Random Forest: Feature Importance Graph')
```

```
Text(0.5, 1.0, 'Random Forest: Feature Importance Graph')
```

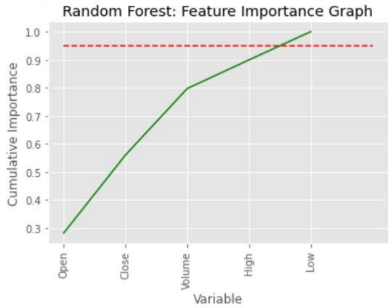

Figure 8.

2.3.5.3 ROC Curve

Last but not least, we plot the Receiver Operating Characteristic (ROC) curve which illustrates the diagnostic ability of our classifier system as its discrimination threshold is varied. As closer to the top-left corner indicates a better performance of our model. Since the AUC scores at 0.73, it means that there is a 73% chance that the model will be able to distinguish between upward and downward movement directions.

```
# Create an ROC Curve plot.
rfc_disp = plot_roc_curve(rf_random, X_test, y_test, alpha = 0.8)
plt.show()
```

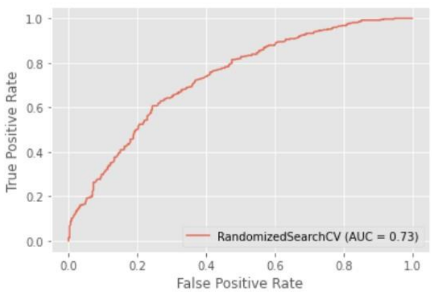

Figure 9.

2.4. Stock Price Predictions with new features

In this section, we pre-process our data in order to achieve better results.

2.4.1. Feature Extraction

After a further study on Stock Market prediction, we are developing new features using the raw data that will accelerate performance. Our ideas for these new features are based on [Predicting the Direction of Stock Market Price Using Tree Based Classifiers](). In our case, we develop seven new features:

- Relative Strength Index (RSI)
- Stochastic Oscillator (SO)
- Williams %R (WilR)
- Moving Average Convergence Divergence (MACD)
- Exponential Moving Average (MACD EMA)
- Price Rate Of Change (PRC)
- On Balance Volume (OBV)

2.4.2 Feature Interpretation

Relative Strength Index (RSI):

RSI is an indicator for determining if a stock is overbought or oversold, over a period of time, and it takes values ranging from 0 to 100. If RSI is below 30 the stock might be oversold and if RSI is above 70 the stock might be overbought. In our case, we use a 14-day period.

Stochastic Oscillator (SO):

SO indicates the level of current price in relation to its price range, over a period of time. It ranges from 0 to 100. In our case, we use a 14-day period.

Williams Percentage Range (WilR):

WilR is an indicator equivalent to SO, mirrored at the 0%-line, when using the same time interval. It ranges from -100 to 0. If WilR is below -80 the stock might be oversold and if WilR is above -20 the stock might be overbought. In our case, we use a 14-day period.

Moving Average Convergence Divergence (MACD):

MACD is a trend-following momentum indicator that shows the relationship between two moving averages. It is calculated by subtracting a 26-day exponential moving average (EMA) from a 12-day EMA.

Exponential Moving Average (MACD EMA):

The EMA of MACD is a single line that serves as a threshold for buy and sell signals. If MACD moves below the line, it gives a sell signal and if MACD moves above the line, it gives a buy signal.In our case, we a 9 day period.

Price Rate of Change (PRC):

PRC measures the percentage change in price between the current price and the price a certain number of periods ago. In our case, we used the price of 14 days ago.

On Balance Volume (OBV):

OBV is used to quantify buying and selling trends of the stock. It is a cumulative value, that initializes from 0 and adds the volume of the day when the stock has an upwards movement direction and subtract the volume of day when the stock has a downwards movement direction.

```
[12] df['Change_of_price'] = df['Adj Close'].diff() # Create a dataframe with the change in price.
     n = 14
     # Calculate the 14 day RSI
     def RSI(n, df):

       up_df, down_df = df['Change_of_price'].copy(), df['Change_of_price'].copy()   # First make a copy of the data frame twice.
       up_df, down_df = pd.DataFrame(up_df), pd.DataFrame(down_df)

       up_df.loc['Change_of_price'] = up_df.loc[(up_df['Change_of_price'] < 0), 'Change_of_price'] = 0   # For up days, if the change is less than 0 set to 0.

       down_df.loc['Change_of_price'] = down_df.loc[(down_df['Change_of_price'] > 0), 'Change_of_price'] = 0  # For down days, if the change is greater than 0 set to 0.

       down_df['Change_in_price'] = down_df['Change_in_price'].abs()   # We need change in price to be absolute.

       ewma_up = up_df['Change_of_price'].transform(lambda x: x.ewm(span = n).mean())   # Calculate the EWMA (Exponential Weighted Moving Average), meaning older values are given less weight compared to newer values.
       ewma_down = down_df['Change_of_price'].transform(lambda x: x.ewm(span = n).mean())

       relative_strength = ewma_up / ewma_down   # Calculate the Relative Strength

       relative_strength_index = 100.0 - (100.0 / (1.0 + relative_strength))   # Calculate the Relative Strength Index

       return down_df, up_df, relative_strength_index

     # Calculate the 14 day Stochastic Oscillator
     def SO(n, df):
       low_n, high_n = df['Low'].copy(), df['High'].copy()   # Make a copy of the high and low column.
       low_n, high_n = pd.DataFrame(low_n), pd.DataFrame(high_n)

       low_n = low_n['Low'].transform(lambda x: x.rolling(window = n).min())  # Apply the rolling function and grab the Min and Max.
       high_n = high_n['High'].transform(lambda x: x.rolling(window = n).max())

       so = 100 * ((df['Adj Close'] - low_n) / (high_n - low_n))  # Calculate the Stochastic Oscillator.

       return low_n, high_n, so

     # Calculate the 14 day Williams %R
     def WilR(n, df):

       low_n, high_n = df['Low'].copy(), df['High'].copy()   # Make a copy of the high and low column.
       low_n, high_n = pd.DataFrame(low_n), pd.DataFrame(high_n)

       low_n = low_n['Low'].transform(lambda x: x.rolling(window = n).min())   # Apply the rolling function and grab the Min.
       high_n = high_n['High'].transform(lambda x: x.rolling(window = n).max())   # Apply the rolling function and grab the Max.

       wr = (-100) * ((high_n - df['Adj Close']) / (high_n - low_n))   # Calculate William %R indicator.

       return wr

     # Calculate the MACD
     ema_26 = df['Adj Close'].transform(lambda x: x.ewm(span = 26).mean())
     ema_12 = df['Adj Close'].transform(lambda x: x.ewm(span = 12).mean())
     macd = ema_12 - ema_26

     # Calculate the EMA
     ema = macd.ewm(span = 9).mean()

     # Calculate the 14 day Price Rate of Change
     def PRC(n, df):

       prc = df['Adj Close'].transform(lambda x: x.pct_change(periods = n))   # Calculate the Rate of Change in the Price, and store it in the Data Frame.

       return prc
```

Figure 10.

```python
# Calculate the On Balance Volume
def OBV(df):

  volume = df['Volume']  # Grab the volume and Change_in_price column.
  change = df['Change_of_price']

  # intialize the previous OBV
  prev_obv = 0
  obv_list = []
  # calculate the On Balance Volume
  for i, j in zip(change, volume):
      if i > 0:
          current_obv = prev_obv + j
      elif i < 0:
          current_obv = prev_obv - j
      else:
          current_obv = prev_obv
      prev_obv = current_obv
      obv_list.append(current_obv)

  return pd.Series(obv_list, index = df.index)  # Return a panda series.

# Add features to the main dataframe
df['Down_days'], df['Up_days'], df['RSI'] = RSI(n, df)
df['low_n'], df['high_n'], df['SO'] = SO(n, df)
df['WilR'] = WilR(n, df)
df['MACD'] = macd
df['MACD_EMA'] = ema
df['PRC'] = PRC(n, df)
df['OBV'] = OBV(df)

df.head(100)
```

Date	High	Low	Open	Close	Volume	Adj Close	Change_of_price	Movements	Change_in_price	Down_days	Up_days	RSI	low_n	high_n	SO	WilR	MACD	MACD_EMA	PRC	OBV
2010-01-04	315.070068	312.432434	313.788788	313.688690	3908488.0	313.688690	NaN	-1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	0.000000	0.000000	NaN	0.0
2010-01-05	314.234222	311.081085	313.903900	312.307312	6003391.0	312.307312	-1.381378	-1	-1.381378	1.381378	0.000000	0.000000	NaN	NaN	NaN	NaN	-0.030992	-0.017218	NaN	-6003391.0
2010-01-06	313.243256	303.483490	313.243256	304.434448	7949443.0	304.434448	-7.872864	-1	-7.872864	7.872864	0.000000	0.000000	NaN	NaN	NaN	NaN	-0.283628	-0.126402	NaN	-13952834.0
2010-01-07	305.305298	296.621613	305.005005	297.347351	12815771.0	297.347351	-7.087097	-1	-7.087097	7.087097	0.000000	0.000000	NaN	NaN	NaN	NaN	-0.647457	-0.302911	NaN	-26768605.0
2010-01-08	301.926941	294.849854	296.296295	301.311310	9439151.0	301.311310	3.963959	1	3.963959	0.000000	3.963959	23.429409	NaN	NaN	NaN	NaN	-0.673791	-0.413240	NaN	-17329454.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2010-05-20	243.033035	237.137131	242.777771	237.742737	9816773.0	237.742737	-9.719727	-1	-9.719727	9.719727	0.000000	22.634659	230.230225	266.726715	20.584204	-79.415796	-7.735602	-6.832130	-0.096424	-103886015.0
2010-05-21	242.742737	232.432434	234.764771	236.261261	19362218.0	236.261261	-1.481476	-1	-1.481476	1.481476	0.000000	21.518822	230.230225	263.633636	18.055151	-81.944849	-8.422856	-7.150276	-0.110347	-123248233.0
2010-05-24	245.140137	238.638641	240.605606	238.818817	8682509.0	238.818817	2.557556	1	2.557556	0.000000	2.557556	28.536446	230.230225	261.671661	27.316158	-72.683842	-8.661383	-7.452497	-0.057685	-114565724.0
2010-05-25	238.963959	232.237244	234.309311	238.773773	6028765.0	238.773773	-0.045044	-1	-0.045044	0.045044	0.000000	28.484686	230.230225	261.671661	27.172895	-72.827105	-8.753142	-7.712626	-0.064128	-120594489.0
2010-05-26	245.125122	237.737732	241.276276	237.972977	6944249.0	237.972977	-0.800797	-1	-0.800797	0.800797	0.000000	27.462865	232.237244	261.671661	19.486484	-80.513516	-8.789128	-7.927927	-0.046524	-127538738.0

Figure 11.

2.4.3. Creating the Prediction Column and Remove NaN Values

After creating our features, we create the classes as we did previously with the raw data predictions and we are removing the NaN values. As we can see, before removing the NaN values, we had 2867 rows. After the NaN values removal, we concluded with 2853 rows.

```
[13] # Create a column we wish to predict

     # Assign 1 when change in price is more than 0 and -1 when it is less that or equal to 0.
     df['Movements'] = np.where(df['Change_in_price'] > 0, 1, -1)
```

Figure 12.

Figure 13.

```
[14] # We need to remove all rows that have an NaN value.
     print(f'Before removing NaN values, we have {df.shape[0]} rows and {df.shape[1]} columns')

     # Any row that has a `NaN` value will be dropped.
     df = df.dropna()

     # Display how much we have left now.
     print(f'After removing NaN values, we have {df.shape[0]} rows and {df.shape[1]} columns')

     df.head()

Before removing NaN values, we have 2867 rows and 20 columns
After removing NaN values, we have 2853 rows and 20 columns
```

Date	High	Low	Open	Close	Volume	Adj Close	Change_of_price	Movements	Change_in_price	Down_days	Up_days	RSI	low_n	high_n	SO	WilR	MACD	MACD_EMA	PRC	OBV
2010-01-25	275.215210	268.023010	273.568573	270.270264	8830960.0	270.270264	-5.010010	-1	-5.010010	5.010010	0.000000	13.574657	267.697693	314.234222	5.528068	-94.471932	-3.641916	-2.033031	-0.138412	-64489845.0
2010-01-26	275.075073	268.413422	269.254242	271.481476	8702289.0	271.481476	1.211212	1	1.211212	0.000000	1.211212	16.961909	267.697693	313.243256	8.307687	-91.692313	-4.268942	-2.493165	-0.130723	-55787556.0
2010-01-27	274.099091	267.922913	270.905914	271.321320	7920871.0	271.321320	-0.160156	-1	-0.160156	0.160156	0.000000	16.861085	267.697693	305.305298	9.635356	-90.364644	-4.704072	-2.945533	-0.108769	-63708427.0
2010-01-28	273.773773	265.565552	272.517517	267.412415	6451742.0	267.412415	-3.908905	-1	-3.908905	3.908905	0.000000	14.443316	265.565552	302.532532	4.995980	-95.004020	-5.226018	-3.409997	-0.100673	-70160169.0
2010-01-29	270.765778	263.068054	269.514526	265.235229	8272719.0	265.235229	-2.177185	-1	-2.177185	2.177185	0.000000	13.224606	263.068054	302.532532	5.491458	-94.508542	-5.703238	-3.875352	-0.119730	-78432888.0

2.4.4. Split our Data

In order to train the model, we pick the features we created from preprocessing. These are:

1. Relative Strength Index (RSI)
2. Stochastic Oscillator (SO)
3. Williams %R (WilR)
4. Moving Average Convergence Divergence (MACD)
5. Exponential Moving Average (MACD EMA)
6. Price Rate Of Change (PRC)
7. On Balance Volume (OBV)

As classes, we are going to use the Classes column and Split our dataset into a train and a test set with ratio 80:20 (Default).

```
[15] # Grab our features & classes and split into train and test sets of the ratio 80:20
     features = df[['RSI','SO','WilR','MACD','MACD_EMA','PRC','OBV']]
     classes = df['Movements']

     X_train, X_test, y_train, y_test = train_test_split(features, classes, random_state = None)
```

Figure 14.

2.4.5. Optimizing Random Forests algorithm

As in the first prediction process, we are creating a “random grid" which will search random combinations of the hyperparameters to find the best solution for tuning our Random Forests model.

```
[16] n_estimators = list(range(200, 2000, 200))  #Setting number of trees.

     max_features = ['auto', 'sqrt', None, 'log2']  # Number of features to consider at every split

     max_depth = list(range(10, 110, 10)) # Maximum number of levels in tree
     max_depth.append(None)

     min_samples_split = [2, 5, 10, 20, 30, 40] # Minimum number of samples required to split a node

     min_samples_leaf = [1, 2, 7, 12, 14, 16 ,20] # Minimum number of samples required at each leaf node

     bootstrap = [True, False] # Method of selecting samples for training each tree

     # Create the random grid
     random_grid = {'n_estimators': n_estimators,
                    'max_features': max_features,
                    'max_depth': max_depth,
                    'min_samples_split': min_samples_split,
                    'min_samples_leaf': min_samples_leaf,
                    'bootstrap': bootstrap}
```

Figure 15.

2.4.6 Building Random Forests model

Also, we create our Random Forests ensemble model and tune its parameters properly with the use of the "random grid". The hyperparameters for the randomized search are also the same as before. After fitting the training data, we filter the best Random Forest estimator of the randomized search. As a criterion for splitting the Decision Trees, we use the GINI impurity.

```
[17] # New Random Forest Classifier to house optimal parameters
     rf = RandomForestClassifier()

     # Specfiy the details of our Randomized Search
     rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid, n_iter = 100, cv = 3, verbose=2, random_state=0, n_jobs = -1)

     # Fit the random search model
     rf_random.fit(X_train, y_train)

     Fitting 3 folds for each of 100 candidates, totalling 300 fits
     [Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
     [Parallel(n_jobs=-1)]: Done  37 tasks      | elapsed:  1.8min
     [Parallel(n_jobs=-1)]: Done 158 tasks      | elapsed:  9.1min
     [Parallel(n_jobs=-1)]: Done 300 out of 300 | elapsed: 16.0min finished
     RandomizedSearchCV(cv=3, error_score=nan,
                        estimator=RandomForestClassifier(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_impurity_split=None,
                                                         min_samples_leaf=1,
                                                         min_samples_split=2,
                                                         min_weight_fraction_leaf=0.0,
                                                         n_estimators=100,
                                                         n_jobs...
                        param_distributions={'bootstrap': [True, False],
                                             'max_depth': [10, 20, 30, 40, 50, 60,
                                                           70, 80, 90, 100, None],
                                             'max_features': ['auto', 'sqrt', None,
                                                              'log2'],
                                             'min_samples_leaf': [1, 2, 7, 12, 14,
                                                                  16, 20],
                                             'min_samples_split': [2, 5, 10, 20, 30,
                                                                   40],
                                             'n_estimators': [200, 400, 600, 800,
                                                              1000, 1200, 1400, 1600,
                                                              1800]},
                        pre_dispatch='2*n_jobs', random_state=0, refit=True,
                        return_train_score=False, scoring=None, verbose=2)

[18] best_estimator = rf_random.best_estimator_
     best_estimator

     RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                            criterion='gini', max_depth=30, max_features=None,
                            max_leaf_nodes=None, max_samples=None,
                            min_impurity_decrease=0.0, min_impurity_split=None,
                            min_samples_leaf=7, min_samples_split=20,
                            min_weight_fraction_leaf=0.0, n_estimators=800,
                            n_jobs=None, oob_score=False, random_state=None,
                            verbose=0, warm_start=False)
```

Figure 16.

2.4.6 Second Prediction Results

Finally, we are making predictions using the test data to evaluate our model. As we can see, we achieve 72.5% accuracy taking into account the new features we create.

```
[20] # Make predictions
     y_pred = best_estimator.predict(X_test)
     print("Accuracy:", accuracy_score(y_test, y_pred, normalize = True) * 100.0)

     Accuracy: 73.24929971988794
```

Figure 17.

2.4.7 Visualizations

In this section, we present some visualizations in order to further understand the model.

2.4.7.1 Confusion Matrix

In the following Confusion Matrix, we can see that the model correctly predicted 74% of the downward movement directions and the 72% of the upward movement directions of Google's stock prices.

```
[21] from sklearn.metrics import confusion_matrix, plot_confusion_matrix

     rf_matrix = confusion_matrix(y_test, y_pred)

     board = plot_confusion_matrix(rf_random, X_test, y_test,
                                   display_labels = ['Down Movements', 'Up Movements'],
                                   normalize = 'true',
                                   cmap=plt.cm.Greens)
     board.ax_.set_title('Confusion Matrix')
     plt.show()
```

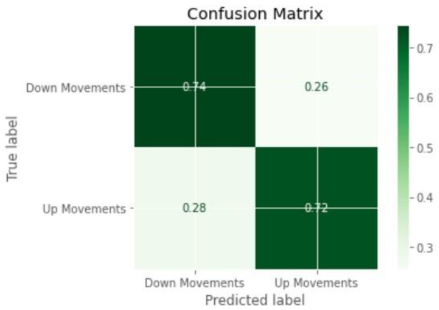

Figure 18.

2.4.7.2 Feature Importance

Here, we see that the features that contribute the most at the Random Forest's performance are William's Percentage Range and Stochastic Oscillator with 21.6% and 21.5% each. We provide a line graph of the cumulative importance of the features for further understanding.

```
[22] # Calculate feature importance and store in pandas series
     feature_imp = pd.Series(best_estimator.feature_importances_, index= features.columns).sort_values(ascending=False)
     feature_imp
```

```
WilR        0.216291
SO          0.215307
RSI         0.146549
MACD        0.136956
PRC         0.114453
OBV         0.089373
MACD_EMA    0.081073
dtype: float64
```

```
[23] # store the values in a list to plot.
     x_values = list(range(len(best_estimator.feature_importances_)))

     # Cumulative importances
     cumulative_importances = np.cumsum(feature_imp.values)

     # Make a line graph
     plt.plot(x_values, cumulative_importances, 'g-')

     # Draw line at 95% of importance retained
     plt.hlines(y = 0.95, xmin = 0, xmax = len(feature_imp), color = 'r', linestyles = 'dashed')

     # Format x ticks and labels
     plt.xticks(x_values, feature_imp.index, rotation = 'vertical')

     # Axis labels and title
     plt.xlabel('Variable')
     plt.ylabel('Cumulative Importance')
     plt.title('Random Forest: Feature Importance Graph')
```

```
Text(0.5, 1.0, 'Random Forest: Feature Importance Graph')
```

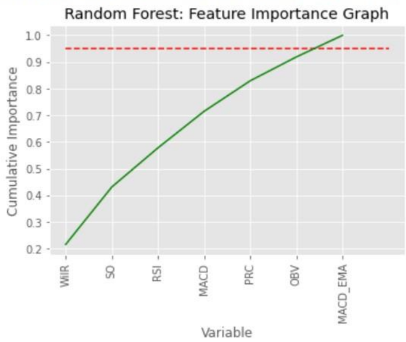

Figure 19.

2.4.7.3 ROC Curve

Finally, we will observe that the AUC scores at 0.8, which means that there is an 80% chance that the model will be able to distinguish between upward and downward movement directions.

```
[24] # Create an ROC Curve plot.
     rfc_disp = plot_roc_curve(rf_random, X_test, y_test, alpha = 0.8)
     plt.show()
```

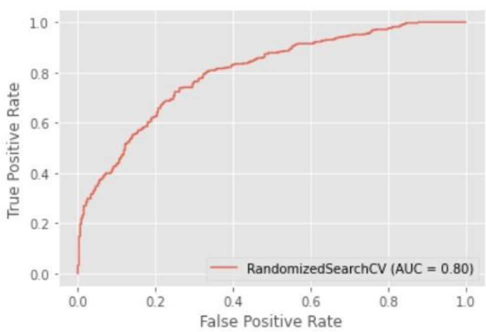

Figure 20.

3. Conclusions

It is clear from the results that the best method for predicting movement directions on stocks is through feature extraction from the raw data. Not only we achieve an astonishing 5.7% increase in accuracy, but also the ROC AUC metric improved significantly too. This leads us to assume that feature extraction was the best choice for optimizing the classification.

Also, we observe that the features that contribute the most on the first case are Open, Close and Volume, and on the second case the features that contribute the most are William's Percentage Range and Stochastic Oscillator, which are features that derive from High and Low features.