Machine Learning for Financial Markets

1 – Trading Apple Stocks with a Binary Classifier

Machine learning techniques have already revolutionized many industries and activities across various fields, and the financial markets are no exception. Traders and investors can now make use of different machine learning algorithms and techniques to develop trading strategies, enabling them to achieve higher returns while minimizing risks and drawdowns.

In this study, I am going to demonstrate the application of a machine learning model for a binary classification task for financial markets. We will cover topics such as data preprocessing, feature engineering, model selection, and performance evaluation, as well as the development of a trading strategy based on the predictions made by the model.

This study is just the first part of an extensive notebook on **Machine Learning for Financial Markets** that I'm still developing on Kaggle, which I invite you to take a look at. This notebook is also a continuation of my previous job **Data Science for Financial Markets**, also available on Kaggle.

With that being said, let's install and import the libraries we're going to use for this project:

```
# Installing Technical Analysis library
!pip install ta

# Installing yfinance library
!pip install yfinance
```

```
# Importing Libraries

# Data Handling
import pandas as pd
import numpy as np

# Data Visualization
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objs as go
import matplotlib.ticker as mtick
from plotly.offline import init_notebook_mode
init_notebook_mode(connected=True)

# Financial Data Analysis
import yfinance as yf
import ta

# Machine Learning
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, roc_curve, auc
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Models
from sklearn.linear_model import LogisticRegression
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from catboost import CatBoostClassifier
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier

# Hiding warnings
import warnings
warnings.filterwarnings("ignore")
```

The goal of this study is to develop a machine learning model that is designed to predict the likelihood of tomorrow's closing price being higher or lower than today's closing price, so we buy shares or short sell them expecting for profits on the following day.

For this experiment, I'm going to use Apple stock's maximum of data available to train and test our model. Below, I download the historic data for Apple stocks using yfinance and plot a candlestick chart of the adjusted closing prices for Apple shares from the 1980's to 2023.

```
# Loading apple stocks until March 24th, 2023
aapl = yf.download('AAPL', end = '2023-03-24')
aapl
```

	Open	High	Low	Close	Adj Close	Volume
Date						
1980-12-12	0.128348	0.128906	0.128348	0.128348	0.099722	469033600
1980-12-15	0.122210	0.122210	0.121652	0.121652	0.094519	175884800
1980-12-16	0.113281	0.113281	0.112723	0.112723	0.087582	105728000
1980-12-17	0.115513	0.116071	0.115513	0.115513	0.089749	86441600
1980-12-18	0.118862	0.119420	0.118862	0.118862	0.092351	73449600
...	...	...	...	...	...	...
2023-03-17	156.080002	156.740005	154.279999	155.000000	155.000000	98862500
2023-03-20	155.070007	157.820007	154.149994	157.399994	157.399994	73641400
2023-03-21	157.320007	159.399994	156.539993	159.279999	159.279999	73938300
2023-03-22	159.300003	162.139999	157.809998	157.830002	157.830002	75701800
2023-03-23	158.830002	161.550003	157.679993	158.929993	158.929993	67622100

10659 rows × 6 columns

Initially, we have information on 10,659 trading days ever since December 12th, 1982. The features we initially have to work on are the 'Open' price, the 'High' price, the 'Low' price, the 'Close' prince, the 'Adjusted Close' price, and 'Volume'. The Adjusted Close price is the historic data adjusted by dividends and stock splits, which is a better representation to capture the historic value of shares over time.

Here's the Candlestick plotted with Plotlty:

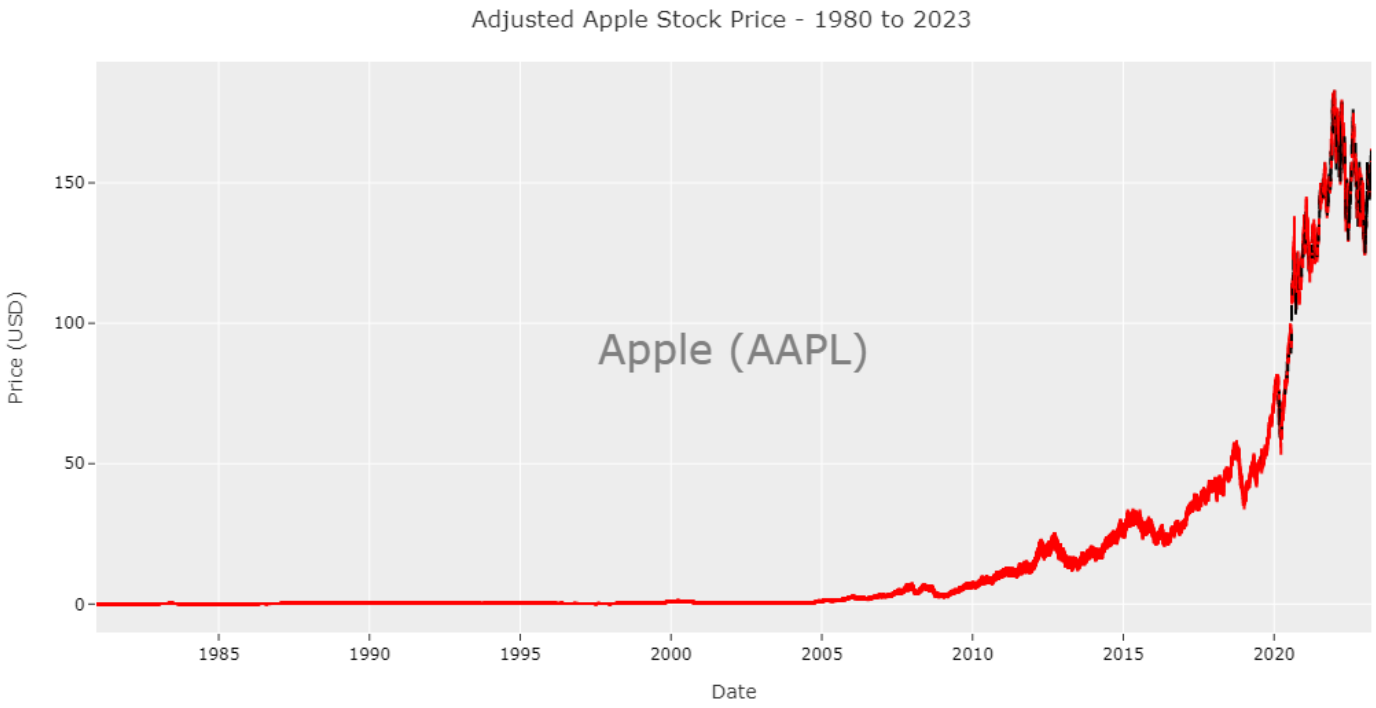

Figure 1 - Adjusted Closing Price from 1980 to 2023

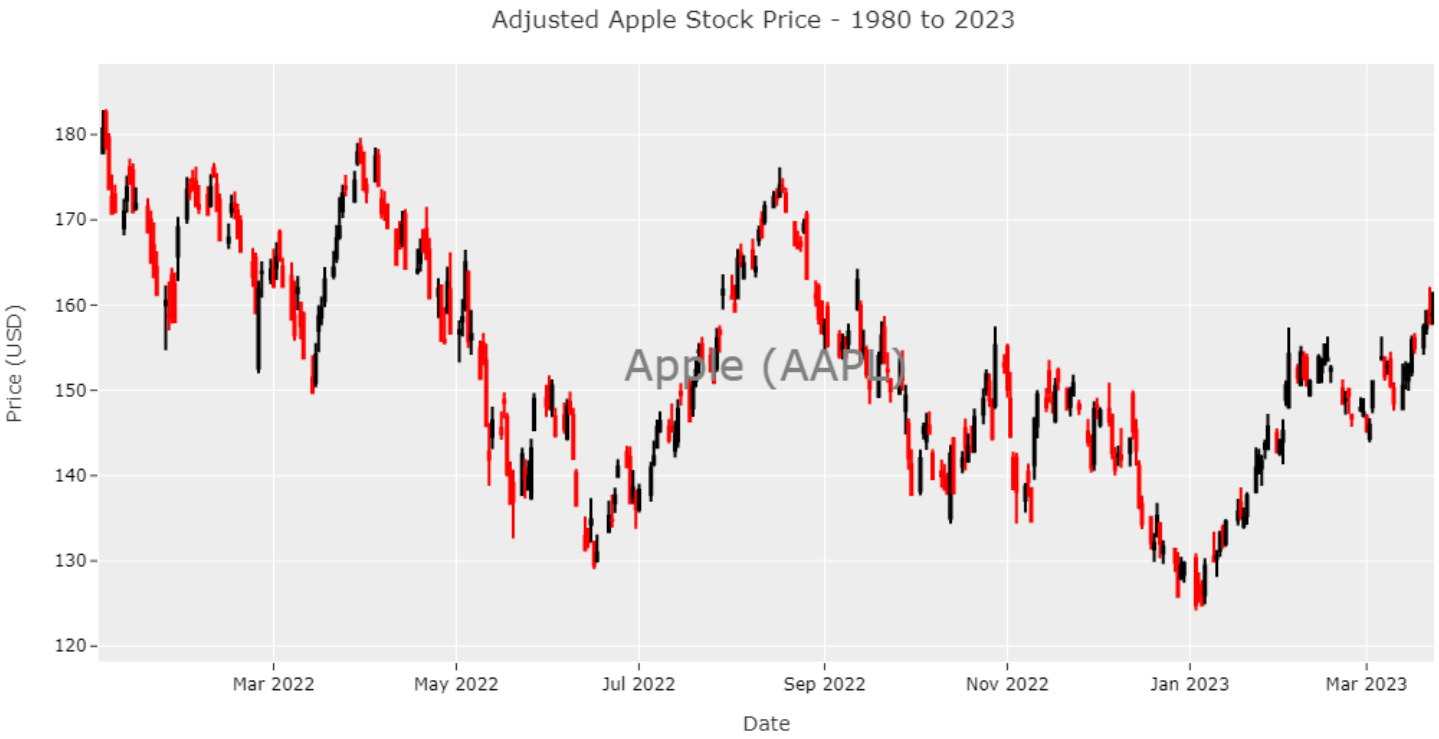

Figure 2- Apple stocks in the last year

To start building our model, we must first split the dataset into training and testing set. I am going to use data from 2010 and earlier to train the model, while the data from 2011 and onwards will be used to test the model's performance.

```
# Data Splitting
train = aapl[aapl.index.year <= 2010]
test = aapl[aapl.index.year >= 2011]
```

Then, I am going to add a new feature to both sets called "Close_Shift". This new feature is going to contain the previous day adjusted closing price. This is a fairly common practice when dealing with time series data, and it serves the purpose to shift forward values one or more steps.

Besides that, I am going to calculate the daily returns, which is the percentage difference between today's closing price and yesterday's closing price, given by the following equation:

$$\left(\frac{Today's\ Closing\ Price}{Yesterday's\ Closing\ Price} - 1\right) \times 100$$

```
# Adding Close_Shift Variable
train['Close_Shift'] = train['Adj Close'].shift(1)
test['Close_Shift'] = test['Adj Close'].shift(1)
```

```
# Adding Daily Returns variable
train['Return'] = (train['Adj Close']/train['Close_Shift'] - 1) * 100
test['Return'] = (test['Adj Close']/test['Close_Shift'] - 1) * 100
```

After doing that, we must create our ***target*** variable, which is the variable the model is going to be trained on and used to make predictions.

The "*target*" variable will receive binary values according the following's day returns. Panda's *shift( )* method will be used to bring the target one row above the daily returns. If the following day's returns are higher than zero, then target will be equal to 1, suggesting that the marked closed in a higher price than the current day. If the next day's returns are below zero, then the target variable will be equal to 0, indicating that the following day was bearish.

```
# Creating target variable on both datasets
train['target'] = np.where(train['Return'].shift(-1) > 0, 1, 0)
test['target'] = np.where(test['Return'].shift(-1) > 0, 1, 0)
```

Here's how our datasets look like now, with these three new attributes:

Warning: Do not forget that **target** is shifted upwards, so the value you see in this column is related to the next row's **Return** values!

	Open	High	Low	Close	Adj Close	Volume	Close_Shift	Return	target
Date									
1980-12-12	0.128348	0.128906	0.128348	0.128348	0.099722	469033600	NaN	NaN	0
1980-12-15	0.122210	0.122210	0.121652	0.121652	0.094519	175884800	0.099722	-5.217067	0
1980-12-16	0.113281	0.113281	0.112723	0.112723	0.087582	105728000	0.094519	-7.339774	1
1980-12-17	0.115513	0.116071	0.115513	0.115513	0.089749	86441600	0.087582	2.475110	1
1980-12-18	0.118862	0.119420	0.118862	0.118862	0.092351	73449600	0.089749	2.899209	1
1980-12-19	0.126116	0.126674	0.126116	0.126116	0.097987	48630400	0.092351	6.102873	1
1980-12-22	0.132254	0.132813	0.132254	0.132254	0.102756	37363200	0.097987	4.866967	1
1980-12-23	0.137835	0.138393	0.137835	0.137835	0.107093	46950400	0.102756	4.219943	1
1980-12-24	0.145089	0.145647	0.145089	0.145089	0.112729	48003200	0.107093	5.262740	1
1980-12-26	0.158482	0.159040	0.158482	0.158482	0.123135	55574400	0.112729	9.230946	1
1980-12-29	0.160714	0.161272	0.160714	0.160714	0.124869	93161600	0.123135	1.408297	0
1980-12-30	0.157366	0.157366	0.156808	0.156808	0.121834	68880000	0.124869	-2.430340	0
1980-12-31	0.152902	0.152902	0.152344	0.152344	0.118366	35750400	0.121834	-2.846803	1
1981-01-02	0.154018	0.155134	0.154018	0.154018	0.119666	21660800	0.118366	1.098777	0
1981-01-05	0.151228	0.151228	0.150670	0.150670	0.117065	35728000	0.119666	-2.173738	0

After creating the target variable, we can successfully divide our data into ***independent variables (X)*** and ***target variable (y)***.

```
X_train = train.drop('target', axis = 1) # Selecting Predictor Variables
y_train = train.target # Selecting Target Variable
X_test = test.drop('target', axis = 1) # Selecting Predictor Variables
y_test = test.target # Selecting Target Variable
```

Predicting the direction of the stock market is an extremely difficult activity and, to have at least some chance of success, we must have the highest amount of data and predictor variables as possible to increase the model's predictive power.

The next function was used to add a bunch of different indicators to the independent features sets, so the model can learn from them and identify relevant patterns to find the probabilities of the direction of the market for the following day.

Some of the new features are simply the most used indicators from Technical Analysis, such as:

. **Simple Moving Averages:** I'm going to add the SMAs for 5, 10, 15, 20, 30, 50, 80, 100, and 200 periods.

. **Relative Strength Index (RSI):** A momentum indicator used to identify overbought and oversold price regions.

. **Bollinger Bands:** A volatility indicator that can also be used to identify oversold and overbought price regions.

. **Commodity Channel Index (CCI):** A momentum indicator also used to identify overbought and oversold price regions.

. **On Balance Volume (OBV):** A volume-based indicator used to measure buying and selling pressure. It is also used to identify potential divergences between price and volume, which may signal a trend reversal.

Beyond that, I've also included some features that are derived from these indicators, such as:

. **Moving Average Ratios:** The Adjusted Close ÷ The Simple Moving Average.

. **RSI Overbought**: A binary feature that is going to be assigned the value of 1 if the RSI is above 70.

. RSI Oversold: A binary feature that is going to be assigned the value of 1 if the RSI is below 30.

. Above BB High: A binary feature that is going to be assigned the value of 1 if the Adj Close price is above the upper band of the Bollinger Bands.

. Below BB Low: A binary feature that is going to be assigned the value of 1 if the Adj Close price is below the lower band of the Bollinger Bands.

. OBV Divergence 10 Days: A feature that subtracts the sum of the differences of the OBV in the last ten days by the sum of the differences of the Adj Close prices in the last ten days.

. OBV Divergence 20 Days: A feature that subtracts the sum of the differences of the OBV in the last twenty days by the sum of the differences of the Adj Close prices in the last twenty days.

. CCI High: A binary feature that is going to be assigned the value of 1 if CCI is higher than 120.

.CCI Low: A binary feature that is going to be assigned the value of 1 if CCI is lower than -120.

. Shorter Moving Average > Longer Moving Average: A binary feature that is going to be assigned the value of 1 if a shorter moving average is above its immediate longer moving average, which indicates a higher trend in prices in that certain period.

```
# Defining feature engineering function
def feature_engineering(df):

    # Adding Simple Moving Averages
    df['sma5'] = ta.trend.sma_indicator(df['Adj Close'],window = 5)
    df['sma10'] = ta.trend.sma_indicator(df['Adj Close'],window = 10)
    df['sma15'] = ta.trend.sma_indicator(df['Adj Close'],window = 15)
    df['sma20'] = ta.trend.sma_indicator(df['Adj Close'],window = 20)
    df['sma30'] = ta.trend.sma_indicator(df['Adj Close'],window = 30)
    df['sma50'] = ta.trend.sma_indicator(df['Adj Close'],window = 50)
    df['sma80'] = ta.trend.sma_indicator(df['Adj Close'],window = 80)
    df['sma100'] = ta.trend.sma_indicator(df['Adj Close'],window = 100)
    df['sma200'] = ta.trend.sma_indicator(df['Adj Close'],window = 200)

    # Adding Price to Simple Moving Averages ratios

    df['sma5_ratio'] = df['Adj Close'] / df['sma5']
    df['sma10_ratio'] = df['Adj Close'] / df['sma10']
    df['sma20_ratio'] = df['Adj Close'] / df['sma20']
```

```
    df['sma30_ratio'] = df['Adj Close'] / df['sma30']
    df['sma50_ratio'] = df['Adj Close'] / df['sma50']
    df['sma80_ratio'] = df['Adj Close'] / df['sma80']
    df['sma100_ratio'] = df['Adj Close'] / df['sma100']
    df['sma200_ratio'] = df['Adj Close'] / df['sma200']

    # Adding RSI, CCI, Bollinger Bands, and OBV

    df['rsi'] = ta.momentum.RSIIndicator(df['Adj Close']).rsi()
    df['cci'] = ta.trend.cci(df['High'], df['Low'], df['Close'], window=20, constant=0
.015)
    bb_indicator = ta.volatility.BollingerBands(df['Adj Close'])
    df['bb_high'] = bb_indicator.bollinger_hband()
    df['bb_low'] = bb_indicator.bollinger_lband()
    df['obv'] = ta.volume.OnBalanceVolumeIndicator(close=df['Adj Close'], volume=df['V
olume']).on_balance_volume()

    # Adding features derived from the indicators above

    df['rsi_overbought'] = (df['rsi'] >= 70).astype(int)
    df['rsi_oversold'] = (df['rsi'] <= 30).astype(int)
    df['above_bb_high'] = (df['Adj Close'] >= df['bb_high']).astype(int)
    df['below_bb_low'] = (df['Adj Close'] <= df['bb_low']).astype(int)
    df['obv_divergence_10_days'] = df['obv'].diff().rolling(10).sum() - df['Adj Close'
].diff().rolling(10).sum()
    df['obv_divergence_20_days'] = df['obv'].diff().rolling(20).sum() - df['Adj Close'
].diff().rolling(20).sum()
    df['cci_high'] = (df['cci'] >= 120).astype(int)
    df['cci_low'] = (df['cci'] <= -120).astype(int)
    df['sma5 > sma10'] = (df['sma5'] > df['sma10']).astype(int)
    df['sma10 > sma15'] = (df['sma10'] > df['sma15']).astype(int)
    df['sma15 > sma20'] = (df['sma15'] > df['sma20']).astype(int)
    df['sma20 > sma30'] = (df['sma20'] > df['sma30']).astype(int)
    df['sma30 > sma50'] = (df['sma30'] > df['sma50']).astype(int)
    df['sma50 > sma80'] = (df['sma50'] > df['sma80']).astype(int)
    df['sma80 > sma100'] = (df['sma80'] > df['sma100']).astype(int)
    df['sma100 > sma200'] = (df['sma100'] > df['sma200']).astype(int)

    # Removing NaN values from the dataframe
    df.dropna(inplace = True)
    return df
```

After creating the function, we can apply it to the X_train and X_test datasets:

```
# Applying function to the X_train and X_test sets
X_train = feature_engineering(X_train)
X_test = feature_engineering(X_test)
```

This leaves us with 46 total predictor features, being those the following:

```
Index(['Open', 'High', 'Low', 'Close', 'Adj Close', 'Volume', 'Close_Shift',
       'Return', 'sma5', 'sma10', 'sma15', 'sma20', 'sma30', 'sma50', 'sma80',
       'sma100', 'sma200', 'sma5_ratio', 'sma10_ratio', 'sma20_ratio',
       'sma30_ratio', 'sma50_ratio', 'sma80_ratio', 'sma100_ratio',
       'sma200_ratio', 'rsi', 'cci', 'bb_high', 'bb_low', 'obv',
       'rsi_overbought', 'rsi_oversold', 'above_bb_high', 'below_bb_low',
       'obv_divergence_10_days', 'obv_divergence_20_days', 'cci_high',
       'cci_low', 'sma5 > sma10', 'sma10 > sma15', 'sma15 > sma20',
       'sma20 > sma30', 'sma30 > sma50', 'sma50 > sma80', 'sma80 > sma100',
       'sma100 > sma200'],
      dtype='object')
```

Before modelling, it's important to equalize the number of samples of the X sets and y sets. In the *feature_engineering* function, we've dropped all NaN values from the X_train and X_test, so we drop these samples from the y sets by linking them to the X's sets indexes:

```
# Removing from y_train and y_test the NaN values that were dropped from X_train and X_t
est by the index
y_train = y_train[X_train.index]
y_test = y_test[X_test.index]
```

Now that we have many predictor variables, we can train some classifiers and compare their performance to see how well they predict probabilities for the following day closing price.

For this task, I will use the Area Under the Curve (AUC) score, which is a popular metric to evaluate the performance of binary classifiers. Overall, an AUC score equal to 0.5 indicates a model that performs just as good as a random guessing, while an AUC score equal to 1.0 indicates a classifier that can correctly predict every single class of our target variable.

```
# Creating a list of different classification models
classifiers = [
    LogisticRegression(random_state = 42),
    XGBClassifier(random_state = 42),
    LGBMClassifier(random_state = 42),
    CatBoostClassifier(random_state = 42, verbose = False),
    AdaBoostClassifier(random_state = 42),
    RandomForestClassifier(random_state = 42)
]

#  Iterating over classifiers in the list above, training, and evaluating them
for clf in classifiers:
    clf.fit(X_train, y_train)
    y_pred = clf.predict_proba(X_test)[:,1]
    auc_score = roc_auc_score(y_test, y_pred)
    print(f'{type(clf).__name__}: AUC Score={auc_score:.3f}')
```

```
LogisticRegression: AUC Score=0.515
XGBClassifier: AUC Score=0.488
LGBMClassifier: AUC Score=0.492
CatBoostClassifier: AUC Score=0.508
AdaBoostClassifier: AUC Score=0.518
RandomForestClassifier: AUC Score=0.478
```

The **AdaBoost Classifier** achieved the highest performance among all the other indicators.

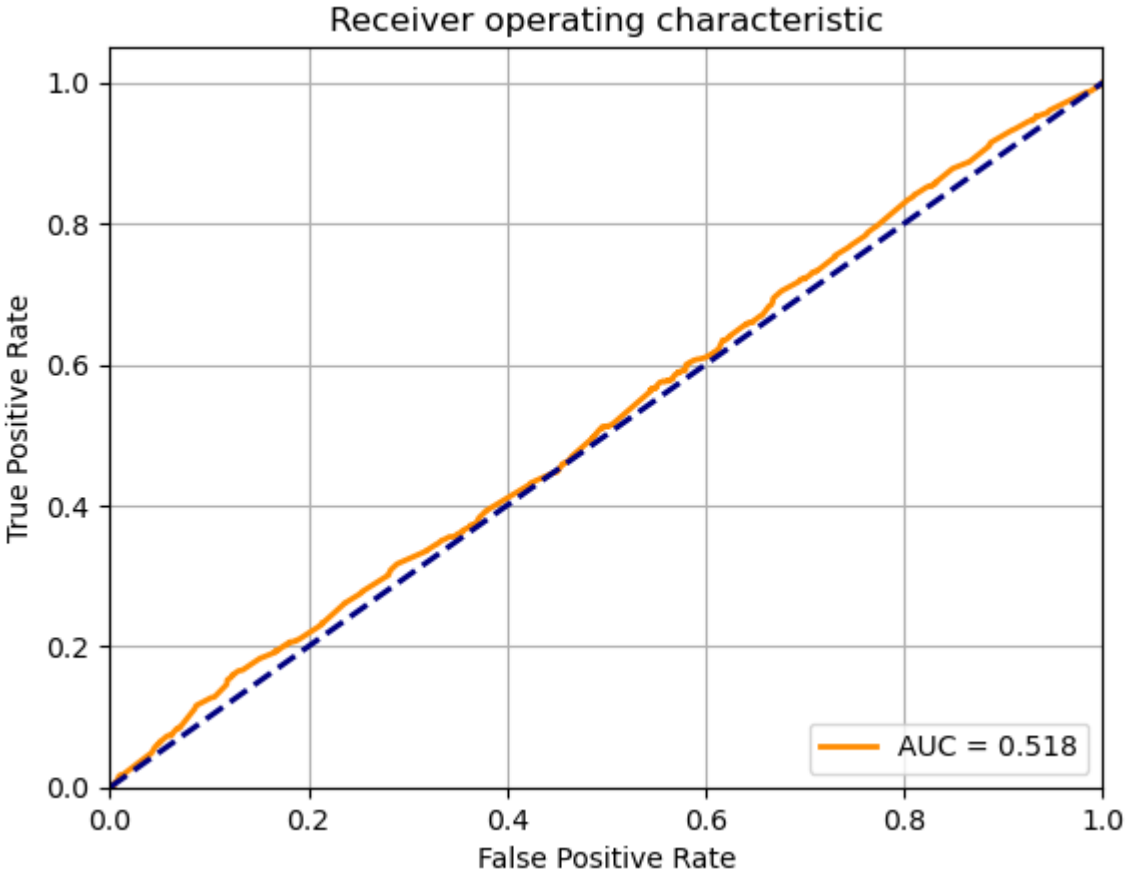

Figure 3- AdaBoost Classificer AUC Score = 0.518

An AUC score of 0.518 indicates a model that performs just slightly better than a random guessing. Although for most machine learning projects this score would be discouraging – to say the least, in financial markets, being slightly better than a simple coin toss is precisely what you need to have higher returns than most traders have.

We can take a look at the model's feature importance plot to see what were the most relevant features for the model's predictions

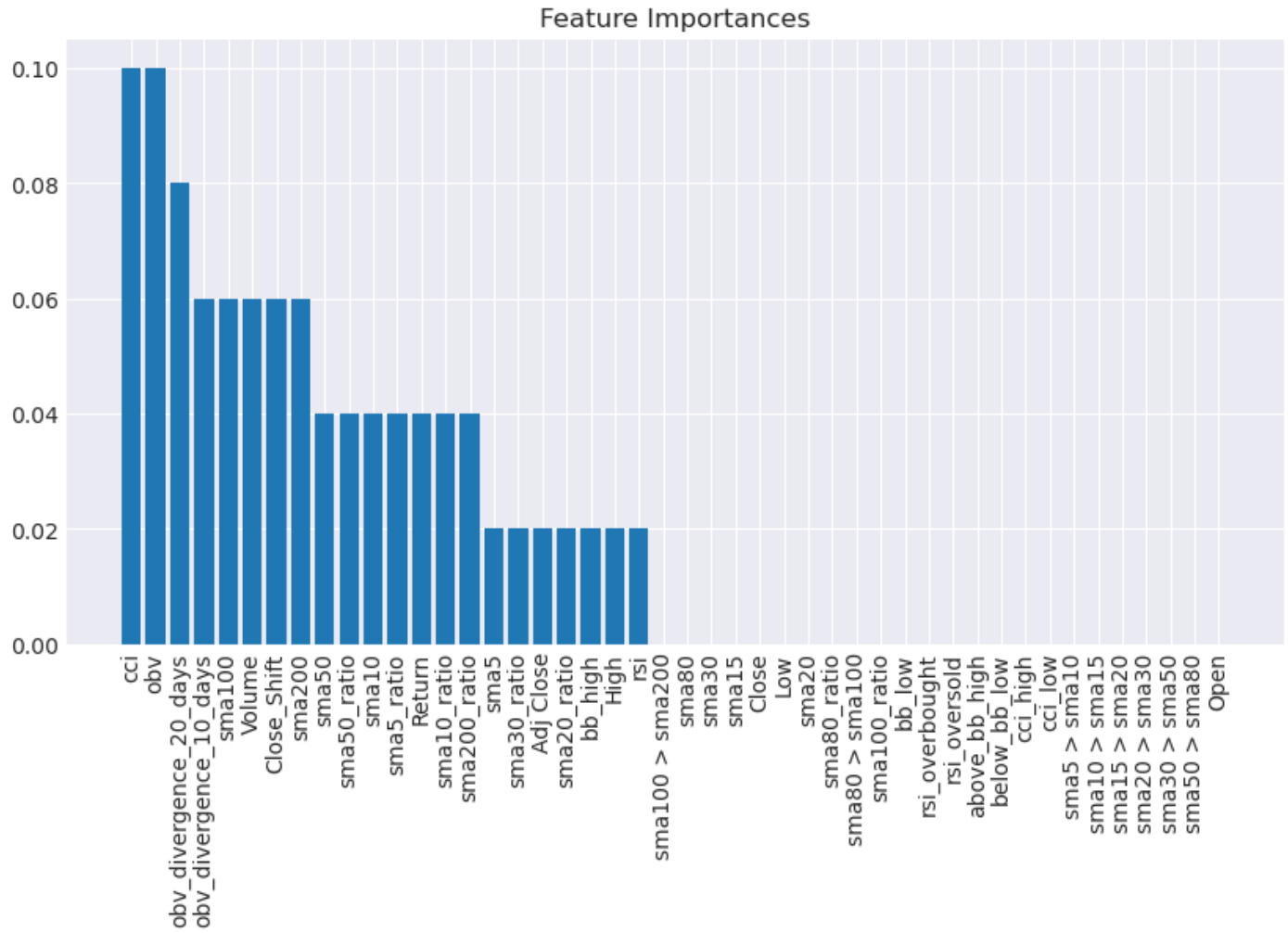

Figure 4- Feature Importances Plot

The CCI and OBV indicators were the most relevant features for predictions, followed by the 10-day and 20-day divergence indicator for the OBV. The simple moving averages and their ratios also seemed to play an important role for predicting probabilities for the following day closing price.

Let's take a deeper look at the output of our model.

```
y_pred # Displaying probabilities
array([0.5183526 , 0.51177782, 0.51949593, ..., 0.51007696, 0.50940466,
0.50940466])
```

The output is an array consisting of different probability values that range from 0.0 to 1.0. Usually, most people simply consider that any value below 0.5 indicates a selling opportunity, while any value above that threshold would suggest a buying opportunity.

We can, however, develop a strategy that is much more robust than that.

By dividing the *y_pred* array into eight groups according to the percentile ranks, we can find different ranges of probability values and compare them to the *y_test* values to see how successful the model was at predicting the market's direction according to each range.

```
quantile_intervals = np.arange(0, 1.125, 0.125) # Creating intervals for the quantiles
quantile_values = np.quantile(y_pred, quantile_intervals) # Dividing y_pred in 8 diffe
rent quantiles
classes = pd.cut(y_pred, quantile_values) # Grouping probabilities according to the qua
ntiles
X_test['y_true'] = y_test # Creating a new column in the X_test set containing the true
label values
table = pd.crosstab(classes, X_test['y_true']) # Creating a table counting true labels
according to each probability range
table
```

y_true	0	1
row_0		
(0.3477, 0.5087]	207	183
(0.5087, 0.5094]	162	186
(0.5094, 0.5112]	160	198
(0.5112, 0.5185]	170	184
(0.5185, 0.5189]	163	185
(0.5189, 0.5195]	183	176
(0.5195, 0.5202]	180	202
(0.5202, 0.5252]	144	193

For instance, we can see that, from prediction values ranging from 0.5189 to 0.5195, there were more bearish days than bullish days. So we're going to develop a strategy where, whenever our model gives us a probability anywhere between these numbers, we go short on Apple stocks. The same is true for ranges where the next day ended up having more bullish than bearish prices.

In the code below, I create a NumPy array that generates signals based on the ranges identified in the table above. To represent bearish ranges, we will assign the value of -1, while for bullish ranges, we will assign the value of 1.

```
sign = np.zeros_like(y_pred) # Creating an array with 0s in the same length as y_pred

# Short selling signal
sign[((y_pred >= 0.3477) & (y_pred < 0.5087)) | ((y_pred > 0.5189) & (y_pred < 0.5195)
)] = -1

# Buying signal
```

```
sign[((y_pred >= 0.5087) & (y_pred <= 0.5189)) |
            ((y_pred >= 0.5195) & (y_pred <= 0.5252))] = 1

# Display signal values
sign
```

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

Now, I create a new "*sign*" column in the X_test containing the values from the array above. So, when "sign" is equal to 1, we buy Apple shares expecting for a raise in prices for the next day. When "sign" is equal to -1, we go short on Apple shares expecting for a decline in prices for the next day.

With this in mind, I will also add a new column called "position", that is going to contain the position we hold for each specific trading day, either we're long or short according to the sign given by the model in the previous day.

Additionally, I'll also add a "model_returns" column that is simply going to multiply the values in the "position" column by the daily return, which is going to indicate whether our position generated profits or losses.

```
X_test['sign'] = sign # Creating new 'sign' column
X_test['position'] = X_test['sign'].shift(1) # Creating 'position' attribute
X_test['model_returns'] = X_test['position'] * X_test['Return'] # Creating 'daily retu
rns' attribute for the strategy
X_test.head(8) # Displaying X_test set
```

Volume	Close_Shift	Return	sma5	sma10	...	sma15 > sma20	sma20 > sma30	sma30 > sma50	sma50 > sma80	sma80 > sma100	sma100 > sma200	y_true	sign	position	model_returns
686044800	12.827568	-0.476288	12.480434	11.975326	...	0	1	1	1	1	1	1	1.0	NaN	NaN
881602400	12.766472	0.535729	12.613877	12.126521	...	0	1	1	1	1	1	0	1.0	1.0	0.535729
1104059600	12.834866	-5.593992	12.592174	12.188440	...	0	1	1	1	1	1	0	-1.0	1.0	-5.593992
549270400	12.116884	-0.830367	12.512412	12.242972	...	0	0	1	1	1	1	0	-1.0	-1.0	0.830367
621244400	12.016270	-0.617216	12.335319	12.313098	...	1	0	1	1	1	1	1	-1.0	-1.0	0.617216
502138000	11.942103	3.283512	12.248869	12.364652	...	1	0	1	1	1	1	0	-1.0	-1.0	-3.283512
430427200	12.334224	-1.971559	12.100106	12.356991	...	1	0	1	1	1	1	1	1.0	-1.0	1.971559
456304800	12.091047	0.711479	12.112143	12.352159	...	1	0	1	1	1	1	1	1.0	1.0	0.711479

8 rows × 50 columns

Figure 5- X_test after including new columns

Now we have a sign, generated by our model's probability predictions, telling us either to go long or short in Apple stocks, a position, which contains whether we're long or short for each specific day, and the daily returns for our strategy.

For instance, on October 18th, 2011 – the second row in the dataframe, our model signed to us that we should buy Apple stocks. On the next day, October 19th, we were long on Apple stocks, however, the stocks decreased in value compared to the previous day, which resulted in losses of 5.59% (third row).

On October 20th, 2011 – fourth row in the dataframe, our model signed a short-selling opportunity. On the 21st, we were short on Apple stocks, which closed that day at -0.61% compared to the previous trading day, resulting in gains of 0.61% for our strategy.

By calculating the daily returns for the strategy, we can look back at the last decade and calculate the cumulative returns of Apple stocks overall and the cumulative returns of our strategy. We can, then, compare a simple Buy and Hold strategy to our model, and see which would give us the highest return in the last 11 years.

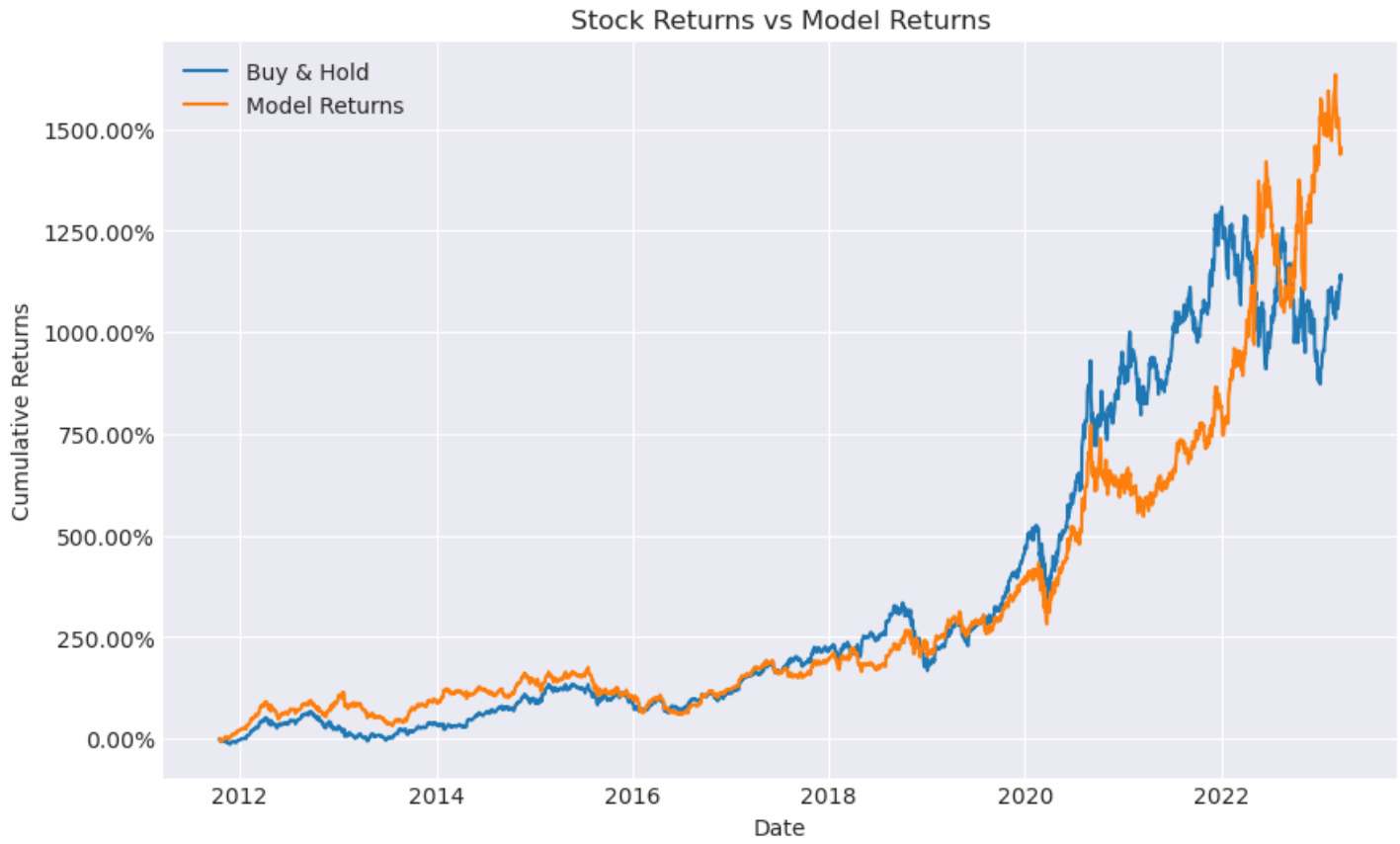

Figure 6- Buy & Hold Performance vs Model Performance Since 2011

```
Buy & Hold Cumulative Returns = 1,138.97%
Model Cumulative Returns = 1,452.22%
```

The line plot above shows a comparison of the performance of a simple Buy & Hold strategy versus a strategy based on the predictions generated by our machine learning model.

It’s evident from the plot, that over time, the model-based strategy outperformed the Buy & Hold approach. However, there were some periods where the Buy & Hold approach performed better.

The overall performance of the strategy resulted in impressive cumulative returns of 1,452.22%, compared to 1,138.97% achieved by the Buy & Hold strategy. However, it's crucial to note that these figures do not account for expenses such as fees, taxes, and other costs associated with implementing the strategy.

The model built in this study is just an initial approach, and there are still many potential improvements to be made. Fine-tuning the model and incorporating additional attributes into the data are just a few examples.

One potential approach to improve the model's performance could be to incorporate the closing prices and returns of other highly correlated stocks, such as those in the same industry as Apple. Another idea could be to include benchmark indices such as the SP 500, as well as other relevant technical indicators, to enhance the model's predictive power. Let your creativity be the limit!

Thanks for reading,

Luís Fernando Torres

Follow me on LinkedIn: https://www.linkedin.com/in/luuisotorres/

Follow me on Kaggle: https://www.kaggle.com/lusfernandotorres