# Random Forest Classification

## Predicting stock price direction with scikit-learn

**Abstract**

This report accompanies the code in the corresponding GitHub repository. We present an implementation of the Random Forest Classifier (RFC) included in the Python library scikit-learn. After giving an overview of the RFC algorithm, we use it to predict the direction of Apple Inc. stock prices, loosely following [1].

# Contents

# 1 Introduction

## 1.1 Overview of Random Forest Classification

Random forest is a supervised machine learning algorithm that uses the outputs of an ensemble of decision trees to reach a classification result. We first define decision tree classification, before explaining how random forest uses decision trees to reach a final result.

### 1.1.1 Decision Trees

A decision tree consists of a set of nodes and a set of branches, whereby decisions are made about data at each node. At each node, a yes/no question is asked of the dataset, and data is filtered according to the answer to said question. For example, if the data is a set of coordinates $(x, y)$ which are split into two types "red" and "blue", a node could filter data according to the question "$x > 3$?". The subset of data in the that satisfies the condition is sent to one node, and the data that doesn't is sent to another. Questions are repeatedly asked of the data, until we reach so-called *leaf nodes*; nodes that contain only one type of data. A typical setup is shown in figure 1.

This is considered a machine learning algorithm because there are many possible choices of conditions that split the data in many possible ways, and so the model needs to learn which splitting conditions lead to an "optimal" split. "Optimal" here means choosing splits that cause leaf nodes to appear as early as possible. The model works this out by choosing splitting conditions at each node that maximise information gain, which can be quantified in a number of ways. The most common ways are called the *Shannon Entropy* (or simply *entropy*) and the *Gini Index* (or *Gini Impurity*). The entropy is given by

$$S := -\sum_i p_i \log(p_i), \tag{1.1}$$

where $p_i$ is the probability of class $i$ [1]. The information gain at a particular node is then given by

$$I := S_{\text{Parent}} - \sum_i w_i \, S_{\text{Child}_i}, \tag{1.2}$$

where the weights $w_i$ are the relative sizes of the child node with respect to the parent, namely

$$w_i = \frac{\text{Number of nodes in child } i}{\text{Number of nodes in the parent}}. \tag{1.3}$$

The Gini Impurity is somewhat simpler, and is defined by

$$G := 1 - \sum_i p_i^2, \tag{1.4}$$

where $p_i$ is the probability of class $i$. The optimum split is then considered to be the split that minimises the Gini impurity.

[1] Namely, the probability of class • for a particular splitting condition ($x > 9$, for example) is given by (Number of • in the node after the condition)/(Total number of nodes)

After training the tree on a training dataset, we can predict the type of a new data point by sending it through the tree, and assigning it the type shared by the points in the leaf node it ends up in. For instance, in the setup shown in figure 1, if we send a new point $(3, 3)$ through the tree, it ends up in the first "1" node, and so we assign it to class •.

| Coordinates | Type |
|---|---|
| (1,1) | • |
| (3,6) | • |
| (2,1) | • |
| (2,5) | • |
| (4,2) | • |

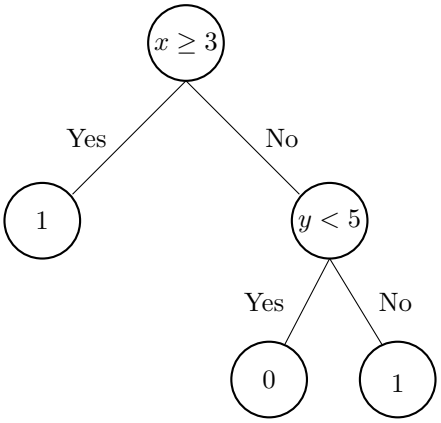


Figure 1: A dataset with a decision tree, classifying the data into red and blue types. Red leaf nodes are denoted by class 1, and blue leaf nodes are denoted by class 0. A new data point $(3, 3)$ ends up in the first leaf node ("1"), and so it is assigned to the • class.

### 1.1.2 Random Forests

A single decision tree turns out to be highly sensitive to the initial data one feeds it. Namely, if we change the training data by only a small amount we end up with a completely different tree. A random forest helps to solve this issue by using many decision trees (a "forest" of trees), and assigning the new data point to the majority output class.

A random forest algorithm can be split into two main steps; *Bootstrapping* and *Aggregating*, together often called *Bagging*.

1. *Bootstrapping:* We first create a number of new datasets from the original by randomly sampling from the dataset with replacement. We then create a tree for each of these datasets. Each new dataset that is created will be smaller than the original, since only a proper subset of the features in the original dataset will be included in the new one. The features used to create the new dataset are chosen at random, and this process is known as "feature selection".

The number of features included is commonly chosen to be the square root of the total number of features, rounded to the nearest integer.

Bootstrapping ensures that we do not use the same data for every tree, meaning that the output of a random forest is less sensitive to small changes in the original training data. The random feature selection helps to reduce correlation between the trees - none of the trees see the entire dataset. If we were to use every feature, then most of the trees would have the same decision nodes and would act very similarly.

Bootstrapping does cause some trees to be trained on less important features, meaning that those trees will produce particularly bad predictions. However, other trees will be trained on more important features, and so will give particularly good predictions. There is therefore no net effect.

2. *Aggregating:* In the aggregation step, we pass a new data point through each tree produced in the bootstrapping step one-by-one, and record the result. We then assign the new data point to the class predicted by the majority of the trees in the forest.

### 1.2 This Report

This report accompanies the code in the corresponding GitHub repository, and showcases a simple implementation of the random forest classification algorithm included in the machine learning python library scikit-learn, loosely following [1]. We use Apple stock price data from Yahoo finance to derive various technical indicators to use as features for the model, and after fitting the model to the training dataset we rank the features from most to least important. After dropping one of the least important features and tuning the hyperparameters of the model, we achieve a precision score of $\sim$91.2% and an accuracy score $\sim$71.9%, respectively.

## 2 Data Processing & Feature Extraction

We first download Apple price data from Yahoo finance from between 2019/01/01 and today, at a time frequency of 1 entry per day. The raw data is a pandas dataframe whose columns are High, Low, Open, Close, Adjusted Close, Volume, Dividends and Stock Splits, the final two we drop.

Before deriving the features for our model, we perform some data preprocessing in the form of a "smoothing" of each quantity in the dataframe. We apply the standard exponential smoothing procedure, whereby denoting the raw dataset as $\{x_t \,|\, t \geq 0\}$ and the smoothed dataset as $\{S_t \,|\, t \geq 0\}$, we define the smoothing by

$$\begin{aligned} S_0 &= x_0 \\ S_t &= \alpha x_t + (1-\alpha) S_{t-1}, \end{aligned} \tag{2.1}$$

where $0 < \alpha < 1$ is a parameter called the *smoothing factor*. The purpose of exponential smoothing is to apply exponentially increasing weights to observations as time increases, such that more recent observations are weighted higher. We show the smoothed close prices as a function of time in figure 2.

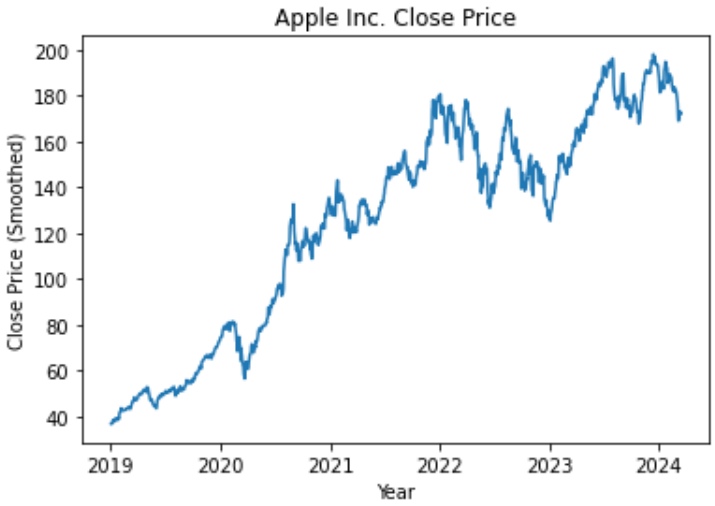


Figure 2: A plot showing the close prices of Apple stock as a function of time, after the exponential smoothing is applied.

After performing the smoothing, we set up the target for the model. We are interested in knowing whether or not a stock bought today will be higher or lower in price after a time horizon of 90 days. Our target column will show -1 if the price is lower than today, and +1 if it is higher than today. Namely, our target is calculated as

$$\text{Target}_i = \text{sgn}(\text{Close}_{i+90} - \text{Close}_i), \tag{2.2}$$

where by "Close" we mean the exponentially smoothed close price of the stock. After setting up the target, we begin feature extraction.

## 2.1 Relative Strength Index (RSI)

The RSI is a momentum indicator[2] used to measure the speed and magnitude of price changes, in an attempt to evaluate whether the stock is under- or over-valued. We first define the ratio between the average gain over the past 90 days and the average loss. Namely,

$$\mathcal{A} := \frac{\text{Average Gain over past 90 days}}{\text{Average Loss over past 90 days}}, \tag{2.3}$$

from which we define the RSI as

$$\boxed{\text{RSI} := 100 - \frac{100}{1+\mathcal{A}}.} \tag{2.4}$$

The RSI is a number between 0 and 100, whereby if RSI $> 70$ the stock is considered overbought and for RSI $< 30$ the stock is considered oversold.

[2]Momentum indicators are a class of indicators used to measure the rate of rise or fall of security prices.

## 2.2 Stochastic Oscillator

A common school of thought is that in a bullish market prices tend to close near a recent high, and in a bearish market prices tend to close near a recent low. The stochastic oscillator is a momentum indicator that compares the current close price of a security with the recent high and low prices over a certain time frame, sometimes called the lookback period. We will use a time frame of 14 days, and so for us the stochastic oscillator is given by

$$\boxed{\mathcal{S} := 100\left(\frac{\text{Close} - \text{Low}_{14}}{\text{High}_{14} - \text{Low}_{14}}\right)}, \tag{2.5}$$

where $\text{High}_{14}$ and $\text{Low}_{14}$ are the highest and lowest close prices over the previous 14 days. Similarly to the relative strength index, $\mathcal{S}$ is a number between 1 and 100, where traditionally $\mathcal{S} > 80$ is thought to indicate an overbought stock and for $\mathcal{S} < 20$ the stock is considered oversold.

## 2.3 Williams Percent Range

The Wiliams Percent Range is very similar to the stochastic oscillator in that it compares the current close price to the high and low over some lookback period, which we take to be 14 days. We define the Williams %R as

$$\boxed{\mathcal{W} := -100\left(\frac{\text{High}_{14} - \text{Close}}{\text{High}_{14} - \text{Low}_{14}}\right)}. \tag{2.6}$$

$\mathcal{W}$ is a number between -100 and 0; $\mathcal{W} > -20$ indicates a sell signal and $\mathcal{W} < -80$ indicates a buy signal.

## 2.4 Moving Average Convergence Difference

The Moving Average Convergence Difference (MACD) is given by a difference of two exponential moving averages;

$$\mathcal{M} := \mu_{12}(\text{Close}) - \mu_{26}(\text{Close}), \tag{2.7}$$

where the subscripts indicate the time period over which the exponential moving average is taken. This, together with a so-called *signal line* calculated via

$$\mathcal{M}_{\text{Signal}} = \mu_9(\mathcal{M}), \tag{2.8}$$

generates a sell signal when $\mathcal{M} < \mathcal{M}_{\text{Signal}}$, and a buy signal when $\mathcal{M} > \mathcal{M}_{\text{Signal}}$.

## 2.5 Price Rate of Change

The Price Rate of Change (PROC) is a simple indicator that measures a recent change in price with respect to a price from $n$ days ago. It is defined by

$$\mathcal{P}_i := \frac{\text{Close}_i - \text{Close}_{i-n}}{\text{Close}_{i-n}}, \tag{2.9}$$

and in this project we chose $n = 90$.

### 2.6 On Balance Volume

On Balance Volume (OBV) is a technical indicator that uses changes in volume to predict changes in stock price. It can be calculated as soon as two observations are available, and is defined by

$$\mathcal{O}_t := \begin{cases} \mathcal{O}_{t-1} + \text{Volume}_t & \text{If Close}_t > \text{Close}_{t-1} \\ \mathcal{O}_{t-1} & \text{If Close}_t = \text{Close}_{t-1} \\ \mathcal{O}_{t-1} - \text{Volume}_t & \text{If Close}_t < \text{Close}_{t-1}. \end{cases} \tag{2.10}$$

We take the starting point to be $\mathcal{O}_0 = 0$.

### 2.7 Garman-Klass Volatility

The Garman-Klass Volatility (GKV) is a technical indicator usually applied in the forex market, and can be used as a measure of the volatility of a stock. We define

$$\mathcal{G} := \frac{(\ln(\text{High}) - \ln(\text{Low}))^2}{2} - (2\ln(2) - 1)(\ln(\text{Close}) - \ln(\text{Open}))^2. \tag{2.11}$$

### 2.8 Additional Features

We also use various ratios as features for our model; in particular we use the ratio between Open and Close, High and Close, Low and Close, and High and Low. In addition, we used the adjusted close price as a feature.

## 3 Implementation & Results

To implement the model, we first split the data into a training set and a test set using the scikit-learn function train_test_split. We use the first 70% of the data to train the model, and the last 30% to test it.

We then use RandomizedSearchCV to run a Random Forest Classifier with a range of hyperparameter values, in order to get a ballpark estimate of the best values. We run the random search on the values in figure 3.

We then use GridSearchCV in order to further narrow down the options for the optimal hyperparameters, before running the RFC on the results of the Grid Search. We fit the RFC to the training data, generate predictions for the above defined target, and generate precision and accuracy scores. The precision is defined

$$\text{Precision} = \frac{T_P}{T_P + F_P}, \tag{3.1}$$

| n_estimators | min_samples_split | max_depth | max_leaf_nodes |
|---|---|---|---|
| 65 | 10 | 12 | 150 |
| 80 | 20 | 20 | 200 |
| 95 | 30 | 28 | 250 |
| 110 | 40 | 36 | 300 |
| 125 | — | — | 350 |
| 140 | — | — | — |
| 155 | — | — | — |
| 170 | — | — | — |

Figure 3: A table showing the parameters used to run the random search, in order to get a ballpark estimate of the optimal hyperparameters for the random forest classifier.

where $T_P$ and $F_P$ are the number of true and false positives, respectively. Intuitively, the precision score measures the ability of the classifier to not record as positive a sample that is negative. The accuracy score is simply the fraction of correctly classified samples. For the hyperparameters n_estimators=110, min_samples_split=50, max_depth=20, max_leaf_nodes=250, and using the gini impurity as our measure of the optimum split, we find precision and accuracy scores of ∼91.4% and ∼64.6%, respectively.

We also calculate importance scores for the features, and visualise these via the bar chart shown in figure 4. Notice that the Open/Close ratio and the Williams Percent Range are the least important features. We find that we get the best accuracy and precision scores if we keep the OCR, but drop $\mathcal{W}$. Retraining the model on the remaining features, and adjusting the hyperparameters such that n_estimators=110, min_samples_split=31, max_depth=20, max_leaf_nodes=250, and using the entropy as our measure of the optimum split, we find precision and accuracy scores of ∼91.2% and ∼71.9%, respectively.

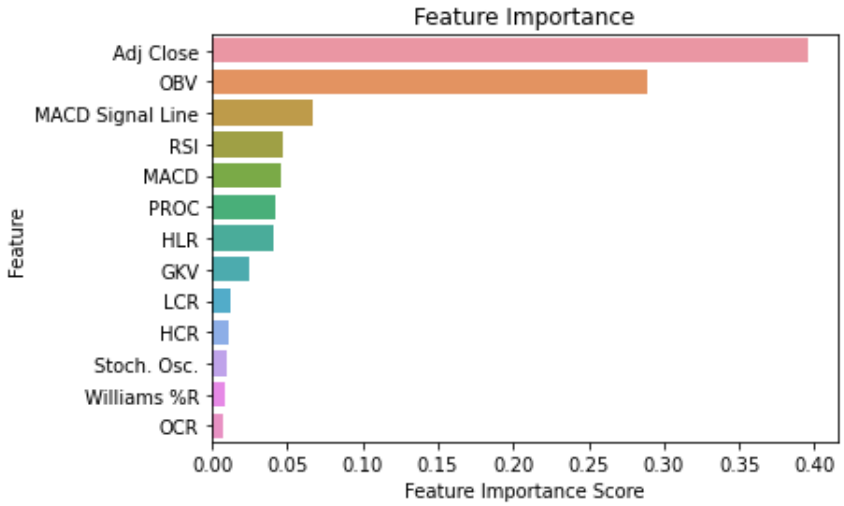


Figure 4: A bar chart showing the importance scores for each feature in the model. Note that the Open/Close ratio and the Williams Percent Range are the least important features.

# References

[1] Khaidem, L., Saha, S., & Dey, S. R.
Predicting the direction of stock market prices using random forest
arXiv:1605.00003. doi:10.48550/arXiv.1605.00003, 2016.