PREDICTING THE DIRECTION OF STOCK PRICES

USING RANDOM FOREST IN PYTHON

January 4th, 2020

Alex Reed

Contents

Introduction:

Before we get started, I wanted to let you know that this entire project was inspired by a research paper I read regarding stock movement prediction using Random Forest. The goal of this project is to recreate a large portion of that paper.

The source GitHub repository for this project can be found at the following link along with repo for the original paper:

- **Original Repo Link**
- **My Project Repo Link**

Additionally, I provide a copy of their repository and the paper in my own GitHub repo so that others can use it as a reference. **In this project, we will create a Random Forest model to predict whether a stock will close up or down based on previous historical technical indicator values.**

Background: What is Random Forest?

Random Forest is a machine learning ensemble method that is widely used because of its flexibility, simplicity, and often quality results. In this tutorial, we will use the Random Forest algorithm to build a classification model that will help us predict whether a stock will close higher or lower based on a range of technical indicators.

Now, I gave you a simple definition of what Random Forest is. However, I think it's crucial to delve into more detail about it before we start coding our model. Now, in short, Random Forests is a supervised machine learning algorithm that uses multiple decision trees in aggregate to help make more stable and accurate predictions. Decision Trees are one of the building blocks of Random Forest, so to understand Random Forest, we first need to understand Decision Trees.

Understanding Random Forest: What are Decision Trees?

Every day of your life, you make decisions. Your decisions can range from what to eat for lunch to what time you leave for work to avoid traffic. However, have you ever taken a step back to understand how you arrive at the decisions you make?

Sometimes you probably do, but other times it feels like auto-pilot, and you just do it. Well, usually, most decisions are based on a set of rules that you follow to arrive at a decision. For example, if I am hungry, I might have a set of rules I used to determine what I'll eat. Let's list my rules:

Rule 1: Determine if I am hungry.
Rule 2: Determine if I have money.
Rule 3: Determine if I want something light or heavy.

If you notice, we flow from one rule to next. In other words, I first must determine if I am even hungry if I am, then I can proceed to rule number 2 and decide whether I have money. There is a flow from one rule to the next.

Additionally, I can further specify my rules by adding criteria to them. For example, at rule number 2, I can have multiple possible levels of having money $50.00, $25.00, or $10.00. Then based on that, I can then move to rule number 3.

Let's walk through a simple example. First, I determine I'm hungry, so I then proceed to rule 2. I determine I have $10.00 to my name, so I can move to rule 3. After determining I have $10.00, I decide I want something light, so I choose to buy soup.

Well, if you were able to follow all of that, you now have the foundation of a Decision Tree. Decision Trees is a flow-like chart structure where each node of the tree is used to test an attribute of the object.

The node in our example up above would represent one of the rules we define. The attribute would be the value we were testing. For instance, I checked my net worth (the amount of money I had), and I also tested my hunger level. The object we are testing is me! I'm the one who gets tested on all the different rules.

Understanding Random Forest: Decision Trees Components

To provide a more formal definition of everything, let's walk through some points again. The first thing I want to mention is that for Decision Trees, there are three kinds of nodes and two kinds of branches.

Decision Node:

Whenever I have to make a choice, we will define this as a "Decision Node," so, for example, one decision node is testing whether I have money or not. A decision node is a point in the tree where a choice must be made.

Root Node:

The starting point of our tree is called the "Root Node," so in our example up above determining if I am hungry or not is the root node. The root node represents the entire population or sample, which is then divided into two or more homogenous sets.

Leaf/Terminal Node:

When we reach the end of our tree, and we don't have to make any more decisions, we have reached a "Leaf Node" also called a "Terminal Node." In my example up above, getting soup would represent a terminal/leaf node. Nodes that do not split are Leaf/Terminal nodes.

Decision Branches:

The branches extending from a Decision Node are called "Decision Branches" and represent one of the possible alternative paths we can take. It's vital to understand that each path must be mutually exclusive. In other words, if I take one path, I can't take the other. Additionally, they are collectively exhaustive; in other words, I've provided all possible paths.

Splitting:

The process of dividing a node into two or more sub-nodes, for example, we split on net-worth.

Parent and Child Node:
A node, which is divided into sub-nodes, is called the parent node of sub-nodes, whereas sub-nodes are the child of the parent node. In our example up above, a parent node would be "determining net worth," and our child would be the balance of my net worth.

Understanding Random Forests: How Decision Trees Work

Decision Trees work by first “Splitting” the data into subsets. The splits are formed on a feature of the data set.

How Decision Trees Work: How is a Split Determined?

The goal is to split/partition the data until each node is homogenous in data and has as little "impurity" as possible. We will cover what impurity is later in the document.

How Decision Trees Work: How is Impurity Calculated?

There are multiple functions you can use to calculate impurity, but one of the most commonly used ones is the Gini Index. However, there are also other ways to calculate impurity to name a few we have Bayes Error and the Cross-Entropy Function. In our model, we will be using the Gini Index.

How Decision Trees Work: Impurity & Calculating the Gini Index

Up above, I mentioned that decision trees work by splitting the data into subsets and that they'll keep splitting the data until the data is homogenous (the same) and has as little impurity as possible. Well, what exactly is impurity?

Impurity refers to the probability of a variable being wrongly classified when randomly chosen. Again, just to reiterate, impurity refers to the **PROBABILITY of a VARIABLE BEING WRONGLY CLASSIFIED**. To better understand this concept, let's start with a **PURE** dataset.

If all the data belongs to a single class, we would call it pure. For example, imagine I have a dataset that contains the breed labels of different dogs. **If all the labels are the SAME BREED, then it's considered pure**. Why you may ask? There isn't a chance to classify it to any other class if there all Labrador retrievers than that's all there is. There isn't a chance to possibly be labeled as a Pug!

The Gini index, or sometimes called Gini impurity, measures the degree of probability of a variable being wrongly classified when it is randomly chosen.

The degree of the Gini index varies between 0 and 1, where 0 denotes that all elements belong to a particular class or if there exists only one class, and 1 indicates that the elements are randomly distributed across various classes. A Gini Index of 0.5 denotes equally distributed elements into some classes.

Here is a simple table for you to remember the values:

Value	Description
0.0	A pure data set. All values belong to a specific class or 1 class.
0.5	All values are EQUALLY distributed among classes.
1.0	All values are RANDOMLY distributed among classes.

The formula for the Gini Index is as follows:

$$Gini = 1 - \sum_{i=1}^{n} (p_i)^2$$

Where pi is the probability of an object being classified to a particular class. I do want to cover the concept of Entropy at this point. Entropy is just another method for determining the split in a decision tree. However, it is calculated slightly differently. Here is the formula:

$$Entropy = -\sum_{i=1}^{n} p_i log_2(p_j)$$

Both methods use the idea of Information gain when it comes to determining the optimal value. Information Gain in this context refers to the difference in information before splitting the parent and information after splitting the child. It should be noted that different impurity measures (Gini index and entropy) usually yield similar results.

How Decision Trees Work: Splitting in Detail

We know that decision trees work by splitting the data so that the Gini Index is minimized, remember lower is better in this case. The Gini Index serves as the metric we use to determine the split, so how is it used? Imagine we have a dataset, and I choose a random spot to split the data.

We now have two impurity measures, one for the left side and one for the right side. We repeat this process many times of splitting the data and then pick the best (lowest impurity) one.

How Decision Trees Work: Pruning

Great, we have a decision tree, and we've determined the number splits, and we've built our decision tree. The next step is the "Pruning" process. Pruning involves reducing the size of the Decision Tree by converting individual branches into leaf nodes and removing leaf nodes.

You may be asking, "Why would we want a smaller decision tree?". The reason is simple. Decision Trees have an easy time overfitting the data. The larger the tree, the more likely this is to happen. Generally, a simpler decision tree avoids over-fitting the data. Additionally, we may find that pruned decision trees provide relatively the same level of accuracy. That means we almost get the entire picture for 70% of the cost.

Understanding Random Forest: Advantages and Disadvantages of Decision Trees

Here are some disadvantages and advantages of decision trees:

Advantages:

- Intuitive in nature.
- Have value even with little hard data.
- Help determine worst, best, and expected values for different scenarios.
- Use a white-box model if a model provides a given result.
- It can be combined with other decision techniques.
- The number of hyper-parameters to be tuned is almost null.

Disadvantages:

- Instability: Even small changes to the input data can have dramatic changes to the overall structure of the decision tree.
- They are often relatively inaccurate. Many other predictors perform better with similar data.
- For data including categorical variables with different numbers of levels, information gain in decision trees is biased in favor of those attributes with more levels.
- Calculations can get very complex, particularly if many values are uncertain and/or if many outcomes are linked.

Understanding Random Forest: Bootstrap Aggregation

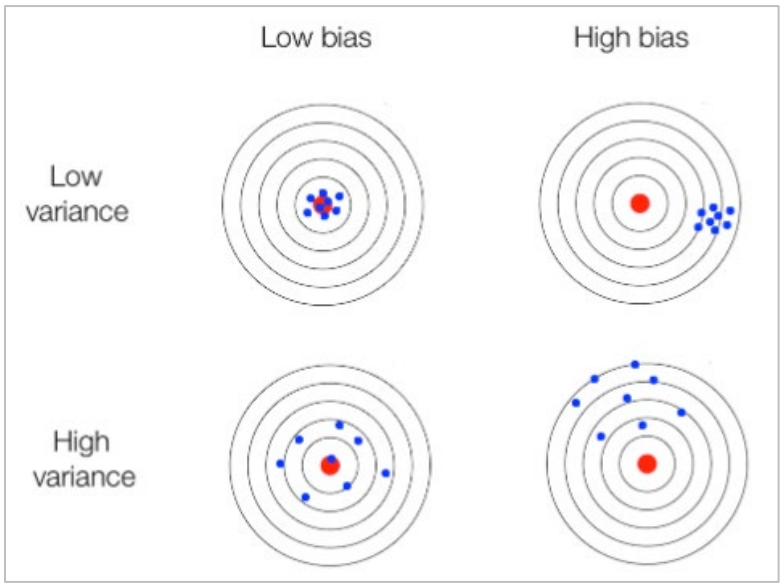
One of the challenges with decision trees is it's hard to model a complex problem with a single decision tree. What was discovered is that we can use multiple decision trees to help make more accurate predictions with models that are complex in nature.

The idea of using the results of many models, aggregating them, and using the output to make a prediction refers to the concept of ensemble learning. An Ensemble Learning model is a model in which decisions are used from **MULTIPLE MODELS** to improve the overall performance of the model. The old idea that two minds are better than 1, perfectly summarizes ensemble learning. We use the results of multiple models to get a better idea of what the true answer is.

Bootstrap Aggregation, also called Bagging, is a machine learning ensemble algorithm used to help improve the stability and accuracy of predictions. Bootstrap Aggregation seems to work well for high-variance, low-bias procedures, such as trees. How Bootstrap Aggregation works depends on the type of problem we are modeling.
For regression, we simply fit the same regression tree many times to "bootstrap-sampled" versions of the training data and average the result. For classification, a committee of trees each cast a vote for the predicted class. While Bootstrap Aggregation reduces the variance of the base learner, it has a limited effect on the bias.

Why does Bootstrap Aggregation reduce the variance? Well, consider we have a single classifier that is unstable; in other words, the variance is high. With Bootstrap Aggregation, the process of repeating a process multiple times and averaging those results is what we can think of as an approximation of the true average.

Why does Bootstrap Aggregation have little impact on the bias? Imagine we have many models that have either low bias, and we aggregate their results. All we are doing is compressing that variance closer together. What that means is if we have a low bias, we still maintain it. Additionally, if we have a high bias, we still retain it. Here is a visual to help you understand:



In the lower right-hand corner, we have high variance and high bias. Say we perform bootstrap aggregations, and we take the results of multiple high bias model all we are doing is compressing the variance, but we are still off target (high bias). Hence, we maintain the bias but only reduce the variance.

Understanding Random Forest: Bootstrap Aggregation Flaws

- Bagging algorithm still uses Decision Trees, which we know have issues.
- Decision Trees uses Gini-Index, a greedy algorithm to find the best split. Greedy meaning, the very best split point is chosen each time
- We end up with trees that are structurally similar to each other. The trees are highly correlated among the predictions. Random Forest addresses this problem

Understanding Random Forest: Why Use it?

We saw Decision Trees have some disadvantages, so we used Bootstrap Aggregation to help improve those weaknesses, but we also saw it has some disadvantages as well. Is there a way to improve Bootstrap Aggregation? We could use Random Forest!

What exactly does Random Forest do to fix these issues? Well, it uses Bootstrap Aggregation but it what it does instead is to create trees that have no correlation or weak correlation with each other. You're probably confused at this point because it doesn't sound like anything really changed.

We are still using Bootstrap aggregation. However, the key here is what we are aggregating before we used **HIGHLY CORRELATED** trees, with Random Forest we devise a way to create **WEAKLY CORRELATED OR UNCORRELATED** trees. What Random Forest tries to do is avoid over-fitting of the data and improve the stability and accuracy of predictions.

Understanding Random Forest: How it works?

With Random Forest we are still doing a lot of the same things, we are still building our forest (creating many trees like we did in bootstrap aggregation) and we are still taking a vote of the trees, at least for classification problems. The key here is how we build the tree. We build a tree doing the following steps:

- Take a random sample of size N with replacement from the data.
- Take a random sample without replacement of the predictors.
- Construct the first CART partition of the data.
- Repeat Step 2 for each subsequent split until the tree is as large as desired. Do not prune.
- Repeat Steps 1–4 many times (e.g., 500).

It's important to understand that step 2 & step 1 is how Random Forest deals with the highly correlated trees generated by bagging. That's the magic that goes on behind the scenes that improves the model! For step 5, I want to be a little clearer on that. It's important to realize that there is no magic number for "many times" it depends on your data. The best advice is to evaluate the model and use cross-validation.

When the training set for the current tree is drawn by sampling with replacement, about one-third of the cases are left out of the sample. This OOB (out-of-bag) data is used to get a running unbiased estimate of the classification error as trees are added to the forest. It is also used to get estimates of variable importance.

After each tree is built, all the data is run down the tree, and proximities are computed for each pair of cases. If two cases occupy the same terminal node, their proximity is increased by one. At the end of the run, the proximities are normalized by dividing by the number of trees.

Proximities are used in replacing missing data, locating outliers, and producing illuminating low-dimensional views of the data.

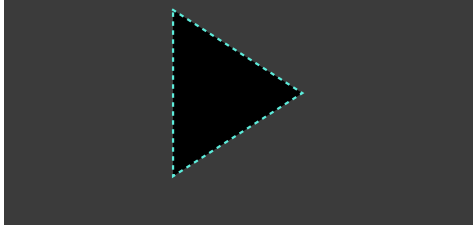
Understanding Random Forest: Closing Notes

With that, we have reached the end of our overview of Decision Trees, Bootstrap Aggregation, and Random Forest. Now with a high-level overview, we can begin the process of building our model.

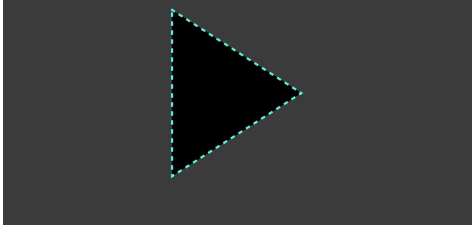
YouTube Series Links – Random Forests

This project has a dedicated YouTube series that goes along with it. If you'd like to watch those videos along with reading the documentation, feel free to visit the links below. They are upper text in each box.

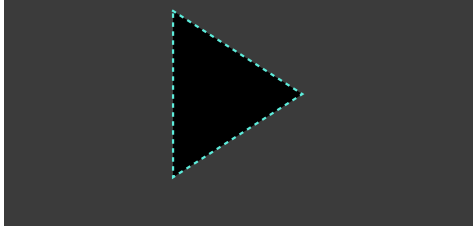
Video 1



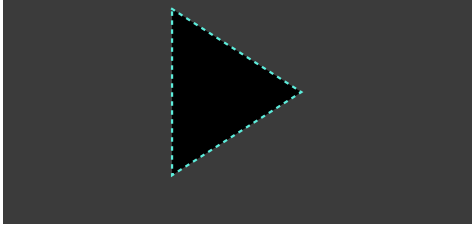
Video 2



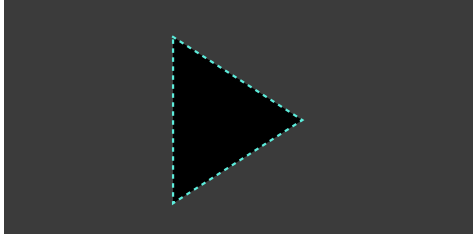
Video 3



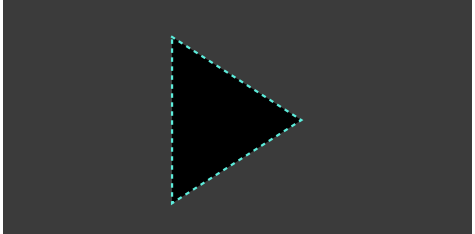
Video 4



Video 5



Video 6



Building a Random Forest Model in Python:

With a high-level overview of the Random forest, we can begin the process of writing the code to build our Random Forest model. Now, to give some structure to this project, I want to break out the main parts of the process:

1. Data Collection & Preprocessing
    - We will collect and clean our data before beginning the process of calculating the indicators used in the paper.
2. Indicator Calculation
    - Calculate all the indicators used in the paper.
3. Building the Model
    - Build the model using **sklearn** Learn.
4. Model Evaluation
    - Explore how to evaluate the model to see how it performed.
5. Reflection and Closing Notes
    - Exploring possible issues that could impact the model.

Libraries:

We will be using different libraries in this project that will help do a range of tasks from loading and cleaning to evaluating the model's performance. For most of the transformation tasks, we will be using the `pandas` library. All of the modeling will be down with `sklearn`, and most of the evaluation and graphing will be using different objects from the `sklearn` library.

For certain data collection tasks, I'll be using the TD Library I developed a while back for a different series. To use that library, I'll need to provide some different credentials related to my TD Account. All of that information will be stored in a separate python file called `config`. Here is the code to import the python libraries:

```
import os
import sys
import requests

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.metrics import plot_roc_curve, accuracy_score, classification_report

from config import ACCOUNT_NUMBER, ACCOUNT_PASSWORD, CONSUMER_ID, REDIRECT_URI
```

SECTION 1

DATA COLLECTION & PRECPROCESSING

Data Collection & Preprocessing: Loading the TD API Library

For those of you who watched my series on building a Python API Client library for TD Ameritrade, here is a chance to put that library to use. I want to collect some price data on a few stocks. Let's use the TD Library to do that. First, because it's not an installed library on my system, I'll need to add the location of the library to my system path. Define the path to the folder, use the `sys.path.insert` method to insert that path and then load the library as usual.

```
# Define path to the TD API folder.
path_to_td_folder = r"PATH_TO_TD_LIBRARY"

# I'll be needing my TD API Client to get some prices, so I'll need to point a path to it.
sys.path.insert(0, path_to_td_folder)

# import the TDClient, may get an Intellisense error but disregard it.
from td.client import TDClient
```

Data Preprocessing: Grabbing Historical Price Data

I have a few stocks I would like to grab historical daily prices for. What I'll do is create a function that will log me into a new TD API Session, loop through a list of ticker symbols, and grab some historical prices using the `“Get_Prices”` endpoint.

After I have my prices, I'll parse the JSON string that's returned so we can get the candle data, store it in. I'll take the final value and store it in a Pandas Data Frame and then save that data frame to a CSV file so we can manually explore the data if need be. The `“get_price_data”` function is provided on the next.

GET PRICE DATA FUNCTION – PART 1

```
def grab_price_data():

    # Create a new session
    TDSession = TDClient(account_number = ACCOUNT_NUMBER,
                         account_password = ACCOUNT_PASSWORD,
                         consumer_id = CONSUMER_ID,
                         redirect_uri = REDIRECT_URI)

    # Login to the session
    TDSession.login()

    # Let's define some tickers we want to get the data for five tickers.

    '''
        HD   - Home Depot
        JPM  - JPMorgan Chase & Co.
        IBM  - International Business Machines Corporation
        ARWR - Arrowhead Pharmaceuticals, Inc.
        COST - Costco Wholesale Corporation
    '''

    # Define the list of tickers
    tickers_list = ['JPM', 'COST', 'IBM', 'HD', 'ARWR']

    # I need to store multiple result sets.
    full_price_history = []

    '''
        REST DOWN BELOW!
    '''
```

GET PRICE DATA FUNCTION – PART 2

```
for ticker in tickers_list:

    # Grab the daily price history for 1 year
    price_history = TDSession.get_price_history(symbol = ticker,
                                                periodType = 'year',
                                                period = 2,
                                                frequency = 1,
                                                frequencyType = 'daily',
                                                needExtendedHoursData = False)

    # grab just the candles, and add them to the list.
    for candle in price_history['candles']:
        candle['symbol'] = price_history['symbol']
        full_price_history.append(candle)

# dump the data to a CSV file, don't have an index column
price_data = pd.DataFrame(full_price_history).to_csv('price_data.csv',
                                                     index_label = False)
```

Data Preprocessing: Load the Data

This portion is a little unnecessary because you take the function up above, and it will return the `price_data` data frame and use that. However, if you don't want to go through the process of pulling the data again. What we can do is load the CSV file we used previously.

In this portion, I check if the price_data.csv file exists in the directory, and if it does, I load it. Otherwise, I call the `get_price_data` and grab the data again. Finally, I print the `price_data` data frame to verify the data was loaded. **YOU WILL NEED TO RUN THE FUNCTION EVERY DAY TO GET THE LATEST PRICE DATA.**

```
if os.path.exists('price_data.csv'):
    # Load the data
    price_data = pd.read_csv('price_data.csv')

else:
    # Grab the data and store it.
    grab_price_data()

    # Load the data
    price_data = pd.read_csv('price_data.csv')

# Display the head before moving on.
price_data.head()
```

	close	datetime	high	low	open	symbol	volume
0	105.51	1513144800000	107.11	105.48	106.70	JPM	15020293
1	104.66	1513231200000	106.45	104.64	106.12	JPM	13034232
2	106.14	1513317600000	106.52	105.20	105.79	JPM	29350598
3	106.96	1513576800000	107.63	106.48	107.19	JPM	12713148
4	106.51	1513663200000	107.49	106.44	107.35	JPM	12149937

With the data now loaded, we can transform so we can calculate some of our trading indicators. The first thing we need to do is sort the data because we have multiple ticker symbols inside of our data frame. Take the data frame and call the `sort_values` method and specify the columns you wish to sort by using the by argument. In our case, we will be using a list of column names to sort by.

The first sort is by the symbol column, and the second sort is by the datetime column. Once we've sorted the data, we need to calculate the change in price from one period to the next. To do this, we will use the `diff()` method. Grab the close column and call the `diff()` method. The `diff()` method will calculate the difference from one row to the next.

```
# I Just need the Close
price_data = price_data[['symbol','datetime','close','high','low','open','volume']]

'''
    First, for average investors, the return of an asset is a complete and scale-free
    summary of the investment opportunity. Second, return series are easier to
    handle than prices series as they have more attractive statistical properties
'''

# sort the values by symbol and then date
price_data.sort_values(by = ['symbol','datetime'], inplace = True)

# calculate the change in price
price_data['change_in_price'] = price_data['close'].diff()
```

Data Preprocessing: Ticker Symbol

Okay, so we've created our `change_in_price` column, but we need to do an extra step. Technically, each row where the ticker symbol changes are incorrect because it's using the price from a different ticker. That means we need to have the first row of each ticker symbol be `NaN` for the `change_in_price` column. To do this, we need to break out into steps.

Step 1: Identify the rows where the ticker symbol changes. If we use the `shift()` method and shift every row down by one, the rows where unshifted column **DOES NOT EQUAL** the shifted column is where the ticker changed. We will store these values in a variable called mask.

Step 2: Change those rows to `NaN` values. We can use the `numpy.where()` method to test our series. The test is simple, wherever the mask variable equals `True`, in other words, wherever the ticker symbol is different, set the `change_in_price` column to `np.nan`.

After doing that, we can filter those `NaN` values; we should only have 5. The code is provided below:

```
# identify rows where the symbol changes
mask = price_data['symbol'] != price_data['symbol'].shift(1)

# For those rows, let's make the value null
price_data['change_in_price'] = np.where(mask == True, np.nan, price_data['change_in_price'])

# print the rows that have a null value, should only be 5
price_data[price_data.isna().any(axis = 1)]
```

	symbol	datetime	low	open	volume	change_in_price
0	ARWR	1513144800000	3.01	3.30	2037395	NaN
1	COST	1513144800000	187.80	188.53	3062856	NaN
2	HD	1513144800000	182.00	182.01	5177363	NaN
3	IBM	1513144800000	153.89	156.60	5661618	NaN
4	JPM	1513144800000	105.48	106.70	15020293	NaN

Data Preprocessing: Smoothing the Data (OPTIONAL)

The following section is optional; in the example below, I will not be doing any smoothing of the data. This is simply to give you the code necessary to reproduce certain results in the paper.

In the paper, they test the model using different windows. For example, they make predictions 30 days out, 60 days out, and 90 days out. To make this type of prediction, we have to transform the data so that when we pass it through the model, it will be able to make those predictions that far out. The transformation they use is a smoothing factor that is defined by the following:

Formula:

$$S_0 = Y_0$$

$$for\ t > 0, S_t = \alpha * Y_t + (1 - \alpha) * S_t - 1$$

Where α is the smoothing factor and 0 < α < 1. Larger values of α reduce the level of smoothing. When α = 1, the smoothed statistic becomes equal to the actual observation.

The goal of smoothing it remove the randomness and noise from our price data. In other words, we don't get spiky up and down graph but instead a smoother one. Additionally, this will help the model to identify long-term trends more easily. Here is how to calculate the smoothed version of the prices using pandas:

```
# define the number of days out you want to predict
days_out = 30

# Group by symbol, then apply the rolling function and grab the Min and Max.
price_data_smoothed = price_data.groupby(['symbol'])[['close','low','high','open','volume']]\
    .transform(lambda x: x.ewm(span = days_out).mean())

# Join the smoothed columns with the symbol and datetime column from the old data frame.
smoothed_df = pd.concat([price_data[['sym-
bol','datetime']], price_data_smoothed], axis=1, sort=False)
```

Data Preprocessing: Signal Flag (OPTIONAL)

If you chose to do the smoothing process, then we need to add an additional column to our data frame. This will serve as a diff column from the original data frame. However, in this case, we don't want one consecutive day to the next we want the number of days we want to predict out.

What we will do is take the window we used up above to calculate our smoothed statistic and use it to calculate our signal flag. We will be using the `numpy.sign()` method which will return a **1.0 if positive**, -**1.0 if negative and 0.0 if no change**.

```
# define the number of days out you want to predict
days_out = 30

# create a new column that will house the flag, and for each group
# calculate the diff compared to 30 days ago. Then use Numpy to define the sign.
smoothed_df['Signal_Flag'] = smoothed_df.groupby('symbol')['close'].\
    transform(lambda x : np.sign(x.diff(days_out)))

# print the first 50 rows
smoothed_df.head(50)
```

SECTION 2

INDICATOR CALCULATION

Indicator Calculation: Relative Strength Index (RSI)

Definition From Paper:

RSI is a popular momentum indicator that determines whether the stock is overbought or oversold. A stock is said to be overbought when the demand unjustifiably pushes the price upwards. This condition is generally interpreted as a sign that the stock is overvalued, and the price is likely to go down. A stock is said to be oversold when the price goes down sharply to a level below its true value. This is a result caused due to panic selling. **RSI ranges from 0 to 100**, and generally, when RSI is above 70, it may indicate that the stock is overbought and when RSI is below 30, it may indicate the stock is oversold.

Formula:

$$RSI = 100 - \frac{100}{1 + RS}$$

Code:

From this point forward, a lot of the calculations will be mostly the same but only differ on the type of calculation we do. Each indicator is calculated using the same few steps:

1. Copy the desired columns and store them in new variables.
2. Group the columns by the symbol, select the column we wish to perform the transformation on, and use the transform method along with a lambda function to calculate the indicator.
3. Store the values in the main data frame.

Now there might be a few extra steps in between, but the general idea is the same across each indicator. Now for the RSI indicator, I need to identify the up days and down days. Well, lucky for us, we already have the `change_in_price` column, so we can use a condition that will set the value to 0 if the price went up for down days and vice versa for up days.

After that, I need to make sure I have the absolute values for down days, or else the calculation won't be correct, so I modify that column and then calculate the EMA of both the Up and Down columns. Finally, I calculate the Relative strength metric and pass that through to the RSI calculation.

RSI Calculation - Code

```
# Calculate the 14 day RSI
n = 14

# First make a copy of the data frame twice
up_df, down_df = price_data[['symbol','change_in_price']].copy(), \
    price_data[['symbol','change_in_price']].copy()

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

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

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

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

ewma_down = down_df.groupby('symbol')['change_in_price']\
    .transform(lambda x: x.ewm(span = n).mean())

# Calculate the Relative Strength
relative_strength = ewma_up / ewma_down

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

# Add the info to the data frame.
price_data['down_days'] = down_df['change_in_price']
price_data['up_days'] = up_df['change_in_price']
price_data['RSI'] = relative_strength_index

# Display the head.
price_data.head(30)
```

RSI Calculation – Output

	symbol	datetime	…	change_in_price	down_days	up_days	RSI
0	ARWR	1513144800000	…	NaN	NaN	NaN	NaN
1	ARWR	1513231200000	…	-0.25	0.25	0.00	0.000000
2	ARWR	1513317600000	…	-0.01	0.01	0.00	0.000000
3	ARWR	1513576800000	…	0.09	0.00	0.09	31.419705
4	ARWR	1513663200000	…	0.03	0.00	0.03	38.813758

Indicator Calculation: Stochastic Oscillator

Definition From Paper:

Stochastic Oscillator follows the speed or the momentum of the price. As a rule, momentum changes before the price changes. It measures the level of the closing price relative to the low-high range over a period of time.

Formula:

$$K = 100 * \frac{(C - L_{14})}{(H_{14} - L_{14})}$$

where,
$C = Current\ Closing\ Price$
$L_{14} = Lowest\ Low\ over\ the\ past\ 14\ days.$
$H_{14} = Highest\ High\ over\ the\ past\ 14\ days.$

Code:

The strategy here is pretty much the same; the only difference is the columns we are copying and the lambda function we are applying. For the RSI, we applied an EMA function, but for the Stochastic Oscillator, we use the rolling function.

With this function, we specify our window, which in this case is 14 periods, and then specify measurement we want to apply to each window. After we obtained the `max` and `min` values, we pass it through our formula and apply the results to the main data frame.

Stochastic Oscillator Calculation - Code:

```
# Calculate the Stochastic Oscillator
n = 14

# Make a copy of the high and low column.
low_14, high_14 = price_data[['symbol','low']].copy(),\
     price_data[['symbol','high']].copy()

# Group by symbol, then apply the rolling function and grab the Min and Max.
low_14 = low_14.groupby('symbol')['low'].\
    transform(lambda x: x.rolling(window = n).min())

high_14 = high_14.groupby('symbol')['high'].\
    transform(lambda x: x.rolling(window = n).max())

# Calculate the Stochastic Oscillator.
k_percent = 100 * ((price_data['close'] - low_14) / (high_14 - low_14))

# Add the info to the data frame.
price_data['low_14'] = low_14
price_data['high_14'] = high_14
price_data['k_percent'] = k_percent

# Display the head.
price_data.head(30)
```

Stochastic Oscillator Calculation – Output:

	symbol	datetime	…	low_14	high_14	k_percent
0	ARWR	1514959200000	…	3.0100	4.88	87.165775
1	ARWR	1515045600000	…	3.0500	5.13	86.538462
2	ARWR	1515132000000	…	3.0800	5.23	83.255814
3	ARWR	1515391200000	…	3.1400	5.23	67.464115
4	ARWR	1515477600000	…	3.2271	5.23	69.544161

Indicator Calculation: Williams %R

Definition From Paper:

Williams %R ranges from -100 to 0. When its value is above -20, it indicates a sell signal, and when its value is below -80, it indicates a buy signal.

Formula:

$$\%R = \frac{(H_{14} - C)}{(H_{14} - L_{14})} * -100$$

where,
$C = Current\ Closing\ Price$
$L_{14} = Lowest\ Low\ over\ the\ past\ 14\ days.$
$H_{14} = Highest\ High\ over\ the\ past\ 14\ days.$

Williams %R Calculation - Code

Identical to the Stochastic Oscillator, we change the arrangement of the formula.

```
# Calculate the Williams %R
n = 14

# Make a copy of the high and low column.
low_14, high_14 = price_data[['symbol','low']].copy(),\
    price_data[['symbol','high']].copy()

# Group by symbol, then apply the rolling function and grab the Min and Max.
low_14 = low_14.groupby('symbol')['low'].\
    transform(lambda x: x.rolling(window = n).min())

high_14 = high_14.groupby('symbol')['high'].\
    transform(lambda x: x.rolling(window = n).max())

# Calculate William %R indicator.
r_percent = ((high_14 - price_data['close']) / (high_14 - low_14)) * - 100

# Add the info to the data frame.
price_data['r_percent'] = r_percent

# Display the head.
price_data.head(30)
```

Williams %R Calculation – Output

	symbol	datetime	…	low_14	high_14	r_percent
0	ARWR	1514959200000	…	3.0100	4.88	-12.834225
1	ARWR	1515045600000	…	3.0500	5.13	-13.461538
2	ARWR	1515132000000	…	3.0800	5.23	-16.744186
3	ARWR	1515391200000	…	3.1400	5.23	-32.535885
4	ARWR	1515477600000	…	3.2271	5.23	-30.455839

Indicator Calculation: Moving Average Convergence Divergence (MACD)

Definition From Paper:

EMA stands for Exponential Moving Average. When the MACD goes below the **SingalLine**, it indicates a sell signal. When it goes above the **SignalLine,** it indicates a buy signal.

Formula:

$$MACD = EMA_{12}(C) - EMA_{26}(C)$$

$$SignalLine = EMA_{9}(MACD)$$

where,
$C = Current\ Closing\ Price$
$MACD = Moving\ Average\ Convergence\ Divergence$
$EMA_n = n\ period\ Exponential\ Moving\ Average.$

Code:

For the MACD, we will need the close column, so grab that and then apply the transform method along with the specified Lambda function. Now calculating an Exponential Moving Average in pandas is easy. First, call the `ewm` (exponential moving weight) function and then specify the span or, in other words, the number of periods to look back.

In this case, we use the definition provided by the formula and specify 26 & 12. Once we've calculated the `ema_26` and `ema_12`, we take the difference between `ema_12` & `ema_26` to get our MACD. Now that we have our MACD, we need to calculate the EMA of the MACD, so we take our MACD series and apply the

same `ewm` function too, but in this case, we specify a span of 9. Finally, we add both the `MACD` and `MACD_EMA` to the main data frame.

MACD Calculation – Code

```
# Calculate the MACD
ema_26 = price_data.groupby('symbol')['close'].transform(lambda x: x.ewm(span = 26).mean())
ema_12 = price_data.groupby('symbol')['close'].transform(lambda x: x.ewm(span = 12).mean())

macd = ema_12 - ema_26

# Calculate the EMA
ema_9_macd = macd.ewm(span = 9).mean()

# Store the data in the data frame.
price_data['MACD'] = macd
price_data['MACD_EMA'] = ema_9_macd

# Print the head.
price_data.head(30)
```

MACD Calculation – Output

	symbol	datetime	…	low_14	MADC	MACD_EMA
0	ARWR	1514959200000	…	3.0100	0.000000	0.000000
1	ARWR	1515045600000	…	3.0500	-0.005609	-0.003116
2	ARWR	1515132000000	…	3.0800	-0.007457	-0.004895
3	ARWR	1515391200000	…	3.1400	-0.004865	-0.004885
4	ARWR	1515477600000	…	3.2271	-0.002161	-0.004075

Indicator Calculation: Price Rate Of Change

Definition From Paper:

It measures the most recent change in price with respect to the price in n days ago.

Formula:

$$PROC_t = \frac{C_t - C_{t-n}}{C_{t-n}}$$

where,
$C_t = Closing\ Price\ at\ time\ t$
$PROC_t = Price\ Rate\ of\ Change\ at\ time\ t$

Code:

The Price Rate of Change is another easy indicator to calculate in pandas because we can leverage a built-in function. In this case, we will use the `pct_change` function and apply it to our all too familiar symbol groups. For the `pct_change` function, we have an argument called periods which specifies how far we need to look back when calculating the rate of change.

In the case of the paper, `n` seemed to change based on the trading window they wanted to predict out. For example, if they wanted to predict the direction 30 days out, then `n` would be 30. Initially, when I wrote the code I didn't see exactly where they defined `n` so I chose 9 as my `n` because it seemed like the standard value. If you want to follow the paper exactly then you need to adjust `n` to match the prediction window.

```
# Calculate the Price Rate of Change
n = 9

# Calculate the Rate of Change in the Price, and store it in the Data Frame.
price_data['Price_Rate_Of_Change'] = price_data.groupby('symbol')['close'].\
    transform(lambda x: x.pct_change(periods = n))

# Print the first 30 rows
price_data.head(30)
```

	symbol	datetime	…	close	MACD_EMA	PROC
0	ARWR	1514959200000	…	3.84	0.025380	0.129412
1	ARWR	1515045600000	…	3.97	0.035711	0.260317
2	ARWR	1515132000000	…	3.68	0.042464	0.171975
3	ARWR	1515391200000	…	3.72	0.047126	0.151703
4	ARWR	1515477600000	……	4.64	0.061321	0.423313

Indicator Calculation: On Balance Volume

Definition From Paper:

On balance volume (OBV) utilizes changes in volume to estimate changes in stock prices. This technical indicator is used to determine buying and selling trends of a stock, by considering the cumulative volume: it cumulatively adds the volumes on days when the prices group, and subtracts the volume on the days when prices go down, compared to the prices of the previous day.

Formula:

$$OBV(t) = \begin{cases} OBV(t-1) + Vol(t) \; if \; C(t) > C(t-1) \\ OBV(t-1) - Vol(t) \; if \; C(t) < C(t-1) \\ OBV(t-1) \; if \; C(t) > C(t-1) \end{cases}$$

where,
$OBV(t) = On\ Balance\ Volume\ at\ time\ t$
$Vol(t) = Trading\ Volume\ at\ time\ t$
$C(t) = Closing\ Price\ at\ time\ t$

Code:

This portion is a little more complicated than the previous ones. However, the idea is still the same. I'm going to be working with groups, but in this case, I'll be using the apply method to apply a custom function I built to calculate the On Balance Volume.

The function simply calculates the `diff` for the closing price and uses a for loop to loop through each row in the volume column. If the change in price was greater than 0, we would add the volume. If it's less than 0 we subtract the volume, and if it's 0 then we leave it alone.

When I return the values, I need to make sure it's a `pandas.Series` object with an index. Once I have the `pandas.Series` I just add it to the data frame like in previous examples.

On Balance Volume Calculation - Code

```
def obv(group):

    # Grab the volume and close column.
    volume = group['volume']
    change = group['close'].diff()

    # intialize the previous OBV
    prev_obv = 0
    obv_values = []

    # 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_values.append(current_obv)

    # Return a panda series.
    return pd.Series(obv_values, index = group.index)


# apply the function to each group
obv_groups = price_data.groupby('symbol').apply(obv)

# add to the data frame, but drop the old index, before adding it.
price_data['On Balance Volume'] = obv_groups.reset_index(level=0, drop=True)

# display the data frame.
price_data.head(30)
```

On Balance Volume Calculation - Output

	symbol	datetime	…	MACD_EMA	PROC	On Balance Volume
0	ARWR	1513144800000	…	0.000000	NaN	0
1	ARWR	1513231200000	…	-0.003116	NaN	-1702102
2	ARWR	1513317600000	…	-0.004895	NaN	-3007748
3	ARWR	1513576800000	…	-0.004885	NaN	-1753443
4	ARWR	1513663200000	…	-0.004075	NaN	-624853

Indicator Calculation: Final Notes

Initially, when I read through the paper I didn't realize why they chose to use all momentum indicators. If you notice up above, almost all of the indicators used are used to find the momentum in price. However, after pondering it for a while it makes sense. If your goal is to just find direction of price, momentum is helpful tool in that process. Some believe that momentum **precedes price which would make it a leading indicator**. Additionally momentum indicators can be useful because they can give **us a better idea of future price movements** because:

- They show whether a trend is strengthening or weakening.
- They show if an asset is overbought or oversold relative to the price range over a given prior period.

SECTION 3

BUILDING THE MODEL

Building the Model: Creating the Prediction Column

Now that we have our technical indicators calculated and our price data cleaned up, we are almost ready to build our model. However, we are missing one critical piece of information that is crucial to the model, the column we wish to predict. Now at this point, our data frame doesn't have that column, but we will create it before we feed the data into the model.

However, before we create it, I want to take some time and understand the exact problem we are trying to solve. Our goal is to predict whether the next day is either a `down_day` or an `up_day`. Based on this knowledge, we are solving a classification problem. If you don't remember, there are two categories of problems in machine learning, classification, and regression.

With classification problems, we try to predict which group new values belong to. For example, based on store sales, are they outperforming store or an underperforming store. Classification problems are problems that have discrete groups. With regression problems, we are trying to predict non-discrete values and, for example, trying to forecast future sales based on previous sales.

In our case, we have a classification problem because we have two discrete groups, `up_days` and `down_days`, and our goal is to take new values (new prices) and classify them into these two groups based on their values.

To create our prediction column, we will group our data frame by each symbol. After we've created our groups, we need to select the close column as this contains the price we need to determine if the stock closed up or down for any given day. Now, we can use a similar logic we used to calculate the price change. However, in this case, we only need to know if the price is higher or lower compared to the previous day.

Take your groups, use the transform method to apply a lambda function to your groups. The lambda function will use the `diff()` function to compare the current price to the previous price. We then wrap the results of that function in the `numpy.sign()` function. That function will return 1.0 for negative values (down days), 1.0 for positive values, and 0.0 for no change (flat days).

```
# Create a column we wish to predict
'''
    In this case, let's create an output column that will be 1 if the closing price
    at time 't' is greater than 't-1' and 0 otherwise. In other words, if the today's
    closing price is greater than yesterday's closing price it would be 1.
'''

# Group by the `Symbol` column, then grab the `Close` column.
close_groups = price_data.groupby('symbol')['close']

# Apply the lambda function which will return -1.0 for down, 1.0 for up and 0.0 for no change.
close_groups = close_groups.transform(lambda x : np.sign(x.diff()))

# add the data to the main dataframe.
price_data['Prediction'] = close_groups

# for simplicity in later sections I'm going to make a change to our prediction column.
# To keep this as a binary classifier I'll change flat days and consider them up days.
price_data.loc[price_data['Prediction'] == 0.0] = 1.0

# print the head
price_data.head(50)

# OPTIONAL CODE: Dump the data frame to a CSV file to examine the data yourself.
# price_data.to_csv('final_metrics.csv')
```

	symbol	datetime	…	volume	change_in_price	Prediction
0	ARWR	1513144800000	…	2037395	NaN	NaN
1	ARWR	1513231200000	…	1702102	-0.25	-1.0
2	ARWR	1513317600000	…	1305646	-0.01	-1.0
3	ARWR	1513576800000	…	1254305	0.09	1.0
4	ARWR	1513663200000	…	1128590	0.03	1.0

Building the Model: Removing `NaN` Values

The random forest can't accept Nan values, so we will need to remove them before feeding the data in. The code below prints the number of rows before dropping the `NaN` values, use the `dropna` method to remove any rows `NaN` values, and then displays the number of rows after dropping the `NaN` values.

```
# We need to remove all rows that have an NaN value.
print('Before NaN Drop we have {} rows and {} columns'.\
    format(price_data.shape[0], price_data.shape[1]))

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

# Display how much we have left now.
print('After NaN Drop we have {} rows and {} columns'.\
    format(price_data.shape[0], price_data.shape[1]))

# Print the head.
price_data.head()
```

Building the Model: Splitting the Data

If you remember back to our series on regression analysis, we have split our data into a training set and testing set. For Random Forest, we need to do the same, so we need to identify our input columns which are the following:

- RSI
- Stochastic Oscillator
- William %R
- Price Rate of Change
- MACD
- On Balance Volume

Those columns will serve as our X, and our Y column will be the Prediction column, the column that specifies whether the stock closed up or down compared to the previous day. Once we've selected our columns, we need to split the data into a training and test set.

`sklearn` makes this easy by providing the `train_test_split` object, which will take our `X_Cols` and `Y_Cols` and split them based on the size we input. In our case, let's have the `test_size` be 20%. For reproducibility, the `train_test_split` object provides the `random_state` argument that will split the data along the same dimensions every time.

After we've split the data, we can create our `RandomForestClassifier` model. Once we've created it, we can fit the training data to the model using the fit method. Finally, with our "trained" model, we can make predictions. Take the `X_test` data set and use it to make predictions.

```
# Grab our X & Y Columns.
X_Cols = price_data[['RSI','k_percent','r_percent','Price_Rate_Of_Change','MACD','On Balance Volume']]
Y_Cols = price_data['Prediction']

# Split X and y into X_
X_train, X_test, y_train, y_test = train_test_split(X_Cols, Y_Cols, random_state = 0)

# Create a Random Forest Classifier
rand_frst_clf = RandomForestClassifier(n_estimators = 100, \
    oob_score = True, criterion = "gini", random_state = 0)

# Fit the data to the model
rand_frst_clf.fit(X_train, y_train)

# Make predictions
y_pred = rand_frst_clf.predict(X_test)
```

SECTION 4

EVALUATING THE MODEL

Model Evaluation: Accuracy

We've built our model, so let's see how accurate it is. `sklearn` learn, again, makes the process of evaluating our model very easy by providing a bunch of built-in metrics that we can call. One of those metrics is the `accuracy_score`.

The `accuracy_score` function computes the accuracy, either the fraction (default) or the count (`normalize=False`) of correct predictions. Accuracy is defined as the number of accurate predictions the model made on the test set. Imagine we had three **TRUE** values [1, 2, 3], and our model predicted the following values [1, 2, 4] we would say the accuracy of our model is 66%.

Now I can almost guarantee that your accuracy will be different than mine because if you don't run this model using the same data set as I did, then it could be higher or lower. In my case, the model accuracy was around **71.1%**, which is pretty high.

```
# Print the Accuracy of our Model.
print('Correct Prediction (%): ', accuracy_score(y_test, \
    rand_frst_clf.predict(X_test), normalize = True) * 100.0)
```

Model Evaluation: Classification Report

To get a more detailed overview of how the model performed, we can build a classification report that will compute the F1_Score, the Precision, the Recall, and the Support. Now, I'm assuming you don't know what these metrics are, so let's take some time to go over them.

Accuracy:

Accuracy measures the portion of all testing samples classified correctly and is defined as the following:

$$Accuracy = \frac{tp + tn}{(tp + tn) - (fp - fn)}$$

$where,$
$tp = True\ Positive$
$tn = True\ Negative$
$fp = False\ Positive$
$fn = False\ Negative$

Recall:

Recall (also known as sensitivity) measures the ability of a classifier to correctly identify positive labels and is defined as the following:

$$Recall = \frac{tp}{(tp + fn)}$$

$where,$
$tp = True\ Positive$
$fn = False\ Negative$

The recall is intuitively the ability of the classifier to find all the positive samples. The best value is 1, and the worst value is 0.

Specificity:

Specificity measures the classifier's ability to correctly identify negative labels and is defined as the following:

$$Specificity = \frac{tn}{(tn + fp)}$$

$where,$
$tn = True\ Negative$
$fp = False\ Positive$

Precision:

Precision measures the proportion of all correctly identified samples in a population of samples which are classified as positive labels and is defined as the following:

$$Percision = \frac{tp}{(tp + fp)}$$

$where,$
$tp = True\ Positive$
$fp = False\ Positive$

The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The best value is 1, and the worst value is 0.

Interpreting the Classification Report:

Now the fun part, interpretation. When it comes to evaluating the model, there we generally look at the accuracy. If our accuracy is high, it means our model is correctly classifying items. In some cases, we will have models that may have low precision or high recall.

It's difficult to compare two models with low precision and high recall or vice versa. To make results comparable, we use a metric called the F-Score. The F-score helps to measure Recall and Precision at the same time. It uses Harmonic Mean in place of Arithmetic Mean by punishing the extreme values more.

```
# Define the traget names
target_names = ['Down Day', 'Up Day']

# Build a classifcation report
report = classification_report(y_true = y_test, \
    y_pred = y_pred, target_names = target_names, output_dict = True)

# Add it to a data frame, transpose it for readability.
report_df = pd.DataFrame(report).transpose()
```

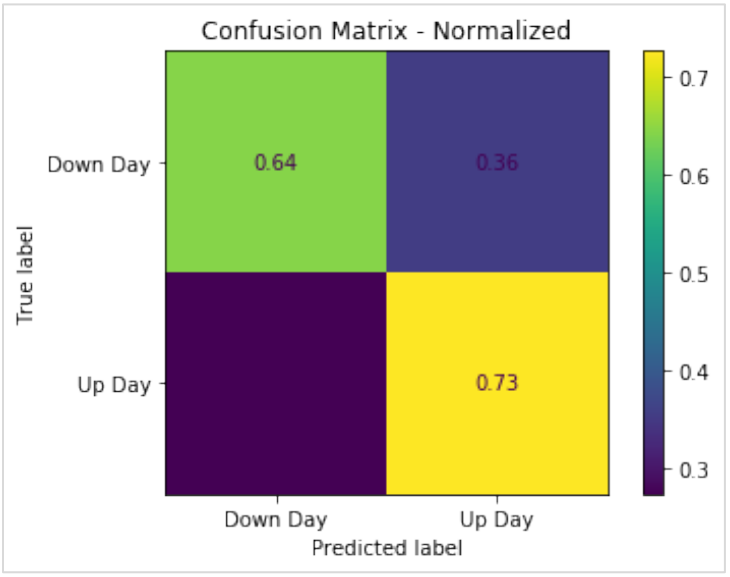
	F1-Score	Precision	Recall	Support
Down Day	0.666667	0.690583	0.644351	239.000000
Up Day	0.703846	0.682836	0.726190	252.000000
accuracy	0.686354	0.686354	0.686354	0.686354
macro avg	0.685256	0.686709	0.685271	491.000000
weighted avg	0.685749	0.686607	0.686354	491.000000

Model Evaluation: Confusion Matrix

The confusion matrix is just another way to visually see the results of our model, now in **sklearn** learn they have a module that makes building a confusion matrix quick. Let's use that.

```
# create a confusion matrix.
rf_matrix = confusion_matrix(y_test, y_pred)

# grab the different parts of the matrix
true_negatives = rf_matrix[0][0]
false_negatives = rf_matrix[1][0]
true_positives = rf_matrix[1][1]
false_positives = rf_matrix[0][1]

# calculate the metrics.
accuracy = (true_negatives + true_positives) / \
    (true_negatives + true_positives + false_negatives + false_positives)

percision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
specificity = true_negatives / (true_negatives + false_positives)

# print out the metrics.
print('Accuracy: {}'.format(float(accuracy)))
print('Percision: {}'.format(float(percision)))
print('Recall: {}'.format(float(recall)))
print('Specificity: {}'.format(float(specificity)))

# display the plot.
disp = plot_confusion_matrix(rand_frst_clf, X_test, y_test, \
    display_labels = ['Down Day', 'Up Day'], normalize = 'true', cmap=plt.cm.Blues)

disp.ax_.set_title('Confusion Matrix - Normalized')
plt.show()
```

Model Evaluation: Confusion Matrix Output



Model Evaluation: Feature Importance

With any model, you want to have an idea of what features are helping explain most of the model, as this can give you insight as to why you're getting the results you are. With Random Forest, we can identify some of our most important features or, in other words, the features that help explain most of the model. In some cases, some of our features might not be very important, or in other words, when compared to additional features, don't explain much of the model.

Why Do We Care About Feature Importance?

What that means is if we were to get rid of those features, our accuracy will go down a little, hopefully, but not significantly. You might be asking, "Why would I want to get rid of a feature if it lowers my accuracy?" Well, it depends, in some cases, you don't care if your model is 95% accurate or 92% accurate. To you, a 92% accurate model is just as good as a 95% accurate model.

However, if you wanted to get a 95% accurate model, you would, in this hypothetical case, have to train your model twice as long. Now, I'm a little extreme in this case, but the idea is the same. The cost doesn't justify the benefit. In the real world, we have to make these decisions all the time, and in some cases, it just doesn't warrant the extra cost for such a minimal increase in the accuracy.

Calculating the Feature Importance

Like all the previous steps, `sklearn` makes this process very easy. Take your `rand_frst_clf` and call the `feature_importances_` property. This will return all of our features and their importance measurement. Store the values in a `Pandas.Series` object and sore the values. Feature importance can be calculated two ways in Random Forest:

- Gini-Based Importance

- Accuracy-Based Importance

With `sklearn` they use the Gini-Importance metric for the Random Forest Algorithm. We can see in our model, that the most important feature is `k_percent` and our least important feature is `Price_Rate_Of_Change`.

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

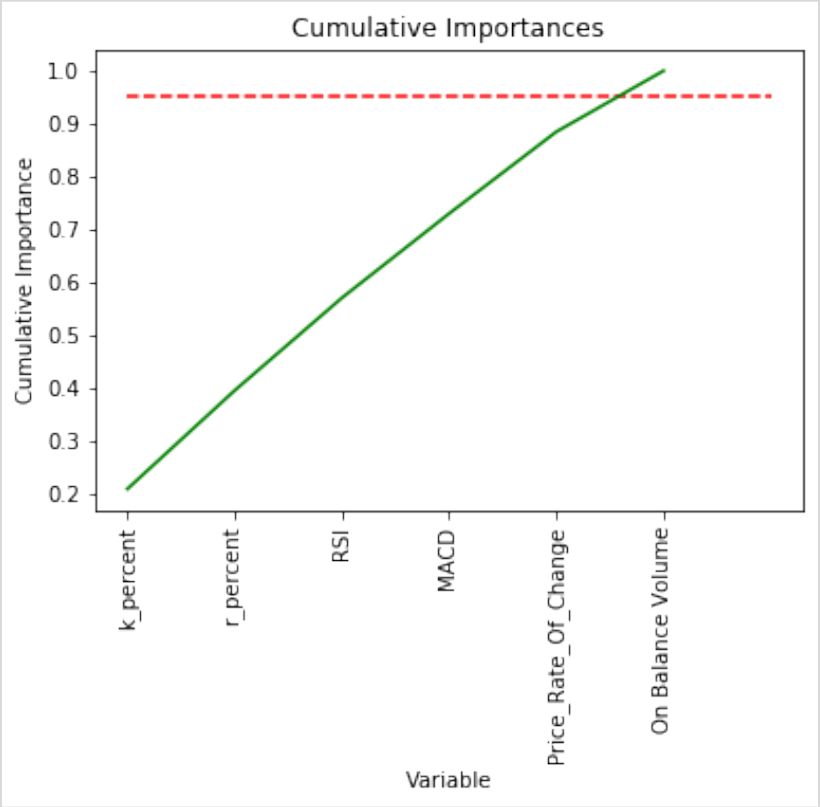
Metric	Feature Importance
K_percent	0.207933
R_percent	0.186390
RSI	0.175436
MACD	0.159717
Price_Rate_Of_Change	0.155012
On Balance Volume	0.115513

Model Evaluation: Feature Importance Graphing

If you want, you can also graph the feature importance of our model, so it's a little easier to visualize the results. What I do in the chart below is chart the cumulative importance or, in other words, how much does each feature add to the total. That way, we can see how much each feature is contributing to the overall importance. Another standard graph that is used is a bar chart.

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

Model Evaluation: Feature Importance Graph

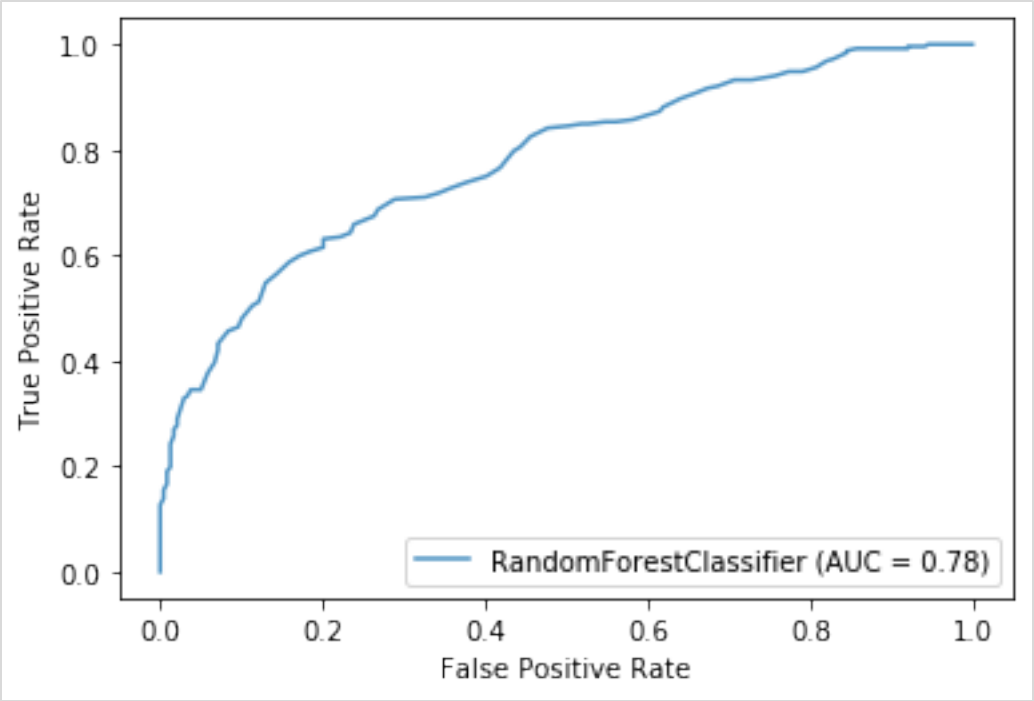


Model Evaluation: ROC Curve

The Receiver Operating Characteristic is a graphical method to evaluate the performance of a binary classifier. A curve is drawn by plotting True Positive Rate (sensitivity) against False Positive Rate (1 - specificity) at various threshold values. ROC curve shows the trade-off between sensitivity and specificity.

When the curve comes closer to the left-hand border and the top border of the ROC space, it indicates that the test is accurate. The closer the curve is to the top and left-hand border, the more accurate the test is. If the curve is close to the 45 degrees diagonal of the ROC space, it means that the test is not accurate. ROC curves can be used to select the optimal model and discard the suboptimal ones.

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



Model Evaluation: Out-Of-Bag Error Score

The `oob_score` uses a sample of "left-over" data that wasn't necessarily used during the model's analysis, and the validation set is a sample of data you decided to subset. In this way, the OOB sample is a little more random than the validation set.

Therefore, the OOB sample (on which the `oob_score` is measured) may be "harder" that the validation set. The `oob_score` may, on average, have a "less good" accuracy score as a consequence. For example, Jeremy and Terence use only the last 2 weeks of grocery store data as a validation set.

The OOB sample may have unused data from across all four years of sales data. The `oob_score`'s sample is much harder because it's more randomized and has more variance. If the `oob_score` never improves, but the validation set score is always excellent.

You need to re-think how to subset the validation set. In the case of Jeremy and Terence, they might decide to take a more random sample of data across all years rather than strictly the last 2 weeks of data.

```
print('Random Forest Out-Of-Bag Error Score: {}'.format(rand_frst_clf.oob_score_))
```

```
Random Forest Out-Of-Bag Error Score: 0.687
```

Model Improvement: Randomized Search

If you remember way up above, I mentioned that there is no magic number of estimators to use with every data set. Instead, with Random Forest, we have to try different values to find what the optimal values should be for each of the parameters.

Fortunately, for us, this can be done using the `RandomizedSearchCV` method provided by `sklearn`. The idea behind this approach is to provide a wide range of possible values for each hyperparameter and then using cross-validation, to try different combinations of these parameters.

With the highest result of these combinations being the one, we should use for our data set. To use this method, we need to first import the `RandomizedSearchCV` object from the sklearn.model_selection module. From there, we need to define a range of values for each of the hyperparameters we wish to test.

To do this appropriately, we need to make sure we understand what each argument means in the model, so let's walk through them:

- **n_estimators**
  - The number of trees in the forest.
- **max_features**
  - The number of features to consider when looking for the best split.
- **max_depth**
  - The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split**
  - The minimum number of samples required to split an internal node.
- **min_samples_leaf**
  - The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.
- **bootstrap**
  - Whether bootstrap samples are used when building trees. If False, the whole dataset is used to construct each tree.

```
'''
    Number of trees in random forest
    Number of trees is not a parameter that should be tuned, but set large enough.
    There is no risk of overfitting in random forest with growing number of # trees,
    as they are trained independently from each other.
'''
n_estimators = list(range(200, 2000, 200))

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

'''
    Maximum number of levels in tree
    Max depth is a parameter that most of the times should be set as high as possible,
    but possibly better performance can be achieved by setting it lower.
'''
max_depth = list(range(10, 110, 10))
max_depth.append(None)

'''
    Minimum number of samples required to split a node
    Higher values prevent a model from learning relations which might
    be highly specific to the particular sample selected for a tree. Too high values
    can also lead to # under-fitting hence depending on the level of
    underfitting or overfitting, you can tune the values for min_samples_split.
'''
min_samples_split = [2, 5, 10, 20, 30, 40]

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

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

# 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}
```

Model Improvement: Running Randomized Search

Now that we've created a range of values for some of our hyperparameters, we can put them to the test. The first thing we need to do is create a new instance of our `RandomForestClassifier` model and pass it through to our `RandomizedSearchCV` object.

When we use the `RandomizedSearchCV`, we need to specify a few additional arguments. The estimator is the model we wish to use; in this case, it's just our `RandomForestClassifier`. The `param_distribution` will get our `random_grid` dictionary.

`n_iter` is an essential argument because it will specify the number of iterations we will do, so the higher it is, the more iterations we will do. cv, defines the cross-validation splitting strategy we will use, `random_state` is used for random uniform sampling. verbose controls the verbosity: the higher, the more messages. `n_jobs` number of jobs to run in parallel. None means 1 unless in a `joblib.parallel_backend` context. -1 means using all processors.

```
# 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=42, n_jobs = -1)

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

```
Fitting 3 folds for each of 100 candidates, totaling 300 fits
[Parallel(n_jobs=-1)]: Using backend             LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  33 tasks            elapsed: 45.0s
[Parallel(n_jobs=-1)]: Done 154 tasks            elapsed: 4.1min
[Parallel(n_jobs=-1)]: Done 300 out of 300       elapsed: 8.8min finished
```

Model Improvement: Using the Improved Model

After that we will now have an improved model based on the optimal parameters from the `RandomizedSearch`. All we need to do at this point is just fit and test the data and see how the model performed.

Using the Improved Model – Code:

```
# With the new Random Classifier trained we can proceed to our regular steps, prediction.
rf_random.predict(X_test)


'''
    ACCURACY
'''
# Once the predictions have been made, then grab the accuracy score.
print('Correct Prediction (%): ', accuracy_score(y_test, rf_random.predict(X_test), \
    normalize = True) * 100.0)


'''
    CLASSIFICATION REPORT
'''
# Define the traget names
target_names = ['Down Day', 'Up Day']

# Build a classifcation report
report = classification_report(y_true = y_test, y_pred = y_pred, \
    target_names = target_names, output_dict = True)

# Add it to a data frame, transpose it for readability.
report_df = pd.DataFrame(report).transpose()
display(report_df)
print('\n')

'''
    FEATURE IMPORTANCE
'''
# Calculate feature importance and store in pandas series
feature_imp = pd.Series(rand_frst_clf.feature_importances_, \
    index=X_Cols.columns).sort_values(ascending=False)
```

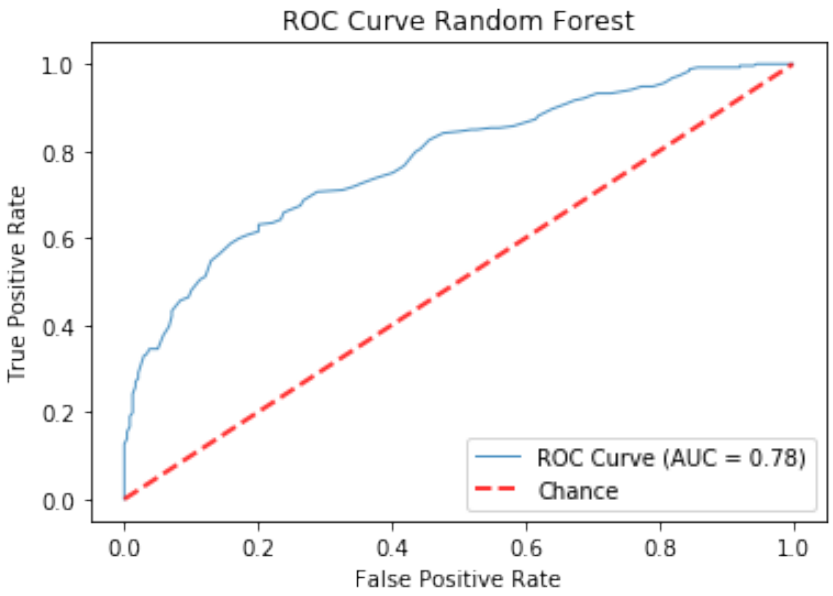
Using the Improved Model – Output:

Correct Prediction	72.7087576

	F1-Score	Precision	Recall	Support
Down Day	0.666667	0.690583	0.644351	239.000000
Up Day	0.703846	0.682836	0.726190	252.000000
accuracy	0.686354	0.686354	0.686354	0.686354
macro avg	0.685256	0.686709	0.685271	491.000000
weighted avg	0.685749	0.686607	0.686354	491.000000

Using the Improved Model – ROC Curve:

```
'''
    ROC CURVE
'''

fig, ax = plt.subplots()

# Create an ROC Curve plot.
rfc_disp = plot_roc_curve(rand_frst_clf, X_test, \
    y_test, alpha = 0.8, name='ROC Curve', lw=1, ax=ax)

# Add our Chance Line
ax.plot([0, 1], [0, 1], linestyle='--', lw=2, color='r', label='Chance', alpha=.8)

# Make it look pretty.
ax.set(xlim=[-0.05, 1.05], ylim=[-0.05, 1.05], title="ROC Curve Random Forest")

# Add the legend to the plot
ax.legend(loc="lower right")

plt.show()
```



SECTION 5

CLOSING NOTES

Considerations for Next Time

Any good project takes some time to reflect on what was done and consider items that might need to be used for next time. When looking at the dataset used in this project there are some important considerations that might impact the results of our model.

1. Is it balanced in the sense of there are enough up days compared to down days or does one dominate the other?
2. Do the stocks represent a single industry, company type, or even size? Our model might perform better with larger companies who tend to be less volatile in price.
3. Do other indicators work better? For the paper, it was easy to see that all the indicators they chose were related to momentum. Do other indicators, when used in aggregate, provide similar results?