Analysis of S&P 500 Companies

Venkata Siva Rajesh Vithanala

Introduction

Wealth is a highly sought after status that many struggle to attain but few achieve. It comes in many forms and has many more uses. Some may choose to save their life earnings to slowly aquire wealth, but some braver may choose to risk their earnings in the stock market.

Among these there are those who try to invest for the long-term profit and those who invest for the short-term gain. The goal of the long-term investors is oftentimes to grow their wealth at a rate exceeding inflation or at the very least to outperform their next best investment (,such as the interest rate banks provide). The invest in companies that they believe will continue to grow, because they believe in the profitability of the companies' business model.

However, the more daring believe in 'beating the market', which is to invest in such a way to outperform the S&P500 index. They believe that at any given point in the market some stocks are overrated, so should be sold, and some are underrated should be bought. In doing so, they will outmaneuver their adversaries, other traders, and profit. Many try, but most fail. More information about 'beating the market' can be found here https://www.investopedia.com/ask/answers/12/beating-the-market.asp#:~:text=The%20phrase%20%22beating%20the%20market,beat%20it%2C%20but%20few%20suc (https://www.investopedia.com/ask/answers/12/beating-the-market.asp#:~:text=The%20phrase%20%22beating%20the%20market,beat%20it%2C%20but%20few%20suc

The S&P500 (Standard & Poor's 500) represents approximately the top 500 largest publicly traded companies in the U.S.. It is often regarded as the best gauge to how well major buisinesses are performing for some time period. To learn more about the S&P500 visit this site https://www.investopedia.com/terms/s/sp500.asp (https://www.investopedia.com/terms/s/sp500.asp)

In the past humans would analyze the trend in the market to make their decisions. However, with the rapid improvement of computer processing speed and machine learning algorithms, trading is largely done by and between computers. Computers will analyze the trends of the market, and try to outperform their adversaries, other computers; the one with the more accuracte algorithm or better luck will profit.

Today, we will walk you through a simple tutorial of a rudimentary algorithm for modeling the stock market, with the goal of 'beating the market'. In the model we train, we will be predicting whether the stocks of a company will go up or down from day to day. Based on this prediction we can determine whether or not to buy or sell stocks.

For example if we sold a stock when we expect a stock to lose value, we can sell stocks today and buy them back tomorrow. If we are correct in our prediction then we would have earned the difference in value times the number of stocks we sold. If we think a stock will gain value, we can buy stocks today and sell them back tomorrow. Again we would have earned the difference in value times the number of stocks bought.

However, some constraints we face is that we do not have access to the most recent data, we are unable to model how our machine's decision influence the market, and we have limited processing power. The goal of this tutorial is not to build a model that will truly outperform the market, but to guide the audience on how to develop such a model.

In [2]:
```
# imports
import requests
from bs4 import BeautifulSoup as bs
import pandas as pd
import datetime as dt
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow import keras
import math
from sklearn.model_selection import cross_validate
```

Data Collection

Web Scraping Wikipedia

In this project, we'll mainly focus on the S&P 100, because we have limited computational resources. That said, there was this really nice table from wikipedia about the S&P 500 (Source: https://en.wikipedia.org/wiki/List_of_S%26P_500_companies (https://en.wikipedia.org/wiki/List_of_S%26P_500_companies)), which contains information about the company and its industry classification. We scrape this wikipedia's table below:

In [4]:
```
r = requests.get('https://en.wikipedia.org/wiki/List_of_S%26P_500_companie
s')
root = bs(r.content, "html")
sp500 = pd.read_html(root.find('table').prettify())[0]
sp500.head()
```

Out[4]:

	Symbol	Security	SEC filings	GICS Sector	GICS Sub-Industry	Headquarters Location	Date first added	CIK	I
0	MMM	3M Company	reports	Industrials	Industrial Conglomerates	St. Paul, Minnesota	1976-08-09	66740	
1	ABT	Abbott Laboratories	reports	Health Care	Health Care Equipment	North Chicago, Illinois	1964-03-31	1800	
2	ABBV	AbbVie Inc.	reports	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	1551152	
3	ABMD	ABIOMED Inc	reports	Health Care	Health Care Equipment	Danvers, Massachusetts	2018-05-31	815094	
4	ACN	Accenture plc	reports	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	1467373	

To isolate just the S&P 100, we need some way to identify them, so here we also scrape another table from wikipedia which lists the S&P 100 (Source: https://en.wikipedia.org/wiki/S%26P_100 (https://en.wikipedia.org/wiki/S%26P_100)):

In [5]:
```
r = requests.get('https://en.wikipedia.org/wiki/S%26P_100')
root = bs(r.content, "html")
sp100 = pd.read_html(root.find_all('table')[2].prettify())[0]
sp100.head()
```

Out[5]:

	Symbol	Name
0	AAPL	Apple Inc.
1	ABBV	AbbVie Inc.
2	ABT	Abbott Laboratories
3	ACN	Accenture
4	ADBE	Adobe Inc.

Pulling from Huge Stock Market Dataset

For this section, all of our stock data comes from a dataset from Kaggle:

https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs (https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs)

This data set is massive, containing information on over a 8500 funds and stocks, each stored in their own files. I've (Neo) downloaded the dataset and put it on a publically available repository to facilitate its use in this project.

We'll only pull the data which corresponds to the S&P 500 to match with the data we pulled from the S&P 500 table since it would be a waste of memory to load in all 8500+ files into dataframes. The raw github repo links are mostly similar, with the individual files named after the stock name. So, below we use common url heads and tails to pull datasets for just the S&P 500 companies found on the list in wikipedia.

```
In [6]: url_common_head = 'https://raw.githubusercontent.com/vvsra/DATA603_Final
_Tutorial_Huge_Stock_Market_Dataset/main/Stocks/'
url_common_tail = '.us.txt'

sp500stocks = pd.DataFrame()
unavailable_stocks = []

for i in range(sp500['Symbol'].size):
  try:
    data = pd.read_csv(url_common_head + sp500['Symbol'][i].lower() + url_c
ommon_tail)
    data['Symbol'] = [sp500['Symbol'][i]]*data.shape[0]
    sp500stocks = pd.concat([sp500stocks, data])
  except:
    print(sp500['Symbol'][i].lower() + url_common_tail + ' does not exist i
n the Huge Stock Market Dataset.')
    unavailable_stocks += [sp500['Symbol'][i]]

print(str(len(unavailable_stocks)) + ' companies\' data could not be found
in the Huge Stock Market Dataset')
sp500stocks.head()
```

```
amcr.us.txt does not exist in the Huge Stock Market Dataset.
aptv.us.txt does not exist in the Huge Stock Market Dataset.
bkr.us.txt does not exist in the Huge Stock Market Dataset.
brk.b.us.txt does not exist in the Huge Stock Market Dataset.
bkng.us.txt does not exist in the Huge Stock Market Dataset.
bf.b.us.txt does not exist in the Huge Stock Market Dataset.
carr.us.txt does not exist in the Huge Stock Market Dataset.
cbre.us.txt does not exist in the Huge Stock Market Dataset.
ctva.us.txt does not exist in the Huge Stock Market Dataset.
dow.us.txt does not exist in the Huge Stock Market Dataset.
evrg.us.txt does not exist in the Huge Stock Market Dataset.
gl.us.txt does not exist in the Huge Stock Market Dataset.
peak.us.txt does not exist in the Huge Stock Market Dataset.
hwm.us.txt does not exist in the Huge Stock Market Dataset.
iqv.us.txt does not exist in the Huge Stock Market Dataset.
j.us.txt does not exist in the Huge Stock Market Dataset.
lhx.us.txt does not exist in the Huge Stock Market Dataset.
lin.us.txt does not exist in the Huge Stock Market Dataset.
lumn.us.txt does not exist in the Huge Stock Market Dataset.
nlok.us.txt does not exist in the Huge Stock Market Dataset.
otis.us.txt does not exist in the Huge Stock Market Dataset.
rtx.us.txt does not exist in the Huge Stock Market Dataset.
tt.us.txt does not exist in the Huge Stock Market Dataset.
tfc.us.txt does not exist in the Huge Stock Market Dataset.
viac.us.txt does not exist in the Huge Stock Market Dataset.
vtrs.us.txt does not exist in the Huge Stock Market Dataset.
vnt.us.txt does not exist in the Huge Stock Market Dataset.
well.us.txt does not exist in the Huge Stock Market Dataset.
28 companies' data could not be found in the Huge Stock Market Dataset
```

Out[6]:

	Date	Open	High	Low	Close	Volume	OpenInt	Symbol
0	1970-01-02	2.1664	2.1749	2.1664	2.1664	84754	0	MMM
1	1970-01-05	2.1664	2.1833	2.1664	2.1749	525466	0	MMM
2	1970-01-06	2.1749	2.2002	2.1749	2.2002	207172	0	MMM
3	1970-01-07	2.2002	2.2172	2.1917	2.2086	193991	0	MMM
4	1970-01-08	2.2086	2.2512	2.2086	2.2427	357846	0	MMM

Now, we have everything we need and just need to do a lot of clean up; cutting out irrelevant data and refining what is left. So, it's on to **Data Processing** from here.

Data Processing

In this section, we make use of Pandas and NumPy to manipulate our dataframes, which are Pandas based objects. Below, I link to the docs for these libraries, so you can explore more of their functionality there:

Pandas: https://pandas.pydata.org/docs/index.html (https://pandas.pydata.org/docs/index.html)

NumPy: https://numpy.org/doc/stable/index.html (https://numpy.org/doc/stable/index.html)

Cleaning sp500 (Wikipedia) Dataframe

As seen when compiling data from the Huge Stock Market Dataset, some of the companies don't have corresponding data in our dataset.

In [7]:
```python
# print inital size
print(sp500.shape[0])

# remove the unavailable stocks
sp500_1 = sp500[~sp500['Symbol'].isin(unavailable_stocks)]

# print new size
print(sp500_1.shape[0])
```

```
505
477
```

Now that that's done, lets late a look at the features of this dataframe:

In [8]:
```python
sp500_1.head()
```

Out[8]:

	Symbol	Security	SEC filings	GICS Sector	GICS Sub-Industry	Headquarters Location	Date first added	CIK	I
0	MMM	3M Company	reports	Industrials	Industrial Conglomerates	St. Paul, Minnesota	1976-08-09	66740	
1	ABT	Abbott Laboratories	reports	Health Care	Health Care Equipment	North Chicago, Illinois	1964-03-31	1800	
2	ABBV	AbbVie Inc.	reports	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	1551152	
3	ABMD	ABIOMED Inc	reports	Health Care	Health Care Equipment	Danvers, Massachusetts	2018-05-31	815094	
4	ACN	Accenture plc	reports	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	1467373	

Most of this looks fine as is, but there are several improvements that can be made:

- SEC filings is just an entire column filled with 'report'. On the Wikipedia webpage, these were hyperlinks to SEC filing records for the company, but in our table, it's useless and thus can be cut from the dataframe.
- Similarly, Central Index Key (CIK) is just a number given to an individual, company, or foreign government by the United States Securities and Exchange Commission. It is completely random and should have no use in our model and thus can also be cut from the dataframe.
- The data in the 'Date first added' column is a time, so it would be best represented by a UTC DateTime object instead of a string.
- The 'Founded' column seems to have a few strange entries and should be cleaned so that all data is in the same format. Also, since it is also time data, it would be best represented by a UTC DateTime object instead of a string.

With these tasks listed one, let's go through them one by one.

Cutting out SEC filings and CIK

In [9]:
```
sp500_2 = sp500_1.drop(columns=['SEC filings'])
sp500_3 = sp500_2.drop(columns=['CIK'])
sp500_3.head()
```

Out[9]:

	Symbol	Security	GICS Sector	GICS Sub-Industry	Headquarters Location	Date first added	Founded
0	MMM	3M Company	Industrials	Industrial Conglomerates	St. Paul, Minnesota	1976-08-09	1902
1	ABT	Abbott Laboratories	Health Care	Health Care Equipment	North Chicago, Illinois	1964-03-31	1888
2	ABBV	AbbVie Inc.	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	2013 (1888)
3	ABMD	ABIOMED Inc	Health Care	Health Care Equipment	Danvers, Massachusetts	2018-05-31	1981
4	ACN	Accenture plc	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	1989

Adjusting Date first added

Apparently, there are some inconsistent formatting in this section as well. Some strings had extra parenthesised dates tailing the initial date, and some are just NaN. Since we already have an abundence of data, we will just ignore these data points and drop them.

In [10]:
```
datetime_date_first_added = []
dropped_companies = []

for r in sp500_3.iterrows():
  try:
    datetime_date_first_added += [datetime.strptime(r[1]['Date first adde
d'], '%Y-%m-%d')]
  except:
    datetime_date_first_added += [np.NaN]
    dropped_companies += [r[1]['Symbol']]

sp500_4 = sp500_3.copy()
sp500_4['Date first added'] = datetime_date_first_added
sp500_4.head()
```

Out[10]:

	Symbol	Security	GICS Sector	GICS Sub-Industry	Headquarters Location	Date first added	Founded
0	MMM	3M Company	Industrials	Industrial Conglomerates	St. Paul, Minnesota	1976-08-09	1902
1	ABT	Abbott Laboratories	Health Care	Health Care Equipment	North Chicago, Illinois	1964-03-31	1888
2	ABBV	AbbVie Inc.	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	2013 (1888)
3	ABMD	ABIOMED Inc	Health Care	Health Care Equipment	Danvers, Massachusetts	2018-05-31	1981
4	ACN	Accenture plc	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	1989

And now to remove the rows with NaN values:

In [11]:
```
# print initial size
print(sp500_4.shape[0])

# remove dropped companies
sp500_5 = sp500_4[~sp500_4['Symbol'].isin(dropped_companies)]

# print new size
print(sp500_5.shape[0])
```

```
477
424
```

Now, just to make sure the type of the object in the 'Date first added' column is a datetime, we print out the dtypes of the dataframe:

In [12]:
```
sp500_5.dtypes
```
Out[12]:
```
Symbol                           object
Security                         object
GICS  Sector                     object
GICS Sub-Industry                object
Headquarters Location            object
Date first added         datetime64[ns]
Founded                          object
dtype: object
```

Cleaning Founded

Similar to how we handled the 'Date first added' column, we will just drop any row which has inconsistent formatting and convert the others into datetime objects. So, first we build the new 'Founded' column:

In [13]:
```
founded = []

for r in sp500_5.iterrows():
  try:
    founded += [datetime.strptime(r[1]['Founded'], '%Y')]
  except:
    founded += [np.NaN]
    dropped_companies += [r[1]['Symbol']]

sp500_6 = sp500_5.copy()
sp500_6['Founded'] = founded
sp500_6.head()
```
Out[13]:

	Symbol	Security	GICS Sector	GICS Sub-Industry	Headquarters Location	Date first added	Founded
0	MMM	3M Company	Industrials	Industrial Conglomerates	St. Paul, Minnesota	1976-08-09	1902-01-01
1	ABT	Abbott Laboratories	Health Care	Health Care Equipment	North Chicago, Illinois	1964-03-31	1888-01-01
2	ABBV	AbbVie Inc.	Health Care	Pharmaceuticals	North Chicago, Illinois	2012-12-31	NaT
3	ABMD	ABIOMED Inc	Health Care	Health Care Equipment	Danvers, Massachusetts	2018-05-31	1981-01-01
4	ACN	Accenture plc	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	1989-01-01

And now to remove the rows with NaN values:

In [14]:
```
# print initial size
print(sp500_6.shape[0])

# remove dropped companies
sp500_7 = sp500_6[~sp500_6['Symbol'].isin(dropped_companies)]

# print new size
print(sp500_7.shape[0])
```

```
424
392
```

Just like before, lets take a look at the dtypes to make sure the 'Founded' column has been successfully converted into datetime objects:

In [15]:
```
sp500_7.dtypes
```

Out[15]:
```
Symbol                            object
Security                          object
GICS  Sector                      object
GICS Sub-Industry                 object
Headquarters Location             object
Date first added          datetime64[ns]
Founded                   datetime64[ns]
dtype: object
```

Joining Wikipedia Dataframes

First, remove unavailable_stocks from sp100

In [16]:
```
# print inital size
print(sp100.shape[0])

# remove the unavailable stocks
sp100_1 = sp100[~sp100['Symbol'].isin(unavailable_stocks)]

# print new size
print(sp100_1.shape[0])
```

```
101
98
```

Now, remove all data points from sp500 and sp500stock which do not share a symbol with sp100:

In [17]:
```
# print inital size
print(sp500_7.shape[0])

# remove the unavailable stocks
sp500_7_1 = sp500_7[sp500_7['Symbol'].isin(sp100_1['Symbol'])]

# print new size
print(sp500_7_1.shape[0])
```

```
392
79
```

In [18]:
```
# print initial size
print(sp500stocks.shape[0])

# remove dropped companies
sp500stocks_0_1 = sp500stocks[sp500stocks['Symbol'].isin(sp100_1['Symbo
l'])]

# print new size
print(sp500stocks_0_1.shape[0])
```

```
2594230
739062
```

With this, only data for companies from the S&P 100 are left in all of our dataframes.

Cleaning sp500stocks (Kaggle) Dataframe

Before anything else, there's a list of dropped companies which need to be removed from this dataframe:

In [19]:
```
# print initial size
print(sp500stocks_0_1.shape[0]) # remove _0_1 if 500

# remove dropped companies
sp500stocks_1 = sp500stocks_0_1[~sp500stocks_0_1['Symbol'].isin(dropped_com
panies)] # remove _0_1 if 500

# print new size
print(sp500stocks_1.shape[0])
```

```
739062
620097
```

Just to be sure, lets compare the unique values of the 'Symbol' column for sp500 and sp500stocks. Since they should have the same unique values, they should be equal after sorting. Since we're working with numpy arrays, a comparison of the sorted arrays should result in an array filled with Trues. Let's verify this:

```
In [20]: sp500su = sp500stocks_1['Symbol'].unique()
         sp500su.sort()

         sp500u = sp500_7_1['Symbol'].unique() # remove _1 if 500
         sp500u.sort()

         sp500u == sp500su
```

```
Out[20]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True,  True,  True,  True,  True,  True,  True])
```

Now, let's take a look at the features of this dataframe and come up with a checklist for cleaning:

```
In [21]: sp500stocks_1.head()
```

Out[21]:

	Date	Open	High	Low	Close	Volume	OpenInt	Symbol
0	1970-01-02	2.1664	2.1749	2.1664	2.1664	84754	0	MMM
1	1970-01-05	2.1664	2.1833	2.1664	2.1749	525466	0	MMM
2	1970-01-06	2.1749	2.2002	2.1749	2.2002	207172	0	MMM
3	1970-01-07	2.2002	2.2172	2.1917	2.2086	193991	0	MMM
4	1970-01-08	2.2086	2.2512	2.2086	2.2427	357846	0	MMM

There are several types of issues to address:

- Dropping useless data
- Data type conversion
- Find overlapping timeframe

Dropping useless data

The 'OpenInt' column doesn't look too useful based off the the head that we displayed earlier. Let's take a more complete look at it to see what unique values it has:

```
In [22]: sp500stocks_1['OpenInt'].unique()

Out[22]: array([0])
```

It appears that the 'OpenInt' column is just a column full of zeroes. Since it has no variation at all between any of the stocks, it is completely useless as a feature for distinguishing between them. So, we can drop it:

In [23]:
```
sp500stocks_2 = sp500stocks_1.drop(columns=['OpenInt'])
sp500stocks_2.head()
```

Out[23]:

	Date	Open	High	Low	Close	Volume	Symbol
0	1970-01-02	2.1664	2.1749	2.1664	2.1664	84754	MMM
1	1970-01-05	2.1664	2.1833	2.1664	2.1749	525466	MMM
2	1970-01-06	2.1749	2.2002	2.1749	2.2002	207172	MMM
3	1970-01-07	2.2002	2.2172	2.1917	2.2086	193991	MMM
4	1970-01-08	2.2086	2.2512	2.2086	2.2427	357846	MMM

Data type conversion

Ideally:

- Date should be a datetime type
- Open, High, Low, and Close should be floats
- Volume should be an integer
- Symbol should be a string

Before anything else, let's see what the current types are for this dataframe:

In [24]:
```
sp500stocks_2.dtypes
```

Out[24]:
```
Date       object
Open      float64
High      float64
Low       float64
Close     float64
Volume      int64
Symbol     object
dtype: object
```

So, it looks like all we need to do is to convert the objects in the 'Date' column into datetime objects.

In case anything has an inconsistent format, we will keep a dataframe of inconsistent rows and deal with it as needed. Unlike in the previous dataset where we could simply drop it, we also have to worry about the date here since we don't want any breaks timewise in the data for any given company.

In [25]:
```
date = []
inconsistent_rows = pd.DataFrame(columns=sp500stocks_2.columns)

#
for r in sp500stocks_2.iterrows():
  try:
    date += [datetime.strptime(r[1]['Date'], '%Y-%m-%d')]
  except:
    date += [np.NaN]
    inconsistent_rows.append(r)


sp500stocks_3 = sp500stocks_2.copy()
sp500stocks_3['Date'] = date
sp500stocks_3.head()
```

Out[25]:

	Date	Open	High	Low	Close	Volume	Symbol
0	1970-01-02	2.1664	2.1749	2.1664	2.1664	84754	MMM
1	1970-01-05	2.1664	2.1833	2.1664	2.1749	525466	MMM
2	1970-01-06	2.1749	2.2002	2.1749	2.2002	207172	MMM
3	1970-01-07	2.2002	2.2172	2.1917	2.2086	193991	MMM
4	1970-01-08	2.2086	2.2512	2.2086	2.2427	357846	MMM

Just to be sure we've converted everything successfully, let's examine the new types for the dataframe:

In [26]:
```
sp500stocks_3.dtypes
```

Out[26]:
```
Date      datetime64[ns]
Open             float64
High             float64
Low              float64
Close            float64
Volume             int64
Symbol            object
dtype: object
```

Next, let's take a look at the inconsistent_rows. Hopefully, it's empty. Otherwise, we'll have some more cleaning to do.

In [27]:
```
inconsistent_rows.head()
```

Out[27]:

Date	Open	High	Low	Close	Volume	Symbol

Thankfully, it's empty, which means we're done with the data conversion section of the cleaning.

Find overlapping timeframe

Next, we need to take a look at the range of dates in this dataframe from each company. The Huge Stock Market Data does not provide the same date range for stock data for all companies. For this project, we need a period of time when all companies have stock data, and preferably a time frame of about 1 year. Depending on the overlapping timeframes for individual companies, this may mean that we need to cut more companies from the two datasets if it comes down to it.

First, lets compile a new dataframe from the data in sp500stocks to find the date ranges for any given stock:

In [28]:
```
date_ranges = pd.DataFrame()
symbols = []
min_dates = []
max_dates = []

for stock_name in sp500stocks_3['Symbol'].unique():
  filtered_rows = sp500stocks_3[sp500stocks_3['Symbol'] == stock_name]
  min_date = filtered_rows.iloc[0]['Date']
  max_date = filtered_rows.iloc[0]['Date']

  for r in filtered_rows.iterrows():
    if min_date > r[1]['Date']:
      min_date = r[1]['Date']
    if max_date < r[1]['Date']:
      max_date = r[1]['Date']

  symbols += [stock_name]
  min_dates += [min_date]
  max_dates += [max_date]

date_ranges['Symbol'] = symbols
date_ranges['Earliest Date'] = min_dates
date_ranges['Latest Date'] = max_dates
date_ranges.head()
```

Out[28]:

	Symbol	Earliest Date	Latest Date
0	MMM	1970-01-02	2017-11-10
1	ABT	1983-04-06	2017-11-10
2	ACN	2005-02-25	2017-11-10
3	ADBE	1986-08-14	2017-11-10
4	ALL	1993-06-03	2017-11-10

Now that we have the start and end dates for all these companies, we can take the max of the 'Earliest Date' column and the min of the 'Latest Date' column to find the timeframe in which all companies have stock data.

In [29]:
```
# max of earliest dates
print(date_ranges['Earliest Date'].max())

# min of latest dates
print(date_ranges['Latest Date'].min())

# print length of timeframe
print(date_ranges['Latest Date'].min() - date_ranges['Earliest Date'].max
())
```

```
2017-10-11 00:00:00
2017-11-09 00:00:00
29 days 00:00:00
```

Unfortunately, this timeframe is too short. This means we'll need to start dropping companies until the overlapping timeframe is an appropriate length. Here, we drop companies until the overlapping timeframe is greater than 365 days:

In [30]:
```
dropped_companies = []
date_ranges_cp = date_ranges.copy()
target_delta_time = dt.timedelta(days=365)

while target_delta_time > date_ranges_cp['Latest Date'].min() - date_ranges
_cp['Earliest Date'].max():
  max_ed_symbol = date_ranges_cp[date_ranges_cp['Earliest Date'] == date_ra
nges_cp['Earliest Date'].max()]['Symbol'].iloc[0]
  min_ld_symbol = date_ranges_cp[date_ranges_cp['Latest Date'] == date_rang
es_cp['Latest Date'].min()]['Symbol'].iloc[0]

  if max_ed_symbol == min_ld_symbol:
    dropped_companies += [max_ed_symbol]
    date_ranges_cp = date_ranges_cp[date_ranges_cp['Symbol'] != max_ed_symb
ol]

  else:
    gain_from_ed = date_ranges_cp['Earliest Date'].max() - date_ranges_cp[d
ate_ranges_cp['Symbol'] != max_ed_symbol]['Earliest Date'].max()
    gain_from_ld = date_ranges_cp[date_ranges_cp['Symbol'] != min_ld_symbo
l]['Latest Date'].min() - date_ranges_cp['Latest Date'].min()
    if gain_from_ed <= gain_from_ld:
      dropped_companies += [max_ed_symbol]
      date_ranges_cp = date_ranges_cp[date_ranges_cp['Symbol'] != max_ed_sy
mbol]
    else:
      dropped_companies += [min_ld_symbol]
      date_ranges_cp = date_ranges_cp[date_ranges_cp['Symbol'] != min_ld_sy
mbol]

print(dropped_companies)
len(dropped_companies)
```

```
['DD']
```

Out[30]:
```
1
```

After removing the one company, 'DD', the length of our overlapping timeframe is now:

```
In [31]: print(date_ranges_cp['Latest Date'].min() - date_ranges_cp['Earliest Dat
         e'].max())
```

```
858 days 00:00:00
```

Next, we have to remove all those companies and their stocks from our two main dataframes (sp500 and sp500stocks):

```
In [32]: # remove from sp500

         # print initial size
         print(sp500_7_1.shape[0]) # remove _1 if 500

         # remove dropped companies
         sp500_8 = sp500_7_1[~sp500_7_1['Symbol'].isin(dropped_companies)] # remove
         _1 if 500

         # print new size
         print(sp500_8.shape[0])
```

```
79
78
```

```
In [33]: # remove from sp500stocks

         # print initial size
         print(sp500stocks_3.shape[0])

         # remove dropped companies
         sp500stocks_4 = sp500stocks_3[~sp500stocks_3['Symbol'].isin(dropped_compani
         es)]

         # print new size
         print(sp500stocks_4.shape[0])
```

```
620097
620083
```

Lastly, we also need to shave off the hanging dates from the various companies so that all of our stock data start and end on the same day for all companies.

```
In [34]: max_ed = date_ranges_cp['Earliest Date'].max()
         min_ld = date_ranges_cp['Latest Date'].min()

         # print initial size
         print(sp500stocks_4.shape[0])

         # shave data
         sp500stocks_5 = sp500stocks_4[(sp500stocks_4['Date'] >= max_ed) & (sp500sto
         cks_4['Date'] <= min_ld)]

         # print new size
         print(sp500stocks_5.shape[0])
```

```
620083
46488
```

Merging Data into a single Master Dataframe

Now that both individual dataframes have been cleaned and refined. we'll merge them together to form the master dataframe for this project. This single dataframe will contain all the data we will need for this project in a tidy format.

First, a review of the shape and features of our two main dataframes which contain all the data we'll use:

In [35]:
```
print(sp500_8.shape)
sp500_8.head()
```

```
(78, 7)
```

Out[35]:

	Symbol	Security	GICS Sector	GICS Sub-Industry	Headquarters Location	Date first added	Founded
0	MMM	3M Company	Industrials	Industrial Conglomerates	St. Paul, Minnesota	1976-08-09	1902-01-01
1	ABT	Abbott Laboratories	Health Care	Health Care Equipment	North Chicago, Illinois	1964-03-31	1888-01-01
4	ACN	Accenture plc	Information Technology	IT Consulting & Other Services	Dublin, Ireland	2011-07-06	1989-01-01
6	ADBE	Adobe Inc.	Information Technology	Application Software	San Jose, California	1997-05-05	1982-01-01
21	ALL	Allstate Corp	Financials	Property & Casualty Insurance	Northfield Township, Illinois	1995-07-13	1931-01-01

In [36]:
```
print(sp500stocks_5.shape)
sp500stocks_5.head()
```

```
(46488, 7)
```

Out[36]:

	Date	Open	High	Low	Close	Volume	Symbol
11479	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM
11480	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM
11481	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM
11482	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM
11483	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM

For each data point in sp500stocks, we want to append the relevant information from the sp500 dataframe using the Symbol column as the common key.

In [37]:
```
MASTER_DF = sp500stocks_5.merge(sp500_8, left_on='Symbol', right_on='Symbo
l')
MASTER_DF.head()
```

Out[37]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Some code to verify its shape and key characteristics:

In [38]:
```
# print shape and compare to sp500stocks (should have same number of rows)

print(MASTER_DF.shape)
print(sp500stocks_5.shape)
```

```
(46488, 13)
(46488, 7)
```

In [39]:
```
# get number of unique companies and compare to sp500 (They should be the s
ame)

print(len(MASTER_DF['Symbol'].unique()))
print(len(sp500_8['Symbol'].unique()))
```

```
78
78
```

With this, the Master dataframe is complete. In the following sections, all data used will come from this one dataframe, but in order to preserve this tidy dataframe, no modification operations should be performed directly on this dataframe (cause it would be too much trouble to recreate if it is damaged). Copies should be used instead. This dataframe variable is in all caps as a reminder of this usage regulation.

Exploratory analysis & Data viz

Now that we have our data, we can start considering how to best use this data to build our stock trading robot. In order to best do that, we need to get a better feel for the contents of our dataframe and how they relate with each other.

We'll start by just taking a look at the general stock trends for all of our data points over the time period that we've isolated. Then, we can go on from there to investigate any trends that appear interesting or relevant to consider.

In this section we use MatPlotLib to generate plots. The general documentation can be explored here: https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.plot.html (https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.plot.html)

Let's first create a copy of our Master Dataframe and review the various features it contains:

In [40]:
```
m_cp_1 = MASTER_DF.copy()
m_cp_1.head()
```

Out[40]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

The features that seem the most relevant for consideration are the **Open**, **High**, **Low**, and **Close** columns for each date and stock. To get a good feel for how these values change for each stock throughout our selected timeframe, let's first create a new column called **Average** which just adds the **Open** and **Close** and divides by 2. We can then use this **Average** as a metric for generally examining the change in stock prices over time.

In [41]:
```
m_cp_1['Average'] = (m_cp_1['Open'] + m_cp_1['Close'])/2
m_cp_1.head()
```

Out[41]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Now that we have an **Average** for each datapoint, we can pivot this dataframe by **Date** and **Symbol** to focus on the values of **Average** for each **Date** and **Symbol**, and graph it accordingly.

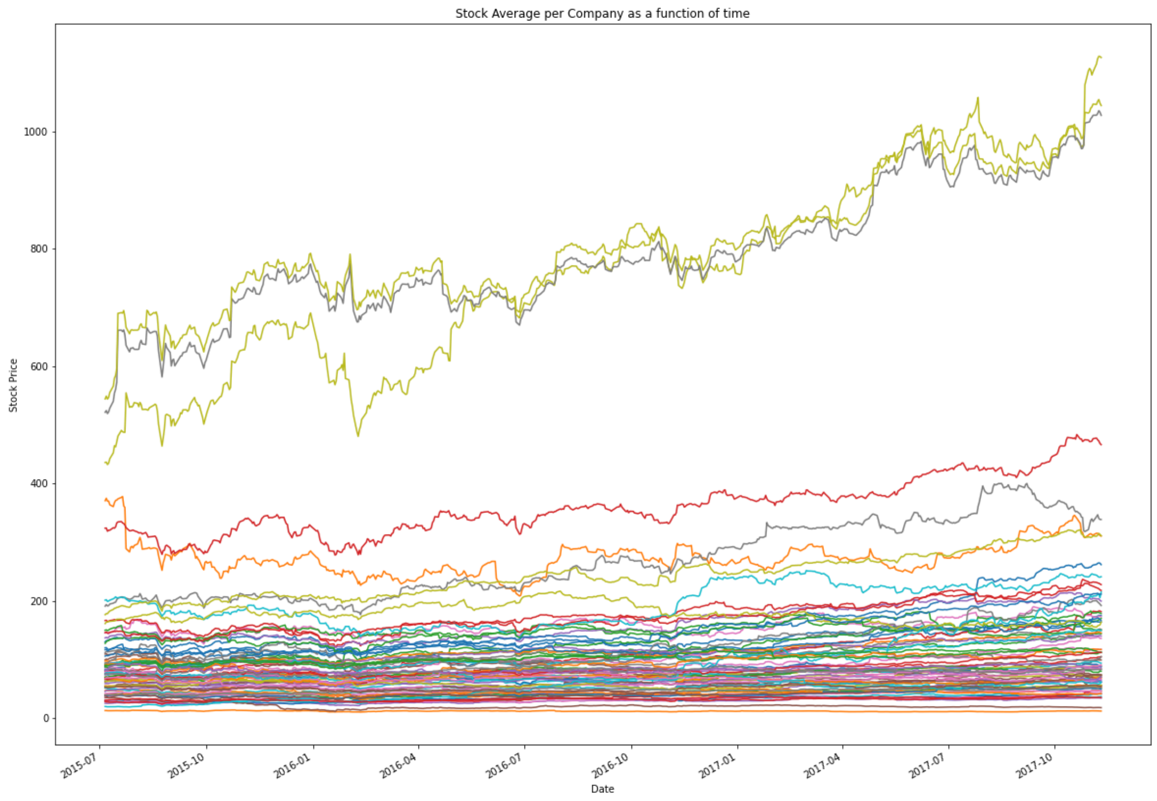
In [42]:
```
# Pivot dataframe
m_cp_1_p = m_cp_1.pivot(index='Date', columns='Symbol', values='Average')
m_cp_1_p.head()
```

Out[42]:

Symbol	AAPL	ABT	ACN	ADBE	AIG	ALL	AMGN	AMT	AMZN	AXP
Date										
2015-07-06	119.74	46.5685	91.1605	80.260	58.8410	61.8695	144.900	90.5335	435.635	74.602
2015-07-07	120.05	47.2350	92.4305	80.630	59.1900	62.5505	146.780	91.3415	436.200	74.589
2015-07-08	117.88	46.6770	92.7080	79.970	58.7020	62.3775	144.800	90.6200	432.025	73.489
2015-07-09	116.39	46.6785	92.4640	80.605	58.9795	62.8565	143.615	90.6385	434.645	73.560
2015-07-10	117.01	46.9470	92.9145	81.045	59.8185	63.3695	145.595	90.9010	442.000	74.353

In [ ]:
```
# graph pivoted dataframe
m_cp_1_p.plot(title="Stock Average per Company as a function of time", lege
nd=False, ylabel="Stock Price", figsize=(20,15))
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7fb3d23b38>
```



From this graph, it looks like there are several outliers which seem to mirror each other in their growth trends. Most of the stock prices are below 200 in price. It appears that there might be a general upward trend in stock prices over time, but it's hard to say for sure from just this graph due to to heavy concentration of lines below the 200 mark.

In [45]:
```
m_cp = MASTER_DF.copy()
m_cp.head()
```

Out[45]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industry
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Additional features that seem like they would be relevant would include the **Sector** and **Growth**, since some sectors may have been more profitable overall than other sectors. And **Growth** represents the percentage increase in stock value day to day. The companies that have high **Growth** represent the companies that were profitable and this is what our machine is trying to predict. In order to calculate **Growth** we will take (**Close** / **Open** - 1) * 100%.

```
In [ ]: m_cp['Growth'] = (m_cp['Close'] / m_cp['Open'] - 1) * 100
        m_cp.head()
```

Out[ ]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Additionally we would like to calculate the **Compounded Growth**, which we will calculate by:

$$\frac{100\%}{\text{Start Date Open Price}} \sum_{i=\text{start date}}^{\text{end date}} [i\text{th Date Close Price} - \text{Start Date Open Price}]$$

Basically, we are modeling how the stock prices on any given day compare to their initial stock price value, and we standarize it to their initial stock price value.

In [ ]:

```
# groupby will group the dataframe based on unique symbol values
# apply will modify the value of each grouping based on formula provided
compounded_growth = m_cp.groupby('Symbol').apply(lambda x: (x['Close'] - fl
oat(x['Open'].head(1))) / float(x['Open'].head(1)))

# reset_index undoes the grouping, returning to original dataframe (keeping
modifications)
compounded_growth = compounded_growth.reset_index()

m_cp['Compounded Growth'] = compounded_growth['Close']
m_cp.head()
```

Out[ ]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Now we will average (mean) the **Compounded Growth** across all companies within the same **Sector**. To do so, we will calculate:

$$\frac{1}{\#\text{ of Companies in the Sector}} \sum_{\text{Companies in the Sector}} [\text{Compounded Growth of Company}]$$

for each sector

```
In [ ]: sectorlst = m_cp['GICS  Sector'].drop_duplicates()
        daterange = m_cp['Date'].drop_duplicates()
        growthplt = pd.DataFrame({'Date': daterange})

        # Calculating unique average sector growth for each sector
        for sector in sectorlst:
          m_sector = m_cp.loc[m_cp['GICS  Sector'] == sector]
          sector_growth = []

          # Calculating the average sector growth for each day
          # by summing up compounded growth for every company
          # of a given sector for that given day
          for date in daterange:
            m_sector_day = m_sector.loc[m_cp['Date'] == date]
            average_growth = m_sector_day['Compounded Growth'].sum() / len(m_sector
        _day)
            sector_growth.append(average_growth)

          growthplt[sector] = sector_growth

        growthplt.head()
```

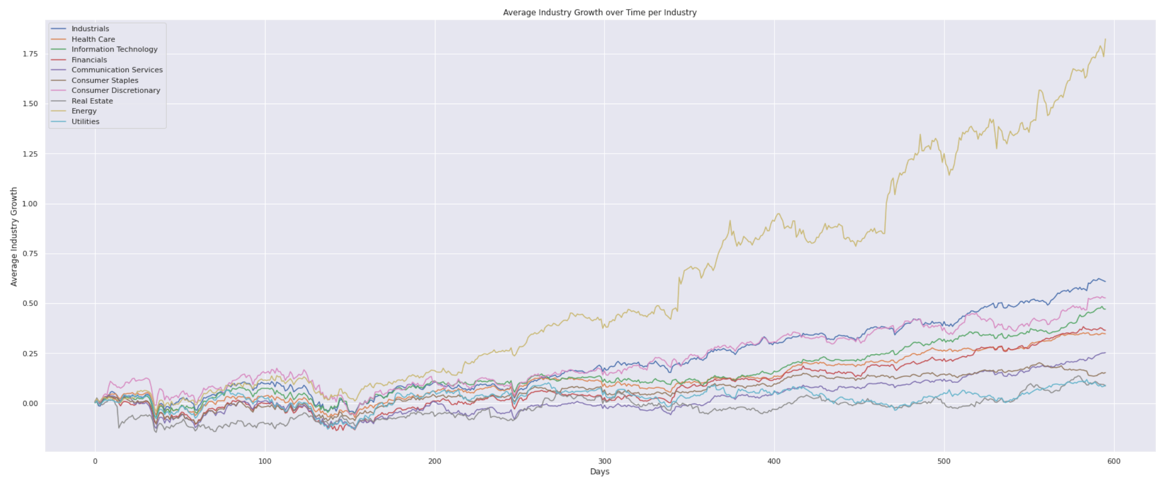
Out[ ]:

	Date	Industrials	Health Care	Information Technology	Financials	Communication Services	Consumer Staples	Consui Discretion
0	2015-07-06	0.007124	0.004889	0.000709	0.004327	0.008994	0.003582	0.006
1	2015-07-07	0.007279	0.014812	0.008542	0.011577	0.016634	0.013809	0.011
2	2015-07-08	-0.013627	0.000610	-0.008218	-0.002968	0.002050	-0.000960	-0.001
3	2015-07-09	-0.015110	0.005726	-0.003634	-0.001134	0.004322	0.004014	0.004
4	2015-07-10	0.003084	0.018934	0.006026	0.011877	0.012308	0.016407	0.018

```
In [ ]: growthplt = growthplt.drop(columns = ['Date'])
```

Now we shall plot the **sector growth** vs **Date**

In [ ]:
```python
growthplt.plot(figsize = (30, 12))
plt.xlabel('Days')
plt.ylabel('Average Sector Growth')
plt.title('Average Sector Growth over Time per Sector')

plt.show()
```

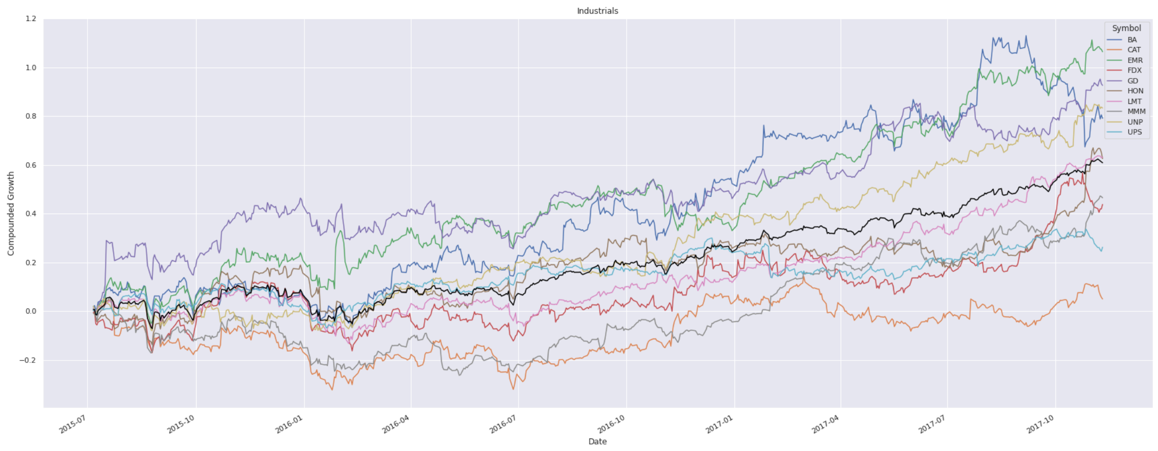


We will notice that the growth of each sector seems closely similar to each other; they seem to all fall under a similar trend (excluding energy). However, you should not be alarmed, since this still falls within our expectations. Typically speaking, the stock market reflects the economic situation of a nation. Therefore fluctuations within one sector will be closely modeled in the fluctuation within another sector. The exception to this is when some sector experiences more growth due to external factors, such as technological advancements and scientific breakthroughs. Since the energy sector seems to diverge from the trend; this might indicate that some breakthrough in energy production occurred during this time period or some national policy (such as the deregulation of this sector) improved the performance of this sector .

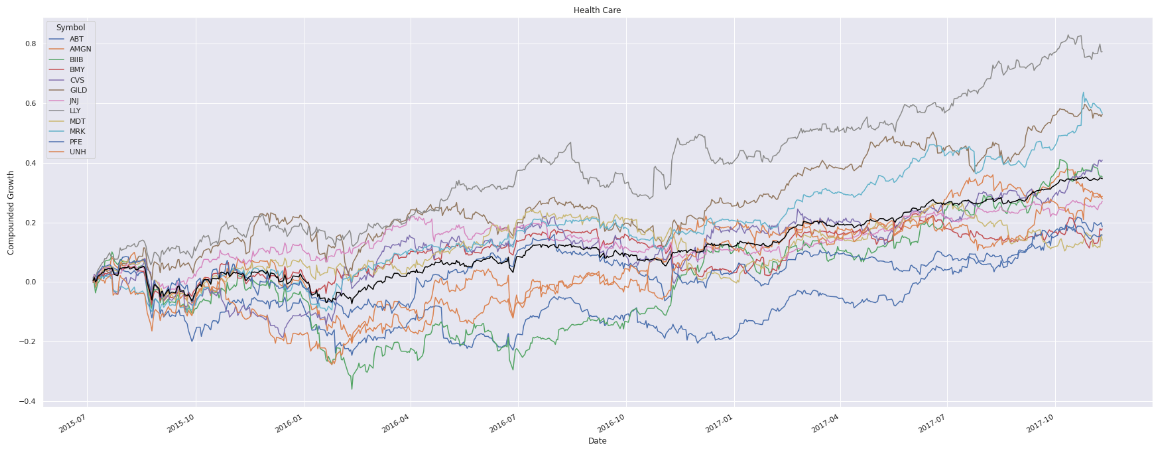
Now lets look at how companies within the same **Sector** compare to each other. We will be creating 10 separate plots for each **Sector**, with the inclusion of the industry average for comparison in black.

In [ ]:
```python
def make_sector_plot(sector):
  pplt = m_cp.loc[(m_cp['GICS  Sector'] == sector)].pivot(index = 'Date', c
olumns = 'Symbol', values = 'Compounded Growth')
  pplt.plot(figsize = (30, 12))

  # plotting the average industry growth for comparison (in black so it sta
nds out)
  plt.plot(daterange, growthplt[sector], color = 'black')

  plt.title(sector)
  plt.xlabel('Date')
  plt.ylabel('Compounded Growth')

  plt.show()
```

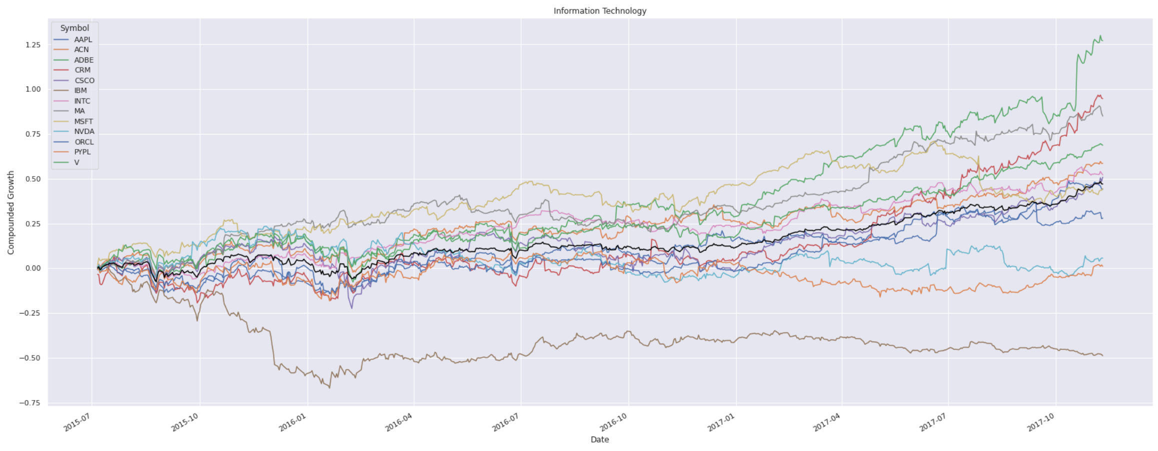
In [ ]:
```
# reindexing so that ith index corresponds to ith sector
sectorlst.index = range(len(sectorlst))

make_sector_plot(sectorlst[0])
```



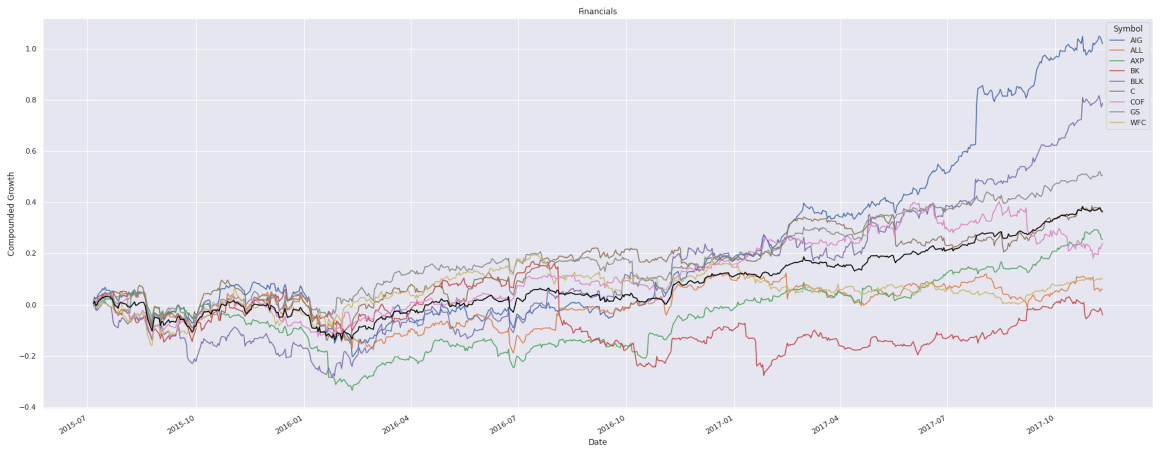
In [ ]:
```
make_sector_plot(sectorlst[1])
```



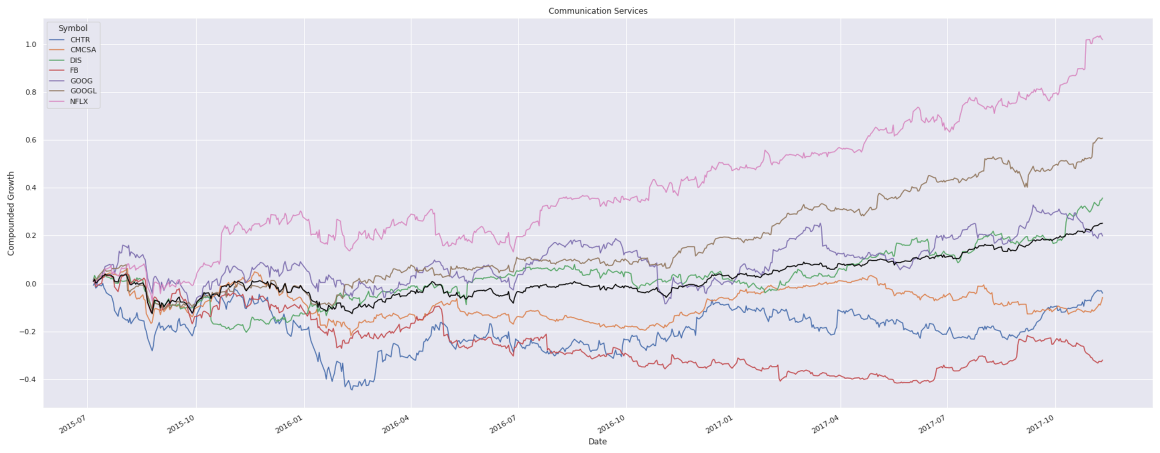
In [ ]:
```
make_sector_plot(sectorlst[2])
```



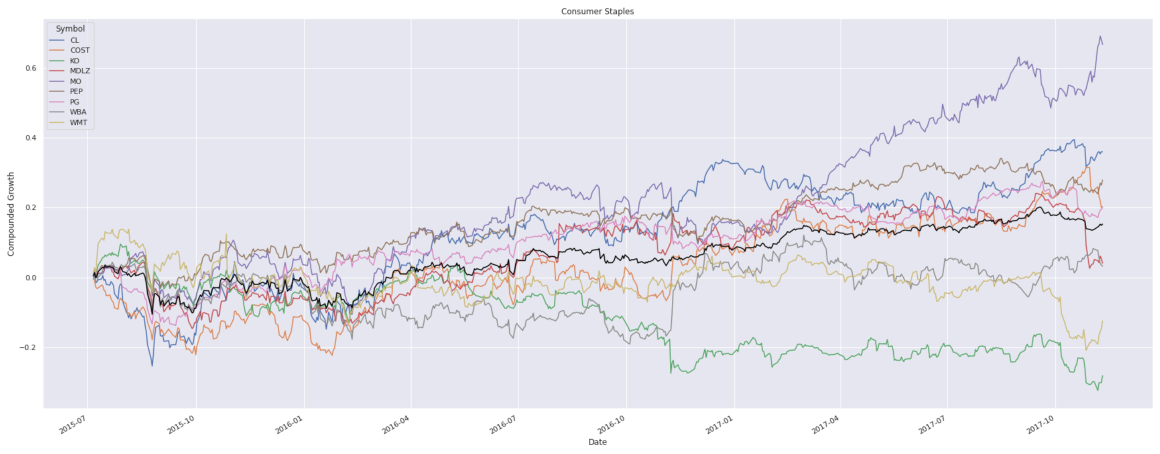
In [ ]:
```
make_sector_plot(sectorlst[3])
```



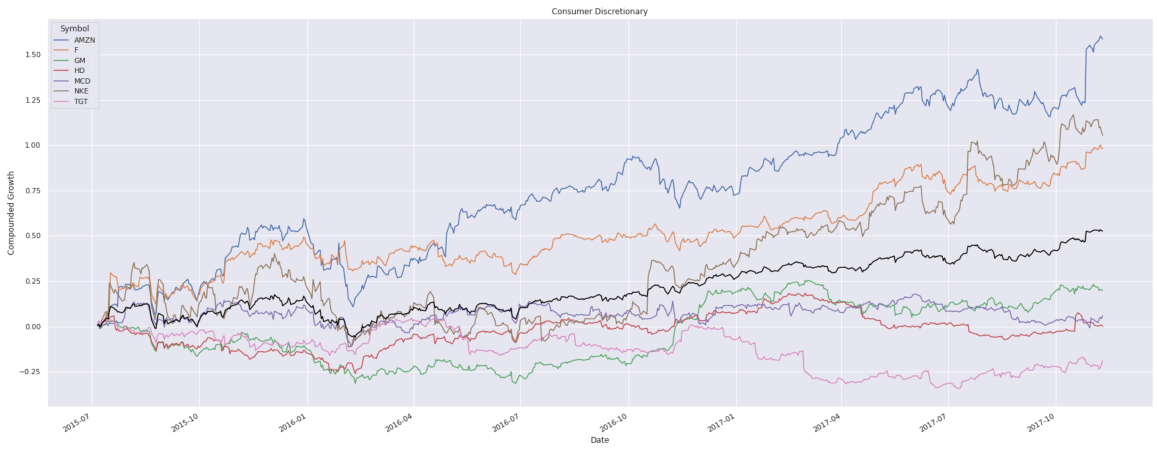
In [ ]:
```
make_sector_plot(sectorlst[4])
```



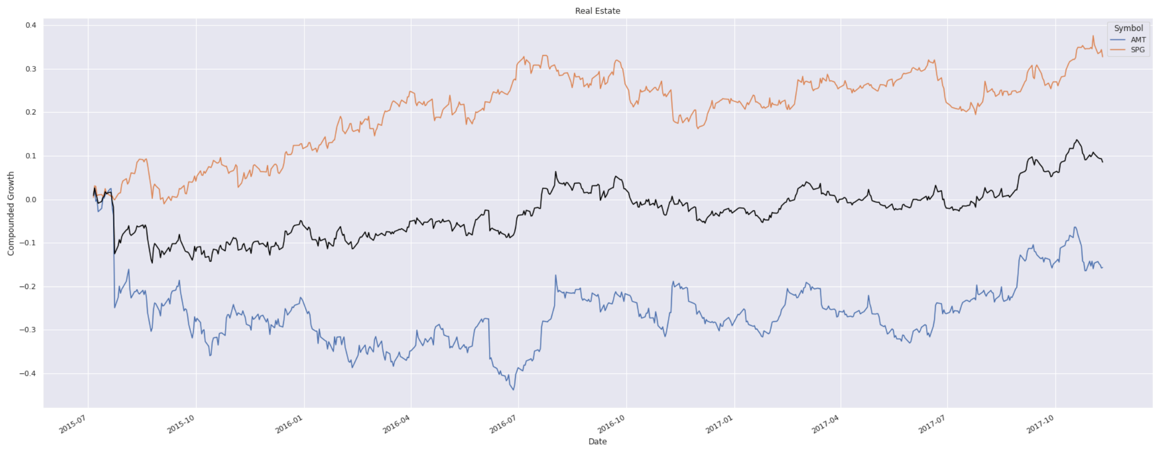
In [ ]:
```
make_sector_plot(sectorlst[5])
```



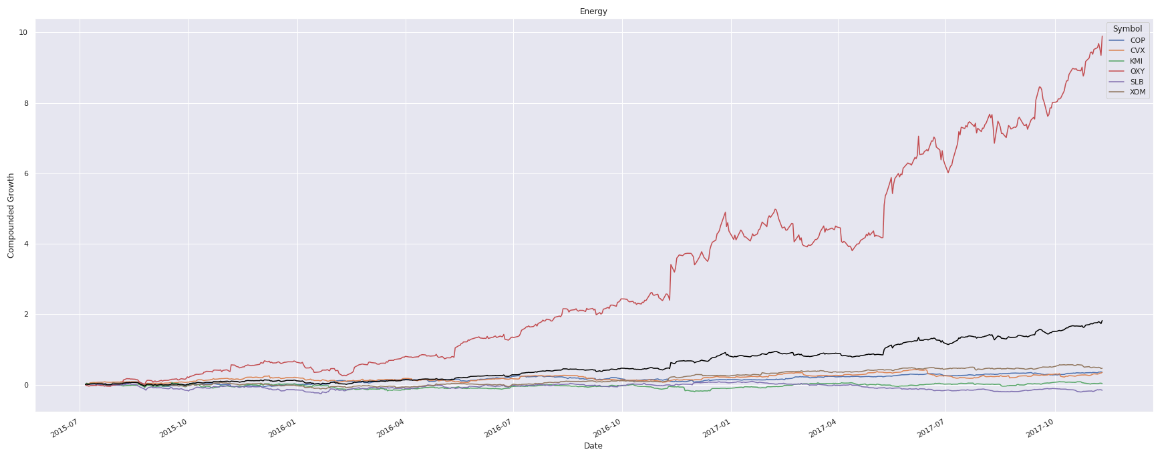
In [ ]:
```
make_sector_plot(sectorlst[6])
```



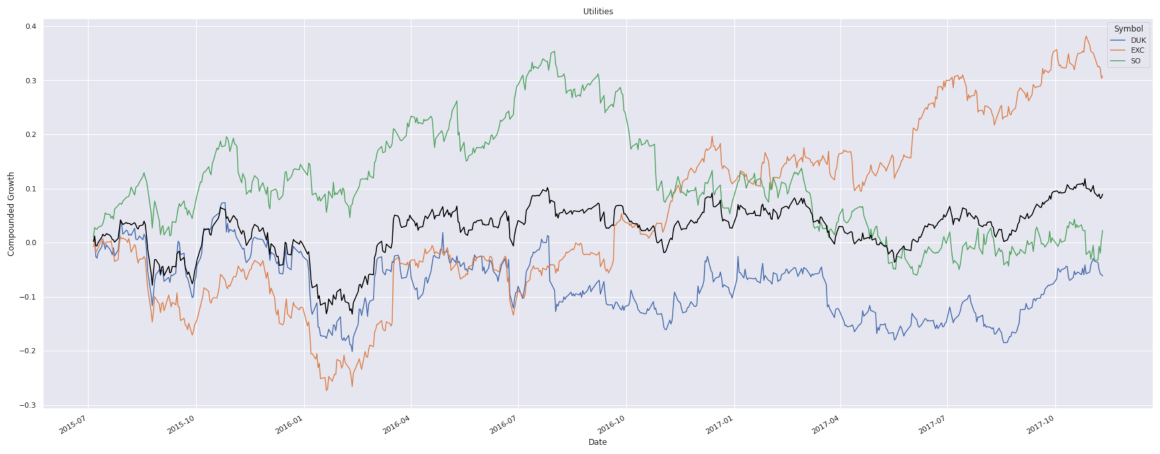
In [ ]:
```
make_sector_plot(sectorlst[7])
```



In [ ]:
```
make_sector_plot(sectorlst[8])
```



In [ ]:
```
make_sector_plot(sectorlst[9])
```



We will notice immediately that the reason behind the massive average industry growth of the energy sector, is because of the company OXY. This might indicate this company is highly profitable, or that this company is overrated. Furthermore we can notice how some companies are underperforming within their sector; this may indicate that they are losing in profitability or that they are underrated.

Additionally, we learn that the profitability of companies within different sector vary significantly. We find that utilities and real estate experience the least growth, whereas energy experiences significant growth. From this we are drawn to believe that the sector a company is from will influence the profitability of that company.

The **Volume** of stocks traded has also been noted to indicate shifts in trader interest in a company. For example, when there is a spike in **Volume** this could be due to panic that a company has loss profitability or has gained in profitability, so everyone is scrambling to buy or sell stocks. Therefore we will shall create a distribution plot of the **Growth** vs **Volume**. First we will define interval ranges of 10 million for the **Volume** of stocks traded.

In [ ]:
```
interval_range = 10000000
m_cp['Volume Range'] = m_cp['Volume'].apply(lambda x: int(x / interval_rang
e))
m_cp.head()
```

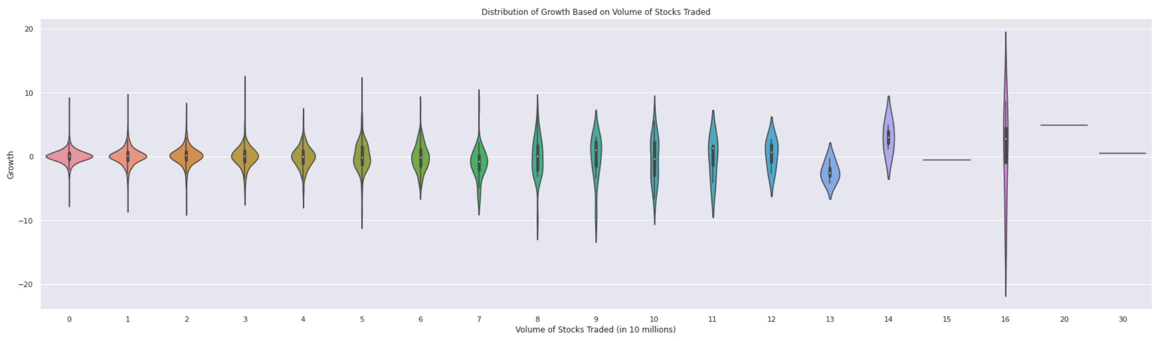
Out[ ]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Now we build distribution plots, to do this we will use python's Seaborn library. For more information on building violin plots https://seaborn.pydata.org/generated/seaborn.violinplot.html (https://seaborn.pydata.org/generated/seaborn.violinplot.html)

```
In [ ]: sns.set(rc = {'figure.figsize':(30,8)})
        ax = sns.violinplot('Volume Range', 'Growth', data = m_cp)
        ax.set(xlabel = 'Volume of Stocks Traded (in 10 millions)', ylabel = 'Growt
        h', title = 'Distribution of Growth Based on Volume of Stocks Traded')

        plt.show()
```

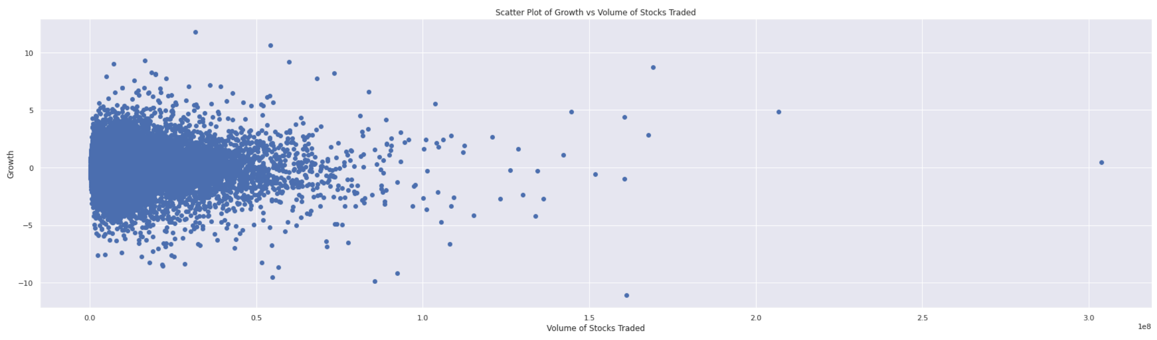
```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorators.py:43: FutureWa
rning: Pass the following variables as keyword args: x, y. From version 0.
12, the only valid positional argument will be `data`, and passing other a
rguments without an explicit keyword will result in an error or misinterpr
etation.
  FutureWarning
```



The variance and range of the distribution seems to increase as the volume of stocks traded increases. However, this may be due to fewer data points for large volume trading.

Therefore we will make a scatter plot for each individual data point. We will use the matplotlib library to do this. To learn more visit this site https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.scatter.html (https://matplotlib.org/3.3.3/api/_as_gen/matplotlib.pyplot.scatter.html)

```
In [ ]: plt.scatter('Volume', 'Growth', data = m_cp)
        plt.xlabel('Volume of Stocks Traded')
        plt.ylabel('Growth')
        plt.title('Scatter Plot of Growth vs Volume of Stocks Traded')

        plt.show()
```



As we had feared, the variation can largely be explained by the lack of data points. However, we might find a more meaningful trend when we look instead at the change in volume of stocks traded. In order to calculate this we will just shift the volumes by 1 day and subtract out the original copy.

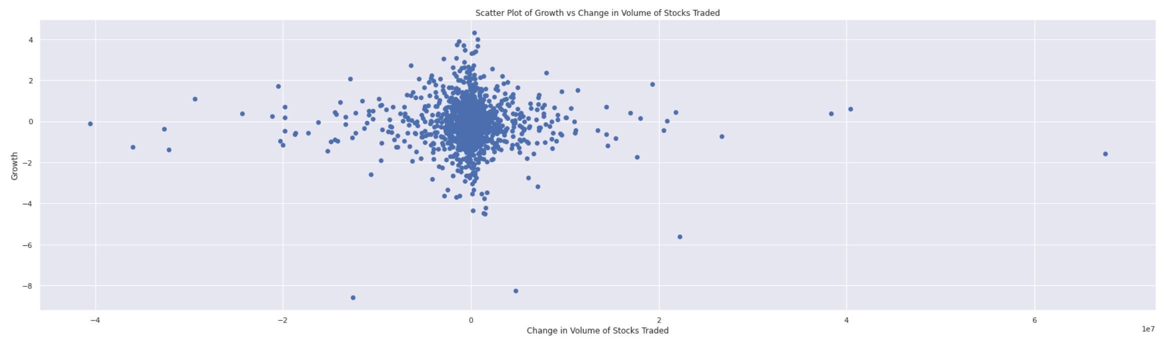
```
In [ ]: # After grouping, volume[i + 1] - volume[i] is assigned to change_volume[i
        + 1] for all i
        change_volume = m_cp.groupby('Symbol').apply(lambda x: x['Volume'].shift(1)
        - x['Volume'])
        change_volume = change_volume.reset_index()
        m_cp['Volume Change'] = change_volume['Volume']

        m_cp.head()
```

Out[ ]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Time to build the scatter plot of **Growth** vs **Volume Change**

```
In [ ]: plt.scatter('Volume Change', 'Growth', data = m_cp)
        plt.xlabel('Change in Volume of Stocks Traded')
        plt.ylabel('Growth')

        plt.title('Scatter Plot of Growth vs Change in Volume of Stocks Traded')

        plt.show()
```



The data seems to indicate the opposite of what our intuition predicted. Large change in volume of trade are typically met with low growth, excluding a few exceptions. However, the largest growth or loss are observed in those with little to no change in volume of stocks traded. From this we find that there seems to be no correlation between growth and volume.

Now to set up a precursor for later analysis, we will average out the growth across all companies, for a metric of how well we would like to perform over this period of time, when testing our model.

In order to do this we will calculate:

$$\frac{1}{\#\text{ of Companies}} \sum_{\text{All Companies}} [\text{Compounded Growth of Company}]$$

This formula should look familiar, since its the same one used to calculate the average per sector, but instead we now want the average for the S&P100.

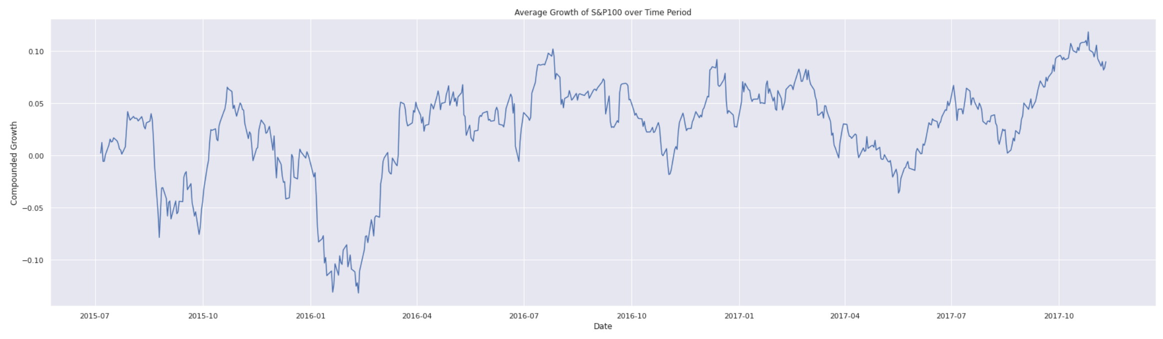
In [ ]:
```
all_growth = []

# summing of compounded growth of all companies for each day to calculate
# average industry growth
for date in daterange:
  m_sector_day = m_sector.loc[m_cp['Date'] == date]
  average_growth = m_sector_day['Compounded Growth'].sum() / len(m_sector_d
ay)
  all_growth.append(average_growth)

growthplt['All'] = all_growth
growthplt.head()
```

Out[ ]:

	Industrials	Health Care	Information Technology	Financials	Communication Services	Consumer Staples	Consumer Discretionary	
0	0.007124	0.004889	0.000709	0.004327	0.008994	0.003582	0.006643	0
1	0.007279	0.014812	0.008542	0.011577	0.016634	0.013809	0.011531	0
2	-0.013627	0.000610	-0.008218	-0.002968	0.002050	-0.000960	-0.001368	0
3	-0.015110	0.005726	-0.003634	-0.001134	0.004322	0.004014	0.004422	0
4	0.003084	0.018934	0.006026	0.011877	0.012308	0.016407	0.018734	-0

Now we plot this against time.

In [ ]:
```
plt.plot(daterange, growthplt['All'])
plt.xlabel('Date')
plt.ylabel('Compounded Growth')
plt.title('Average Growth of S&P100 over Time Period')

plt.show()
```



And the exact measure of how much the S&P100 grew over our designated time period, would just be the average **Compounded Growth** value on the end of this period.

In [ ]:
```
growthplt['All'][-1:]
```

Out[ ]:
```
595    0.089655
Name: All, dtype: float64
```

Which would be around 9%. We will also calculate the average magnitude of Growth each day, to approximate how much profit correct predictions vs incorrect predictions in the market would be.

In order to calculate this we will:

$$\frac{1}{\#\text{ of Entries}} \sum_{\text{All Entries}} [|\text{Growth of Entry}|]$$

In [ ]:
```
sum_magnitude = 0

for i in m_cp['Growth']:
  sum_magnitude += abs(i)

print('Sum magnitude of growth: ', sum_magnitude)
print('Average magnitude of growth: ', sum_magnitude / len(m_cp))
```

```
Sum magnitude of growth:  37327.34809400359
Average magnitude of growth:  0.8029458805283856
```

From this we find that for each correct prediction we expect to see an increase in profit of around 0.8%.

Analysis, hypothesis testing, & ML

In this section, we will apply a neural network to predict the opening price of any given stock given the open, high, low, closing, and volume of that stock in previous days. The model used here is based on the TensorFlow Kertas library tutorial, which is linked here: https://www.tensorflow.org/guide/keras/sequential_model (https://www.tensorflow.org/guide/keras/sequential_model)

Designing Model

Since the neural net (NN) can only take a set number of inputs, we will have to choose a limit for the backlog of stock data that we feed into the model. For now, we'll set this limit to be 20 days.

In [43]:
```
backlog = dt.timedelta(days=20)
backlog.days
```

Out[43]:
```
20
```

Now that we've set the backlog, let's consider how the data will be passed in and what sort of output we should expect. This will help us determine the shape of our model.

For input, for each company, there will be 20 days worth of data with each day containing the five metrics: open, high, low, close, and volume. This means that for each company, we'll need 100 nodes in the input layer. Taking a look at the number of companies we have:

In [46]:
```
len(m_cp['Symbol'].unique())
```

Out[46]:
```
78
```

We have 78 companies, so the input layer will need to have 7810 nodes as input. We have an extra 10 nodes to denote the industry of the company whose stock we're interested in.

For output, we'll just have 2 nodes, one holding the probability that that company's opening stock will have increased and one holding the probability that that company's opening stock did not increase.

So, based on all of that, we can build our model:

```
In [47]: model = keras.Sequential([
             keras.layers.Dense(backlog.days*5*len(m_cp['Symbol'].unique()) + 10),
         # input layer (1)
             keras.layers.Dense(len(m_cp['Symbol'].unique())*backlog.days, activatio
         n='relu'),    # hidden layer (2)
             keras.layers.Dense(2, activation='softmax')
         # output layer (3)
         ])

         model.compile(optimizer='adam',
                       loss='sparse_categorical_crossentropy',
                       metrics=['accuracy'])
```

Notice that we also have a hidden layer between our input and output layer. In this case, we just arbitrarily decided that this hidden layer should have 1560 nodes.

All the layers are densely connected, which means that every node fomr the previous layer is connected to every node in the next layer.

With this, our model is build and ready to be trained.

Preparing Data

Now that we have our model, we need to prepare our data to fit in into our model. More specifically we need to take the data from our master dataframe and split it into a set of x and y values which correspond with the input and output of the model. Let's start by first making a copy of the Master dataframe to work with:

```
In [48]: m_cp = MASTER_DF.copy()
         m_cp.head()
```

Out[48]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

All values that the NN accepts as input and outputs are within the [0,1]. But our data potentially spans the entire real number line. In order to bring everything closer to the [0,1] range, we'll create normalized versions of the columns we'll use:

In [49]:
```
m_cp['N_Open'] = (m_cp['Open'].mean() - m_cp['Open']) / m_cp['Open'].std()
m_cp['N_High'] = (m_cp['High'].mean() - m_cp['High']) / m_cp['High'].std()
m_cp['N_Low'] = (m_cp['Low'].mean() - m_cp['Low']) / m_cp['Low'].std()
m_cp['N_Close'] = (m_cp['Close'].mean() - m_cp['Close']) / m_cp['Close'].st
d()
m_cp['N_Volume'] = (m_cp['Volume'].mean() - m_cp['Volume']) / m_cp['Volum
e'].std()

m_cp.head()
```
Out[49]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Now, everything is closer to [0,1], but not completely within the acceptable range. So, we'll use a sigmoid encoding to force everything to fit within the appropriate range.

In [50]:
```
m_cp['SE_N_Open'] = 1 / (1 + np.exp(-m_cp['N_Open']))
m_cp['SE_N_High'] = 1 / (1 + np.exp(-m_cp['N_High']))
m_cp['SE_N_Low'] = 1 / (1 + np.exp(-m_cp['N_Low']))
m_cp['SE_N_Close'] = 1 / (1 + np.exp(-m_cp['N_Close']))
m_cp['SE_N_Volume'] = 1 / (1 + np.exp(-m_cp['N_Volume']))

m_cp.head()
```
Out[50]:

	Date	Open	High	Low	Close	Volume	Symbol	Security	GICS Sector	GICS Sub Industr
0	2015-07-06	145.19	146.78	145.13	146.32	2302102	MMM	3M Company	Industrials	Industria Conglomerate
1	2015-07-07	146.37	146.83	144.53	146.64	3037428	MMM	3M Company	Industrials	Industria Conglomerate
2	2015-07-08	145.38	145.78	144.07	144.08	2769709	MMM	3M Company	Industrials	Industria Conglomerate
3	2015-07-09	145.87	146.71	145.05	145.05	2193042	MMM	3M Company	Industrials	Industria Conglomerate
4	2015-07-10	146.01	147.06	145.78	146.22	2184670	MMM	3M Company	Industrials	Industria Conglomerate

Now that all the considered data is within the appropriate range, we will pivot this data by date and symbol to focus on the values we care about:

```
In [51]: m_cp_p = m_cp.pivot(index='Date', columns='Symbol', values=['SE_N_Open', 'S
         E_N_High', 'SE_N_Low', 'SE_N_Close', 'SE_N_Volume'])
         m_cp_p.head()
```

Out[51]:

	SE_N_Open								
Symbol	**AAPL**	**ABT**	**ACN**	**ADBE**	**AIG**	**ALL**	**AMGN**	**AMT**	**AMZ**
Date									
2015-07-06	0.504787	0.624515	0.552369	0.570067	0.604764	0.600008	0.463687	0.553083	0.10
2015-07-07	0.503244	0.623117	0.550245	0.568998	0.604197	0.599251	0.458804	0.551435	0.10
2015-07-08	0.505524	0.623447	0.547666	0.570182	0.604152	0.598539	0.460553	0.552011	0.10
2015-07-09	0.506530	0.623354	0.548683	0.568883	0.604181	0.597752	0.462953	0.551802	0.10
2015-07-10	0.509581	0.623607	0.548794	0.567632	0.602804	0.597305	0.460753	0.552563	0.10

5 rows × 390 columns

Now, let's filter out the corresponding x's and y's. Each y will contain either a 0 or a 1, representing the presencse of an increase in opening stock price. For each element in y, there will be a corresponding array in x of length 7810 containing the encoding for the industry and the 20 previous days of market data.

But, first, we need to pick a company from our 78 options:

```
In [52]: options = m_cp['Symbol'].unique()
         options.sort()
         options
```

```
Out[52]: array(['AAPL', 'ABT', 'ACN', 'ADBE', 'AIG', 'ALL', 'AMGN', 'AMT', 'AMZN',
                'AXP', 'BA', 'BIIB', 'BK', 'BLK', 'BMY', 'C', 'CAT', 'CHTR', 'CL',
                'CMCSA', 'COF', 'COP', 'COST', 'CRM', 'CSCO', 'CVS', 'CVX', 'DIS',
                'DUK', 'EMR', 'EXC', 'F', 'FB', 'FDX', 'GD', 'GILD', 'GM', 'GOOG',
                'GOOGL', 'GS', 'HD', 'HON', 'IBM', 'INTC', 'JNJ', 'KMI', 'KO',
                'LLY', 'LMT', 'MA', 'MCD', 'MDLZ', 'MDT', 'MMM', 'MO', 'MRK',
                'MSFT', 'NFLX', 'NKE', 'NVDA', 'ORCL', 'OXY', 'PEP', 'PFE', 'PG',
                'PYPL', 'SLB', 'SO', 'SPG', 'TGT', 'UNH', 'UNP', 'UPS', 'V', 'WBA',
                'WFC', 'WMT', 'XOM'], dtype=object)
```

We decided to just go with 'AAPL' for our tutorial.

```
In [53]: target_company = 'AAPL'
```

Here, we'll also do some prep work for encoding industries:

In [54]:
```
def encode_target_industry():
  target_industry = m_cp[m_cp['Symbol'] == target_company]['GICS  Sector'].
unique()[0]
  industries = m_cp['GICS  Sector'].unique()
  for i in range(len(industries)):
    if industries[i] == target_industry:
      break
  encoding = np.zeros(10)
  encoding[i] = 1
  return encoding
```

Now, time to prepare the x and y data from the dataframe:

In [55]:
```
xs = []
ys = []

for i in range(m_cp_p.shape[0] - 20):
  ### prepare xs
  x = np.array([])
  # handle industry stuff
  x = np.append(x, encode_target_industry())
  # handle prev 20 days data
  for j in range(20):
    params = np.append(
        [m_cp_p.iloc[i + j]['SE_N_Open']],
        np.append(
            [m_cp_p.iloc[i + j]['SE_N_High']],
            np.append(
                [m_cp_p.iloc[i + j]['SE_N_Low']],
                np.append(
                    [m_cp_p.iloc[i + j]['SE_N_Close']],
                    [m_cp_p.iloc[i + j]['SE_N_Volume']],
                )
            )
        )
    )
    x = np.append(x, params)
  xs += [x]

  # prepare ys
  if m_cp_p.iloc[i + 20]['SE_N_Open']['AAPL'] > m_cp_p.iloc[i + 19]['SE_N_C
lose']['AAPL']:
    ys += [1]
  else:
    ys += [0]

np_xs = np.array(xs)
np_ys = np.array(ys)
```

Now, let's verify that the chape for both np_xs and np_ys are correct:

In [56]:
```
np_xs.shape
```

Out[56]:
```
(576, 7810)
```

```
In [57]: np_ys.shape
```

```
Out[57]: (576,)
```

The shape is as expected and we can see that we have 576 sets of x y data. From this, we'll choose a certain portion to be training data and a certain portion to be testing data. Preferably, we would like the training data to be ~10 times larger than the testing data.

```
In [58]: split_index = math.floor(np_ys.shape[0]/11*10)

         train_xs = np_xs[:split_index]
         train_ys = np_ys[:split_index]

         test_xs = np_xs[split_index:]
         test_ys = np_ys[split_index:]
```

Now that we have our model, training data, and testing data, we can move on to fitting our model.

Fitting Model

The TensorFlow library handles all the fitting for us, and will follow the shape and optimization configurations that we established in the designing model section. The only new thing we need to input is the number of epochs, which is just the number of times the model goes through our training data in the learning process.

But first, let's verify the shape of the training sets:

```
In [59]: train_xs.shape
```

```
Out[59]: (523, 7810)
```

```
In [60]: train_ys.shape
```

```
Out[60]: (523,)
```

And now, to fit the model:

In [61]:
```
model.fit(train_xs, train_ys, epochs=80)
```

```
Epoch 1/80
17/17 [==============================] - 9s 451ms/step - loss: 71.3265 - a
ccuracy: 0.5063
Epoch 2/80
17/17 [==============================] - 8s 460ms/step - loss: 17.4929 - a
ccuracy: 0.4826
Epoch 3/80
17/17 [==============================] - 8s 461ms/step - loss: 17.3955 - a
ccuracy: 0.5284
Epoch 4/80
17/17 [==============================] - 8s 466ms/step - loss: 10.8554 - a
ccuracy: 0.5544
Epoch 5/80
17/17 [==============================] - 8s 455ms/step - loss: 1.9552 - ac
curacy: 0.4969
Epoch 6/80
17/17 [==============================] - 8s 454ms/step - loss: 1.9658 - ac
curacy: 0.5171
Epoch 7/80
17/17 [==============================] - 8s 454ms/step - loss: 1.1666 - ac
curacy: 0.5004
Epoch 8/80
17/17 [==============================] - 8s 457ms/step - loss: 1.1108 - ac
curacy: 0.5265
Epoch 9/80
17/17 [==============================] - 8s 457ms/step - loss: 0.9079 - ac
curacy: 0.5008
Epoch 10/80
17/17 [==============================] - 8s 455ms/step - loss: 0.8271 - ac
curacy: 0.5131
Epoch 11/80
17/17 [==============================] - 8s 456ms/step - loss: 0.9121 - ac
curacy: 0.5342
Epoch 12/80
17/17 [==============================] - 8s 453ms/step - loss: 0.9263 - ac
curacy: 0.4838
Epoch 13/80
17/17 [==============================] - 8s 457ms/step - loss: 0.8020 - ac
curacy: 0.5104
Epoch 14/80
17/17 [==============================] - 8s 455ms/step - loss: 0.9597 - ac
curacy: 0.5154
Epoch 15/80
17/17 [==============================] - 8s 456ms/step - loss: 0.8316 - ac
curacy: 0.5795
Epoch 16/80
17/17 [==============================] - 8s 453ms/step - loss: 0.7835 - ac
curacy: 0.4729
Epoch 17/80
17/17 [==============================] - 8s 456ms/step - loss: 1.1001 - ac
curacy: 0.4773
Epoch 18/80
17/17 [==============================] - 8s 461ms/step - loss: 1.0517 - ac
curacy: 0.4342
Epoch 19/80
17/17 [==============================] - 8s 460ms/step - loss: 1.2229 - ac
curacy: 0.4534
Epoch 20/80
17/17 [==============================] - 8s 455ms/step - loss: 1.3041 - ac
curacy: 0.5077
Epoch 21/80
```

```
17/17 [==============================] - 8s 459ms/step - loss: 21.4450 - a
ccuracy: 0.5416
Epoch 22/80
17/17 [==============================] - 8s 456ms/step - loss: 229.5566 -
accuracy: 0.5707
Epoch 23/80
17/17 [==============================] - 8s 457ms/step - loss: 31.5583 - a
ccuracy: 0.4543
Epoch 24/80
17/17 [==============================] - 8s 453ms/step - loss: 6.4483 - ac
curacy: 0.5169
Epoch 25/80
17/17 [==============================] - 8s 453ms/step - loss: 5.5663 - ac
curacy: 0.5002
Epoch 26/80
17/17 [==============================] - 8s 453ms/step - loss: 4.4033 - ac
curacy: 0.5012
Epoch 27/80
17/17 [==============================] - 8s 455ms/step - loss: 1.6212 - ac
curacy: 0.5320
Epoch 28/80
17/17 [==============================] - 8s 452ms/step - loss: 0.9008 - ac
curacy: 0.4545
Epoch 29/80
17/17 [==============================] - 8s 452ms/step - loss: 0.7669 - ac
curacy: 0.5117
Epoch 30/80
17/17 [==============================] - 8s 452ms/step - loss: 0.7713 - ac
curacy: 0.5385
Epoch 31/80
17/17 [==============================] - 8s 452ms/step - loss: 0.7505 - ac
curacy: 0.4967
Epoch 32/80
17/17 [==============================] - 8s 451ms/step - loss: 0.7907 - ac
curacy: 0.4894
Epoch 33/80
17/17 [==============================] - 8s 452ms/step - loss: 0.7838 - ac
curacy: 0.4896
Epoch 34/80
17/17 [==============================] - 8s 454ms/step - loss: 0.7398 - ac
curacy: 0.5267
Epoch 35/80
17/17 [==============================] - 8s 455ms/step - loss: 0.7137 - ac
curacy: 0.4873
Epoch 36/80
17/17 [==============================] - 8s 456ms/step - loss: 0.7011 - ac
curacy: 0.4634
Epoch 37/80
17/17 [==============================] - 8s 453ms/step - loss: 0.6976 - ac
curacy: 0.5518
Epoch 38/80
17/17 [==============================] - 8s 452ms/step - loss: 0.7010 - ac
curacy: 0.5576
Epoch 39/80
17/17 [==============================] - 8s 453ms/step - loss: 0.6975 - ac
curacy: 0.5419
Epoch 40/80
17/17 [==============================] - 8s 456ms/step - loss: 0.6930 - ac
curacy: 0.5447
Epoch 41/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6995 - ac
```

```
curacy: 0.5379
Epoch 42/80
17/17 [==============================] - 8s 456ms/step - loss: 0.6908 - ac
curacy: 0.5394
Epoch 43/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6902 - ac
curacy: 0.5442
Epoch 44/80
17/17 [==============================] - 8s 466ms/step - loss: 0.6940 - ac
curacy: 0.5145
Epoch 45/80
17/17 [==============================] - 8s 464ms/step - loss: 0.7025 - ac
curacy: 0.5092
Epoch 46/80
17/17 [==============================] - 8s 454ms/step - loss: 0.7091 - ac
curacy: 0.5112
Epoch 47/80
17/17 [==============================] - 8s 458ms/step - loss: 0.6901 - ac
curacy: 0.5696
Epoch 48/80
17/17 [==============================] - 8s 458ms/step - loss: 0.7000 - ac
curacy: 0.4955
Epoch 49/80
17/17 [==============================] - 8s 458ms/step - loss: 0.6846 - ac
curacy: 0.5673
Epoch 50/80
17/17 [==============================] - 8s 460ms/step - loss: 0.6954 - ac
curacy: 0.5052
Epoch 51/80
17/17 [==============================] - 8s 458ms/step - loss: 0.6979 - ac
curacy: 0.5026
Epoch 52/80
17/17 [==============================] - 8s 459ms/step - loss: 0.6886 - ac
curacy: 0.5519
Epoch 53/80
17/17 [==============================] - 8s 456ms/step - loss: 0.6900 - ac
curacy: 0.5322
Epoch 54/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6854 - ac
curacy: 0.5644
Epoch 55/80
17/17 [==============================] - 8s 455ms/step - loss: 0.6929 - ac
curacy: 0.5251
Epoch 56/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6933 - ac
curacy: 0.5355
Epoch 57/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6884 - ac
curacy: 0.5507
Epoch 58/80
17/17 [==============================] - 8s 474ms/step - loss: 0.6863 - ac
curacy: 0.5599
Epoch 59/80
17/17 [==============================] - 8s 456ms/step - loss: 0.6873 - ac
curacy: 0.5628
Epoch 60/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6919 - ac
curacy: 0.5332
Epoch 61/80
17/17 [==============================] - 8s 456ms/step - loss: 0.6892 - ac
curacy: 0.5454
```

```
Epoch 62/80
17/17 [==============================] - 8s 458ms/step - loss: 0.6896 - ac
curacy: 0.5410
Epoch 63/80
17/17 [==============================] - 8s 456ms/step - loss: 0.6912 - ac
curacy: 0.5324
Epoch 64/80
17/17 [==============================] - 8s 458ms/step - loss: 0.6857 - ac
curacy: 0.5661
Epoch 65/80
17/17 [==============================] - 8s 457ms/step - loss: 0.6968 - ac
curacy: 0.5073
Epoch 66/80
17/17 [==============================] - 8s 459ms/step - loss: 0.6903 - ac
curacy: 0.5331
Epoch 67/80
17/17 [==============================] - 8s 458ms/step - loss: 0.6884 - ac
curacy: 0.5519
Epoch 68/80
17/17 [==============================] - 8s 455ms/step - loss: 0.6981 - ac
curacy: 0.5032
Epoch 69/80
17/17 [==============================] - 8s 458ms/step - loss: 0.6914 - ac
curacy: 0.5351
Epoch 70/80
17/17 [==============================] - 8s 456ms/step - loss: 0.6865 - ac
curacy: 0.5604
Epoch 71/80
17/17 [==============================] - 8s 453ms/step - loss: 0.6860 - ac
curacy: 0.5614
Epoch 72/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6885 - ac
curacy: 0.5519
Epoch 73/80
17/17 [==============================] - 8s 455ms/step - loss: 0.6906 - ac
curacy: 0.5348
Epoch 74/80
17/17 [==============================] - 8s 453ms/step - loss: 0.6917 - ac
curacy: 0.5319
Epoch 75/80
17/17 [==============================] - 8s 455ms/step - loss: 0.6898 - ac
curacy: 0.5431
Epoch 76/80
17/17 [==============================] - 8s 451ms/step - loss: 0.6864 - ac
curacy: 0.5589
Epoch 77/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6898 - ac
curacy: 0.5418
Epoch 78/80
17/17 [==============================] - 8s 455ms/step - loss: 0.6897 - ac
curacy: 0.5407
Epoch 79/80
17/17 [==============================] - 8s 454ms/step - loss: 0.6862 - ac
curacy: 0.5590
Epoch 80/80
17/17 [==============================] - 8s 455ms/step - loss: 0.6868 - ac
curacy: 0.5600
```

```
Out[61]: <tensorflow.python.keras.callbacks.History at 0x7f289c7f46d8>
```

Evaluating Model

We will run a rudimentary test on our testing data to measure how well our model predicts stocks.

```
In [62]: test_loss, test_acc = model.evaluate(test_xs,  test_ys, verbose=1)
         print('Test accuracy:', test_acc)
```

```
2/2 [==============================] - 0s 117ms/step - loss: 0.6857 - accu
racy: 0.5660
Test accuracy: 0.5660377144813538
```

Although our accuracy is pretty decent of around 0.566, this may be due to chance. It should be noted that if the model can predict with accuracy of around 0.566 then that means there are approximately 13 more correct predictions than incorrect prediction for every 100 prediction. Then we should expect to see around 95 additional correct predictions for AAPL over this 2 year period. If we assume correct and incorrect predictions yield approximately equal gain/loss. Then with an average gain of 0.8% per prediction, we calculate (1.008)^95 is approximately 2.132. This far exceeds the market of only 1.09. If we take into account a high tax rate of around 20% on our profit. Then we'll get an average yield of 0.64%, so a net profit of around (1.0064)^95 or approximately 1.833. In theory we would have beaten the market, but it is very likely that we just got lucky in this instance. The formula for approximating the profitability based on accuracy was used:

$$(1 + (\text{Approximate Growth} * (1 - \text{Tax Rate}))^{((2 * \text{Accuracy} - 1) * \#\text{Days})}$$

For now we will calculate a 10-fold validation test. For more information about K-Fold Validation visit this https://machinelearningmastery.com/k-fold-cross-validation/ (https://machinelearningmastery.com/k-fold-cross-validation/)

In [63]:
```
number_folds = 10

# # of iterations we train our model over the provided training set
number_epochs = 20

kfolds = np.array_split(np_xs, number_folds)

# slices represents the index for how the k folds will be partitioned
slices = [0]
index = 0
for i in range(10):
  index += len(kfolds[i])
  slices.append(index)

scorelst = np.empty(number_folds)

# runs the k iterations
for i in range(number_folds):
  training_data = np.append(np_xs[:slices[1]], np_xs[slices[i + 1]:], axis
= 0)
  training_target = np.append(np_ys[:slices[1]], np_ys[slices[i + 1]:], axi
s = 0)

  testing_data = np_xs[slices[i]:slices[i + 1]]
  testing_target = np_ys[slices[i]:slices[i + 1]]

  model.fit(training_data, training_target, epochs = 20)

  loss, score = model.evaluate(testing_data, testing_target, verbose = 1)
  scorelst[i] = score

for i in range(len(kfolds)):
  print(f'Score of {i}th fold' , scorelst[i])

print('Average accuracy of Model: ', scorelst.mean())
```

```
Epoch 1/20
18/18 [==============================] - 9s 462ms/step - loss: 0.6902 - ac
curacy: 0.5451
Epoch 2/20
18/18 [==============================] - 8s 462ms/step - loss: 0.6897 - ac
curacy: 0.5451
Epoch 3/20
18/18 [==============================] - 8s 462ms/step - loss: 0.6892 - ac
curacy: 0.5451
Epoch 4/20
18/18 [==============================] - 9s 475ms/step - loss: 0.6897 - ac
curacy: 0.5451
Epoch 5/20
18/18 [==============================] - 8s 458ms/step - loss: 0.6893 - ac
curacy: 0.5451
Epoch 6/20
18/18 [==============================] - 8s 458ms/step - loss: 0.6904 - ac
curacy: 0.5451
Epoch 7/20
18/18 [==============================] - 8s 459ms/step - loss: 0.6901 - ac
curacy: 0.5451
Epoch 8/20
18/18 [==============================] - 8s 460ms/step - loss: 0.6898 - ac
curacy: 0.5451
Epoch 9/20
18/18 [==============================] - 8s 457ms/step - loss: 0.6888 - ac
curacy: 0.5451
Epoch 10/20
18/18 [==============================] - 8s 461ms/step - loss: 0.6896 - ac
curacy: 0.5451
Epoch 11/20
18/18 [==============================] - 8s 457ms/step - loss: 0.6894 - ac
curacy: 0.5451
Epoch 12/20
18/18 [==============================] - 8s 457ms/step - loss: 0.6905 - ac
curacy: 0.5451
Epoch 13/20
18/18 [==============================] - 8s 458ms/step - loss: 0.6939 - ac
curacy: 0.5451
Epoch 14/20
18/18 [==============================] - 8s 457ms/step - loss: 0.6911 - ac
curacy: 0.5451
Epoch 15/20
18/18 [==============================] - 8s 461ms/step - loss: 0.6903 - ac
curacy: 0.5451
Epoch 16/20
18/18 [==============================] - 8s 467ms/step - loss: 0.6887 - ac
curacy: 0.5451
Epoch 17/20
18/18 [==============================] - 8s 470ms/step - loss: 0.6898 - ac
curacy: 0.5451
Epoch 18/20
18/18 [==============================] - 8s 458ms/step - loss: 0.6900 - ac
curacy: 0.5451
Epoch 19/20
18/18 [==============================] - 8s 460ms/step - loss: 0.6902 - ac
curacy: 0.5451
Epoch 20/20
18/18 [==============================] - 8s 458ms/step - loss: 0.6898 - ac
curacy: 0.5451
2/2 [==============================] - 0s 120ms/step - loss: 0.7097 - accu
```

```
racy: 0.4483
Epoch 1/20
17/17 [==============================] - 8s 456ms/step - loss: 0.6896 - ac
curacy: 0.5425
Epoch 2/20
17/17 [==============================] - 8s 458ms/step - loss: 0.6907 - ac
curacy: 0.5425
Epoch 3/20
17/17 [==============================] - 8s 457ms/step - loss: 0.6926 - ac
curacy: 0.5425
Epoch 4/20
17/17 [==============================] - 8s 455ms/step - loss: 0.6910 - ac
curacy: 0.5425
Epoch 5/20
17/17 [==============================] - 8s 456ms/step - loss: 0.6894 - ac
curacy: 0.5425
Epoch 6/20
17/17 [==============================] - 8s 456ms/step - loss: 0.6906 - ac
curacy: 0.5425
Epoch 7/20
17/17 [==============================] - 8s 457ms/step - loss: 0.6901 - ac
curacy: 0.5425
Epoch 8/20
17/17 [==============================] - 8s 456ms/step - loss: 0.6902 - ac
curacy: 0.5425
Epoch 9/20
17/17 [==============================] - 8s 459ms/step - loss: 0.6899 - ac
curacy: 0.5425
Epoch 10/20
17/17 [==============================] - 8s 457ms/step - loss: 0.6895 - ac
curacy: 0.5425
Epoch 11/20
17/17 [==============================] - 8s 456ms/step - loss: 0.6899 - ac
curacy: 0.5425
Epoch 12/20
17/17 [==============================] - 8s 459ms/step - loss: 0.6901 - ac
curacy: 0.5425
Epoch 13/20
17/17 [==============================] - 8s 457ms/step - loss: 0.6930 - ac
curacy: 0.5425
Epoch 14/20
17/17 [==============================] - 8s 458ms/step - loss: 0.6912 - ac
curacy: 0.5425
Epoch 15/20
17/17 [==============================] - 8s 458ms/step - loss: 0.6909 - ac
curacy: 0.5425
Epoch 16/20
17/17 [==============================] - 8s 458ms/step - loss: 0.6898 - ac
curacy: 0.5425
Epoch 17/20
17/17 [==============================] - 8s 459ms/step - loss: 0.6900 - ac
curacy: 0.5425
Epoch 18/20
17/17 [==============================] - 8s 460ms/step - loss: 0.6907 - ac
curacy: 0.5425
Epoch 19/20
17/17 [==============================] - 8s 457ms/step - loss: 0.6909 - ac
curacy: 0.5425
Epoch 20/20
17/17 [==============================] - 8s 457ms/step - loss: 0.6897 - ac
curacy: 0.5425
```

```
2/2 [==============================] - 0s 119ms/step - loss: 0.6857 - accu
racy: 0.5690
Epoch 1/20
15/15 [==============================] - 7s 454ms/step - loss: 0.6904 - ac
curacy: 0.5391
Epoch 2/20
15/15 [==============================] - 7s 460ms/step - loss: 0.6901 - ac
curacy: 0.5391
Epoch 3/20
15/15 [==============================] - 7s 462ms/step - loss: 0.6904 - ac
curacy: 0.5391
Epoch 4/20
15/15 [==============================] - 7s 467ms/step - loss: 0.6901 - ac
curacy: 0.5391
Epoch 5/20
15/15 [==============================] - 7s 465ms/step - loss: 0.6904 - ac
curacy: 0.5391
Epoch 6/20
15/15 [==============================] - 7s 455ms/step - loss: 0.6907 - ac
curacy: 0.5391
Epoch 7/20
15/15 [==============================] - 7s 458ms/step - loss: 0.6902 - ac
curacy: 0.5391
Epoch 8/20
15/15 [==============================] - 7s 455ms/step - loss: 0.6913 - ac
curacy: 0.5391
Epoch 9/20
15/15 [==============================] - 7s 456ms/step - loss: 0.6904 - ac
curacy: 0.5391
Epoch 10/20
15/15 [==============================] - 7s 455ms/step - loss: 0.6905 - ac
curacy: 0.5391
Epoch 11/20
15/15 [==============================] - 7s 456ms/step - loss: 0.6904 - ac
curacy: 0.5391
Epoch 12/20
15/15 [==============================] - 7s 452ms/step - loss: 0.6901 - ac
curacy: 0.5391
Epoch 13/20
15/15 [==============================] - 7s 454ms/step - loss: 0.6904 - ac
curacy: 0.5391
Epoch 14/20
15/15 [==============================] - 7s 453ms/step - loss: 0.6906 - ac
curacy: 0.5391
Epoch 15/20
15/15 [==============================] - 7s 453ms/step - loss: 0.6903 - ac
curacy: 0.5391
Epoch 16/20
15/15 [==============================] - 7s 455ms/step - loss: 0.6904 - ac
curacy: 0.5391
Epoch 17/20
15/15 [==============================] - 7s 455ms/step - loss: 0.6907 - ac
curacy: 0.5391
Epoch 18/20
15/15 [==============================] - 7s 461ms/step - loss: 0.6903 - ac
curacy: 0.5391
Epoch 19/20
15/15 [==============================] - 7s 455ms/step - loss: 0.6901 - ac
curacy: 0.5391
Epoch 20/20
15/15 [==============================] - 7s 455ms/step - loss: 0.6906 - ac
```

```
curacy: 0.5391
2/2 [==============================] - 0s 127ms/step - loss: 0.6850 - accu
racy: 0.5690
Epoch 1/20
13/13 [==============================] - 6s 453ms/step - loss: 0.6916 - ac
curacy: 0.5323
Epoch 2/20
13/13 [==============================] - 6s 453ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 3/20
13/13 [==============================] - 6s 453ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 4/20
13/13 [==============================] - 6s 456ms/step - loss: 0.6910 - ac
curacy: 0.5323
Epoch 5/20
13/13 [==============================] - 6s 456ms/step - loss: 0.6910 - ac
curacy: 0.5323
Epoch 6/20
13/13 [==============================] - 6s 452ms/step - loss: 0.6913 - ac
curacy: 0.5323
Epoch 7/20
13/13 [==============================] - 6s 455ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 8/20
13/13 [==============================] - 6s 453ms/step - loss: 0.6908 - ac
curacy: 0.5323
Epoch 9/20
13/13 [==============================] - 6s 454ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 10/20
13/13 [==============================] - 6s 458ms/step - loss: 0.6911 - ac
curacy: 0.5323
Epoch 11/20
13/13 [==============================] - 6s 453ms/step - loss: 0.6924 - ac
curacy: 0.5323
Epoch 12/20
13/13 [==============================] - 6s 455ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 13/20
13/13 [==============================] - 6s 455ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 14/20
13/13 [==============================] - 6s 456ms/step - loss: 0.6911 - ac
curacy: 0.5323
Epoch 15/20
13/13 [==============================] - 6s 454ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 16/20
13/13 [==============================] - 6s 453ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 17/20
13/13 [==============================] - 6s 457ms/step - loss: 0.6911 - ac
curacy: 0.5323
Epoch 18/20
13/13 [==============================] - 6s 452ms/step - loss: 0.6911 - ac
curacy: 0.5323
Epoch 19/20
13/13 [==============================] - 6s 455ms/step - loss: 0.6912 - ac
curacy: 0.5323
Epoch 20/20
```

```
13/13 [==============================] - 6s 454ms/step - loss: 0.6911 - ac
curacy: 0.5323
2/2 [==============================] - 0s 119ms/step - loss: 0.6850 - accu
racy: 0.5862
Epoch 1/20
11/11 [==============================] - 5s 453ms/step - loss: 0.6905 - ac
curacy: 0.5378
Epoch 2/20
11/11 [==============================] - 5s 454ms/step - loss: 0.6905 - ac
curacy: 0.5378
Epoch 3/20
11/11 [==============================] - 5s 453ms/step - loss: 0.6905 - ac
curacy: 0.5378
Epoch 4/20
11/11 [==============================] - 5s 455ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 5/20
11/11 [==============================] - 5s 455ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 6/20
11/11 [==============================] - 5s 456ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 7/20
11/11 [==============================] - 5s 455ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 8/20
11/11 [==============================] - 5s 454ms/step - loss: 0.6905 - ac
curacy: 0.5378
Epoch 9/20
11/11 [==============================] - 5s 457ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 10/20
11/11 [==============================] - 5s 451ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 11/20
11/11 [==============================] - 5s 457ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 12/20
11/11 [==============================] - 5s 455ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 13/20
11/11 [==============================] - 5s 455ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 14/20
11/11 [==============================] - 5s 454ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 15/20
11/11 [==============================] - 5s 455ms/step - loss: 0.6903 - ac
curacy: 0.5378
Epoch 16/20
11/11 [==============================] - 5s 458ms/step - loss: 0.6903 - ac
curacy: 0.5378
Epoch 17/20
11/11 [==============================] - 5s 467ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 18/20
11/11 [==============================] - 5s 467ms/step - loss: 0.6904 - ac
curacy: 0.5378
Epoch 19/20
11/11 [==============================] - 5s 467ms/step - loss: 0.6903 - ac
curacy: 0.5378
```

```
Epoch 20/20
11/11 [==============================] - 5s 467ms/step - loss: 0.6903 - ac
curacy: 0.5378
2/2 [==============================] - 0s 123ms/step - loss: 0.6955 - accu
racy: 0.5000
Epoch 1/20
9/9 [==============================] - 4s 458ms/step - loss: 0.6873 - accu
racy: 0.5594
Epoch 2/20
9/9 [==============================] - 4s 457ms/step - loss: 0.6873 - accu
racy: 0.5594
Epoch 3/20
9/9 [==============================] - 4s 457ms/step - loss: 0.6872 - accu
racy: 0.5594
Epoch 4/20
9/9 [==============================] - 4s 457ms/step - loss: 0.6872 - accu
racy: 0.5594
Epoch 5/20
9/9 [==============================] - 4s 457ms/step - loss: 0.6871 - accu
racy: 0.5594
Epoch 6/20
9/9 [==============================] - 4s 454ms/step - loss: 0.6870 - accu
racy: 0.5594
Epoch 7/20
9/9 [==============================] - 4s 455ms/step - loss: 0.6869 - accu
racy: 0.5594
Epoch 8/20
9/9 [==============================] - 4s 458ms/step - loss: 0.6869 - accu
racy: 0.5594
Epoch 9/20
9/9 [==============================] - 4s 455ms/step - loss: 0.6868 - accu
racy: 0.5594
Epoch 10/20
9/9 [==============================] - 4s 455ms/step - loss: 0.6867 - accu
racy: 0.5594
Epoch 11/20
9/9 [==============================] - 4s 457ms/step - loss: 0.6867 - accu
racy: 0.5594
Epoch 12/20
9/9 [==============================] - 4s 454ms/step - loss: 0.6867 - accu
racy: 0.5594
Epoch 13/20
9/9 [==============================] - 4s 459ms/step - loss: 0.6866 - accu
racy: 0.5594
Epoch 14/20
9/9 [==============================] - 4s 454ms/step - loss: 0.6865 - accu
racy: 0.5594
Epoch 15/20
9/9 [==============================] - 4s 457ms/step - loss: 0.6865 - accu
racy: 0.5594
Epoch 16/20
9/9 [==============================] - 4s 456ms/step - loss: 0.6865 - accu
racy: 0.5594
Epoch 17/20
9/9 [==============================] - 4s 468ms/step - loss: 0.6865 - accu
racy: 0.5594
Epoch 18/20
9/9 [==============================] - 4s 463ms/step - loss: 0.6864 - accu
racy: 0.5594
Epoch 19/20
9/9 [==============================] - 4s 458ms/step - loss: 0.6864 - accu
```

```
racy: 0.5594
Epoch 20/20
9/9 [==============================] - 4s 457ms/step - loss: 0.6864 - accu
racy: 0.5594
2/2 [==============================] - 0s 120ms/step - loss: 0.7109 - accu
racy: 0.4310
Epoch 1/20
8/8 [==============================] - 4s 454ms/step - loss: 0.6831 - accu
racy: 0.5764
Epoch 2/20
8/8 [==============================] - 4s 452ms/step - loss: 0.6830 - accu
racy: 0.5764
Epoch 3/20
8/8 [==============================] - 4s 458ms/step - loss: 0.6829 - accu
racy: 0.5764
Epoch 4/20
8/8 [==============================] - 4s 459ms/step - loss: 0.6829 - accu
racy: 0.5764
Epoch 5/20
8/8 [==============================] - 4s 455ms/step - loss: 0.6828 - accu
racy: 0.5764
Epoch 6/20
8/8 [==============================] - 4s 460ms/step - loss: 0.6827 - accu
racy: 0.5764
Epoch 7/20
8/8 [==============================] - 4s 458ms/step - loss: 0.6827 - accu
racy: 0.5764
Epoch 8/20
8/8 [==============================] - 4s 457ms/step - loss: 0.6827 - accu
racy: 0.5764
Epoch 9/20
8/8 [==============================] - 4s 455ms/step - loss: 0.6826 - accu
racy: 0.5764
Epoch 10/20
8/8 [==============================] - 4s 456ms/step - loss: 0.6827 - accu
racy: 0.5764
Epoch 11/20
8/8 [==============================] - 4s 457ms/step - loss: 0.6824 - accu
racy: 0.5764
Epoch 12/20
8/8 [==============================] - 4s 457ms/step - loss: 0.6823 - accu
racy: 0.5764
Epoch 13/20
8/8 [==============================] - 4s 460ms/step - loss: 0.6822 - accu
racy: 0.5764
Epoch 14/20
8/8 [==============================] - 4s 460ms/step - loss: 0.6822 - accu
racy: 0.5764
Epoch 15/20
8/8 [==============================] - 4s 458ms/step - loss: 0.6821 - accu
racy: 0.5764
Epoch 16/20
8/8 [==============================] - 4s 454ms/step - loss: 0.6821 - accu
racy: 0.5764
Epoch 17/20
8/8 [==============================] - 4s 458ms/step - loss: 0.6820 - accu
racy: 0.5764
Epoch 18/20
8/8 [==============================] - 4s 452ms/step - loss: 0.6820 - accu
racy: 0.5764
Epoch 19/20
```

```
8/8 [==============================] - 4s 459ms/step - loss: 0.6820 - accu
racy: 0.5764
Epoch 20/20
8/8 [==============================] - 4s 456ms/step - loss: 0.6819 - accu
racy: 0.5764
2/2 [==============================] - 0s 126ms/step - loss: 0.7031 - accu
racy: 0.4912
Epoch 1/20
6/6 [==============================] - 3s 443ms/step - loss: 0.6922 - accu
racy: 0.5349
Epoch 2/20
6/6 [==============================] - 3s 444ms/step - loss: 0.6923 - accu
racy: 0.5349
Epoch 3/20
6/6 [==============================] - 3s 446ms/step - loss: 0.6922 - accu
racy: 0.5349
Epoch 4/20
6/6 [==============================] - 3s 444ms/step - loss: 0.6921 - accu
racy: 0.5349
Epoch 5/20
6/6 [==============================] - 3s 439ms/step - loss: 0.6920 - accu
racy: 0.5349
Epoch 6/20
6/6 [==============================] - 3s 445ms/step - loss: 0.6920 - accu
racy: 0.5349
Epoch 7/20
6/6 [==============================] - 3s 445ms/step - loss: 0.6918 - accu
racy: 0.5349
Epoch 8/20
6/6 [==============================] - 3s 445ms/step - loss: 0.6918 - accu
racy: 0.5349
Epoch 9/20
6/6 [==============================] - 3s 443ms/step - loss: 0.6917 - accu
racy: 0.5349
Epoch 10/20
6/6 [==============================] - 3s 444ms/step - loss: 0.6916 - accu
racy: 0.5349
Epoch 11/20
6/6 [==============================] - 3s 441ms/step - loss: 0.6916 - accu
racy: 0.5349
Epoch 12/20
6/6 [==============================] - 3s 439ms/step - loss: 0.6916 - accu
racy: 0.5349
Epoch 13/20
6/6 [==============================] - 3s 444ms/step - loss: 0.6915 - accu
racy: 0.5349
Epoch 14/20
6/6 [==============================] - 3s 448ms/step - loss: 0.6915 - accu
racy: 0.5349
Epoch 15/20
6/6 [==============================] - 3s 442ms/step - loss: 0.6914 - accu
racy: 0.5349
Epoch 16/20
6/6 [==============================] - 3s 442ms/step - loss: 0.6914 - accu
racy: 0.5349
Epoch 17/20
6/6 [==============================] - 3s 443ms/step - loss: 0.6914 - accu
racy: 0.5349
Epoch 18/20
6/6 [==============================] - 3s 447ms/step - loss: 0.6914 - accu
racy: 0.5349
```

```
Epoch 19/20
6/6 [==============================] - 3s 442ms/step - loss: 0.6913 - accu
racy: 0.5349
Epoch 20/20
6/6 [==============================] - 3s 440ms/step - loss: 0.6913 - accu
racy: 0.5349
2/2 [==============================] - 0s 121ms/step - loss: 0.6570 - accu
racy: 0.7018
Epoch 1/20
4/4 [==============================] - 2s 444ms/step - loss: 0.6957 - accu
racy: 0.5130
Epoch 2/20
4/4 [==============================] - 2s 449ms/step - loss: 0.6957 - accu
racy: 0.5130
Epoch 3/20
4/4 [==============================] - 2s 440ms/step - loss: 0.6956 - accu
racy: 0.5130
Epoch 4/20
4/4 [==============================] - 2s 442ms/step - loss: 0.6955 - accu
racy: 0.5130
Epoch 5/20
4/4 [==============================] - 2s 444ms/step - loss: 0.6955 - accu
racy: 0.5130
Epoch 6/20
4/4 [==============================] - 2s 442ms/step - loss: 0.6954 - accu
racy: 0.5130
Epoch 7/20
4/4 [==============================] - 2s 446ms/step - loss: 0.6953 - accu
racy: 0.5130
Epoch 8/20
4/4 [==============================] - 2s 443ms/step - loss: 0.6952 - accu
racy: 0.5130
Epoch 9/20
4/4 [==============================] - 2s 447ms/step - loss: 0.6952 - accu
racy: 0.5130
Epoch 10/20
4/4 [==============================] - 2s 447ms/step - loss: 0.6951 - accu
racy: 0.5130
Epoch 11/20
4/4 [==============================] - 2s 443ms/step - loss: 0.6950 - accu
racy: 0.5130
Epoch 12/20
4/4 [==============================] - 2s 442ms/step - loss: 0.6949 - accu
racy: 0.5130
Epoch 13/20
4/4 [==============================] - 2s 446ms/step - loss: 0.6949 - accu
racy: 0.5130
Epoch 14/20
4/4 [==============================] - 2s 445ms/step - loss: 0.6949 - accu
racy: 0.5130
Epoch 15/20
4/4 [==============================] - 2s 446ms/step - loss: 0.6947 - accu
racy: 0.5130
Epoch 16/20
4/4 [==============================] - 2s 445ms/step - loss: 0.6947 - accu
racy: 0.5130
Epoch 17/20
4/4 [==============================] - 2s 443ms/step - loss: 0.6946 - accu
racy: 0.5130
Epoch 18/20
4/4 [==============================] - 2s 443ms/step - loss: 0.6945 - accu
```

```
racy: 0.5130
Epoch 19/20
4/4 [==============================] - 2s 437ms/step - loss: 0.6944 - accu
racy: 0.5130
Epoch 20/20
4/4 [==============================] - 2s 440ms/step - loss: 0.6944 - accu
racy: 0.5130
2/2 [==============================] - 0s 124ms/step - loss: 0.6836 - accu
racy: 0.5789
Epoch 1/20
2/2 [==============================] - 1s 434ms/step - loss: 0.7048 - accu
racy: 0.4483
Epoch 2/20
2/2 [==============================] - 1s 434ms/step - loss: 0.7047 - accu
racy: 0.4483
Epoch 3/20
2/2 [==============================] - 1s 430ms/step - loss: 0.7044 - accu
racy: 0.4483
Epoch 4/20
2/2 [==============================] - 1s 436ms/step - loss: 0.7043 - accu
racy: 0.4483
Epoch 5/20
2/2 [==============================] - 1s 433ms/step - loss: 0.7040 - accu
racy: 0.4483
Epoch 6/20
2/2 [==============================] - 1s 435ms/step - loss: 0.7038 - accu
racy: 0.4483
Epoch 7/20
2/2 [==============================] - 1s 443ms/step - loss: 0.7036 - accu
racy: 0.4483
Epoch 8/20
2/2 [==============================] - 1s 470ms/step - loss: 0.7033 - accu
racy: 0.4483
Epoch 9/20
2/2 [==============================] - 1s 438ms/step - loss: 0.7031 - accu
racy: 0.4483
Epoch 10/20
2/2 [==============================] - 1s 438ms/step - loss: 0.7028 - accu
racy: 0.4483
Epoch 11/20
2/2 [==============================] - 1s 435ms/step - loss: 0.7025 - accu
racy: 0.4483
Epoch 12/20
2/2 [==============================] - 1s 440ms/step - loss: 0.7024 - accu
racy: 0.4483
Epoch 13/20
2/2 [==============================] - 1s 429ms/step - loss: 0.7020 - accu
racy: 0.4483
Epoch 14/20
2/2 [==============================] - 1s 430ms/step - loss: 0.7018 - accu
racy: 0.4483
Epoch 15/20
2/2 [==============================] - 1s 429ms/step - loss: 0.7015 - accu
racy: 0.4483
Epoch 16/20
2/2 [==============================] - 1s 434ms/step - loss: 0.7013 - accu
racy: 0.4483
Epoch 17/20
2/2 [==============================] - 1s 442ms/step - loss: 0.7010 - accu
racy: 0.4483
Epoch 18/20
```

```
2/2 [==============================] - 1s 437ms/step - loss: 0.7007 - accu
racy: 0.4483
Epoch 19/20
2/2 [==============================] - 1s 464ms/step - loss: 0.7005 - accu
racy: 0.4483
Epoch 20/20
2/2 [==============================] - 1s 434ms/step - loss: 0.7004 - accu
racy: 0.4483
2/2 [==============================] - 0s 117ms/step - loss: 0.6861 - accu
racy: 0.5789
Score of 0th fold 0.4482758641242981
Score of 1th fold 0.568965494632721
Score of 2th fold 0.568965494632721
Score of 3th fold 0.5862069129943848
Score of 4th fold 0.5
Score of 5th fold 0.43103447556495667
Score of 6th fold 0.4912280738353729
Score of 7th fold 0.7017543911933899
Score of 8th fold 0.5789473652839661
Score of 9th fold 0.5789473652839661
Average accuracy of Model:  0.5454325437545776
```

From our 10 fold validation we find that the average accuracy is quite high around 0.545. Using above method for calculation, we get an expected profitability of around 1.52. This may seem to indicate that our model is quite good, but the reality is that our data is outdated and a lot of the results were approximated. The actual calculations are far more complicated and is beyond the scope of this tutorial. If you want to learn more about some of the common trading practices visit here https://www.investopedia.com/options-basics-tutorial-4583012#:~:text=A%20call%20option%20gives%20the,right%20to%20sell%20a%20stock (https://www.investopedia.com/options-basics-tutorial-4583012#:~:text=A%20call%20option%20gives%20the,right%20to%20sell%20a%20stock).

Insight & Policy Decision

I hope that after this tutorial you have developed a better understanding of how to train a machine to make predictions on the stock market. In all likelihood machines will make better predictions than humans, at least with regard to short term investments. As our computational speeds increases, our machine learning algorithms can be trained on larger datasets with far more features, increasing the accuracy of their predictions. It is very likely that the presence of computer trader will only increase with technological advancements. When that happens, it may very well become a race for computational power to determine those that will 'beat the market' and the rest that will fail.