Using Machine Learning To Decide Which Stocks To Buy

Evan Okin

March 2020

Overview

The objective of this project is to use fundamental analysis to predict which stocks will perform well. Each publicly traded company reports financial data, commonly found in their 10-K filings.

The goal is to create a mathematical ("machine learning") model that will help identify which stocks to buy. Along the way, we can compare various algorithms and relate them to the dataset. The analysis will include both prediction/regression models (try to predict the magnitude of stock movements) and classification models (try to categorize stocks into two categories, buy or do not buy).

If we can successfully build an insightful model, then we can have a leg-up on the majority of investors. Whether or not we can build a profitable model, this can serve as a guide to those looking to improve their machine learning skills.

Summary of Results

After assessing various models, the best performing model was a Random Forest with an accuracy of over 67% at predicting whether a stock will increase or decrease during the course of a year based on key financial variables. Our model has found hidden insights in the data. This is significant becaues any edge in the market is hard to obtain, and we can utilize this model to predict stock prices in 2020 and beyond. Implementation can be found in Section 7 of the report.

Table of Contents

Section 7: Next Steps and Implementation

Section 1: Overview of OSEMN Framework

The OSEMN Framework is an acronym for solving problems in Data Science in a systematic mattern. The O stands for Obtaining data from relevant sources. The S stands for Scrubbing data to a format that our machine understands. The E stands for Exploring data to find significant patterns and trends. The M stands for Modeling data to make predictions and forecasts. The N stands for iNterpreting the results.

Section 2: O: Obtain data from relevant resources

Kaggle.com is home to many data sets for aspiring data scientists. Many of the data sets are financial in nature, due to the desire to beat the markets.

The dataset is obtained from: [https://www.kaggle.com/cnic92/200-financial-indicators-of-us-stocks-20142018 (https://www.kaggle.com/cnic92/200-financial-indicators-of-us-stocks-20142018)](https://www.kaggle.com/cnic92/200-financial-indicators-of-us-stocks-20142018)

The data can be obtained from Financial Modeling Prep API and pandas_datareader. Pandas_datareader is an API that.

Import necessary packages to run our API

We need to import our packages to run our Application Program Interface (API). Pandas is useful for dealing with dataframes, which are essentially excel-like tables. Numpy is useful for numerical/mathematical computation. Datetime is useful for dealing with variables that have dates or times. Pandas Datareader is useful for pulling in stock returns from the web. Warnings is useful for turning off warnings so we can run our code without interruption (assuming no errors).

In [1]:
```
import pandas as pd
import numpy as np
import datetime
import pandas_datareader.data as web
import warnings
warnings.simplefilter("ignore")
```

Use an API to pull in monthly stock prices

We can input a list and pull in monthly stock prices using the Yahoo Finance API. For the purpose of this project, we will not need this (because we have access to the data in csv files), but it's included here for completeness.

In [2]:
```
#Pull in "FANG" Stocks
list_of_stocks_to_pull=['FB','AMZN','NFLX','GOOG']
start=datetime.datetime(2019,10,1)
end=datetime.datetime(2020,2,1)
df=web.get_data_yahoo(list_of_stocks_to_pull, start, end,interval='m')
```

Preview the data

We can preview any amount of rows of the data. To make the output manageable, we will only show 2 rows when we preview the data.

In [3]:
```
df.head(2)
```

Out[3]:

Attributes	Adj Close				Close		
Symbols	AMZN	FB	GOOG	NFLX	AMZN	FB	GOOG
Date							
2019-10-01	1776.660034	191.649994	1260.109985	287.410004	1776.660034	191.649994	1260.109985
2019-11-01	1800.800049	201.639999	1304.959961	314.660004	1800.800049	201.639999	1304.959961

2 rows × 24 columns

Use an API to pull in weekly stock prices

We can pull in weekly stock returns using the Yahoo Finance API.

In [4]:
```
df=web.get_data_yahoo(list_of_stocks_to_pull, start, end,interval='w')
```

Preview the data

In [5]:
```
df.head(2)
```

Out[5]:

Attributes	Adj Close				Close		
Symbols	AMZN	FB	GOOG	NFLX	AMZN	FB	GOOG
Date							
2019-09-30	1739.650024	180.449997	1209.000000	272.790009	1739.650024	180.449997	1209.000000
2019-10-07	1731.920044	184.190002	1215.449951	282.929993	1731.920044	184.190002	1215.449951

2 rows × 24 columns

Use an API to pull in daily stock prices

We can pull in daily stock returns using the Yahoo Finance API.

In [6]:
```
df=web.get_data_yahoo(list_of_stocks_to_pull, start, end,interval='d')
```

Preview the data

In [7]:
```
df.head(2)
```

Out[7]:

Attributes	Adj Close				Close			
Symbols	AMZN	FB	GOOG	NFLX	AMZN	FB	GOOG	
Date								
2019-10-01	1735.650024	175.809998	1205.099976	269.579987	1735.650024	175.809998	1205.099976	
2019-10-02	1713.229980	174.600006	1176.630005	268.029999	1713.229980	174.600006	1176.630005	

2 rows × 24 columns

Convert data to percentage changes to analyze returns

Often, we care less about the stock prices themselves and more about how much the stock price changes over time. We can transform our data into percent changes.

In [8]:
```
df=df.pct_change()
df.dropna
df.head(3)
```

Out[8]:

Attributes	Adj Close				Close				Hi
Symbols	AMZN	FB	GOOG	NFLX	AMZN	FB	GOOG	NFLX	AI
Date									
2019-10-01	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
2019-10-02	-0.012917	-0.006882	-0.023625	-0.005750	-0.012917	-0.006882	-0.023625	-0.005750	-0
2019-10-03	0.006532	0.027377	0.009519	0.000448	0.006532	0.027377	0.009519	0.000448	-0

3 rows × 24 columns

Read in financial data from csv files into python

We have five separate dataframes, each corresponding to a different year of data. We can read in the data from separate csv files into separate tables.

In [9]:
```
import pandas as pd
df_2014=pd.read_csv('2014_Financial_Data.csv')
df_2015=pd.read_csv('2015_Financial_Data.csv')
df_2016=pd.read_csv('2016_Financial_Data.csv')
df_2017=pd.read_csv('2017_Financial_Data.csv')
df_2018=pd.read_csv('2018_Financial_Data.csv')
```

Confirm that each of the data files has the same number of columns/variables

We can check to make sure that all of the files have the same number of columns, which would be a good start to making sure that we have consistent variables of data.

In [10]:
```
(len(df_2014.columns)==len(df_2015.columns)==len(df_2016.columns)
==len(df_2017.columns)==len(df_2018.columns))
```

Out[10]:
```
True
```

Section 3: S: Scrub data to a format that our machine understands

Rename columns to be more descriptive

While not required for our model, we can rename columns so that we (and anyone reading our code) have a better understanding of what the variables represent.

In [11]:
```
df_2014.rename(columns = {'Unnamed: 0':'Company'}, inplace = True)
df_2015.rename(columns = {'Unnamed: 0':'Company'}, inplace = True)
df_2016.rename(columns = {'Unnamed: 0':'Company'}, inplace = True)
df_2017.rename(columns = {'Unnamed: 0':'Company'}, inplace = True)
df_2018.rename(columns = {'Unnamed: 0':'Company'}, inplace = True)
```

In [12]:
```
df_2014.rename(columns = {'2015 PRICE VAR [%]':'Price Change'},
               inplace = True)
df_2015.rename(columns = {'2016 PRICE VAR [%]':'Price Change'},
               inplace = True)
df_2016.rename(columns = {'2017 PRICE VAR [%]':'Price Change'},
               inplace = True)
df_2017.rename(columns = {'2018 PRICE VAR [%]':'Price Change'},
               inplace = True)
df_2018.rename(columns = {'2019 PRICE VAR [%]':'Price Change'},
               inplace = True)
```

Create a new variable

We can add a new variable (year) so that we can more easily identify the data.

In [13]:
```
df_2014['Year']=2014
df_2015['Year']=2015
df_2016['Year']=2016
df_2017['Year']=2017
df_2018['Year']=2018
```

Concatenate the data into one dataframe

We can combine the data into one concise dataframe.

In [14]:
```
import pandas as pd
df=pd.concat([df_2014, df_2015, df_2016, df_2017, df_2018],
             ignore_index=True)
```

Find the count of a variable

We can analyze a variable and see the breakdown by count.

In [15]:
```
df['Year'].value_counts()
```

Out[15]:
```
2017    4960
2016    4797
2018    4392
2015    4120
2014    3808
Name: Year, dtype: int64
```

We can analyze the percentage breakdown of a column, and round its values.

In [16]:
```
round(df['Year'].value_counts(normalize=True),3)
```

Out[16]:
```
2017    0.225
2016    0.217
2018    0.199
2015    0.187
2014    0.172
Name: Year, dtype: float64
```

Find the number of rows and columns of the data

In [17]:
```
df.shape
```

Out[17]:
```
(22077, 226)
```

Get descriptive statistics for the data

We can get descriptive statistics for each column that has numeric data. This is helpful because it allows us to see a range of values (and potentially observe if outlier values exist).

In [18]:
```
1 df.describe()
```

Out[18]:

	Revenue	Revenue Growth	Cost of Revenue	Gross Profit	R&D Expenses	SG&A Expense	
count	2.090600e+04	19989.000000	2.030600e+04	2.087000e+04	1.993900e+04	2.040800e+04	2
mean	5.161619e+09	3.622214	3.258565e+09	1.970452e+09	1.033333e+08	8.699279e+08	1
std	3.197314e+10	312.648170	2.583092e+10	8.735750e+09	7.676062e+08	3.804283e+09	5
min	-6.276160e+08	-12.769300	-2.986888e+09	-1.280800e+10	-1.098000e+08	-1.401594e+08	-5
25%	5.880737e+07	-0.014700	3.211750e+06	3.035575e+07	0.000000e+00	1.768550e+07	3
50%	4.352510e+08	0.057600	1.530115e+08	1.943525e+08	0.000000e+00	8.062450e+07	1
75%	2.287259e+09	0.182400	1.180224e+09	8.806035e+08	1.220150e+07	3.699722e+08	6
max	1.886894e+12	42138.663900	1.581527e+12	4.621600e+11	2.883700e+10	1.856830e+11	3

8 rows × 224 columns

Find the percentage of null values among the data

We can observe the proportion of null values for each column of our data, to later decide what to do with missing data.

In [19]:
```
1 round(df.isna().sum()/len(df),3).head(2)
```

Out[19]:
```
Company    0.000
Revenue    0.053
dtype: float64
```

Set the index of the data

We can set the index of our data to something more meaningful. By setting the index of our data to year, we can easily compare data within years.

In [20]:
```
1 df.index=df['Year']
```

Preview the top value counts

We can confirm that no company appears more than five times. We don't expect any company to appear more than five times, because there are only five years of data.

In [21]:
```
1 df['Company'].value_counts().head(2)
```

Out[21]:
```
GFED    5
SINO    5
Name: Company, dtype: int64
```

Preview the bottom value counts

In [22]:
```
df['Company'].value_counts().tail(2)
```

Out[22]:
```
STDY    1
KTWO    1
Name: Company, dtype: int64
```

We see that each company appears no more than 5 times, but no less than 1 time. This is good and as expected.

Preview the columns

We can observe the columns of the data which correspond to the variables we have available.

In [23]:
```
df.columns
```

Out[23]:
```
Index(['Company', 'Revenue', 'Revenue Growth', 'Cost of Revenue',
       'Gross Profit', 'R&D Expenses', 'SG&A Expense', 'Operating Expense
s',
       'Operating Income', 'Interest Expense',
       ...
       'Inventory Growth', 'Asset Growth', 'Book Value per Share Growth',
       'Debt Growth', 'R&D Expense Growth', 'SG&A Expenses Growth', 'Sect
or',
       'Price Change', 'Class', 'Year'],
      dtype='object', length=226)
```

Preview a column

In [24]:
```
df['Revenue Growth'].value_counts().head(2)
```

Out[24]:
```
 0.0    991
-1.0     82
Name: Revenue Growth, dtype: int64
```

Create a conditional

We can create a column using a conditional.

In [25]:
```
import numpy as np
df['Class_Check']=np.where(df['Price Change']>=0,1,0)
```

Check a condition

We can confirm that a column is doing what we want it to, such as whether or not the 'Class' variable is the percentage of increases in the data.

In [26]:
```
df['Class'].sum()==df['Class_Check'].sum()
```

Out[26]: True

Drop a column

We can drop a column that we no longer need for our prediction.

In [27]:
```
df.drop('Class_Check',axis=1,inplace=True)
```

Drop a list of columns

We can drop a list of columns that we think do not add predictive value. It is likely that absolute values (such as revenue) will not have strong predictor value as relative values (such as revenue growth), so we can keep columns that are either growth or ratios. It is also helpful to delete some of these columns because of multicollinearity - if two variables are both highly correlated with each other, then an increase in one variable could mask what variable is truly increasing our target variable.

In [28]:
```
df.drop(['Gross Profit','R&D Expenses','SG&A Expense', 'Operating Exper
         'Earnings before Tax','Income Tax Expense','Net Income - Non-C
         'Net Income - Discontinued ops','Net Income','Preferred Divide
        'EPS Diluted','Weighted Average Shs Out','Weighted Average Shs
        'EBITDA','EBIT','Consolidated Income','Earnings Before Tax Marg
        'Cash and cash equivalents','Short-term investments','Cash and
        'Receivables','Inventories','Total current assets','Property, P
        'Goodwill and Intangible Assets','Long-term investments','Tax a
        'Total assets','Payables','Short-term debt','Total current liab
        'Total debt','Deferred revenue','Tax Liabilities','Deposit Liab
        'Total liabilities','Other comprehensive income','Retained earn
         'Total shareholders equity','Investments','Net Debt','Other As
        'Depreciation & Amortization','Stock-based compensation','Opera
        'Acquisitions and disposals','Investment purchases and sales','
        'Issuance (repayment) of debt','Issuance (buybacks) of shares',
        'Effect of forex changes on cash','Net cash flow / Change in ca
        'operatingProfitMargin','daysOfSalesOutstanding','daysOfInvento
        'daysOfPayablesOutstanding','cashConversionCycle','Tangible Boo
        'Market Cap','Enterprise Value','PB ratio','PTB ratio','Graham
        'Tangible Asset Value','Net Current Asset Value','Invested Capi
        'Average Payables','Average Inventory','Days Sales Outstanding'
        'Days of Inventory on Hand','EPS Diluted Growth','Weighted Aver
        '10Y Revenue Growth (per Share)','5Y Revenue Growth (per Share)
        '5Y Operating CF Growth (per Share)','10Y Net Income Growth (pe
        '10Y Shareholders Equity Growth (per Share)','5Y Shareholders E
        '10Y Dividend per Share Growth (per Share)','5Y Dividend per Sh
        'Revenue','Cost of Revenue'],axis=1,inplace=True)
```

Preview the data

In [29]:
```
1 df.head(2)
```

Out[29]:

Year	Company	Revenue Growth	EPS	Dividend per Share	Gross Margin	EBITDA Margin	Profit Margin	Free Cash Flow margin	Net Profit Margin	priceBookVa
2014	PG	-0.0713	4.1900	2.448	0.4754	0.2470	0.1560	0.1359	0.1565	
2014	VIPS	1.1737	0.2396	0.000	0.2487	0.0107	0.0058	0.0704	0.0364	

2 rows × 125 columns

Section 4: E: Explore data to find significant patterns and trends

Find the number of rows and columns of the data

In [30]:
```
1 df.shape
```

Out[30]:
```
(22077, 125)
```

Get information on the data

We can get information on the columns, including the data types of our variables.

In [31]:
```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 22077 entries, 2014 to 2018
Columns: 125 entries, Company to Year
dtypes: float64(121), int64(2), object(2)
memory usage: 21.2+ MB
```

Find the types of the columns

In [32]:
```
1 df.dtypes.head(2)
```

Out[32]:
```
Company            object
Revenue Growth    float64
dtype: object
```

Get descriptive statistics on the data

In [33]:
```
1 df.describe()
```

Out[33]:

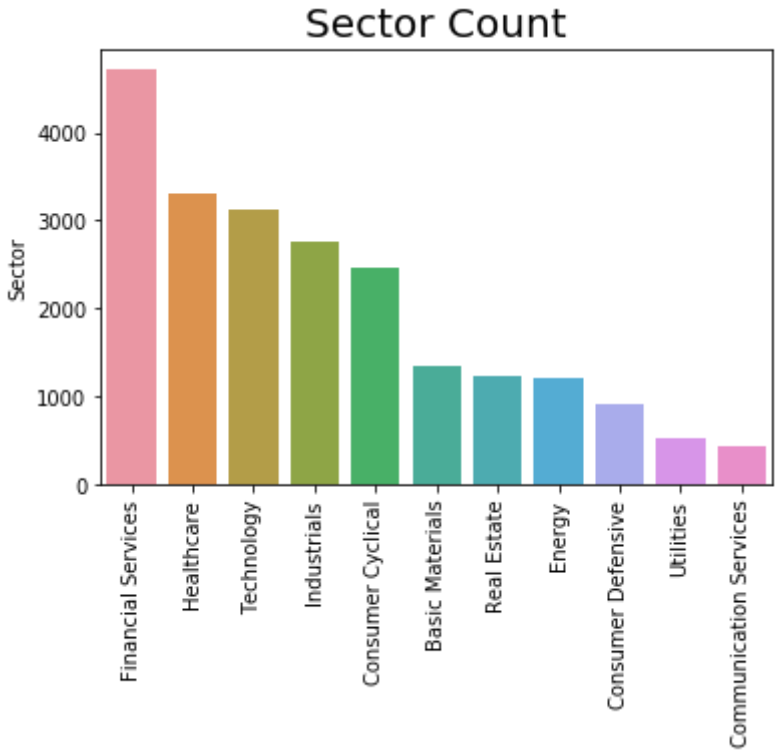
	Revenue Growth	EPS	Dividend per Share	Gross Margin	EBITDA Margin	Profit Margin	Fl
count	19989.000000	2.077600e+04	19818.000000	20878.000000	19630.000000	19633.000000	197
mean	3.622214	-1.065748e+04	1.215392	0.487844	-8.880591	-9.307057	
std	312.648170	8.960977e+05	72.200951	0.945601	239.625361	243.205468	2
min	-12.769300	-1.018709e+08	0.000000	-74.319100	-24207.000000	-24414.000000	-232
25%	-0.014700	-3.900000e-01	0.000000	0.250700	0.021000	-0.042000	
50%	0.057600	6.400000e-01	0.000000	0.460750	0.129909	0.043000	
75%	0.182400	2.020000e+00	0.720000	0.802775	0.302800	0.133000	
max	42138.663900	8.028004e+06	10100.664000	31.000000	3090.870000	3090.870000	6

8 rows × 123 columns

Create a histogram distribution plot

We can create a a histogram for our distribution, so that we can analyze the count by sector.

In [34]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
df_sector = df['Sector'].value_counts()
sns.barplot(np.arange(len(df_sector)), df_sector)
plt.xticks(np.arange(len(df_sector)),
           df_sector.index.values.tolist(), rotation=90)
plt.title('Sector Count', fontsize=20)
plt.show()
```



Analyze a segment of the data

We can analyze a segment of our data, such as honing in on one stock (Amazon).

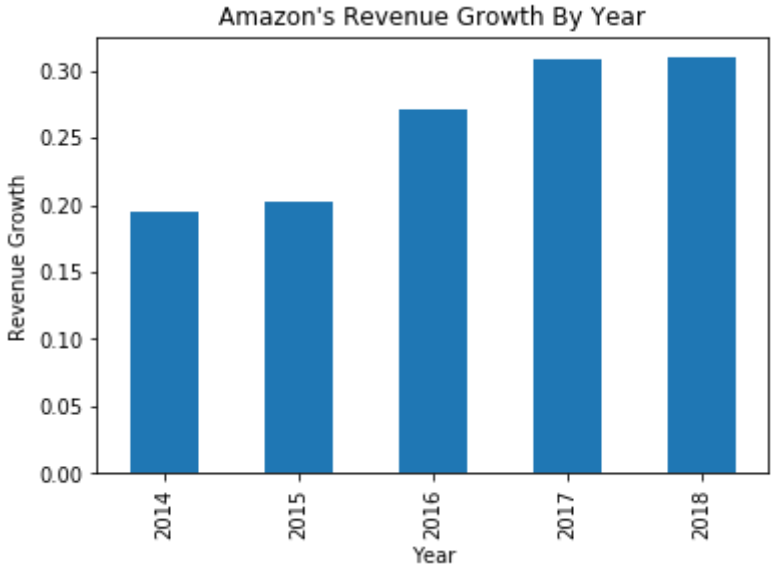
In [35]:
```
df_Amazon=df.loc[df['Company']=='AMZN']
```

Plot a histogram

In addition to plotting through the Matplotlib library as we did above, we can also create a histogram plot using Python's built-in functionality.

In [36]:
```
df_Amazon['Revenue Growth'].plot(kind='bar')
plt.title('Amazon\'s Revenue Growth By Year')
plt.xlabel('Year')
plt.ylabel('Revenue Growth')
```

Out[36]: Text(0, 0.5, 'Revenue Growth')



In [37]:
```
df_Amazon['Profit Margin']
df_Amazon['Profit Margin'].plot(kind='bar')
plt.title('Amazon\'s Profit Margin By Year')
plt.xlabel('Year')
plt.ylabel('Profit Margin')
```

Out[37]:
```
Text(0, 0.5, 'Profit Margin')
```

Find the number of unique entries of a variable

In [38]:
```
df['Sector'].nunique()
```

Out[38]:
```
11
```

Find the names of the unique entries of a variable

In [39]:
```
df['Sector'].unique()
```

Out[39]:
```
array(['Consumer Defensive', 'Basic Materials', 'Healthcare',
       'Consumer Cyclical', 'Industrials', 'Real Estate',
       'Communication Services', 'Energy', 'Financial Services',
       'Utilities', 'Technology'], dtype=object)
```

Find mean for a group

In [40]:
```
1 df.groupby(['Sector']).mean()['Class'].sort_values(ascending=False)
```

Out[40]:
```
Sector
Utilities                  0.702703
Financial Services         0.631568
Real Estate                0.630137
Technology                 0.575496
Consumer Defensive         0.567245
Industrials                0.558526
Consumer Cyclical          0.529340
Communication Services     0.514607
Basic Materials            0.508185
Healthcare                 0.442965
Energy                     0.394412
Name: Class, dtype: float64
```

In [41]:
```
1 df.groupby(['Sector']).mean()['Price Change'].sort_values(ascending=Fal
```

Out[41]:
```
Sector
Real Estate                1968.771288
Technology                  337.480040
Financial Services          332.580219
Healthcare                  219.976921
Basic Materials              63.708856
Consumer Cyclical            16.142191
Utilities                    13.989712
Consumer Defensive           12.501579
Industrials                   7.357903
Communication Services        5.310088
Energy                       -3.023544
Name: Price Change, dtype: float64
```

Run a hypothesis test

If we have a theory that we want to test, we can run a hypothesis test to see if it's accurate. For instance, based on the above, we might theorize that Real Estate stocks outperform the aggregate of all other sectors of stocks at a statistically significant level. We can see if the data supports this theory.

Null hypothesis (Ho): There is no significant difference in returns

Alternative hypothesis (Ha): There is a significant difference in returns

In [42]:
```
1 import pandas as pd
2 import numpy as np
3 from scipy import stats
4 import statsmodels.api as sm
5 from statsmodels.formula.api import ols
```

In [43]:
```
1 control=df.loc[df['Sector']!='Real Estate']['Price Change']
2 experimental=df.loc[df['Sector']=='Real Estate']['Price Change']
```

In [44]:
```
1 control.mean()
```

Out[44]:
```
168.70321342821495
```

In [45]:
```
1 experimental.mean()
```

Out[45]:
```
1968.771287861904
```

In [46]:
```
1 stats.ttest_ind(experimental, control,equal_var=False)
```

Out[46]:
```
Ttest_indResult(statistic=0.922949241505085, pvalue=0.35621268191746924)
```

The p-value is high, and it far exceeds a typical 5% threshold. We fail to reject the null hypothesis. Results are statistically insignificant with p-value nearly 36%. Real Estate stocks do not have a significantly different return profile than all other stocks, even though they do appear to be higher.

We can test if Real Estate stocks statistically outperform Energy stocks.

In [47]:
```
1 control=df.loc[df['Sector']=='Energy']['Price Change']
2 experimental=df.loc[df['Sector']=='Real Estate']['Price Change']
```

In [48]:
```
1 stats.ttest_ind(experimental, control,equal_var=False)
```

Out[48]:
```
Ttest_indResult(statistic=1.0117372213864555, pvalue=0.3118610869757253)
```

Although slightly more significant, we still see that we can not reject the null hypothesis - there is not a highly significant difference in returns based on the Real Estate versus Energy sector.

Save a dataframe to an excel file

Before we make any adjustments, we can save our data to an excel file, so that we know we have a good data file to come back to.

In [49]:
```
1 df.to_excel('Original_Data.xlsx')
```

Create categorical/dummy variables

We will want to see if our variables are continuous or if they are discrete (take on specific values). If they are discrete we will have to create dummy variables. Luckily for us, we kept variables that represent growth rates, margins, or ratios. These are inately scaled and continuous. We need to change Sectors into dummy variables. For instance, we will create one column for each sector. If a stock is in the "Energy" sector, then it will have a 1 for the "Energy" column, and if not, it will have a 0 for this column.

In [50]:
```
import pandas as pd
df = pd.concat([df, pd.get_dummies(df['Sector'],
                                   drop_first=True)], axis=1)
```

Drop a column

In [51]:
```
df.drop('Sector',axis=1,inplace=True)
```

Find the percentage of null values by column

In [52]:
```
(df.isna().sum()/len(df)).head(2)
```

Out[52]:
```
Company           0.000000
Revenue Growth    0.094578
dtype: float64
```

Write a loop and append values that meet a certain criterion to a list

We can see which columns have a lot of empty values. If a column has over 15% null values, we can drop the variable (the 15% figure is arbitrary so we will use it as a starting point). This is fine because we have 125 columns/variables and many will be correlated with each other, so anything to carefully reduce the dimensions of our data is helpful. We can create a list of variables that have over 15% empty data.

In [53]:
```
bad_data_list=[]
for i in df.columns:
    if df[i].isna().sum()/len(df)>0.15:
        bad_data_list.append(i)
len(bad_data_list)
```

Out[53]:
```
37
```

Drop a list from dataframe

Now that we have a list of variables that include "bad data", we can drop them from our dataframe.

In [54]:
```
df.drop(bad_data_list,axis=1,inplace=True)
```

Find what percentage of rows we maintain in our dataset by dropping rows with null values in at least one column

In [55]:
```
len(df.columns)
```

Out[55]:
```
97
```

We can see if we will still have a lot of data if we dropped all null values from our data.

In [56]:
```
original_length=len(df)
df.dropna(inplace=True)
len(df)/original_length
```

Out[56]: 0.6787154051728043

In [57]:
```
len(df)
```

Out[57]: 14984

There are still nearly 15,000 rows of data, so we'll keep our data with these null values dropped.

Describe the data

In [58]:
```
df.describe()
```

Out[58]:

	Revenue Growth	EPS	Dividend per Share	Gross Margin	EBITDA Margin	Profit Margin	Fr Flow
count	14984.000000	1.498400e+04	14984.000000	14984.000000	14984.000000	14984.000000	14984
mean	4.328576	-9.136009e+02	1.433802	0.497083	-5.313065	-5.539541	-5
std	360.045377	8.127908e+04	83.002638	0.754436	109.939738	108.999081	114
min	-0.999600	-9.457551e+06	0.000000	-74.319100	-7322.000000	-7030.000000	-8011
25%	-0.010900	-2.400000e-01	0.000000	0.273800	0.029000	-0.030000	-0
50%	0.064600	8.300000e-01	0.000000	0.456000	0.127000	0.043000	0
75%	0.183525	2.230000e+00	0.760500	0.750025	0.282000	0.121000	0
max	42138.663900	1.444920e+05	10100.664000	1.896500	1060.404000	940.041000	121

8 rows × 96 columns

Drop outliers from data

We can see in the data description above that there are big outliers. We can set a ceiling and floor for the top 2% and bottom 2% to account for outliers, so that they don't wildly move our results.

In [59]:
```
upper_2_percent = df.quantile(0.98)
outliers_top = (df > upper_2_percent)
lower_2_percent = df.quantile(0.02)
outliers_bottom = (df < lower_2_percent)
df=df.mask(outliers_top, upper_2_percent, axis=1)
df=df.mask(outliers_bottom, lower_2_percent, axis=1)
```

Describe the data

In [60]:
```
1 df.describe()
```

Out[60]:

	Revenue Growth	EPS	Dividend per Share	Gross Margin	EBITDA Margin	Profit Margin	Fre Flow
count	14984.000000	14984.000000	14984.000000	14984.000000	14984.000000	14984.000000	14984.
mean	0.143214	0.864734	0.531567	0.516702	-0.438882	-0.605210	-0.
std	0.378893	3.408584	0.860382	0.297826	2.821927	2.970357	2.
min	-0.459470	-12.900600	0.000000	0.041666	-17.417420	-18.472120	-15.
25%	-0.010900	-0.240000	0.000000	0.273800	0.029000	-0.030000	-0.
50%	0.064600	0.830000	0.000000	0.456000	0.127000	0.043000	0.
75%	0.183525	2.230000	0.760500	0.750025	0.282000	0.121000	0.
max	1.948204	9.573400	3.736800	1.000000	0.781000	0.428000	0.

8 rows × 96 columns

We can see that when we replace outliers, our data is much smoother and much more reasonable.

Find the number of rows of the data

In [61]:
```
1 len(df)
```

Out[61]: 14984

Create a dataframe of features for machine learning algorithms

We need to remove certain values that we don't want to use for our prediction algorithm. We want to drop the Company name because we want to use other variables (not the name of a company) for prediction. We want to drop price change and class because these will be what we are trying to predict from the rest of the data.

In [62]:
```
1 features=df.drop(['Company','Price Change','Class'],axis=1)
```

Preview a dataframe

In [63]:
```
1 features.head(2)
```

Out[63]:

Year	Revenue Growth	EPS	Dividend per Share	Gross Margin	EBITDA Margin	Profit Margin	Free Cash Flow margin	Net Profit Margin	priceToSalesRatio	priceE
2014	-0.0713	4.19	2.448	0.4754	0.247	0.156	0.1359	0.1565	2.8583	
2014	0.0076	2.90	1.550	0.3557	0.201	0.102	0.1052	0.1019	1.8610	

2 rows × 94 columns

Set up a column that you want to predict with the machine learning algorithm

We will want to predict the price change for our stocks.

In [64]:
```
1 target_regression=df['Price Change']
```

Preview the data

In [65]:
```
1 target_regression.head(2)
```

Out[65]:
```
Year
2014    -9.323276
2014    12.897715
Name: Price Change, dtype: float64
```

Set up a column that you want to predict with the machine learning algorithm

We will want to predict whether a stock increased or decreased over the course of a one-year period.

In [66]:
```
1 target_classification=df['Class']
```

Preview the data

In [67]:
```
1 target_classification.head(2)
```

Out[67]:
```
Year
2014    0
2014    1
Name: Class, dtype: int64
```

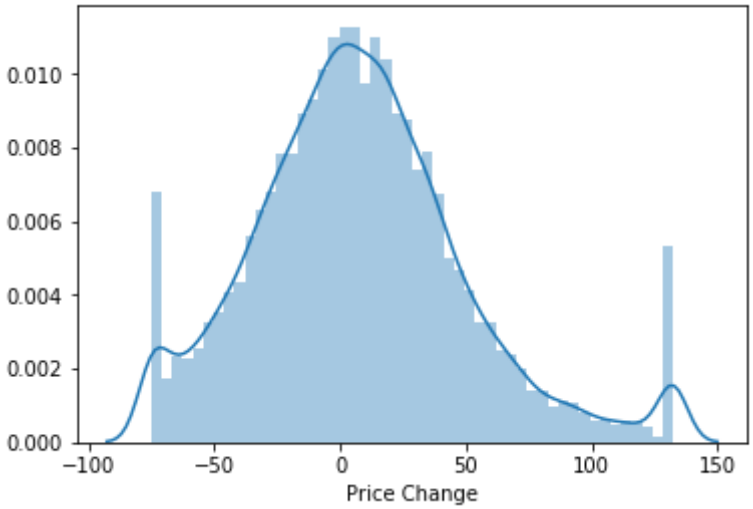
Section 5: M: Model data to make predictions and forecasts

Create a distribution plot of a variable

We can plot our variable for "Price Change" that we are trying to predict.

In [68]:
```
1 import seaborn as sns
2 sns.distplot(df['Price Change'])
```

Out[68]: <matplotlib.axes._subplots.AxesSubplot at 0x1a237fd8d0>



We can see that the distribution is roughly normal, which means that our machine learning models are appropriate. We can see spikes on the end of the spectrum, which is a result of our outlier removal (all outliers are capped at these values).

In [69]:
```
1 y=df['Price Change']
```

Set random seed for reproducability of same results

We can set a random seed so that we can reproduce the same results each time that we run our code. We can use any number for the seed.

In [70]:
```
1 import numpy as np
2 np.random.seed(42)
```

Set up a train-test split for Machine Learning

As a rule of thumb across our models, we will need to split the data into our training set and our testing set. We will build the model based on the data from our training set, and then we will test it out on our testing set to see how our model performs. We need to split it up because otherwise, we would be overfitting the model by using the actual results to predict the results! A standard training set size is between 70%-80% of the data. We can use 80%, because doing so, we will still have a lot of data left for our testing set.

In [71]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(
    features,target_regression,test_size=0.2,random_state=42)
```

Confirm the length of the training/testing set

In [72]:
```
print(format(len(X_train)/((len(X_train)+len(X_test))),".4%"))
```

```
79.9987%
```

In [73]:
```
len(df)%5 #won't be zero
```

Out[73]:
```
4
```

We can see that the length of the training set is slightly off from 80%. This is because the aggregate data is not a multiple of 5, so it cannot be split into an exact 80/20 split.

Run a Linear Regression

The most basic and common form of machine learning model is multiple linear regression. We draw a line that "fits" the points. The goal is to minimize how far away the distance is between prediction and actual results. This common method is referred to as Ordinary Least Squares (OLS), where the goal is to minimize the sum of squares of residuals (error terms) from our line of best fit.

In [74]:
```
from sklearn.linear_model import LinearRegression
lm=LinearRegression()
lm.fit(X_train,y_train)
```

Out[74]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False)
```

Print the intercept

In [75]:
```
print(lm.intercept_)
```

```
-1768.1233723757296
```

Print the regression coefficients

In [76]:
```
print(lm.coef_)
```

```
[-1.48092467e+00  1.54081315e+00 -8.97029967e-01  1.50168841e+01
 -2.44810388e+00  3.16555300e+00  1.78706900e-01  5.17185728e+00
 -1.73250183e-02  1.66308988e-02  8.92814970e-04  1.12807977e-02
  2.68547964e+01  7.39169383e-01 -1.33520020e+01 -6.78772172e-01
 -6.30615349e+00  9.90675455e-01  7.39169383e-01  5.33439715e-04
 -3.04036217e-02  7.00492086e-01 -3.22178675e-01  1.01070356e-01
 -8.71641529e-03 -3.39236653e+00 -8.89151110e-02  1.90096698e+01
 -1.15486631e+01 -1.90845324e-03  3.47497807e-02  9.82999350e-02
 -2.27750585e-02 -1.07278171e+00 -1.21636530e-02  2.25820836e-02
 -4.81789224e-03 -2.51504242e-03  3.47497807e-02  9.82999350e-02
 -2.27750585e-02 -1.73735839e-01  1.42625397e-02 -3.84348598e-02
  1.66308988e-02 -1.73250183e-02  1.12807977e-02  8.92814973e-04
 -3.55066325e+00 -2.24583372e+00 -8.89151110e-02 -3.39236653e+00
 -1.90845323e-03 -1.37384437e-02  2.45671253e+01 -1.07278171e+00
 -1.61240217e+00  1.08983028e+00  8.31895487e-01 -1.12129215e+00
  3.01615541e+00 -3.79366607e-01  2.58705893e+00  1.02827783e-02
  5.33439716e-04  9.90675455e-01 -4.12410261e-02 -1.81991356e+00
  7.06176630e-02 -7.05249152e-01  9.87576897e-01 -1.49660976e+00
 -1.43830516e+00  1.03125965e+00  6.70159051e-01  3.30244543e-01
 -8.51928303e-01 -3.96000690e+00 -6.32130762e-01 -9.90468201e-01
  1.71441971e-01  2.29071318e-01  1.08162387e+00  8.81546693e-01
 -2.98644304e+00 -5.85313906e+00 -4.59140237e+00 -1.46954366e+01
  2.80968705e+00  3.10645713e+00 -4.42421771e-01  2.08882878e+00
  4.97767669e+00  6.06707121e+00]
```

Create a dataframe of coefficients

We can create a table of our coefficients which will make it easier to analyze.

In [77]:
```
import pandas as pd
coef_df=pd.DataFrame(lm.coef_,features.columns,
                     columns=['Coefficients'])
coef_df.head(2)
```

Out[77]:

	Coefficients
Revenue Growth	-1.480925
EPS	1.540813

Create predictions for testing set

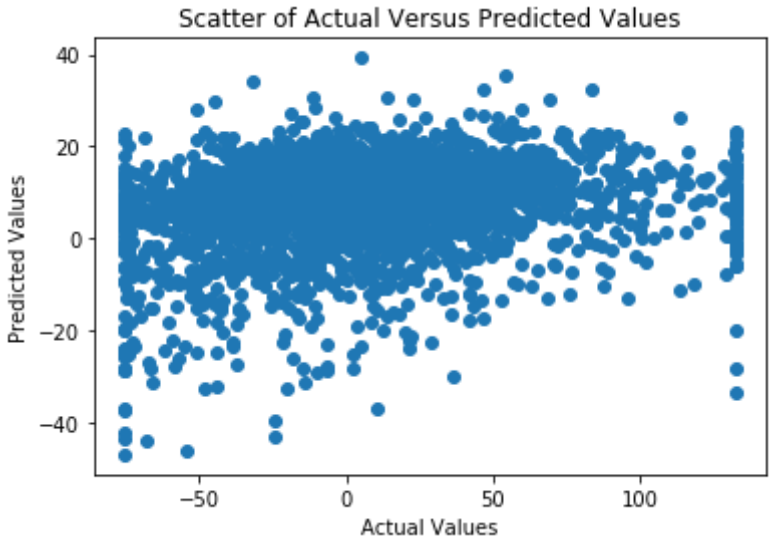
In [78]:
```
predictions = lm.predict(X_test)
predictions
```

Out[78]:
```
array([ 11.9380055 ,  -0.61381451, -12.29130921, ...,   2.61742468,
        32.328953  ,  16.21766817])
```

Plot a scatter of actual values for the testing set against predictions

In [79]:
```
1 import matplotlib.pyplot as plt
2 %matplotlib inline
3 plt.scatter(y_test,predictions)
4 plt.xlabel('Actual Values')
5 plt.ylabel('Predicted Values')
6 plt.title('Scatter of Actual Versus Predicted Values')
```

Out[79]:
```
Text(0.5, 1.0, 'Scatter of Actual Versus Predicted Values')
```

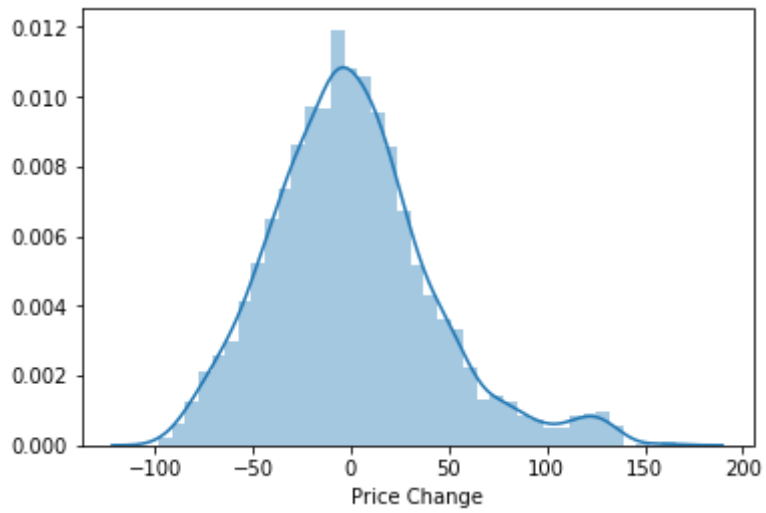


Ideally we would like to see a line sloping upwards where actual values = predicted values, which is unfortunately not what we are seeing. Luckily for us, we can analze the data as a classification problem instead of as a continuous/prediction problem.

Plot a histogram of residuals

In [80]:
```
1 import seaborn as sns
2 sns.distplot((y_test-predictions))
```

Out[80]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1c2d4b6630>
```



Evaluate the Mean Squared Error

We can evaluate our regression by looking at the Mean Squared Error (MSE), which is the mean of squared errors. It is popular because it punishes larger errors.

In [81]:
```
from sklearn import metrics
metrics.mean_squared_error(y_test,predictions)
```

Out[81]: 1726.9493745748991

Evaluate the Root Mean Squared Error

We can also evaluate our regression by looking at the Root Mean Squared Error (RMSE), which is the square root of the MSE. The RMSE is more popular than the MSE because it is easier to interpret.

In [82]:
```
import numpy as np
np.sqrt(metrics.mean_squared_error(y_test,predictions))
```

Out[82]: 41.556580400399874

Import necessary libraries

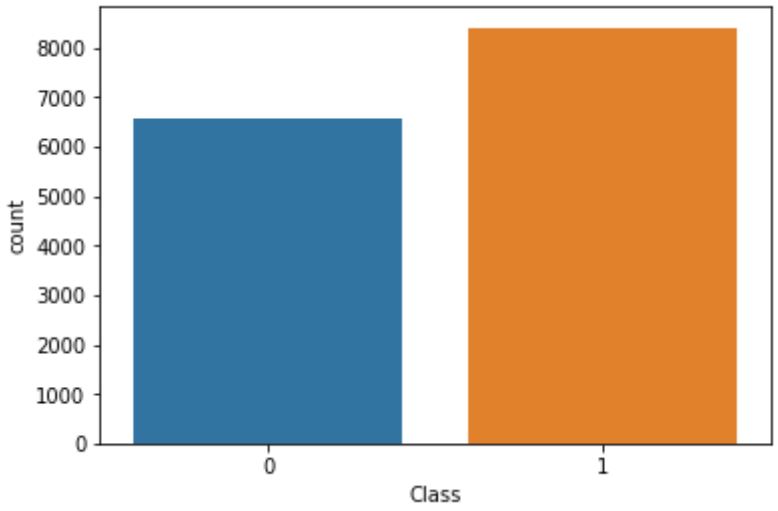
In [83]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Find a count by class

We can begin to model out our predictions as a classification problem - we will predict whether or not a stock will increase (class = 1) or decrease (class = 0) over the course of a year.

In [84]:
```
1 import seaborn as sns
2 sns.countplot(x='Class',data=df)
```

Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2cf56f98>



Set up a train-test split for Machine Learning

In [85]:
```
1 from sklearn.model_selection import train_test_split
2 X_train,X_test,y_train,y_test=train_test_split(
3     features,target_classification,test_size=0.2,random_state=42)
```

Run a Logistic Regression

Logistic Regression is often appropriate for binary classification. In our case, we'll classify our target variable as a 1 if a stock is expected to increase and a 0 otherwise. The logistic regression curve can only go between 0 and 1, using what is referred to as the sigmoid function. If the function gives a value below 0.5, the classification is 0 (stock will decrease). If the function gives a value above 0.5, the classification is 1 (stock will increase).

In [86]:
```
from sklearn.linear_model import LogisticRegression
logmodel=LogisticRegression()
logmodel.fit(X_train,y_train)
```

Out[86]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tr
ue,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

Create predictions for testing set

In [87]:
```
predictions=logmodel.predict(X_test)
```

Run a classification report

We can run a classification report to analyze our results.

In [88]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.56      0.33      0.42      1284
           1       0.62      0.80      0.70      1713

   micro avg       0.60      0.60      0.60      2997
   macro avg       0.59      0.57      0.56      2997
weighted avg       0.59      0.60      0.58      2997
```

Run a confusion matrix

We can also run a confusion matrix to analyze our results. This will allow us to see a grid of actual versus predicted values, so that we can see true positives, true negatives, false positives, and false negatives.

In [89]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,predictions))
```

```
[[ 428  856]
 [ 342 1371]]
```

Display model accuracy

We can display how accurate our model is at predicting true positives and true negatives. Luckily for us, accuracy is a good measure for prediction, because our data is fairly balanced between stocks that increase and stocks that decrease. Therefore, we will be using model accuracy to compare our various models, the first being Logistic Regression.

In [90]:
```
from sklearn.metrics import accuracy_score
logistic_accuracy_score=accuracy_score(y_test, predictions)
print(format(logistic_accuracy_score,".2%"))
```

```
60.03%
```

Scale the features

We can scale the features so that they dictate the model with appropriate weight.

In [91]:
```
from sklearn.preprocessing import StandardScaler
scaler=StandardScaler()
scaler.fit(features)
```

```
/Users/evanokin/anaconda3/lib/python3.7/site-packages/sklearn/preprocessi
ng/data.py:645: DataConversionWarning: Data with input dtype uint8, int6
4, float64 were all converted to float64 by StandardScaler.
  return self.partial_fit(X, y)
```

Out[91]:
```
StandardScaler(copy=True, with_mean=True, with_std=True)
```

In [92]:
```
scaled_features = scaler.transform(features)
scaled_features
```

```
/Users/evanokin/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.
py:1: DataConversionWarning: Data with input dtype uint8, int64, float64
were all converted to float64 by StandardScaler.
  """Entry point for launching an IPython kernel.
```

Out[92]:
```
array([[-0.56617922,  0.97558893,  2.22749649, ..., -0.22467026,
        -0.45905386, -0.17002611],
       [-0.35793421,  0.59712003,  1.18373893, ..., -0.22467026,
        -0.45905386, -0.17002611],
       [-0.37535394,  1.14281938,  3.72548618, ..., -0.22467026,
        -0.45905386, -0.17002611],
       ...,
       [-0.06496594, -0.23609832, -0.61784706, ..., -0.22467026,
        -0.45905386, -0.17002611],
       [-1.46276769, -0.52361733, -0.61784706, ..., -0.22467026,
        -0.45905386, -0.17002611],
       [-0.313857  , -0.62336883, -0.61784706, ..., -0.22467026,
        -0.45905386, -0.17002611]])
```

Create a dataframe of scaled features

In [93]:
```
df_feat=pd.DataFrame(scaled_features,columns=features.columns)
```

Preview the data

In [94]:
```
1 df_feat.head(2)
```

Out[94]:

	Revenue Growth	EPS	Dividend per Share	Gross Margin	EBITDA Margin	Profit Margin	Free Cash Flow margin	Net Profit Margin	priceToSalesRa
0	-0.566179	0.975589	2.227496	-0.138682	0.243063	0.256277	0.253228	0.251475	-0.196
1	-0.357934	0.597120	1.183739	-0.540608	0.226761	0.238097	0.241363	0.233801	-0.255

2 rows × 94 columns

Set up a train-test split for Machine Learning

In [95]:
```
1 from sklearn.model_selection import train_test_split
2 X_train,X_test,y_train,y_test = train_test_split(
3     features,target_classification,test_size=0.2,random_state=42)
```

Run a K-Nearest Neighbors

We can run a K-Nearest Neighbors algorithm to segment the data. We will start with 1 "neighbor" and tweak this parameter later.

In [96]:
```
1 from sklearn.neighbors import KNeighborsClassifier
2 knn=KNeighborsClassifier(n_neighbors=1)
3 knn.fit(X_train,y_train)
```

Out[96]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=1, p=2,
           weights='uniform')
```

In [97]:
```
1 pred = knn.predict(X_test)
```

Run a confusion matrix

In [98]:
```
1 from sklearn.metrics import confusion_matrix
2 print(confusion_matrix(y_test,pred))
```

```
[[ 598  686]
 [ 695 1018]]
```

Run a classification report

In [99]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.46      0.47      0.46      1284
           1       0.60      0.59      0.60      1713

   micro avg       0.54      0.54      0.54      2997
   macro avg       0.53      0.53      0.53      2997
weighted avg       0.54      0.54      0.54      2997
```

Display model accuracy

In [100]:
```
from sklearn.metrics import accuracy_score
KNN_accuracy_score=accuracy_score(y_test,pred)
print(format(KNN_accuracy_score,".2%"))
```

```
53.92%
```

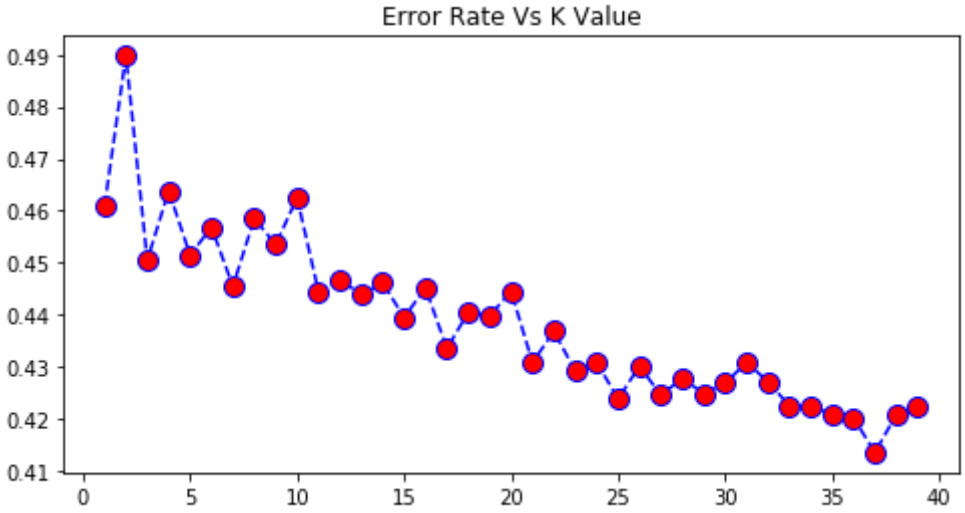
Assess various amount of neighbors for KNN

We can assess what our average error rate would be under various amount of neighbors, and then choose a value which minimizes our error rate.

In [101]:
```
error_rate=[]
for i in range(1,40):
    knn=KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train,y_train)
    pred_i=knn.predict(X_test)
    error_rate.append(np.mean(pred_i!=y_test)) #avg error rate
```

In [102]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.figure(figsize=(8,4))
plt.plot(range(1,40),error_rate,color='blue',linestyle='dashed',
         marker='o',markerfacecolor='red',markersize=10)
plt.title('Error Rate Vs K Value')
```

Out[102]: Text(0.5, 1.0, 'Error Rate Vs K Value')



It looks like the n=37 is the optimal number of neighbors, which we can confirm.

In [103]:
```
min(error_rate)==error_rate[36]
#list indexing starts at 0, so the 37th element is number 36
```

Out[103]: True

Run a K-Nearest Neighbors

Knowing that n=37 is the optimal number of neighbors to minimize error rate, we can run a better KNN model.

In [104]:
```
knn=KNeighborsClassifier(n_neighbors=37)
knn.fit(X_train,y_train)
pred = knn.predict(X_test)
```

Run a classification report

In [105]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.53      0.33      0.41      1284
           1       0.61      0.78      0.68      1713

   micro avg       0.59      0.59      0.59      2997
   macro avg       0.57      0.55      0.54      2997
weighted avg       0.57      0.59      0.56      2997
```

Run a confusion matrix

In [106]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(y_test,pred))
```

```
[[ 423  861]
 [ 378 1335]]
```

Display model accuracy

In [107]:
```
from sklearn.metrics import accuracy_score
KNN_accuracy_score=accuracy_score(y_test,pred)
print(format(KNN_accuracy_score,".2%"))
```

```
58.66%
```

Shuffle the dataframe to eliminate order bias

The dataframe is sorted with 2014 at the top and 2018 at the bottom. We can shuffle our dataframe to make sure that we eliminate bias in our splitting of the data into training and testing. This step is not necessary because train-test split automatically shuffles the data.

In [108]:
```
df = df.sample(frac=1)
```

Run a Decision Tree

We can run a decision tree to classify our data.

In [109]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
X_train, X_test, Y_train, Y_test = train_test_split(features,
                                target_classification, test_size=0.2)
model = DecisionTreeClassifier(max_depth=10)
scores = cross_val_score(model, features, target_classification,
                            scoring="accuracy", cv=5)
scores= cross_val_score(model,X_train,Y_train,scoring="accuracy",cv=5)
print ("Cross Validated Accuracy: %0.3f +/- %0.3f" %
       (scores.mean(), scores.std()))
```

```
Cross Validated Accuracy: 0.668 +/- 0.006
```

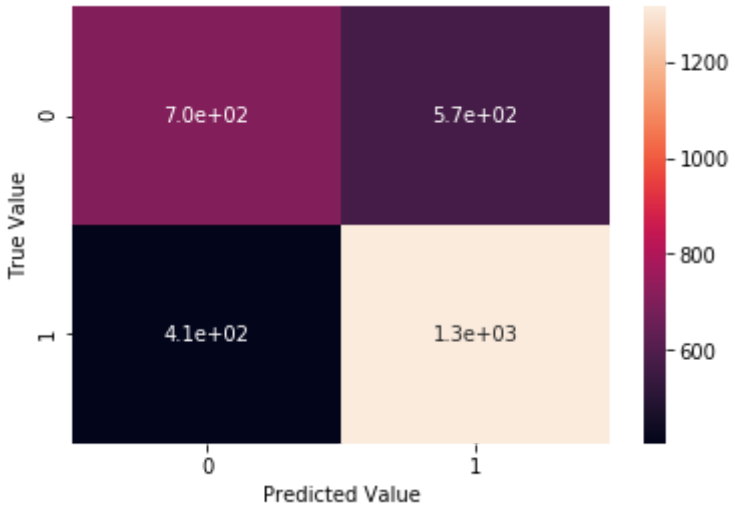
In [110]:
```
model = DecisionTreeClassifier(max_depth=10)
model.fit(X_train, Y_train)
predictions=model.predict(X_test)
```

Run a heatmap of a confusion matrix

We can create a heatmap for our confusion matrix so that it's easier to visualize.

In [111]:
```
import seaborn as sns
from sklearn.metrics import confusion_matrix
predictions=model.predict(X_test)
sns.heatmap(confusion_matrix(Y_test,predictions),annot=True)
plt.xlabel('Predicted Value')
plt.ylabel('True Value')
```

Out[111]:
```
Text(33.0, 0.5, 'True Value')
```



Display model accuracy

In [112]:
```
from sklearn.metrics import accuracy_score
DT_accuracy_score=accuracy_score(Y_test,predictions)
print(format(DT_accuracy_score,".2%"))
```

```
67.37%
```

Run a Random Forest

We can run a Random Forest (which is a collection of decision trees) to classify our data.

In [113]:
```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=200)
scores = cross_val_score(model, features, target_classification,
                         scoring="accuracy", cv=5)
scores= cross_val_score(model,X_train,Y_train,scoring="accuracy",cv=5)
print ("Cross Validated Accuracy: %0.3f +/- %0.3f" % (scores.mean(),
                                                      scores.std()))
```

```
Cross Validated Accuracy: 0.667 +/- 0.005
```

In [114]:
```
model.fit(X_train,Y_train)
```

Out[114]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gin
i',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

In [115]:
```
predictions=model.predict(X_test)
```

Print a classification report

In [116]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.63      0.55      0.59      1277
           1       0.69      0.76      0.72      1720

   micro avg       0.67      0.67      0.67      2997
   macro avg       0.66      0.65      0.66      2997
weighted avg       0.67      0.67      0.67      2997
```

Print a confusion matrix

In [117]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(Y_test,predictions))
```

```
[[ 701  576]
 [ 416 1304]]
```

Display model accuracy

In [118]:
```
from sklearn.metrics import accuracy_score
RF_accuracy_score=accuracy_score(Y_test,predictions)
print(format(RF_accuracy_score,".2%"))
```

```
66.90%
```

Run a Grid Search to optimize hyper-parameters

For many maching learning algorithms, there are several parameters of the model that can be hard to interpret. We can run what is referred to as a GridSearchCV to loop through different parameters so that we can compare the model under different parameters. This will allow us to choose the best model based on the parameters we've tested.

In [119]:
```
from sklearn.model_selection import GridSearchCV
```

In [120]:
```
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 2000,
                                            stop = 2000, num = 1)]
# Number of features to consider at every split
#max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(60, 60, num = 1)]
#max_depth.append(None)
# Minimum number of samples required to split a node
#min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [5]
# Method of selecting samples for training each tree
#bootstrap = [True, False]
```

In [121]:
```
# Create the random grid
param_grid = {'n_estimators': n_estimators,
              # 'max_features': max_features,
               'max_depth': max_depth,
              #'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf}
              #'bootstrap': bootstrap}
print(param_grid)
```

```
{'n_estimators': [2000], 'max_depth': [60], 'min_samples_leaf': [5]}
```

In [122]:
```
grid = GridSearchCV(RandomForestClassifier(),param_grid,verbose=5)
```

In [123]:
```
1 grid.fit(X_train,Y_train)
```

```
Fitting 3 folds for each of 1 candidates, totalling 3 fits
[CV] max_depth=60, min_samples_leaf=5, n_estimators=2000 .............

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent w
orkers.

[CV]  max_depth=60, min_samples_leaf=5, n_estimators=2000, score=0.662496
8726544909, total= 1.5min
[CV] max_depth=60, min_samples_leaf=5, n_estimators=2000 .............

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:  1.6min remaining:
0.0s

[CV]  max_depth=60, min_samples_leaf=5, n_estimators=2000, score=0.668085
1063829787, total= 1.4min
[CV] max_depth=60, min_samples_leaf=5, n_estimators=2000 .............

[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:  3.0min remaining:
0.0s

[CV]  max_depth=60, min_samples_leaf=5, n_estimators=2000, score=0.667834
7934918648, total= 1.4min

[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:  4.5min remaining:
0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:  4.5min finished
```

Out[123]:
```
GridSearchCV(cv='warn', error_score='raise-deprecating',
       estimator=RandomForestClassifier(bootstrap=True, class_weight=Non
e, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators='warn', n_jobs=Non
e,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'n_estimators': [2000], 'max_depth': [60], 'min_sample
s_leaf': [5]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=5)
```

In [124]:
```
1 grid.best_params_
```

Out[124]:
```
{'max_depth': 60, 'min_samples_leaf': 5, 'n_estimators': 2000}
```

In [125]:
```
grid.best_estimator_
```

Out[125]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gin
i',
                       max_depth=60, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=5, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=2000, n_jobs=None,
                       oob_score=False, random_state=None, verbose=0,
                       warm_start=False)
```

In [126]:
```
grid_predictions=grid.predict(X_test)
```

Run a confusion matrix

In [127]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(Y_test,grid_predictions))
```

```
[[ 678  599]
 [ 380 1340]]
```

Run a classification report

In [128]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test,grid_predictions))
```

```
              precision    recall  f1-score   support

           0       0.64      0.53      0.58      1277
           1       0.69      0.78      0.73      1720

   micro avg       0.67      0.67      0.67      2997
   macro avg       0.67      0.66      0.66      2997
weighted avg       0.67      0.67      0.67      2997
```

Display model accuracy

In [129]:
```
from sklearn.metrics import accuracy_score
RF_accuracy_score=accuracy_score(Y_test,grid_predictions)
print(format(RF_accuracy_score,".2%"))
```

```
67.33%
```

Run a Support Vector Machine (SVM)

We can run a Support Vector Machine (SVM) for classification.

In [130]:
```
from sklearn.svm import SVC
model=SVC()
X_train, X_test, Y_train, Y_test = train_test_split(features,
                                    target_classification, test_size=0.2)
model.fit(X_train,Y_train)
```

Out[130]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

In [131]:
```
predictions=model.predict(X_test)
```

Run a confusion matrix

In [132]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(Y_test,predictions))
```

```
[[  97 1210]
 [ 119 1571]]
```

Run a classification report

In [133]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.45      0.07      0.13      1307
           1       0.56      0.93      0.70      1690

   micro avg       0.56      0.56      0.56      2997
   macro avg       0.51      0.50      0.42      2997
weighted avg       0.51      0.56      0.45      2997
```

Run a Grid Search to optimize hyper-parameters

Because it is difficult to have an intuition for the parameters for SVM, we can run a grid search to find the parameters, just like we did for our Random Forest.

In [134]:
```
from sklearn.model_selection import GridSearchCV
param_grid={'C':[0.1,1,10,100,1000],'gamma':[1,0.1,0.01,0.001,0.0001]}
grid = GridSearchCV(SVC(),param_grid,verbose=3)
grid.fit(X_train,Y_train)
```

```
[CV] C=10, gamma=0.001 ...............................................
[CV] ...... C=10, gamma=0.001, score=0.5586690017513135, total=  17.0s
[CV] C=10, gamma=0.001 ...............................................
[CV] ...... C=10, gamma=0.001, score=0.5514392991239049, total=  16.3s
[CV] C=10, gamma=0.001 ...............................................
[CV] ...... C=10, gamma=0.001, score=0.5399249061326659, total=  16.3s
[CV] C=10, gamma=0.0001 ..............................................
[CV] ..... C=10, gamma=0.0001, score=0.5701776332249187, total=  13.5s
[CV] C=10, gamma=0.0001 ..............................................
[CV] ..... C=10, gamma=0.0001, score=0.5829787234042553, total=  13.5s
[CV] C=10, gamma=0.0001 ..............................................
[CV] ..... C=10, gamma=0.0001, score=0.5777221526908636, total=  13.1s
[CV] C=100, gamma=1 ..................................................
[CV] ......... C=100, gamma=1, score=0.5601701275956967, total=  28.5s
[CV] C=100, gamma=1 ..................................................
[CV] ......... C=100, gamma=1, score=0.5602002503128911, total=  27.5s
[CV] C=100, gamma=1 ..................................................
[CV] ......... C=100, gamma=1, score=0.5602002503128911, total=  27.4s
[CV] C=100, gamma=0.1 ................................................
[CV] ....... C=100, gamma=0.1, score=0.5599199399549663, total=  33.6s
```

In [135]:
```
grid.best_params_
```

Out[135]:
```
{'C': 1, 'gamma': 0.0001}
```

In [136]:
```
grid.best_estimator_
```

Out[136]:
```
SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.0001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

In [137]:
```
grid_predictions=grid.predict(X_test)
```

Run a confusion matrix

In [138]:
```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(Y_test,grid_predictions))
```

```
[[ 372  935]
 [ 297 1393]]
```

Run a classification report

In [139]:
```
from sklearn.metrics import classification_report
print(classification_report(Y_test,grid_predictions))
```

```
              precision    recall  f1-score   support

           0       0.56      0.28      0.38      1307
           1       0.60      0.82      0.69      1690

   micro avg       0.59      0.59      0.59      2997
   macro avg       0.58      0.55      0.53      2997
weighted avg       0.58      0.59      0.56      2997
```

Display model accuracy

In [140]:
```
from sklearn.metrics import accuracy_score
SVM_accuracy_score=accuracy_score(Y_test,grid_predictions)
print(format(SVM_accuracy_score,".2%"))
```

```
58.89%
```

Section 6: Interpret Results

We can create a dataframe comparing accuracy across all of our machine learning models.

In [141]:
```
df = pd.DataFrame({"Machine Learning Algorithm":
                   ['Logistic Regression',
                    'K-Nearest Neighbors',
                    'Decision Trees','Random Forest',
                    'Support Vector Machine',
                    'Buy All Stocks','Monkey Guessing'],
"Accuracy": [logistic_accuracy_score,KNN_accuracy_score,
             DT_accuracy_score,RF_accuracy_score,SVM_accuracy_score,
             target_classification.value_counts(normalize=True)[1],.5]}
```

In [142]:
```
df
```

Out[142]:

	Machine Learning Algorithm	Accuracy
0	Logistic Regression	0.600267
1	K-Nearest Neighbors	0.586587
2	Decision Trees	0.673674
3	Random Forest	0.673340
4	Support Vector Machine	0.588922
5	Buy All Stocks	0.560932
6	Monkey Guessing	0.500000

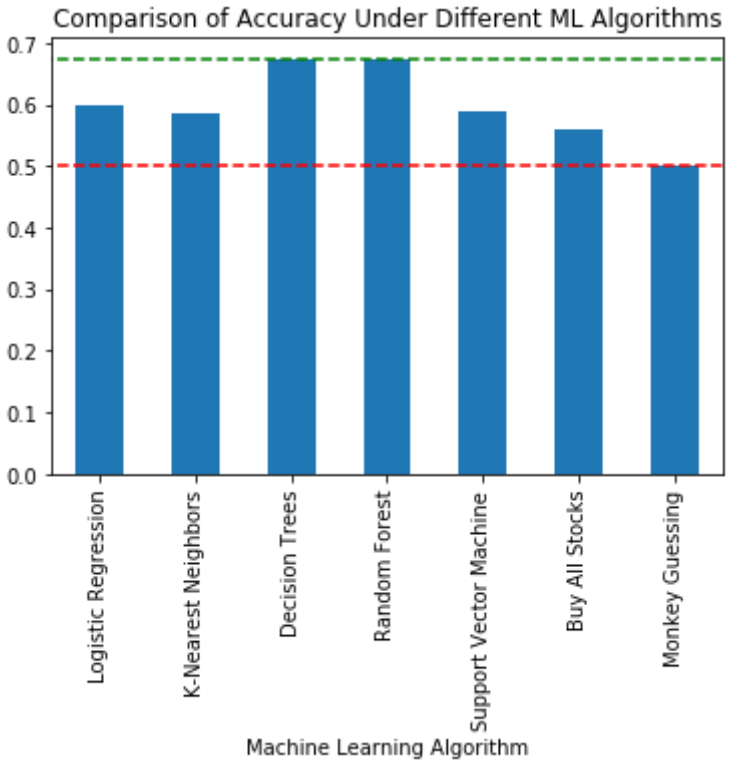
In [143]:
```
df.set_index('Machine Learning Algorithm',inplace=True)
pd.options.display.float_format = '{:.2f}%'.format
df['Accuracy']=round(df['Accuracy'],3)
df
```

Out[143]:

Machine Learning Algorithm	Accuracy
Logistic Regression	0.60%
K-Nearest Neighbors	0.59%
Decision Trees	0.67%
Random Forest	0.67%
Support Vector Machine	0.59%
Buy All Stocks	0.56%
Monkey Guessing	0.50%

In [144]:
```
df.plot(kind='bar',legend=False)
plt.hlines(y=0.5,xmin=-100,xmax=100,linestyle='dashed',color='red')
plt.hlines(y=max(df['Accuracy']),xmin=-100,xmax=100,
           linestyle='dashed',color='green')
plt.ylabel('Accuracy (%)',color='white')
plt.xlabel('Machine Learning Algorithm')
plt.title('Comparison of Accuracy Under Different ML Algorithms')
```

Out[144]: `Text(0.5, 1.0, 'Comparison of Accuracy Under Different ML Algorithms')`



The best performing model is a Random Forest with parameters n_estimators=2000,

min_sample_leafs=5, and max_depth=60. We can utilize this model on 2019 data when available, and see if we can continue to predict stock market performance with 67.13% accuracy.

Section 7: Next Steps and Implementation

We should be excited by the fact that we were able to achieve 67% accuracy. It is very difficult to predict stock market movement, and any edge that we have over baseline is great. Despite this, there are some next steps/follow-ups that would be helpful to improve our analysis going forward:

- Include more data, from years dating further back than 2014.
- Use all of 2014-2018 for our training data and use 2019 for our testing data.
- Evaluate more machine learning models, such as Deep Learning/Neural Networks.
- Dig into Principal Component Analysis (PCA) to reduce dimensionality (essentially, find out which variables are most significant), which would substantially improve run-time.
- Include tax implications for model implementation.
- Try to create a model that provides more granularity than yearly periods of buy/sell (this will be difficult given the financial data).

Implementation would be to create an equally-weighted portfolio of stocks that the model tells you to get at the start of the next period. Then, one year later, re-run the model, sell off the stocks that the model predicts will go down that year, and hold the stocks that the model predicts will go up that year. Rinse and repeat each year. Each year, re-run the model (with an extra year's data) to refine the model and make sure that the model is working accurately.