MACHINE LEARNING TECHNIQUES IN FINANCE

Introduction

From its conception, ***Finance*** has been a data-driven industry. Financial institutions gather large amounts of data that provide them with important insights, and in recent years, ***Machine Learning*** has become an increasingly important tool for making sense of that data. Since these algorithms make quick and fairly accurate decisions, they are used for various purposes, such as fraud detection and prevention, portfolio management, stock price predictions and so on.

It is common knowledge that ***Machine Learning*** is primarily divided into two categories:

- ***Unsupervised Machine Learning***, and
- ***Supervised Machine Learning***.

Let us now give an example for each of these two categories and use some of their most reputable algorithms.

Section 1: Unsupervised Machine Learning.

Unsupervised Machine Learning is a process that seeks a grouping of data (if it exists) into an (unknown) number of classes from a given dataset.

The goal of such algorithms is to identify latent groupings of observations, which allow us to predict the class of observations even without a labelled response vector. There are many clustering algorithms and they have a plethora of approaches to identifying the clusters in data.

Our Example:

It is said that companies from the same industry behave similarly in their daily stock movements, as they are affected in an analogous way by market forces. Thus, what we would logically expect is, for instance, to see stocks like Microsoft (*MSFT*) and Apple (*AAPL*) to be clustered in the same group. So, what we want, is to create such algorithms that will aggregate companies into reasonably separated clusters, based on their daily stock movements.

Let us take the following 21 stocks from the stock market:

- From the **Technological Sector**: Amazon (*AMZN*), Tesla (*TSLA*), Advanced Micro Devices (*AMD*), NVIDIA (*NVDA*), Microsoft (*MSFT*) and Apple (*AAPL*).
- From the **Financial Services Sector**: Morningstar (*MORN*), S&P Global (*SPGI*), Moody's (*MCO*), Nasdaq (*NDAQ*) and MSCI (*MSCI*).

- From the **Automotive Sector**: Toyota (*TM*), Honda (*HMC*), Blue Bird (*BLBD*), Ford Motor Company (*F*) and General Motors Company (*GM*).

- From the **Pharmaceutical Sector**: Pfizer (*PFE*), GlaxoSmithKlyne (*GSK*), Astrazeneca PLC (*AZN*), Amgen (*AMGN*) and Abbvie (*ABBV*).

It should be mentioned that this distinction of the 21 stocks into these four categories will be unknown in this example. This is a measuring stick, to see if our example works and to check the performance of the algorithms.

What we want to do, is to put these stocks into groups, using their daily movement. We will do that in the following manner:

Importing the Necessary Libraries:

In [1]:
```
from pandas_datareader import data
import matplotlib.pyplot as plt
import pandas as pd
import datetime as dt
import numpy as np
```

Getting the Data and Calculating the Daily Stock Movements:

In [2]:
```
companies = {
  'Amazon':'AMZN',
  'Tesla':'TSLA',
  'Advanced Micro Devices':'AMD',
  'NVIDIA':'NVDA',
  'Microsoft':'MSFT',
  'Apple':'AAPL',
  'Morningstar':'MORN',
  'S&P Global':'SPGI',
  'Moody\'s':'MCO',
  'Nasdaq':'NDAQ',
  'MSCI':'MSCI',
  'Toyota':'TM',
  'Honda':'HMC',
  'Blue Bird':'BLBD',
  'Ford Motor Company':'F',
  'General Motors Company':'GM',
  'Pfizer':'PFE',
  'GlaxoSmithKlyne':'GSK',
  'Astrazeneca PLC':'AZN',
  'Amgen':'AMGN',
  'Abbvie':'ABBV'
  }
```

Since we know the sector each one of these stocks belongs to, we can use that information to evaluate our clustering algorithms. Thus, we make an array with the **Ground Truth**, which will be used to see how well we have categorized our stocks.

In [3]:
```
a = np.array([[0, 6], [1, 5], [2, 5], [3, 5]])
ground_truth = np.repeat(a[:, 0], a[:, 1])
```

In [4]:
```
# Let us get the data for a 2-year period (Start of 2019 - End of 2020):
start_date = dt.datetime(2019, 1, 1)
end_date = dt.datetime(2020, 12, 31)
df = data.DataReader(list(companies.values()), 'yahoo', start_date, end_date)

# If we want, we can check if there are any missing values in 'Open' and 'Close' columns per stock:
#np.isnan(df.loc[:,['Open', 'Close']]).any() # There are no missing values.
```

In [5]:
```
# Get the 'Open' and 'Close' values and put them into an array:
stock_open = np.array(df['Open']).T
stock_close = np.array(df['Close']).T

# Calculating the daily movements of the stocks:
movements = stock_close - stock_open
```

Something to consider: What we have now are the daily movements of each stock, so if we add them all up, over the course of two years (2019 - 2020), then we could consider investing in some of them. For those with positive values, it will be advisable to go long, while shorting the stocks with negative values.

In [6]:
```
movement_sums = np.sum(movements, 1) # Sum on columns.
for i in range(len(companies)):
  print('company:{}, Change:{}'.format(df['Close'].columns[i], movement_sums[i]))
```

```
company:AMZN, Change:3.6958541870117188
company:TSLA,  Change:250.12969589233398
company:AMD,   Change:11.810003280639648
company:NVDA,  Change:16.21504020690918
company:MSFT,  Change:40.149993896484375
company:AAPL,  Change:44.462425231933594
company:MORN, Change:150.70996856689453
company:SPGI,   Change:11.260147094726562
company:MCO,   Change:50.80989074707031
```

```
company:NDAQ,  Change:6.929862976074219
company:MSCI,   Change:187.7698974609375
company:TM,      Change:-18.199981689453125
company:HMC,    Change:-5.090019226074219
company:BLBD,  Change:-10.937999725341797
company:F,         Change:-3.680009365081787
company:GM,      Change:-10.63003158569336
company:PFE,     Change:-29.422351837158203
company:GSK,    Change:-7.519994735717773
company:AZN,    Change:-15.360057830810547
company:AMGN, Change:45.63993835449219
company:ABBV,  Change:-21.480079650878906
```

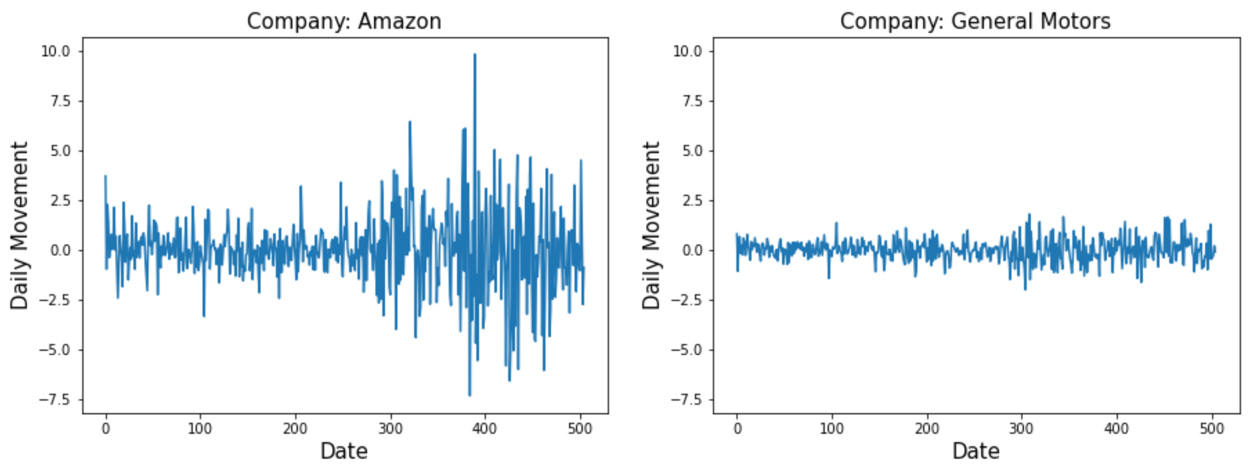
Thus, looking at the output above, it would be advisable to go long on stocks such as Tesla (*TSLA*) and Morningstar (*MORN*), while shorting Pfizer (*PFE*) and Abbvie (*ABBV*).

Normalizing our Feature Array:

Do we really need to normalize the daily movements of stocks? Let us take two stocks from different categories (say **Amazon** and **General Motors**) and see how they differ in their daily variations.

In [7]:
```python
plt.figure(figsize = (15,5))

ax1 = plt.subplot(1,2,1)
plt.title('Company: Amazon', fontsize = 15)
plt.xticks(fontsize = 10)
plt.yticks(fontsize = 10)
plt.xlabel('Date', fontsize = 15)
plt.ylabel('Daily Movement', fontsize = 15)
plt.plot(movements[0])

plt.subplot(1,2,2, sharey = ax1)
plt.title('Company: General Motors', fontsize = 15)
plt.xticks(fontsize = 10)
plt.yticks(fontsize = 10)
plt.xlabel('Date', fontsize = 15)
plt.ylabel('Daily Movement', fontsize = 15)
plt.plot(movements[15])
```

Out [7]:
```
[<matplotlib.lines.Line2D at 0x25d69eb9580>]
```



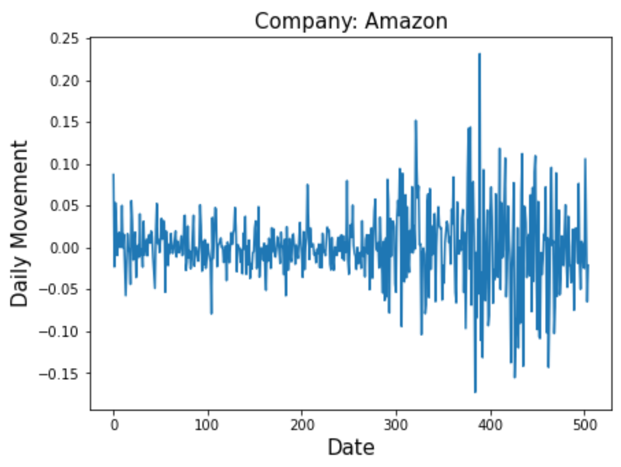
As we can plainly see, these two stocks differ a lot as far as their scale is concerned. Amazon's stock seems more volatile than the stock for General Motors. Therefore it is logical, before our analysis to normalize the values of the daily movements so as to account for this volatile behaviour and have both of these stocks contribute equally to the analysis.

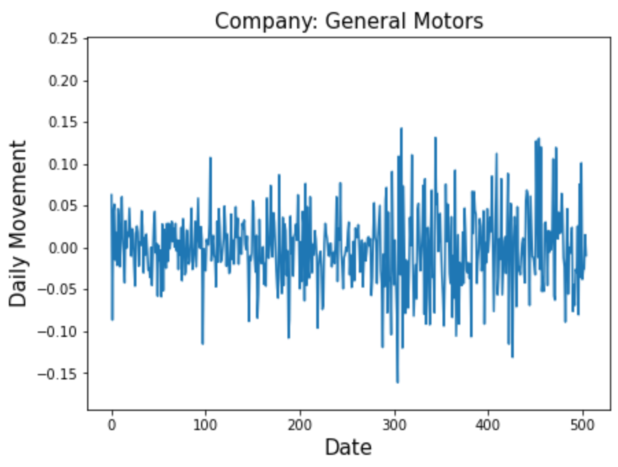
In [8]:
```
from sklearn.preprocessing import Normalizer

c_movements = movements.copy()
normalizer = Normalizer()
X = normalizer.fit_transform(c_movements)
```

Now let us see whether things got a little better.

In [9]:
```
plt.figure(figsize = (15,5))

ax1 = plt.subplot(1,2,1)
plt.title('Company: Amazon', fontsize = 15)
plt.xticks(fontsize = 10)
plt.yticks(fontsize = 10)
plt.xlabel('Date', fontsize = 15)
plt.ylabel('Daily Movement', fontsize = 15)
plt.plot(X[0])

plt.subplot(1,2,2, sharey = ax1)
plt.title('Company: General Motors', fontsize = 15)
plt.xticks(fontsize = 10)
plt.yticks(fontsize = 10)
plt.xlabel('Date', fontsize = 15)
plt.ylabel('Daily Movement', fontsize = 15)
plt.plot(X[15])
```

Out [9]:
```
[<matplotlib.lines.Line2D at 0x25d6b9c3340>]
```





These are not perfect, but compared to the graphs above, they are much better. At the very least, these two stocks (as well as all the other stocks) are now similarly affecting our analysis.

After doing that, we are now ready to begin clustering our data. So first up, is the **Hierarchical Clustering** technique.

A. Hierarchical Clustering Analysis (Agglomerative Clustering Method).

This procedure attempts to identify and build a hierarchy of relatively homogeneous groups of cases (or variables) based on selected characteristics of data. It should be noted that we do not know in advance how many clusters we want, but in order to determine just that, we use tree-like visual representation of the observations. All the clusters are combined (or separated) based on distance measures between and within clusters.

Hierarchical Clustering Analysis can be done in two ways:

- **Divisive Hierarchical Clustering**: Here all observations start as part of one big cluster, and splits are performed over and over again, as one moves down the hierarchy. This approach is also known as a "top-down" approach. Should a cluster be separated, these two new pieces can never go back to being one again. That is why this procedure is called hierarchical, because there is a structure to the separation of the clusters
- **Agglomerative Hierarchical Clustering**: This is the very opposite of the Divisive procedure described above. Each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. It is also known as a "bottom-up" approach. One particular characteristic of this procedure is that once a cluster is formed, it can not be separated again.

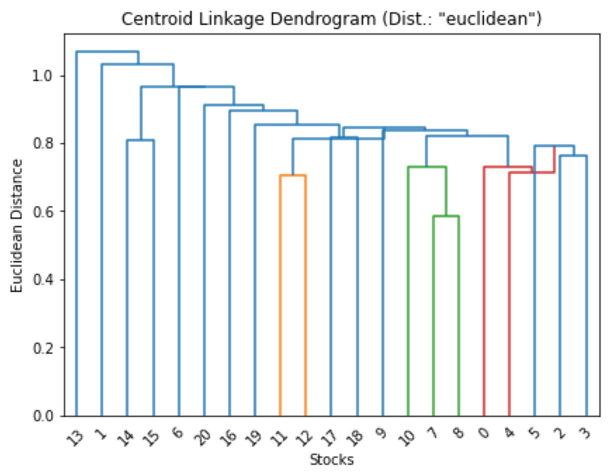
Since these two approaches get the same results, we shall use the **Agglomerative Hierarchical Clustering** method to separate our stocks into groups. In order to merge new clusters, this procedure uses two distance-based metrics:

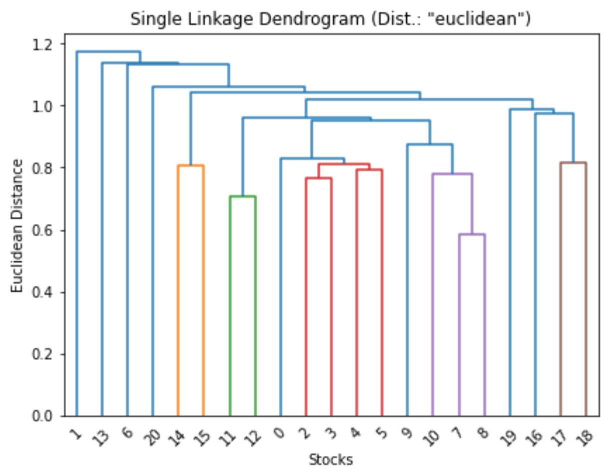
- ***metric*** or ***affinity***: This distance is the pairwise distance between individual observations and it can take values from the ***pdist*** function from the ***scipy*** library. Such values could be: ***'euclidean'***, ***'minkowski'***, ***'cityblock'***, etc. (more on: https://docs.scipy.org/doc/scipy-0.17.0/reference/generated/generated/scipy.spatial.distance.pdist.html).

- ***method*** or ***linkage***: This distance metric determines how the distance between clusters will be measured. It takes values such as: ***'single'***, ***'complete'***, ***'average'***, ***'ward'***, etc. (more on: https://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html).

Let us now see this method in action. First of all, we shall form the dendrograms in order to see if we can detect the number of clusters in our data.

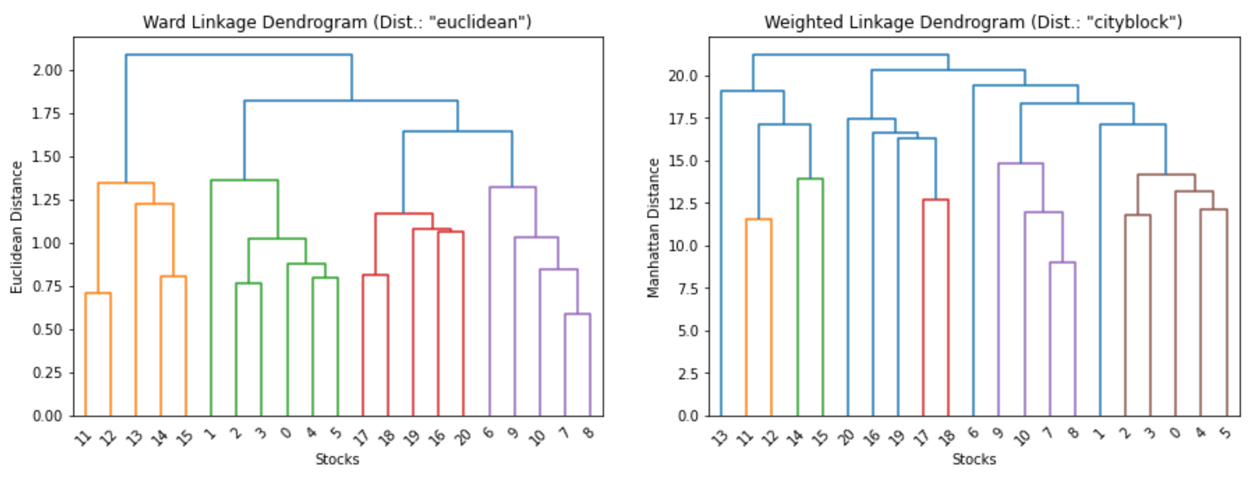
In [10]:
```
import scipy.cluster.hierarchy as sch

plt.figure(figsize = (15,5))

ax1 = plt.subplot(121)
dendrogram = sch.dendrogram(sch.linkage(X, method = 'centroid', metric = 'euclidean'))
plt.title('Centroid Linkage Dendrogram (Dist.: "euclidean")')
plt.xlabel('Stocks')
plt.ylabel('Euclidean Distance')

ax2 = plt.subplot(122)
dendrogram = sch.dendrogram(sch.linkage(X, method = 'single', metric = 'euclidean'))
plt.title('Complete Linkage Dendrogram (Dist.: "euclidean")')
plt.xlabel('Stocks')
plt.ylabel('Euclidean Distance')
plt.show()
```

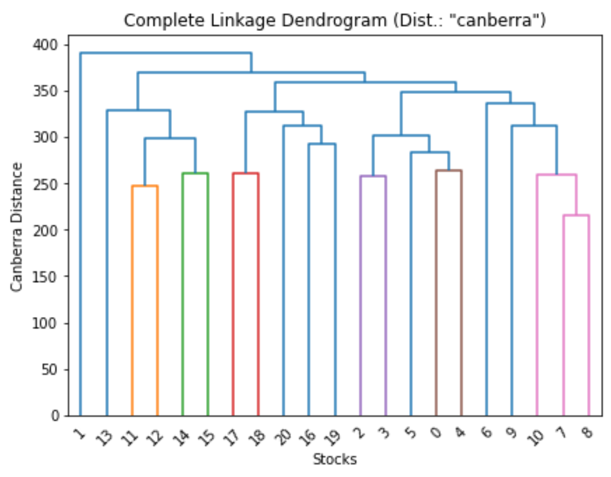


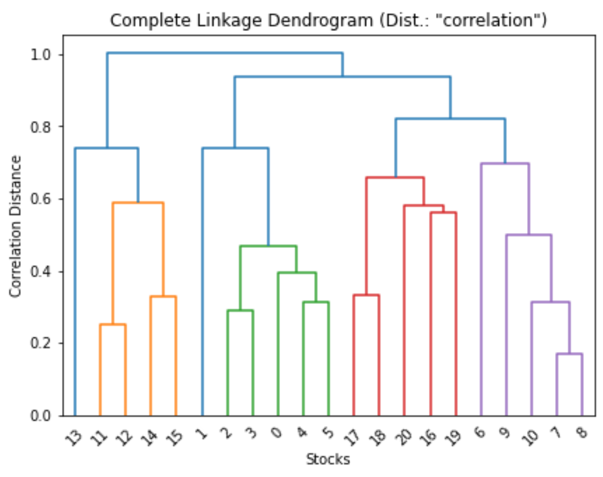


It appears that some of these methods have fallen for the so called *'chaining effect'*. One observation is connected to another, then the second observation to a third and so on. These dendrograms do not form proper clusters and could indicate outliers, but since we want to use all the 21 stocks, we shall not be getting rid of any observations and thus, ignore these methods and use others that produce adequate clustering results.

In [11]:
```python
import scipy.cluster.hierarchy as sch

plt.figure(figsize = (15,5))

ax1 = plt.subplot(121)
dendrogram1 = sch.dendrogram(sch.linkage(X, method = 'ward', metric = 'euclidean'))
plt.title('Ward Linkage Dendrogram (Dist.: "braycurtis")')
plt.xlabel('Stocks')
plt.ylabel('Euclidean Distance')

ax2 = plt.subplot(122)
dendrogram2 = sch.dendrogram(sch.linkage(X, method = 'weighted', metric = 'cityblock'))
plt.title('Weighted Linkage Dendrogram (Dist.: "cityblock")')
plt.xlabel('Stocks')
plt.ylabel('Manhattan Distance')
plt.show()

plt.figure(figsize = (15,5))
ax1 = plt.subplot(121)
dendrogram3 = sch.dendrogram(sch.linkage(X, method = 'complete', metric = 'canberra'))
plt.title('Complete Linkage Dendrogram (Dist.: "canberra")')
plt.xlabel('Stocks')
plt.ylabel('Canberra Distance')

ax2 = plt.subplot(122)
dendrogram4 = sch.dendrogram(sch.linkage(X, method = 'complete', metric = 'correlation'))
plt.title('Complete Linkage Dendrogram (Dist.: "correlation")')
plt.xlabel('Stocks')
plt.ylabel('Correlation Distance')
plt.show()
```







As we can see from the dendrograms above, we definitely have two clusters, but it is strongly indicated that we could also have three, or four clusters. This is a very arduous procedure, as we can never be certain about the exact number of clusters that we have. In our example, most of the dendrograms that did not fall into 'chaining effect', represented three to four clusters. Thus, in this exercise, we shall choose to believe that we have four clusters.

After all, we did know that we would have somewhere around four of them, but it still is very comforting to know that even if we did not, the dendrograms would eventually point us in the right direction, if we were sharp enough to notice.

The most promising one of them all looks to be the one that uses complete linkage and computes distance based on the correlation of the data. So let us use it and see how well it can cluster our stocks.

First for three clusters:

In [12]:
```
from sklearn.cluster import AgglomerativeClustering

hc = AgglomerativeClustering(n_clusters = 3, affinity = 'correlation', linkage = 'complete')
labels = hc.fit_predict(X)
```

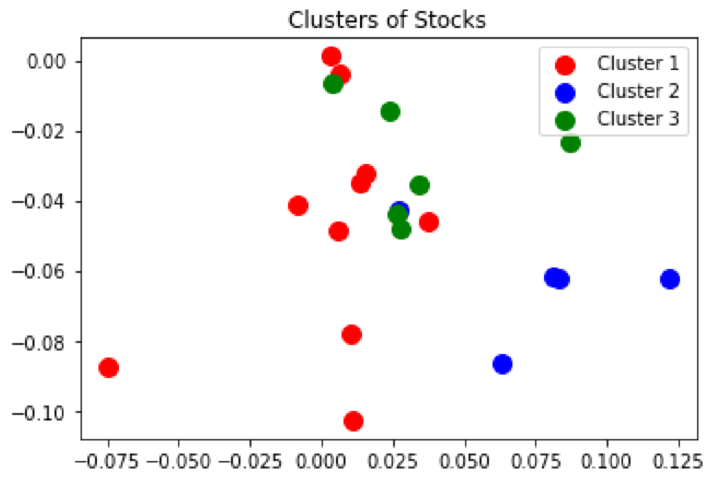
In [13]:
```
df1 = pd.DataFrame({'labels':labels,'companies':list(companies)}).sort_values(by=['labels'],axis = 0)
df1
```

Out [13]:

	labels	companies
10	0	MSCI
18	0	Astrazeneca PLC
17	0	GlaxoSmithKlyne
16	0	Pfizer
19	0	Amgen
9	0	Nasdaq
8	0	Moody's
20	0	Abbvie
6	0	Morningstar
7	0	S&P Global

11	1	Toyota
12	1	Honda
13	1	Blue Bird
14	1	Ford Motor Company
15	1	General Motors Company
5	2	Apple
4	2	Microsoft
3	2	NVIDIA
2	2	Advanced Micro Devices
1	2	Tesla
0	2	Amazon

In [14]:
```
plt.scatter(X[labels == 0, 0], X[labels == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
plt.scatter(X[labels == 1, 0], X[labels == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
plt.scatter(X[labels == 2, 0], X[labels == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.title('Clusters of Stocks')
plt.legend()
plt.show()
```



As expected, two groups got merged into one, as they had no other cluster to belong to (the **Financial Services Sector** and the **Pharmaceutical Sector**). The **Automotive Sector** as well as the **Technological Sector** got classified correctly.

Therefore, all in all, this was not such a big disaster. If we did not attempt anything else, we would have gotten two sectors correctly classified.

Now, let us try the same thing, but for four clusters.

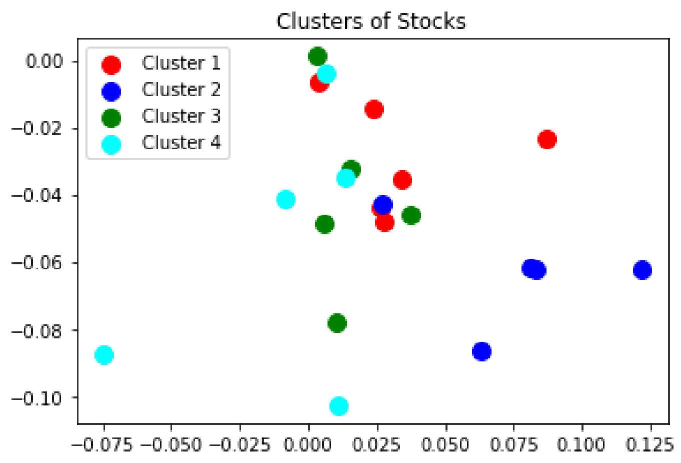
In [15]:
```
from sklearn.cluster import AgglomerativeClustering

hc = AgglomerativeClustering(n_clusters = 4, affinity = 'correlation', linkage = 'complete')
labels = hc.fit_predict(X)
```

In [16]:
```
df2 = pd.DataFrame({'labels':labels,'companies':list(companies)}).sort_values(by=['labels'],axis = 0)
df2
```

Out [16]:

	labels	companies
0	0	Amazon
1	0	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft
5	0	Apple
15	1	General Motors Company
14	1	Ford Motor Company
13	1	Blue Bird
12	1	Honda
11	1	Toyota
10	2	MSCI
9	2	Nasdaq
8	2	Moody's
7	2	S&P Global
6	2	Morningstar
16	3	Pfizer
17	3	GlaxoSmithKlyne
18	3	Astrazeneca PLC
19	3	Amgen
20	3	Abbvie

In [17]:
```
plt.scatter(X[labels == 0, 0], X[labels == 0, 1], s = 100, c = 'red', label = 'Cluster 1')
plt.scatter(X[labels == 1, 0], X[labels == 1, 1], s = 100, c = 'blue', label = 'Cluster 2')
plt.scatter(X[labels == 2, 0], X[labels == 2, 1], s = 100, c = 'green', label = 'Cluster 3')
plt.scatter(X[labels == 3, 0], X[labels == 3, 1], s = 100, c = 'cyan', label = 'Cluster 4')
plt.title('Clusters of Stocks')
plt.legend()
plt.show()
```



It definitely looks better than before, but let us use the **Ground Truth** and see how good this approach really is. What we shall do, is construct a **Confusion Matrix**, which is a specific table layout that will visually show us the performance of our algorithm. In order to do that, first we will sort our resulting data frame (*df2*) based on the index.

In [18]:
```
df3 = pd.DataFrame.sort_index(df2)
df3
```

Out [18]:

	labels	companies
0	0	Amazon
1	0	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft
5	0	Apple
6	2	Morningstar
7	2	S&P Global
8	2	Moody's
9	2	Nasdaq
10	2	MSCI
11	1	Toyota
12	1	Honda
13	1	Blue Bird
14	1	Ford Motor Company
15	1	General Motors Company
16	3	Pfizer
17	3	GlaxoSmithKlyne
18	3	Astrazeneca PLC
19	3	Amgen
20	3	Abbvie

Now we have to switch some labels. See, the **Financial Sector** got mostly grouped in group 3 (*label:2*), while the **Automotive Sector** in group 2(*label:1*). This sort of thing would mix up our Confusion matrix and make it show like they were misclassified, which they were not. Hence, let us switch those labels.

In [19]:
```
df3.loc[df3['labels']==2, 'labels'] = 4
df3.loc[df3['labels']==1, 'labels'] = 2
df3.loc[df3['labels']==4, 'labels'] = 1
df3
```

Out [19]:

	labels	companies
0	0	Amazon
1	0	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft

5	0	Apple
6	1	Morningstar
7	1	S&P Global
8	1	Moody's
9	1	Nasdaq
10	1	MSCI
11	2	Toyota
12	2	Honda
13	2	Blue Bird
14	2	Ford Motor Company
15	2	General Motors Company
16	3	Pfizer
17	3	GlaxoSmithKlyne
18	3	Astrazeneca PLC
19	3	Amgen
20	3	Abbvie

Confusion Matrix:

In [20]:
```
pd.crosstab(ground_truth, df3['labels'], rownames=['Actual'], colnames=['Predicted'])
```

Out [20]:

Predicted	**0**	**1**	**2**	**3**
Actual				
0	6	0	0	0
1	0	5	0	0
2	0	0	5	0
3	0	0	0	5

A perfect classification with no errors whatsoever. Usually, grouping using **Hierarchical Clustering Analysis** is a tricky procedure and often does not lead to desired results. And keep in mind that we used the best-case scenario. There are so many options that we can get lost, while trying to pick the one that will give us adequate results. Even though, Ward's Method along with Euclidean Distance would end up with the same results.

Since we have the **Ground Truth**, let us also calculate the **Adjusted Rand Index**. The Rand Index computes a similarity measure between two clusters by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusters. It takes values between 0 and 1 and the closer it is to 1, the better our clustering worked.

Adjusted Rand Index:

In [21]:
```
from sklearn.metrics.cluster import adjusted_rand_score

adjusted_rand_score(ground_truth, df3['labels'])
```

Out [21]:
```
1.0
```

Perfect score, all the stocks got grouped exactly where they belonged, just as we suspected from the **Confusion Matrix** above. In conclusion, this was **Hierarchical Clustering Analysis**. It can be tricky especially since one has to look at a lot of dendrograms to get a feeling for the approximate number of clusters, but it gives adequate and quick results. Although mostly they will not be as perfect as in our example.

Next up, is **Partitioning Clustering**.

B. Partitioning Clustering (K-Means Clustering).

As we have previously mentioned, when we cluster the observations of a data set, what we essentially need is to partition them into distinct groups so that the observations within each group are similar to each other, while observations in different groups are different from each other.

One of the most common partitioning clustering techniques, is **K-Means Clustering**. What it does, is it partitions a data set into K distinct, non-overlapping clusters. It differs from **Hierarchical Clustering Analysis** in two ways:

- In order to use **K-Means Clustering**, we have to know the number of total clusters before we run our algorithm.
- If two observations form a cluster, they can be divided later on and become parts of different clusters. This merging and division of cluster observations is repeated until the algorithm converges.

The algorithm is pretty simple and works in the following way:

1. A number of K cluster "centers" are formed at random locations, or based on initial values.
2. For every observation of our data, the distance between each observation and the k center points is calculated and each of those observations are assigned to a center, which is nearest to them.
3. Based on the observations in each cluster, new center points are calculated and assigned to each cluster.
4. Steps 2 and 3 are repeated until we have reached convergence, meaning that no observation changes in cluster membership.

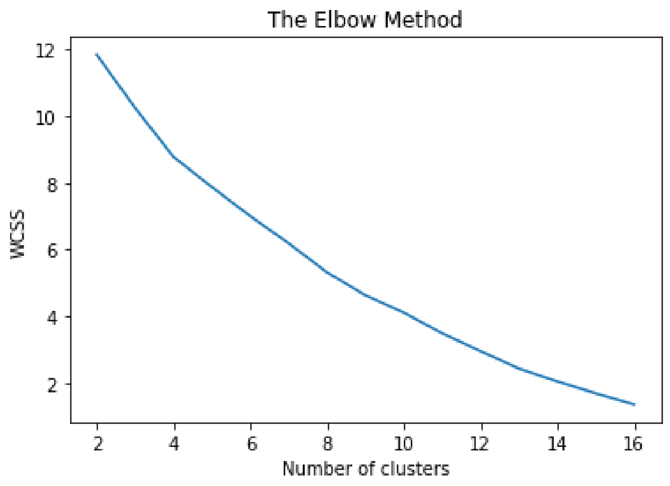
We should also keep in mind that in order for this technique to work successfully, it helps if the clusters are convex shaped, also if our data are scaled equally and finally, if each cluster has a similar number of observations.

Picking the Right Number of Clusters:

As we have said, this technique demands to know the number of clusters beforehand. Based on the dendrograms, which we depicted in **Hierarchical Clustering Analysis**, we ended up picking either three or four clusters. We expect that this technique will also perform adequately under those two options. Fortunately though, **K-Means Clustering** has its own techniques to determine the optimal number of clusters into which our data may be grouped.

Let us try the so called **"Elbow Method"**:

In [22]:
```
from sklearn.cluster import KMeans
wcss = []
for i in range(2, 17):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 300)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(2, 17), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



This method works by utilizing something called inertia, which is within-cluster sums of squares (*WCSS*). In other words, it is the sum of squared distances of samples to their closest cluster center.

In the graph we are looking for the point at which our line bends in a discernible way (much like an elbow). Here it looks like this point is at four, or maybe five or even six clusters. If we are being candid, there were little to no evidence of having five clusters, while we ran **Hierarchical Clustering Analysis**, and therefore it is highly unlikely to have five or more clusters, thus it is probably four.

K-Means Clustering (for four clusters):

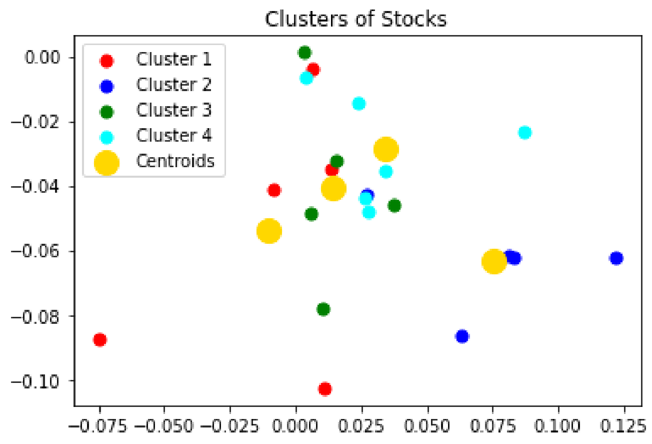
In [23]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters = 4, max_iter = 1000, random_state = 300)
labels = kmeans.fit_predict(X)
```

In [24]:
```
df4 = pd.DataFrame({'labels':labels,'companies':list(companies)}).sort_values(by=['labels'],axis = 0)
df4
```

Out [24]:

	labels	companies
20	0	Abbvie
18	0	Astrazeneca PLC
17	0	GlaxoSmithKlyne
16	0	Pfizer
19	0	Amgen
15	1	General Motors Company
14	1	Ford Motor Company
13	1	Blue Bird
12	1	Honda
11	1	Toyota
10	2	MSCI
8	2	Moody's
7	2	S&P Global
6	2	Morningstar
9	2	Nasdaq
5	3	Apple
4	3	Microsoft
3	3	NVIDIA
2	3	Advanced Micro Devices
1	3	Tesla
0	3	Amazon

In [25]:
```
plt.scatter(X[labels == 0, 0], X[labels == 0, 1], s = 50, c = 'red', label = 'Cluster 1')
plt.scatter(X[labels == 1, 0], X[labels == 1, 1], s = 50, c = 'blue', label = 'Cluster 2')
plt.scatter(X[labels == 2, 0], X[labels == 2, 1], s = 50, c = 'green', label = 'Cluster 3')
plt.scatter(X[labels == 3, 0], X[labels == 3, 1], s = 50, c = 'cyan', label = 'Cluster 4')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s = 200, c = 'gold', label =
'Centroids')
plt.title('Clusters of Stocks')
plt.legend()
plt.show()
```



Seems like a perfect fit, but to be certain, we will once again calculate and present the **Confusion Matrix** and the **Adjusted Rand Index**. Before we do that, we shall need to re-label all the labels of our data frame above (*df4*) so that they match the ones used in our **Ground Truth** array.

In [26]:
```
df5 = pd.DataFrame.sort_index(df4)
df5
```

Out [26]:

	labels	companies
0	3	Amazon
1	3	Tesla
2	3	Advanced Micro Devices
3	3	NVIDIA
4	3	Microsoft
5	3	Apple
6	2	Morningstar
7	2	S&P Global
8	2	Moody's
9	2	Nasdaq
10	2	MSCI
11	1	Toyota
12	1	Honda
13	1	Blue Bird
14	1	Ford Motor Company
15	1	General Motors Company
16	0	Pfizer
17	0	GlaxoSmithKlyne
18	0	Astrazeneca PLC
19	0	Amgen
20	0	Abbvie

In [27]:
```
df5.loc[df5['labels']==3, 'labels'] = 4
df5.loc[df5['labels']==0, 'labels'] = 3
df5.loc[df5['labels']==4, 'labels'] = 0
df5.loc[df5['labels']==1, 'labels'] = 5
df5.loc[df5['labels']==2, 'labels'] = 1
df5.loc[df5['labels']==5, 'labels'] = 2
df5
```

Out [27]:

	labels	companies
0	0	Amazon
1	0	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft
5	0	Apple
6	1	Morningstar
7	1	S&P Global
8	1	Moody's
9	1	Nasdaq
10	1	MSCI
11	2	Toyota
12	2	Honda
13	2	Blue Bird
14	2	Ford Motor Company
15	2	General Motors Company
16	3	Pfizer
17	3	GlaxoSmithKlyne
18	3	Astrazeneca PLC
19	3	Amgen
20	3	Abbvie

<u>Confusion Matrix:</u>

In [28]:
```
pd.crosstab(ground_truth, df5['labels'], rownames=['Actual'], colnames=['Predicted'])
```

Out [28]:

Predicted	**0**	**1**	**2**	**3**
Actual				
0	6	0	0	0
1	0	5	0	0
2	0	0	5	0
3	0	0	0	5

<u>Adjusted Rand Index:</u>

In [29]:
```
from sklearn.metrics.cluster import adjusted_rand_score

adjusted_rand_score(ground_truth, df5['labels'])
```

Out [29]:
```
1.0
```

Once again, a perfect fit. All the stocks got clustered into the correct groups, just as we wanted. This result of course, was dependent on where we initiate our algorithm. Here it initiated on random initial values. If we tinker a bit with the ***random_state*** of the algorithm, we could get to another solution which is not as good as this one.

For example:

In [30]:
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters = 4, max_iter = 1000, random_state = 500)
labels = kmeans.fit_predict(X)
```

In [31]:
```
df6 = pd.DataFrame({'labels':labels,'companies':list(companies)}).sort_values(by=['labels'],axis = 0)
df6
```

Out [31]:

	labels	companies
0	0	Amazon
1	0	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft
5	0	Apple
9	1	Nasdaq
8	1	Moody's
10	1	MSCI
6	1	Morningstar
7	1	S&P Global
12	2	Honda
13	2	Blue Bird
14	2	Ford Motor Company
15	2	General Motors Company
19	3	Amgen
11	3	Toyota
16	3	Pfizer
17	3	GlaxoSmithKlyne
18	3	Astrazeneca PLC
20	3	Abbvie

Confusion Matrix:

In [32]:
```
pd.crosstab(ground_truth, df6['labels'], rownames=['Actual'], colnames=['Predicted'])
```

Out [32]:

Predicted	0	1	2	3
Actual				
0	6	0	0	0
1	0	5	0	0
2	0	0	4	1
3	0	0	0	5

<u>Adjusted Rand Index:</u>

In [33]:
```
adjusted_rand_score(ground_truth, df6['labels'])
```

Out [33]:
```
0.87374749498998
```

Just because we initiated our algorithm on another starting point, now the centers that were formed included an erroneously grouped observation (Toyota into the **Pharmaceutical Sector**). Therefore, we should note this, when running the **K-Means** algorithm.

Let us now explore how **Density-Based Clustering** groups our data.

C. Density-Based Clustering.

Yet another famous clustering technique is the **Density-Based Clustering**. Just as its name suggests, it views clusters in forms of density. In our data, observations with similar characteristics should stick close to each other, as opposed to observations that differ from each other. That is why the logic of this algorithm is that our data will be separated into high and low density areas. The high density areas will be considered as individual clusters and if the observations are not a part of that area, they are believed to be outliers.

As opposed to the **K-Means Clustering** algorithm, **Density-Based Clustering** does not assume that the shape of our clusters will be convex. Instead it is very flexible in this way, letting clusters be of any shape possible, as long as they are highly densed. This could also be considered a downside of this algorithm, because if there are clusters of varying density, then this can make it hard for this algorithm to identify them.

Density-Based Clustering does not need to know beforehand the number of clusters. It does however need to know the minimum number of data points that constitute a cluster, as well as, the maximum distance between each observation point.

It works like this:

1. One of the observations is chosen at random.
2. Based on the distance and the minimum number of neighbours, this observation is considered whether it should be a part of a cluster.
3. It repeats recursively for each of the neighbouring data points to the chosen observation.

4. The next observation is chosen, taking us back to *Step 1*.

Let us now see how **Density-Based Clustering** performs on our data:

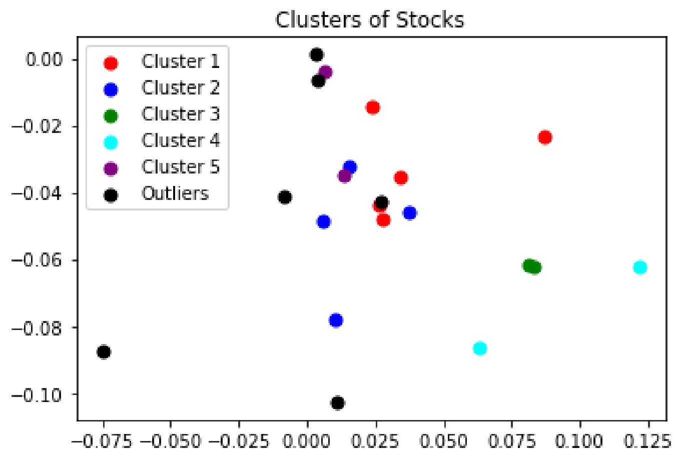
In [34]:
```
from sklearn.cluster import DBSCAN

dbscan = DBSCAN(eps=0.95, min_samples=2)
labels = dbscan.fit_predict(X)
```

In [35]:
```
df7 = pd.DataFrame({'labels':labels,'companies':list(companies)}).sort_values(by=['labels'],axis = 0)
df7
```

Out [35]:

	labels	companies
20	-1	Abbvie
1	-1	Tesla
16	-1	Pfizer
13	-1	Blue Bird
19	-1	Amgen
6	-1	Morningstar
0	0	Amazon
4	0	Microsoft
3	0	NVIDIA
2	0	Advanced Micro Devices
5	0	Apple
7	1	S&P Global
8	1	Moody's
9	1	Nasdaq
10	1	MSCI
11	2	Toyota
12	2	Honda
14	3	Ford Motor Company
15	3	General Motors Company
17	4	GlaxoSmithKlyne
18	4	Astrazeneca PLC

In [36]:
```
plt.scatter(X[labels == 0, 0], X[labels == 0, 1], s = 50, c = 'red', label = 'Cluster 1')
plt.scatter(X[labels == 1, 0], X[labels == 1, 1], s = 50, c = 'blue', label = 'Cluster 2')
plt.scatter(X[labels == 2, 0], X[labels == 2, 1], s = 50, c = 'green', label = 'Cluster 3')
plt.scatter(X[labels == 3, 0], X[labels == 3, 1], s = 50, c = 'cyan', label = 'Cluster 4')
plt.scatter(X[labels == 4, 0], X[labels == 4, 1], s = 50, c = 'purple', label = 'Cluster 5')
plt.scatter(X[labels == -1, 0], X[labels == -1, 1], s = 50, c = 'black', label = 'Outliers')
plt.title('Clusters of Stocks')
plt.legend()
plt.show()
```



Compared to the previous clustering techniques, this one performs poorly. We have six observations that were not grouped in a cluster and thus were considered outliers. All the groups lost at least one stock, while the **Automotive Sector** got split up into two separate clusters. This is of course due to the parameters that we chose, there may certainly be other combinations that could give us a more adequate result.

In [37]:
```
df8 = pd.DataFrame.sort_index(df7)
df8
```

Out [37]:

	labels	companies
0	0	Amazon
1	-1	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft
5	0	Apple
6	-1	Morningstar
7	1	S&P Global
8	1	Moody's
9	1	Nasdaq
10	1	MSCI
11	2	Toyota
12	2	Honda
13	-1	Blue Bird
14	3	Ford Motor Company
15	3	General Motors Company
16	-1	Pfizer
17	4	GlaxoSmithKlyne
18	4	Astrazeneca PLC
19	-1	Amgen
20	-1	Abbvie

In [38]:
```
df8.loc[df8['labels']==3, 'labels'] = 5
df8.loc[df8['labels']==4, 'labels'] = 3
df8.loc[df8['labels']==5, 'labels'] = 4
df8
```

Out [38]:

	labels	companies
0	0	Amazon
1	-1	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft
5	0	Apple
6	-1	Morningstar
7	1	S&P Global
8	1	Moody's
9	1	Nasdaq
10	1	MSCI
11	2	Toyota
12	2	Honda
13	-1	Blue Bird
14	4	Ford Motor Company
15	4	General Motors Company
16	-1	Pfizer
17	3	GlaxoSmithKlyne
18	3	Astrazeneca PLC
19	-1	Amgen
20	-1	Abbvie

Confusion Matrix:

In [39]:
```
pd.crosstab(ground_truth, df8['labels'], rownames=['Actual'], colnames=['Predicted'])
```

Out [39]:

Predicted	-1	0	1	2	3	4
Actual						
0	1	5	0	0	0	0
1	1	0	4	0	0	0
2	1	0	0	2	0	2
3	3	0	0	0	2	0

Adjusted Rand Index:

In [40]:
```
adjusted_rand_score(ground_truth, df8['labels'])
```

Out [40]:
```
0.4567627494456763
```

This was something expected from what we deduced from the data frame above. **Density-Based Clustering** algorithm did not provide as good results as the previous two and this was perhaps due to the nature of our data. The points are too close to each other and some even cross over to other clusters, making it difficult to discern if they belong to one cluster, or another, thus rightfully classifying them as outliers. Still the groups that were clustered correctly, had the right stocks in them. Therefore it was an alright technique, but definitely lacked the precision of the previous two.

Finally, let us see **Gaussian Mixture Model Clustering**.

D. Gaussian Mixture Model Clustering.

This approach assumes that our groups are normally distributed and uses a probabilistic approach to model the labels of the population that we are interested in. It is frequently used on multimodal data with precise results. Each cluster is described in terms of a Gaussian density, which has a centroid (as in **K-Means Clustering**), and a covariance matrix. Knowing the number (K) of components (or centroids), the **Gaussian Mixture Model Clustering** algorithm uses the ***Expectation-Maximization*** technique to estimate the model's parameters. This is a is a numerical technique for maximum likelihood estimation and it works like this:

1. In the E-step (or Expectation step), each observation is assigned a responsibility or weight for each cluster, based on the likelihood of each of the corresponding Gaussians. Observations close to the center of a cluster will most likely get weight 1 for that cluster, and weight 0 for every other cluster, while observations in-between the two clusters divide their weight accordingly.
2. In the M-step (or Maximization step), the algorithm maximizes the expectations calculated in the previous step with respect to the model parameters. Each observation contributes to the weighted means (and covariances) for every cluster.

These steps are repeated until the algorithm reaches convergence.

This algorithm can start off from a random set of initial values and reach convergence from there, but it can also start off of K-Means initial values, which are said to improve its clustering capabilities. Let us try both:

Random Starting Values:

In [41]:
```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=4, random_state=300)
labels = gmm.fit_predict(X)
```

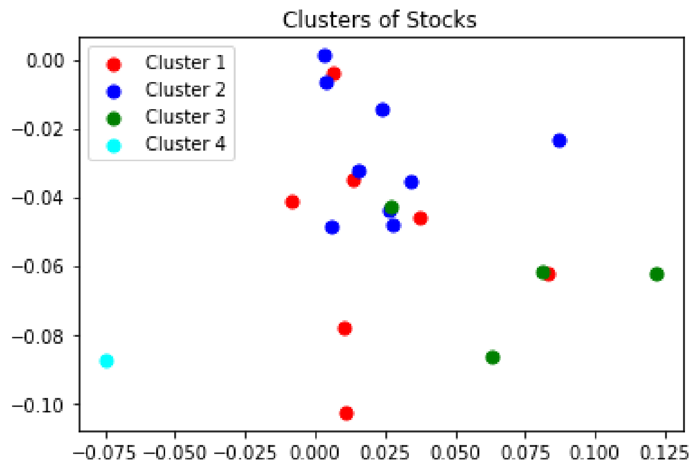
```
df9 = pd.DataFrame({'labels':labels,'companies':list(companies)}).sort_values(by=['labels'],axis = 0)
df9
```

Out [41]:

	labels	companies
18	0	Astrazeneca PLC
17	0	GlaxoSmithKlyne
16	0	Pfizer
11	0	Toyota
19	0	Amgen
9	0	Nasdaq
8	0	Moody's
0	1	Amazon
10	1	MSCI
6	1	Morningstar
5	1	Apple
4	1	Microsoft
3	1	NVIDIA
2	1	Advanced Micro Devices
1	1	Tesla
7	1	S&P Global
12	2	Honda
13	2	Blue Bird
14	2	Ford Motor Company
15	2	General Motors Company
20	3	Abbvie

In [42]:

```
plt.scatter(X[labels == 0, 0], X[labels == 0, 1], s = 50, c = 'red', label = 'Cluster 1')
plt.scatter(X[labels == 1, 0], X[labels == 1, 1], s = 50, c = 'blue', label = 'Cluster 2')
plt.scatter(X[labels == 2, 0], X[labels == 2, 1], s = 50, c = 'green', label = 'Cluster 3')
plt.scatter(X[labels == 3, 0], X[labels == 3, 1], s = 50, c = 'cyan', label = 'Cluster 4')
plt.title('Clusters of Stocks')
plt.legend()
plt.show()
```



Even without the calculation of a **Confusion Matrix**, we can see that this was a disaster. All the groups got mixed together and the clustering makes no sense whatsoever. So let us see if we can improve on it by changing the starting values. Using random starting values in this case does not work as well as we would have hoped. Maybe increasing the number of iterations could do the algorithm some good.

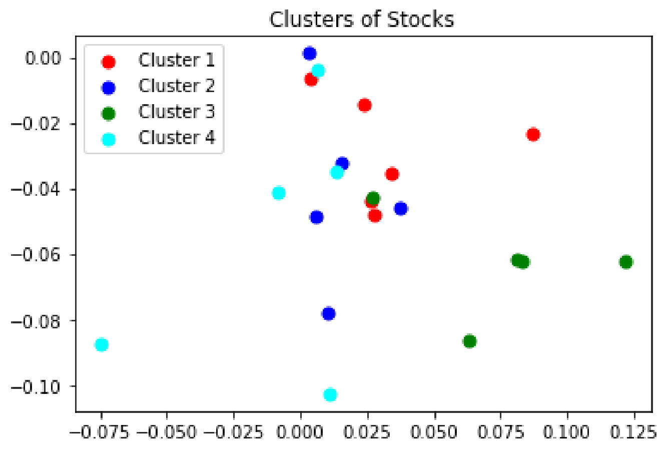
K-Means Starting Values:

In [43]:
```
from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n_components=4, n_init = 10, init_params='kmeans', random_state=8)
labels = gmm.fit_predict(X)

df10 = pd.DataFrame({'labels':labels,'companies':list(companies)}).sort_values(by=['labels'],axis =
0)
df10
```

Out [43]:

	labels	companies
0	0	Amazon
1	0	Tesla
2	0	Advanced Micro Devices
3	0	NVIDIA
4	0	Microsoft
5	0	Apple
9	1	Nasdaq
8	1	Moody's
10	1	MSCI
6	1	Morningstar
7	1	S&P Global
11	2	Toyota
12	2	Honda
13	2	Blue Bird
14	2	Ford Motor Company
15	2	General Motors Company
19	3	Amgen
16	3	Pfizer
17	3	GlaxoSmithKlyne
18	3	Astrazeneca PLC
20	3	Abbvie

In [44]:
```
plt.scatter(X[labels == 0, 0], X[labels == 0, 1], s = 50, c = 'red', label = 'Cluster 1')
plt.scatter(X[labels == 1, 0], X[labels == 1, 1], s = 50, c = 'blue', label = 'Cluster 2')
plt.scatter(X[labels == 2, 0], X[labels == 2, 1], s = 50, c = 'green', label = 'Cluster 3')
plt.scatter(X[labels == 3, 0], X[labels == 3, 1], s = 50, c = 'cyan', label = 'Cluster 4')
plt.title('Clusters of Stocks')
plt.legend()
plt.show()
```



Confusion Matrix:

In [45]:
```
pd.crosstab(ground_truth, df10['labels'], rownames=['Actual'], colnames=['Predicted'])
```

Out [45]:

Predicted	**0**	**1**	**2**	**3**
Actual				
0	6	0	0	0
1	0	5	0	0
2	0	0	5	0
3	0	0	0	5

Adjusted Rand Index:

In [46]:
```
adjusted_rand_score(ground_truth, df10['labels'])
```

Out [46]:
```
1.0
```

A perfect fit, which means that this way the algorithm works like a charm.

Conclusion to Section 1:

We saw that truly stocks can be aggregated on their daily movements into groups. That could potentially help us diversify our portfolio and mitigate some of the risk that might involve our investment.

Out of all the techniques we saw, only **Density-Based Clustering** produced sub-optimal results. All the other techniques worked perfectly. That being said, we would probably choose to use **K-Means Clustering**, or perhaps **Gaussian Mixture Models** to group our stocks. While **Hierarchical Clustering Analysis** was an okay approach, it all depended on choosing the right *linkage* and *distance metric* to get the right results. And with a good tuning, **K-Means Clustering** does the selection automatically. We would however use dendrograms in addition to other techniques to get the feeling of the right number of clusters.

This is definitely not an easy procedure, but given the chance, it can lead us to interesting results.

Section 2: Supervised Machine Learning.

Supervised Machine Learning is a procedure that uses ***labeled*** datasets to train algorithms that are used to either classify data or predict outcomes. The training data provided to our computer work as the supervisor that guide our machine with the intention of predicting accurate results. **Supervised Machine Learning** is divided into two main categories:

- ***Regression:*** Here the response variable is a continuous, numerical value. We may have a number of features and based on those what we want is to observe how our output behaves.
- ***Classification:*** In this case, the response variable is discreet and used to divide groups based on the features that are provided to us.

So let us try and give two examples for each of those categories.

1. Our Example for Regression:

Since one of the merits of **Supervised Machine Learning**, is that it can predict outcomes, then one good application in **Finance** would be the prediction of ***Stock Closing Prices***, based on the ***Stock Opening Prices*** and their ***Volume***. If we can see a good pattern between the real and the predicted values, then our algorithms will have succeeded, but it must be hard achieving that, because otherwise we would all be swimming in money.

Let us take a stock, say that of S&P500 (*GSPC*) for the duration of three business years (start of 2019 - end of 2021) and see if we can predict the ***Stock Closing Prices*** at the end of 2021.

Hence, we will import the necessary libraries and data:

Importing the Necessary Libraries:

In [47]:
```
import numpy as np
import pandas as pd
from pandas_datareader import data
import matplotlib.pyplot as plt
```

Getting and Visualizing the Data:

In [48]:
```
ticker = '^GSPC'
start_date = dt.datetime(2019, 1, 1)
end_date = dt.datetime(2021, 12, 31)

df = data.DataReader(ticker, 'yahoo', start_date, end_date)
df
```

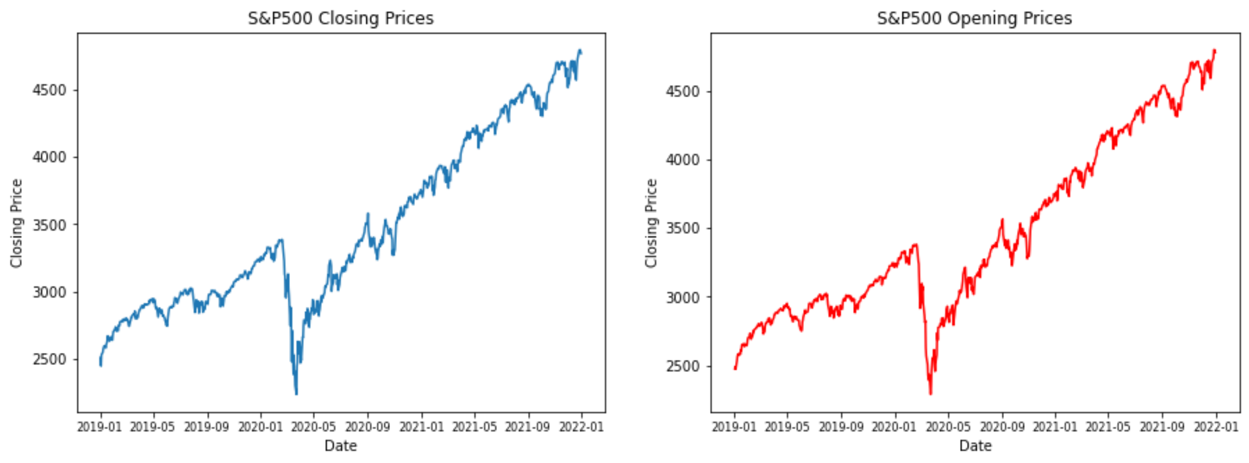
Out [48]:

Date	High	Low	Open	Close	Volume	Adj Close
2019-01-02	2519.489990	2467.469971	2476.959961	2510.030029	3733160000	2510.030029
2019-01-03	2493.139893	2443.959961	2491.919922	2447.889893	3822860000	2447.889893
2019-01-04	2538.070068	2474.330078	2474.330078	2531.939941	4213410000	2531.939941
2019-01-07	2566.159912	2524.560059	2535.610107	2549.689941	4104710000	2549.689941
2019-01-08	2579.820068	2547.560059	2568.110107	2574.409912	4083030000	2574.409912
...	...	...	...	...	...	...
2021-12-27	4791.490234	4733.990234	4733.990234	4791.189941	2264120000	4791.189941
2021-12-28	4807.020020	4780.040039	4795.490234	4786.350098	2217050000	4786.350098
2021-12-29	4804.060059	4778.080078	4788.640137	4793.060059	2369370000	4793.060059
2021-12-30	4808.930176	4775.330078	4794.229980	4778.729980	2390990000	4778.729980
2021-12-31	4786.830078	4765.750000	4775.209961	4766.180176	2446190000	4766.180176

757 rows × 6 columns

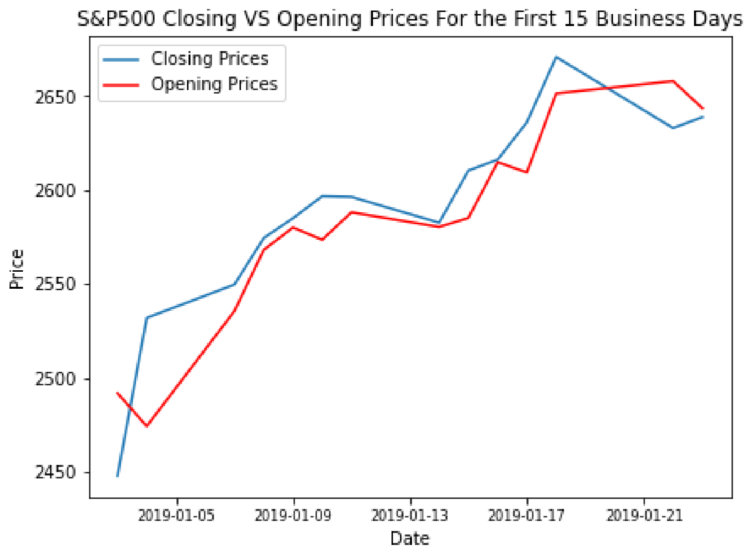
In [49]:

```
plt.figure(figsize = (15,5))
plt.rc('xtick', labelsize = 8)

ax1 = plt.subplot(121)
plt.plot(df['Close'])
plt.title('S&P500 Closing Prices')
plt.ylabel('Closing Price')
plt.xlabel('Date')

ax2 = plt.subplot(122)
plt.plot(df['Open'], color='red')
plt.title('S&P500 Opening Prices')
plt.ylabel('Closing Price')
plt.xlabel('Date')
plt.show()
```



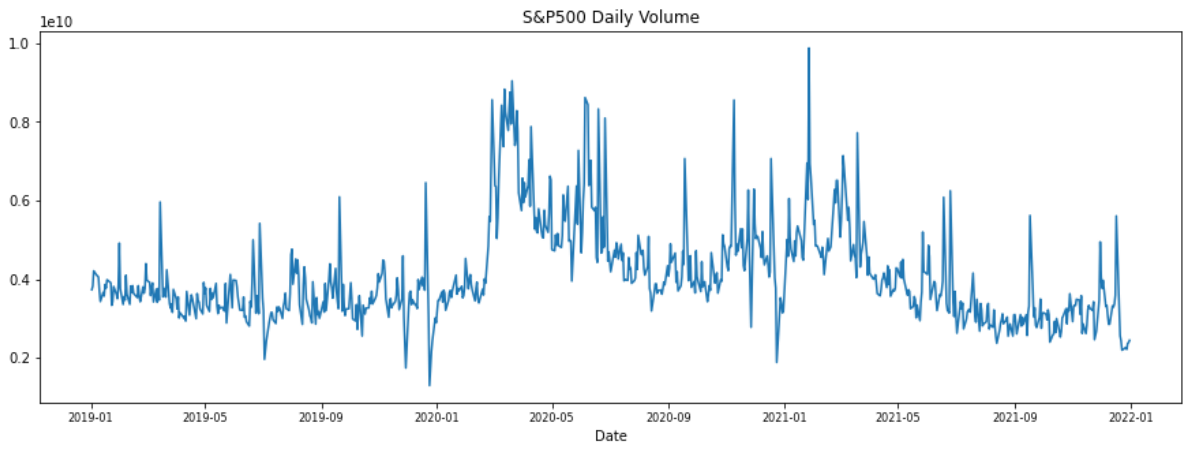
Admittedly, these two graphs look like they are identical to each other and there is some truth to it, as opening prices are pretty close to closing prices and they adjust day by day. However, if we look a bit closer, we will see that these two sets of values are not the same. Let us, for instance, take the first fifteen days and visualize the closing and the opening prices.

In [50]:
```
plt.figure(figsize = (15,5))

ax1 = plt.subplot(121)
plt.rc('xtick', labelsize = 8)
plt.plot(df['Close'][1:15])
plt.plot(df['Open'][1:15], color='red')
plt.title('S&P500 Closing VS Opening Prices For the First 15 Business Days')
plt.ylabel('Price')
plt.xlabel('Date')
plt.legend(['Closing Prices', 'Opening Prices'])
plt.show()
```



Therefore, even those these two prices move in the same direction most of the time, they are not identical.

In [51]:
```
plt.figure(figsize = (15,5))

plt.rc('xtick', labelsize = 8)
plt.plot(df['Volume'])
plt.title('S&P500 Daily Volume')
plt.xlabel('Date')
plt.show()
```



Scaling our Features and Response variable:

We shall standardize our data so as to help our algorithms function properly. ***Multiple Linear Regression*** would be fine for instance, but other algorithms (such as ***Support Vector Regression***) need this step, because otherwise they might neglect some feature.

In [52]:
```
X = pd.DataFrame(df[['Open', 'Volume']])
y = df[['Close']]

from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
sc_y = StandardScaler()
X = pd.DataFrame(sc_X.fit_transform(X))
y = sc_y.fit_transform(y)
```

Splitting our Data into Train and Test parts:

Having a sense of how the values of our variables move, it is now time to split our data into train and test parts so as to measure and see the performance of each algorithm, when prediction is performed on the **Closing Prices**. In total we have 757 observations, thus we shall use the first 657 in order to predict the last 100.

In [53]:
```
# The Feature matrix train-test split:
X_train = X[:657] # Starts from 0.
X_test = X[657:]

# The response vector train-test split:
y_train = y[:657]
y_test = y[657:]
```

A. (Multiple) Linear Regression.

This is a very old and, in all probability, the first statistical model that tries to represent a linear relationship between the features and the target vector. When we have more than one explanatory variables (like we do in this example, we are talking about a **Multiple Linear Regression**. This model for our problem can be written in the following form:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \epsilon_i$$

where x_{i1} represent the ***Stock Opening Prices***, while x_{i2} is the ***Volume***.

Now, let us apply this model on our data and see how well it will predict our closing stock prices.

In [54]:
```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr.fit(X_train, y_train)
lr_pred = lr.predict(X_test)
```

In [55]:
```
df_lr = pd.DataFrame(y_test)
df_lr['Predictions'] = lr_pred
df_lr.columns = ['Real Values', 'Predictions']
df_lr
```

Out [55]:

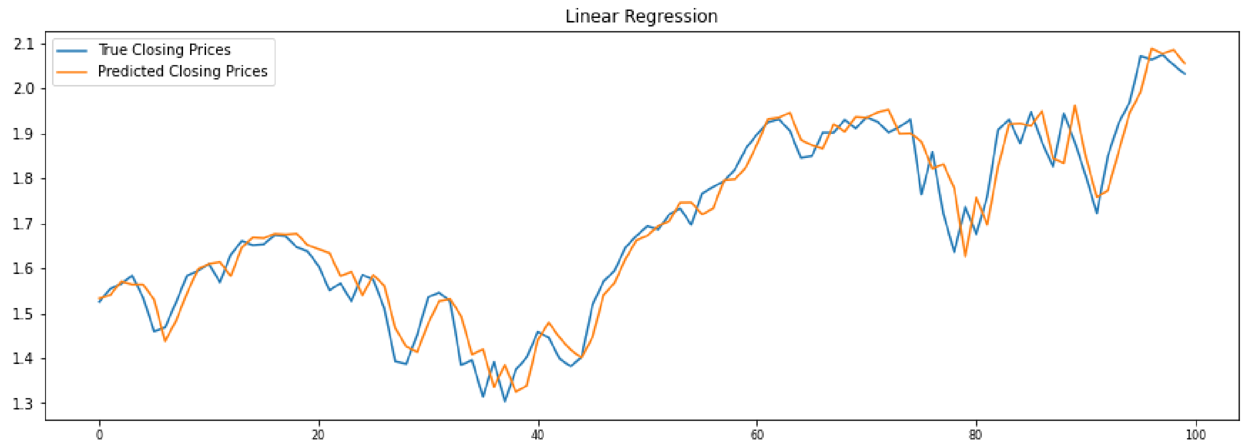
	Real Values	Predictions
0	1.525524	1.533010
1	1.554358	1.540466
2	1.565582	1.570704
3	1.583912	1.563957
4	1.534400	1.563766
...	...	...
95	2.071496	1.992140
96	2.063919	2.088554
97	2.074423	2.077044
98	2.051991	2.085672
99	2.032346	2.055641

<u>Root Mean Square Error (RMSE):</u>

In [56]:
```
from sklearn.metrics import mean_squared_error
rmse_lr = np.sqrt(mean_squared_error(df_lr['Real Values'], df_lr['Predictions']))
rmse_lr
```

Out [56]:
```
0.049638482026864404
```

In [57]:
```
plt.figure(figsize=(15,5))
plt.plot(df_lr)
plt.title('Linear Regression')
plt.legend(['True Closing Prices', 'Predicted Closing Prices'])
plt.show()
```

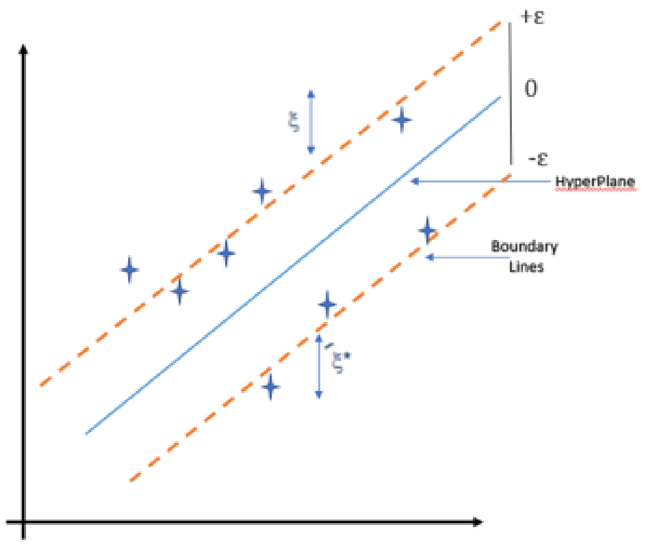


All in all, this was not a bad fit. There are of course some deviations and it is not exactly as the real **Stock Closing Prices**, but this prediction works well. The orange line is behaving very similarly to the blue one. Plus, ***RMSE*** is quite small, but as always, we are looking for the smallest of them all.

Now let's see how the other algorithms fare. Next up is ***Support Vector Regression (SVR)***.

B. Support Vector Regression (SVR).

This is a generalization of the **Support Vector Machine** algorithm, but it works in a similar way. The reason it was introduced is, of course, the need to work on continuous response variables. As we recall, the optimization problem in **Support Vector Machine** is a problem of finding the maximum margin separating the hyperplane, while correctly classifying as many training points as possible. This is achieved similarly in **Support Vector Regression**, with the introduction of an ϵ-*insensitive* region around the function, called the ϵ-*tube*. Let us get a better understanding by looking at the graph below:



As we can see, the hyperplane separates the two classes as best as it can. There are some data points that fall inside the ϵ-*tube* for which we will be disregarding the error when we do our calculations. The only error that we are going to take into account is of those points (ξ_i and ξ_i^*) that are outside the tube. Essentially, what we want is to minimize the sum of the error points outside of the ϵ -*tube*. So some-what like a Linear Regression, we perform an optimization by first defining a convex ϵ -*insensitive* loss function to be minimized and finding the flattest tube that contains most of the training instances. Hence, what we end up with, is a function which consists of the loss function and the geometrical properties of the tube. Mathematically, the optimization problem of **Support Vector Regression** is formed like this:

$$\min\left\{\frac{1}{2}||w||^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)\right\}, \qquad for\ i = 1, \dots, N.$$

subject to:

$$y_i - w^T x_i \leq \epsilon + \xi_i^*, \qquad fori = 1, \dots, N.$$

$$w^T x_i - y_i \leq \epsilon + \xi_i, \qquad for\ i = 1, \dots, N.$$

$$\xi_i^*, \xi_i \geq 0, \qquad for\ i = 1, \dots, N.$$

Here $||w||$ is the magnitude of the normal vector to the surface that is being approximated, while C is a regularization — thus, a tunable parameter that gives more weight to minimizing the flatness, or the error, for this minimization problem.

Now let us run it and see what results we shall get:

In [58]:
```
from sklearn.svm import SVR
svr = SVR(kernel = 'linear', C = 40, epsilon = 0.0001) # High l2 penalty, high precision.
svr.fit(X_train, y_train.ravel())
svr_pred = svr.predict(X_test)
```

In [59]:
```
df_svr = pd.DataFrame(y_test)
df_svr['Predictions'] = svr_pred
df_svr.columns = ['Real Values', 'Predictions']
df_svr
```

Out [59]:

	Real Values	Predictions
0	1.525524	1.532851
1	1.554358	1.539958
2	1.565582	1.569969
3	1.583912	1.563676
4	1.534400	1.563724
...	...	...
95	2.071496	1.991366
96	2.063919	2.087741

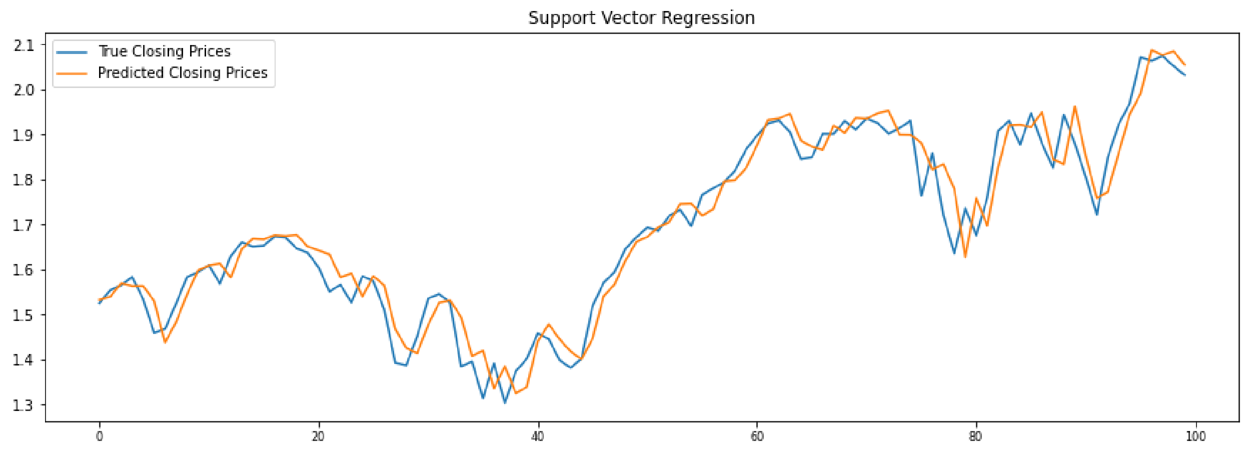
97	2.074423	2.076435
98	2.051991	2.085094
99	2.032346	2.055131

Root Mean Square Error (RMSE):

In [60]:
```
from sklearn.metrics import mean_squared_error
rmse_svr = np.sqrt(mean_squared_error(df_svr['Real Values'], df_svr['Predictions']))
rmse_svr
```

Out [60]:
```
0.04985604916709515
```

In [61]:
```
plt.figure(figsize=(15,5))
plt.plot(df_svr)
plt.title('Support Vector Regression')
plt.legend(['True Closing Prices', 'Predicted Closing Prices'])
plt.show()
```



That is an okay fit, much like the one we had in **Linear Regression**, albeit a bit worse, when comparing the two ***RMSE*** values. Looking at the graph it does come pretty close to the real values.

C. Random Forest Regression.

Random forest is an ensemble algorithm and what it means is that it uses several algorithms together with the aim of improving predictions in general. What it specifically does, is it combines several decision trees, where each tree provides a single vote toward the final prediction.

The final random forest calculates a final output as an average of the results of all the trees it is composed of.

It works in the following manner:

1. It selects a random number of K data points from the training set.
2. It builds the decision tree associated to these K data points. Each of the trees makes its own individual prediction.
3. It repeats steps 1 and 2 for the number of trees we have chosen to build.
4. The predictions made by the decision trees are then averaged to produce a single result.

So this is simple enough, let us see how it works:

In [62]:
```
from sklearn.ensemble import RandomForestRegressor
rfr = RandomForestRegressor(n_estimators = 500, random_state = 42)
rfr.fit(X_train, y_train.ravel())
rfr_pred = rfr.predict(X_test)
```

In [63]:
```
df_rfr = pd.DataFrame(y_test)
df_rfr['Predictions'] = rfr_pred
df_rfr.columns = ['Real Values', 'Predictions']
df_rfr
```

Out [63]:

	Real Values	Predictions
0	1.525524	1.508850
1	1.554358	1.506958
2	1.565582	1.506958
3	1.583912	1.508380
4	1.534400	1.492408
...	...	...
95	2.071496	1.506958
96	2.063919	1.506958
97	2.074423	1.506958
98	2.051991	1.506958
99	2.032346	1.506958

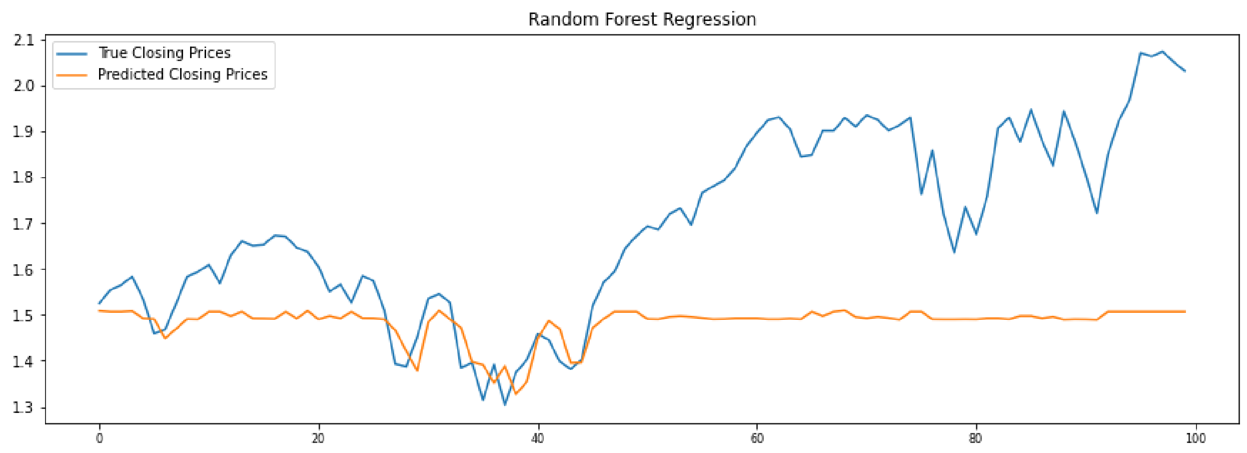
100 rows × 2 columns

<u>Root Mean Square Error (RMSE):</u>

In [64]:
```
from sklearn.metrics import mean_squared_error
rmse_rfr = np.sqrt(mean_squared_error(df_rfr['Real Values'], df_rfr['Predictions']))
rmse_rfr
```

Out [64]:
```
0.27581848029895234
```

In [65]:
```
plt.figure(figsize=(15,5))
plt.plot(df_rfr)
plt.title('Random Forest Regression')
plt.legend(['True Closing Prices', 'Predicted Closing Prices'])
plt.show()
```



Admittedly this was a disaster. So far this was the worst prediction yet. See the **Random Forest** algorithm has an issue, it cannot extrapolate. This is to say that when the Random Forest Regressor is tasked with the problem of predicting for values not previously seen, it will always predict an average of the values seen previously. And unfortunately, the average of a sample can not go beyond the highest and lowest values in the sample.

D. Stacking Regressor.

After taking into account all those results, there is a better thing that we can do. What if we could aggregate all those previously mentioned regressors into a single model and have this much more complex model do our final predictions? If we did that, we would get the so called stacked generalization. And that is exactly what we shall do for our regression problem, we will put the strengths of each of the algorithms previously used and make this model do a prediction for us. Since we will not be specifying a particular algorithm as an output for this ensemble, the output will be calculated using Cross-Validated Ridge Regression.

In [66]:
```
from sklearn.ensemble import StackingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
```

```
level0 = list()
level0.append(('svr', SVR()))
level0.append(('rfr', RandomForestRegressor()))
level0.append(('lr', LinearRegression()))

sr = StackingRegressor(estimators=level0, cv=10)
sr.fit(X_train, y_train.ravel())
sr_pred = sr.predict(X_test)
```

In [67]:
```
df_sr = pd.DataFrame(y_test)
df_sr['Predictions'] = sr_pred
df_sr.columns = ['Real Values', 'Predictions']
df_sr
```

Out [67]:

	Real Values	Predictions
0	1.525524	1.540330
1	1.554358	1.545649
2	1.565582	1.572512
3	1.583912	1.569415
4	1.534400	1.569499
...	...	...
95	2.071496	1.968559
96	2.063919	2.057918
97	2.074423	2.049303
98	2.051991	2.057616
99	2.032346	2.030283

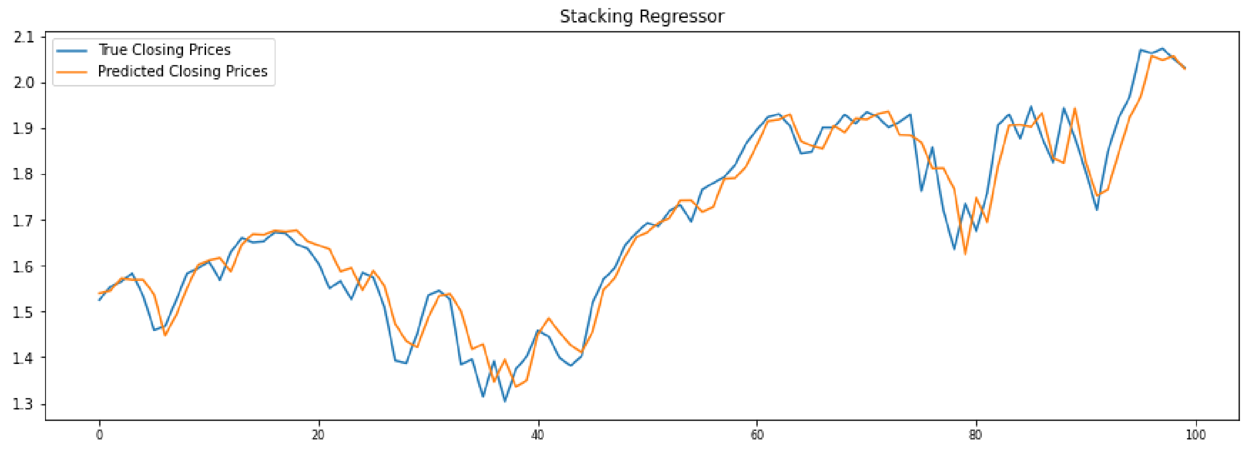
100 rows × 2 columns

Root Mean Square Error (RMSE):

In [68]:
```
from sklearn.metrics import mean_squared_error
rmse_sr = np.sqrt(mean_squared_error(df_sr['Real Values'], df_sr['Predictions']))
rmse_sr
```

Out [68]:
```
0.049434741386953654
```

In [69]:
```
plt.figure(figsize=(15,5))
plt.plot(df_sr)
plt.title('Stacking Regressor')
plt.legend(['True Closing Prices', 'Predicted Closing Prices'])
plt.show()
```



This is the best performing algorithm so far and, should we do away with the **Random Forest Regressor**, which seems to mix up our algorithm, then it becomes slightly better at predicting the **Closing Stock Prices**.

E. Artificial Neural Networks (Deep Learning).

Artificial Neural Network (*ANN*) is an efficient computing system whose central theme is borrowed from the analogy of biological neural networks. It incorporates a set of interconnected nodes that operate in parallel at each layer of the program. Its complexity depends on the architecture that we shall provide it each time. At this time and for this rather simple problem, we shall require a simple *Sequential* model, which is very straightforward (a simple list of layers), but is limited to single-input, single-output stacks of layers (as the name gives away).

At this point we shall choose to use 2 layers in total (one input, one hidden) with 5 nodes each and an output layer of one value as the *Sequential* model requires. There is no need to go extravagant on this model as it is quite simple. It has only two features and around 750 observations.

So let us apply this model on our data and see what we come up with.

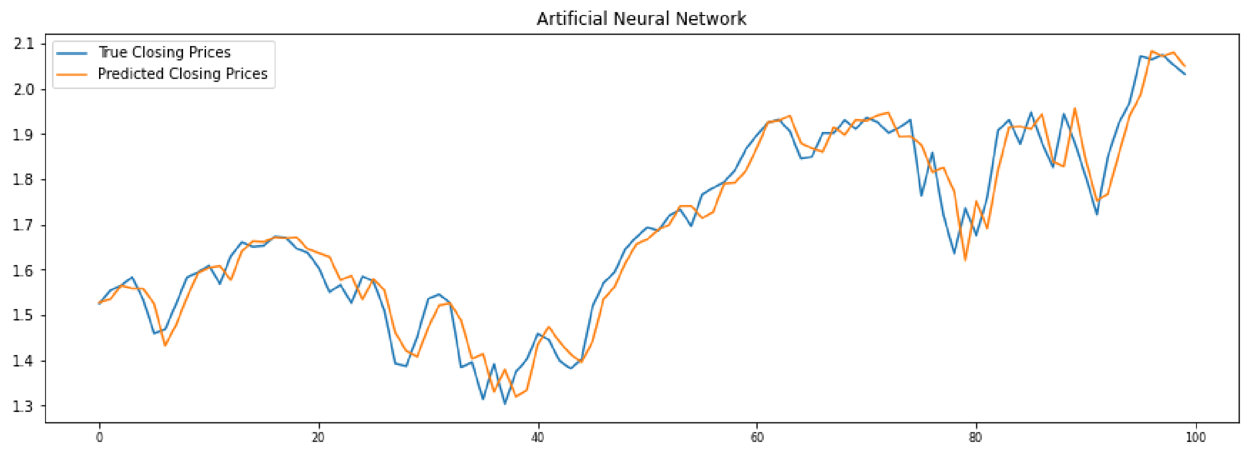
In [70]:
```
import tensorflow as tf

ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=5, activation='linear'))
ann.add(tf.keras.layers.Dense(units=5, activation='linear'))
ann.add(tf.keras.layers.Dense(units=1, activation='linear'))
ann.compile(optimizer = 'Adam', loss = 'MeanSquaredError', metrics = ['RootMeanSquaredError'])
ann.fit(X_train, y_train, epochs = 100)
ann_pred = ann.predict(X_test)
```

In [71]:
```
df_ann = pd.DataFrame(y_test)
df_ann['Predictions'] = ann_pred
df_ann.columns = ['Real Values', 'Predictions']
df_ann
```
Out [71]:

	Real Values	Predictions
0	1.525524	1.527736
1	1.554358	1.535230
2	1.565582	1.565467
3	1.583912	1.558669
4	1.534400	1.558447
...	...	...
95	2.071496	1.986500
96	2.063919	2.082826
97	2.074423	2.071301
98	2.051991	2.079916
99	2.032346	2.049906

100 rows × 2 columns

__Root Mean Square Error (RMSE):__

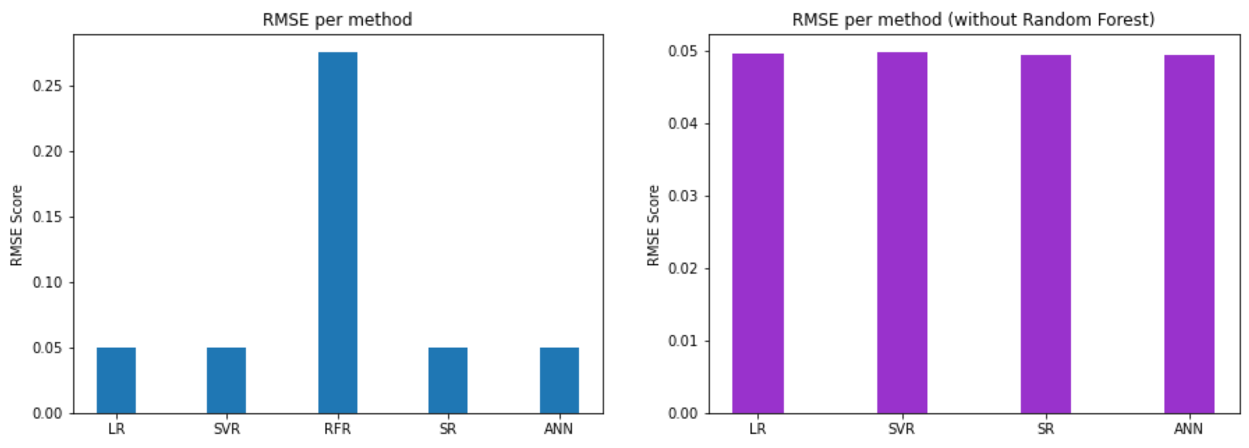
In [72]:
```
from sklearn.metrics import mean_squared_error
rmse_ann = np.sqrt(mean_squared_error(df_ann['Real Values'], df_ann['Predictions']))
rmse_ann
```
Out [72]:
```
0.04940682625245221
```
In [73]:
```
plt.figure(figsize=(15,5))
plt.plot(df_ann)
plt.title('Artificial Neural Network')
plt.legend(['True Closing Prices', 'Predicted Closing Prices'])
plt.show()
```


It did pretty well. let us not forget that an optimization is involved, so the results will vary. At times this program outperforms all the rest that we have already discussed and at others it is the second worst, right after the Random Forest Regressor. It provided us with adequate results for this setup, but there are others that could be even better inside the vast Keras API.

Conclusion to Section 2, Exercise 1:

Let us make the following histogram to help us sum thing up:

In [74]:
```
mylist = [rmse_lr, rmse_svr, rmse_rfr, rmse_sr, rmse_ann]
names = ['LR', 'SVR', 'RFR', 'SR', 'ANN']

mylist1 = [rmse_lr, rmse_svr, rmse_sr, rmse_ann]
names1 = ['LR', 'SVR', 'SR', 'ANN']


plt.figure(figsize = (15,5))
plt.rc('xtick', labelsize = 10)
width = 0.35

ax1 = plt.subplot(121)
ax1.bar(names, mylist, width)
ax1.set_ylabel('RMSE Score')
ax1.set_title('RMSE per method')

ax2 = plt.subplot(122)
ax2.bar(names1, mylist1, width, color='darkorchid')
ax2.set_ylabel('RMSE Score')
ax2.set_title('RMSE per method (without Random Forest)')

plt.show()
```



In conclusion, all methods apart from the **Random Forest Regressor** succeeded in coming very close to predicting the real **Closing Stock Prices**. The best ones were the **Stacked Regressor** and the **Artificial Neural Network**, with the other two methods just behind. If we had to choose one to do this procedure all over again, then the best one would be the **Stacked Regressor**, because due to optimization in **Artificial Neural Network**, that procedure is a bit more volatile and sometimes gives results that are a bit more off than the one we would choose. And after all, the improvement is not that significant most of the times. After all, as we can see in the bar graphs above, the differences are imperceptible and these four options in the graph to the right are quite similar to one another. Some are better than other but only by a fraction.

---

2. Our Example for Classification:

The prices in our previous example were trying to be specific, but maybe a more tangible question that could get probable results would be: Can we predict whether the price of a stock shall move up or down, based on the information that we have on stock returns from previous days and the volume of the stocks?

Let us try and tackle this question in this exercise. For that very reason, we shall take the S&P500 (*GSPC*) for the duration of five business years (start of 2015 - end of 2019) and see if we can predict the direction at which its **Adjusted Closing Price** will move based on the parameters that we have already discussed.

Importing the Necessary Libraries:

In [75]:
```
import numpy as np
import pandas as pd
from pandas_datareader import data
import datetime as dt
```

Getting the Data:

In [76]:
```
ticker = '^GSPC'
start_date = dt.datetime(2015, 1, 1)
end_date = dt.datetime(2019, 12, 31)

mydata = data.DataReader(ticker, 'yahoo', start_date, end_date)
mydata
```

Out [76]:

	High	Low	Open	Close	Volume	Adj Close
Date						
2015-01-02	2072.360107	2046.040039	2058.899902	2058.199951	2708700000	2058.199951
2015-01-05	2054.439941	2017.339966	2054.439941	2020.579956	3799120000	2020.579956
2015-01-06	2030.250000	1992.439941	2022.150024	2002.609985	4460110000	2002.609985

2015-01-07	2029.609985	2005.550049	2005.550049	2025.900024	3805480000	2025.900024
2015-01-08	2064.080078	2030.609985	2030.609985	2062.139893	3934010000	2062.139893
...	...	...	...	...	...	...
2019-12-24	3226.429932	3220.510010	3225.449951	3223.379883	1296540000	3223.379883
2019-12-26	3240.080078	3227.199951	3227.199951	3239.909912	2160680000	3239.909912
2019-12-27	3247.929932	3234.370117	3247.229980	3240.020020	2428670000	3240.020020
2019-12-30	3240.919922	3216.570068	3240.090088	3221.290039	3013290000	3221.290039
2019-12-31	3231.719971	3212.030029	3215.179932	3230.780029	2893810000	3230.780029

1258 rows × 6 columns

We shall focus on the Daily returns of the S&P500, as far as the **Adjusted Closing Price** is concerned, therefore:

In [77]:
```
temp = mydata['Adj Close'].pct_change()*100 # Get the daily percent returns.
temp = temp.rename('Today')
temp = temp.reset_index()
temp
```

Out [77]:

	Date	Today
0	2015-01-02	NaN
1	2015-01-05	-1.827811
2	2015-01-06	-0.889347
3	2015-01-07	1.162984
4	2015-01-08	1.788828
...	...	...
1253	2019-12-24	-0.019545
1254	2019-12-26	0.512817
1255	2019-12-27	0.003398
1256	2019-12-30	-0.578082
1257	2019-12-31	0.294602

1258 rows × 2 columns

Create the Lagged Columns:

In [78]:
```
for i in range(1,6):
  temp['Lag '+str(i)] = temp['Today'].shift(i)
temp
```

Out [78]:

	Date	Today	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5
0	2015-01-02	NaN	NaN	NaN	NaN	NaN	NaN
1	2015-01-05	-1.827811	NaN	NaN	NaN	NaN	NaN
2	2015-01-06	-0.889347	-1.827811	NaN	NaN	NaN	NaN
3	2015-01-07	1.162984	-0.889347	-1.827811	NaN	NaN	NaN
4	2015-01-08	1.788828	1.162984	-0.889347	-1.827811	NaN	NaN
...	...	...	...	...	...	...	...

1253	2019-12-24	-0.019545	0.086614	0.494478	0.445929	-0.043230	0.033529
1254	2019-12-26	0.512817	-0.019545	0.086614	0.494478	0.445929	-0.043230
1255	2019-12-27	0.003398	0.512817	-0.019545	0.086614	0.494478	0.445929
1256	2019-12-30	-0.578082	0.003398	0.512817	-0.019545	0.086614	0.494478
1257	2019-12-31	0.294602	-0.578082	0.003398	0.512817	-0.019545	0.086614

1258 rows × 7 columns

Get the shifted (by one day) Volume values:

In [79]:

```
#---------------------------------------------------------------
# Dividing by 1.000.000.000 is making us count Volume in billions
# and plus, now we do not have to scale our data.
#---------------------------------------------------------------
temp['Volume'] = mydata.Volume.shift(1).values/1000000000

# Dropping the NaN values.
temp = temp.dropna()
temp
```

Out [79]:

	Date	Today	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Volume
6	2015-01-12	-0.809369	-0.840381	1.788828	1.162984	-0.889347	-1.827811	3.36414
7	2015-01-13	-0.257856	-0.809369	-0.840381	1.788828	1.162984	-0.889347	3.45646
8	2015-01-14	-0.581307	-0.257856	-0.809369	-0.840381	1.788828	1.162984	4.10730
9	2015-01-15	-0.924788	-0.581307	-0.257856	-0.809369	-0.840381	1.788828	4.37868
10	2015-01-16	1.342420	-0.924788	-0.581307	-0.257856	-0.809369	-0.840381	4.27672
...	...	...	...	...	...	...	...	...
1253	2019-12-24	-0.019545	0.086614	0.494478	0.445929	-0.043230	0.033529	3.06061
1254	2019-12-26	0.512817	-0.019545	0.086614	0.494478	0.445929	-0.043230	1.29654
1255	2019-12-27	0.003398	0.512817	-0.019545	0.086614	0.494478	0.445929	2.16068
1256	2019-12-30	-0.578082	0.003398	0.512817	-0.019545	0.086614	0.494478	2.42867
1257	2019-12-31	0.294602	-0.578082	0.003398	0.512817	-0.019545	0.086614	3.01329

1252 rows × 8 columns

Create the Direction column:

Here if the sign of the column *"Today"* is negative, then the direction is down (or zero) and if it is positive, then the direction is up (or one).

In [80]:

```
df = temp.copy()
df['Direction'] = [1 if i>0 else 0 for i in df['Today']]
df
```

Out [80]:

	Date	Today	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Volume	Direction
6	2015-01-12	-0.809369	-0.840381	1.788828	1.162984	-0.889347	-1.827811	3.36414	0
7	2015-01-13	-0.257856	-0.809369	-0.840381	1.788828	1.162984	-0.889347	3.45646	0
8	2015-01-14	-0.581307	-0.257856	-0.809369	-0.840381	1.788828	1.162984	4.10730	0
9	2015-01-15	-0.924788	-0.581307	-0.257856	-0.809369	-0.840381	1.788828	4.37868	0
10	2015-01-16	1.342420	-0.924788	-0.581307	-0.257856	-0.809369	-0.840381	4.27672	1
...	...	...	...	...	...	...	...	...	...
1253	2019-12-24	-0.019545	0.086614	0.494478	0.445929	-0.043230	0.033529	3.06061	0
1254	2019-12-26	0.512817	-0.019545	0.086614	0.494478	0.445929	-0.043230	1.29654	1
1255	2019-12-27	0.003398	0.512817	-0.019545	0.086614	0.494478	0.445929	2.16068	1
1256	2019-12-30	-0.578082	0.003398	0.512817	-0.019545	0.086614	0.494478	2.42867	0
1257	2019-12-31	0.294602	-0.578082	0.003398	0.512817	-0.019545	0.086614	3.01329	1

1252 rows × 9 columns

Splitting our Data into Train and Test parts:

In our dataset, we have a total of 1252 observations, and thus we shall use the first 1100 in order to predict the last 152.

In [81]:
```
X = df[df.columns[2:8]] # The first two columns are not used.
y = df[df.columns[8]]

# The Feature matrix train-test split:
X_train = X[:1100] # Starts from 0.
X_test = X[1100:]

# The response vector train-test split:
y_train = y[:1100]
y_test = y[1100:]
```

A. Logistic Regression.

Logistic Regression (also called *Logit Regression*) is a probabilistic statistical classification model that predicts the probability of the occurrence of an event. If the estimated probability is greater than 50%, then the model predicts that the observation belongs to that particular class (to the class labeled "1"), and otherwise it predicts that it does not (or to the class labeled "0"). This makes it a binary classifier. The (multiple) logistic function, in our example, can be written as:

$$P(Y|X) = \frac{exp(\beta_0 + \beta_1 X_i + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \beta_6 X_6)}{1 + exp(\beta_0 + \beta_1 X_i + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 + \beta_5 X_5 + \beta_6 X_6)}$$

So let us use it and try to predict whether the price of a stock shall move up or down:

In [82]:
```
from sklearn.linear_model import LogisticRegression
LR = LogisticRegression(random_state = 5)
LR.fit(X_train, y_train)
logr_pred = LR.predict(X_test)
```

In [83]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, logr_pred)
print(cm)
logr_score = accuracy_score(y_test, logr_pred)
logr_score
```

```
[[10 52]
 [ 5  85]]
```

Out [83]:
```
0.625
```

Not bad. Prediction, especially in the stock market, is a very challenging task, so to be able to get even that kind of accuracy is good enough. In total, 57 observations are misclassified, but 95 get right where they belong.

B. K-Nearest Neighbours.

Yet another famous methodology, is the **K-Nearest Neighbors** (*K-NN*) classification methodology. What it does, is it identifies a group of *K* objects in the training set that are closest to the test object and assigns a label based on the most dominant class in this neighborhood. So for example if we have chosen a neighbourhood of five objects and three of them were labeled as 1, then the new observation would also be labelled as 1.

The basic idea is that if we want to predict the value of a new observation we look for observations similar to this one and we use as predictors their values (or a function of them). Specifically, for classification problems as ours, we use the mode of the classes of the nearest data.

So without further ado, let us see how well **K-Nearest Neighbors** algorithm predicts our classes.

In [84]:
```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 3)
knn.fit(X_train, y_train)
knn_pred = knn.predict(X_test)
```

In [85]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, knn_pred)
print(cm)
```

```
knn_score = accuracy_score(y_test, knn_pred)
knn_score
```

```
[[37 25]
 [41 49]]
```

Out [85]:
```
0.5657894736842105
```

Again, not bad. Even though it has a worse accuracy than **Logistic Regression** before it, it is still trying its best to classify a difficult problem. Here the result was dependent on two things: the distance metric we used and the number of neighbours we used, thus there might be better solutions, but this one was the best we could come up with.

C. Naive Bayes.

Naive Bayes is a simple probabilistic classifier that applies Bayes' theorem with strong (naive) assumption of independence, such that the presence of an individual feature of a class is unrelated to the presence of another feature.

Assume that input features x_1, x_2, ..., x_n, are conditionally independent of each other, given the class label Y, such that:

$$P(x_1, x_2, \dots, x_n|Y) = \prod_{i=1}^{n} P(x_i|Y)$$

Therefore, if we only had two classes, like in our problem, then we would define $P(i|x)$, for $i = 0, 1$ as the probability that measurement vector $x = \{x_1, x_2, \dots, x_n\}$ belongs to class i. While the classification score would be defined as:

$$\frac{P(1|x)}{P(0|x)} = \frac{\prod_{i=1}^{n} f(x_i|1)P(1)}{\prod_{i=1}^{n} f(x_i|0)P(0)} = \frac{P(1)}{P(0)} \prod_{i=1}^{n} \frac{f(x_i|1)}{f(x_i|0)}$$

The naive Bayes model is surprisingly effective and immensely appealing, owing to its simplicity and robustness. Because this algorithm does not require application of complex iterative parameter estimation schemes to large datasets, it is very useful and relatively easy to construct and use. It's somewhat similar to **K-Nearest Neighbors** in the sense that it makes some assumptions that might oversimplify reality, but still, it performs well in many cases.

Thus, let us apply this algorithm and see what we get.

In [86]:
```
from sklearn.naive_bayes import GaussianNB
NB = GaussianNB()
NB.fit(X_train, y_train)
nb_pred = NB.predict(X_test)
```

In [87]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, nb_pred)
print(cm)
nb_score = accuracy_score(y_test, nb_pred)
nb_score
```

```
[[40 22]
 [51 39]]
```

Out [87]:
```
0.5197368421052632
```

This result is a bit worse than the previous two, as 73 observations get misclassified. Having the other two options, maybe we would not choose this one as a classifier for this problem. After all, **Naive Bayes** works best for large datasets, but the "naive" assumption that the variables are independent does not work well here, because we have lagged variables.

D. Support Vector Machine.

Support Vector Machine is a powerful, highly flexible modeling technique. It classifies the data by finding the hyperplane (meaning an $n - 1$ subspace in an n-dimensional space) that maximizes the margin between the classes in the training data. In two dimensions, for example, the hyperplane is a line that separates the data. Thus, a good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class.

Let us see then, how well it separates our data.

In [88]:
```
from sklearn.svm import SVC
svm = SVC(kernel = 'linear', random_state = 0)
svm.fit(X_train, y_train)
svm_pred = svm.predict(X_test)
```

In [89]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, svm_pred)
print(cm)
svm_score = accuracy_score(y_test, svm_pred)
svm_score
```

```
[[ 0 62]
 [ 0 90]]
```

Out [89]:
```
0.5921052631578947
```

Not bad at all. This is the second-best result so far accuracy-wise, right after **Logistic Regression**. It should be mentioned that other *kernels* performed even worse than the *linear* so it really was the best that we could do.

E. Random Forest Classification.

We have already mentioned this technique in the previous exercise, but let us repeat how this procedure works once again. **Random Forest** is an ensemble learning approach for classification, in which *"weak learners"* collaborate to form *"strong learners*", using a large collection of decorrelated decision trees (the random forest). Instead of developing a solution based on the output of a single deep tree, however, random forest aggregates the output from a number of shallow trees, forming an additional layer to bagging. Bagging constructs n predictors, using independent successive trees, by bootstrapping samples of the dataset. The n predictors are combined to solve a classification or estimation problem through averaging. Although individual classifiers are weak learners, all the classifiers combined form a strong learner.

Let us see how it handles our data.

In [90]:
```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators = 20, criterion = 'entropy', random_state = 42)
rf.fit(X_train, y_train)
rf_pred = rf.predict(X_test)
```

In [91]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, rf_pred)
print(cm)
rf_score = accuracy_score(y_test, rf_pred)
rf_score
```

```
[[29 33]
 [39 51]]
```

Out [91]:
```
0.5263157894736842
```

Not bad, not good. It has an okay predictive power, based on the accuracy of the confusion matrix. It is dependent on the number of decision trees we choose to apply in total and the criterion we choose to measure the quality of a split. So changing all those parameters could make our algorithm better or worse.

F. Artificial Neural Networks.

Here we will repeat what we already said about this procedure in ***Exercise 1***.

Artificial Neural Network (*ANN*) is an efficient computing system whose central theme is borrowed from the analogy of biological neural networks. It incorporates a set of interconnected nodes that operate in parallel at each layer of the program. Its complexity depends on the architecture that we shall provide it each time. At this time and for this rather simple problem, we shall require a simple *Sequential* model, which is very straightforward (a simple list of layers), but is limited to single-input, single-output stacks of layers (as the name gives away).

At this point we shall choose to use 2 layers in total (one input, one hidden) with 5 nodes each and an output layer of one value as the *Sequential* model requires. Since this is a simple classification, the output layer will be a *sigmoid* function.

So let us apply this model on our data and see what we come up with.

In [92]:
```
import tensorflow as tf

ann = tf.keras.models.Sequential()
ann.add(tf.keras.layers.Dense(units=5, activation='relu'))
ann.add(tf.keras.layers.Dense(units=5, activation='relu'))
ann.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
ann.compile(optimizer = 'Adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
ann.fit(X_train, y_train, batch_size = 32, epochs = 200)
ann_pred = ann.predict(X_test)
ann_pred = np.round(ann_pred)
```

In [93]:
```
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, ann_pred)
print(cm)
ann_score = accuracy_score(y_test, ann_pred)
ann_score
```

```
[[ 0 62]
 [ 0 90]]
```

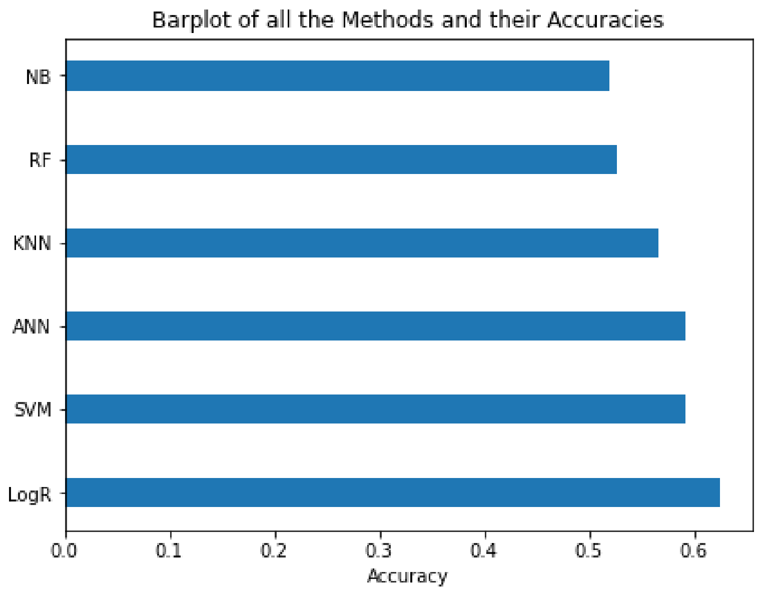
Out [93]:
```
0.5921052631578947
```

Not bad, but also not the best. This procedure does have an optimization to it and so will give us different answers at different runs. Still no matter how many runs we had, it never surpassed the accuracy of our best model, the **Logistic Regression**.

Conclusion to Section 2, Exercise 2:

Let us do another bar chart in order to make this synopsis easier.

In [94]:
```
mylist = [logr_score, svm_score, ann_score, knn_score, rf_score, nb_score]
names = ['LogR', 'SVM', 'ANN', 'KNN', 'RF', 'NB']


plt.figure(figsize = (15,5))
plt.rc('xtick', labelsize = 10)
width = 0.35

ax1 = plt.subplot(121)
ax1.barh(names, mylist, width)
ax1.set_xlabel('Accuracy')
ax1.set_title('Barplot of all the Methods and their Accuracies')
plt.show()
```



We saw that even predicting the direction at which a stock shall move the next day can be a difficult task to tackle. The worst technique proved to be **Naive Bayes**, however it was for understandable reasons, as we discussed in the final comments of that method. Out of all the methods used, the **Logistic Regression** gave the most promising results, even though they were not perfect. Still, it would be the technique we would use in such a problem, because it outperformed all others.

BIBLIOGRAPHY

Albon Chris. "Machine Learning with Python Cookbook Practical Solutions from Preprocessing to Deep Learning". O'Reilly Media (2018).

Avila Julian and Hauck Trent. "Scikit-Learn Cookbook Over 80 Recipes for Machine Learning in Python With Scikit-Learn". 2nd Revised edition. Packt Publishing Ltd (2017).

James Gareth, Witten Daniela, Hastie Trevor and Tibshirani Robert. "An Introduction to Statistical Learning with Applications in R". Springer Science & Business Media (2021).

Ester, Martin, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise." KDD (1996).

Hastie Trevor, Tibshirani Robert and Jerome Friedman. "The Elements of Statistical Learning: Data Mining, Inference, and Prediction". Springer (2013).

Géron Aurélien. "Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems". O'Reilly Media (2019).

Kuhn Max and Johnson Kjell. "Applied Predictive Modeling". Springer (2013).

Awad Mariette and Khanna Rahul. "Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers". 1st Edition. Apress (2015)

Wolpert David H. "Stacked generalization". Neural networks 5.2 (1992): 241-259.