A Comparative Study of Various Clustering Algorithms

This article involves the use of different cluster analysis algorithms, with the target of clustering/grouping the senators using their votes as features for 15 different bills in the 114th United States Congress. Finally, the performance of the algorithms in clustering will be compared. I used the data and was inspired by the studies here.

Wikipedia's definition for cluster analysis is: "Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups."

Another (more machine learning-wise) definition is: "Clustering or cluster analysis is an unsupervised learning problem." This technique is most often used to understand the interesting patterns in data, such as groups of people based on their behavior.

Cluster analysis can be achieved by various algorithms that differ significantly in their understanding of what constitutes a cluster and how to efficiently find them. Cluster analysis is not an automatic task, but an iterative process of optimization that involves trial and failure. It is often necessary to modify data pre-processing and model parameters until the result achieves the desired properties.

There are many different algorithms for cluster analysis - some of them going back 1930s, so it may be beneficial to try different clustering algorithms in machine learning and compare their efficiency in this specific clustering task. In this paper, I will introduce the results of the following algorithms:

- K Means Clustering,
- Hierarchical Clustering,
- Gaussian Mixture Models.

I tried to concentrate on the following issues for the selected dataset and algorithms:

- How many clusters (parties) are there?
- Was the algorithm successful in detecting the clusters?
- Comparison of the performances of these algorithms.

Visual exploratory analysis of the results using Pandas and Seaborn give a better picture about the efficiency of clustering algorithms.

Data Cleaning

The first step is to import and clean the data (if needed) using pandas before starting the cluster analysis.

```python
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.filterwarnings("ignore", category=FutureWarning)
```

```python
df = pd.read_csv('114_congress.csv')
```

```python
df.head()
```

	name	party	state	00001	00004	00005	00006	00007	00008	00009	00010	00020	00026	00032	00038	00039	00044	00047
0	Alexander	R	TN	0.0	1.0	1.0	1.0	1.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
1	Ayotte	R	NH	0.0	1.0	1.0	1.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
2	Baldwin	D	WI	1.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0
3	Barrasso	R	WY	0.0	1.0	1.0	1.0	1.0	0.0	1.0	1.0	1.0	1.0	0.0	0.0	1.0	0.0	0.0
4	Bennet	D	CO	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0

```python
df.shape
```

```
(100, 18)
```

According to the dataset, there were 54 Republicans, 44 Democrats and 2 Independents.

```python
df.party.value_counts()
```

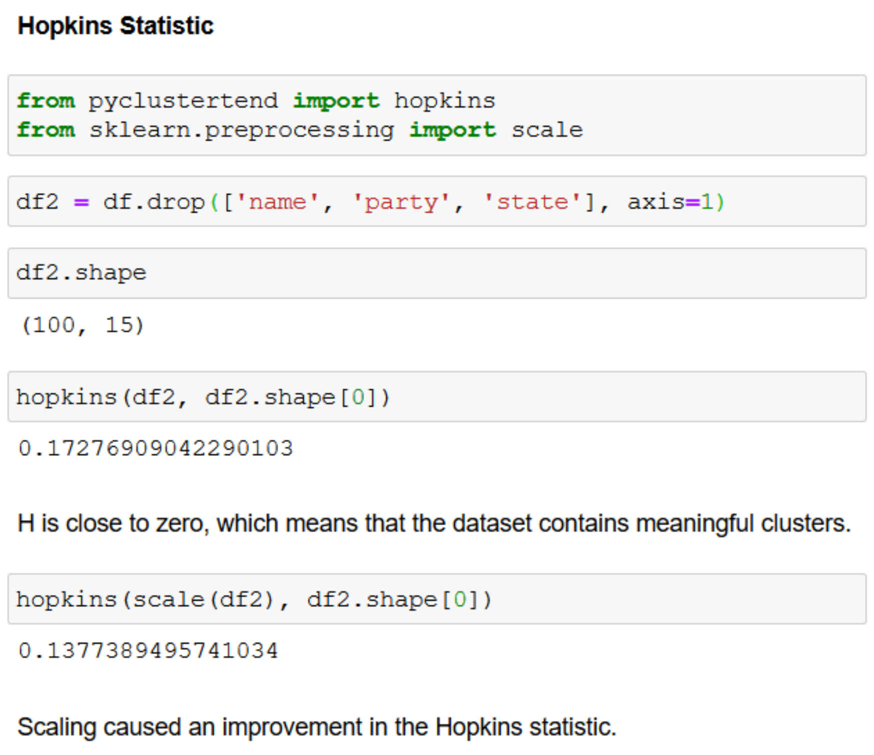
```
R    54
D    44
I     2
Name: party, dtype: int64
```

There are 100 rows (senators) and 18 columns (attributes) of data. 15 columns cover the votes of the senators for the bills and the first three columns are the name, party and state of the senator. Our target is to estimate the party, so the first three columns will be dropped.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 18 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   name    100 non-null    object 
 1   party   100 non-null    object 
 2   state   100 non-null    object 
 3   00001   100 non-null    float64
 4   00004   100 non-null    float64
 5   00005   100 non-null    float64
 6   00006   100 non-null    float64
 7   00007   100 non-null    float64
 8   00008   100 non-null    float64
 9   00009   100 non-null    float64
 10  00010   100 non-null    float64
 11  00020   100 non-null    float64
 12  00026   100 non-null    float64
 13  00032   100 non-null    float64
 14  00038   100 non-null    float64
 15  00039   100 non-null    float64
 16  00044   100 non-null    float64
 17  00047   100 non-null    float64
dtypes: float64(15), object(3)
memory usage: 14.2+ KB
```

The dataset is clean (there are no NaNs, Dtype are correct), so we will directly start by checking the Hopkins statistic in order to have an idea about the cluster tendency of the dataset. The rule of the thumb is that if the Hopkins statistic is above 0.5 than the dataset is not clusterable. If the value of H is close to zero, then we can reject null hypothesis and deduce that dataset contains meaningful clusters.



Hopkins Statistic

```
from pyclustertend import hopkins
from sklearn.preprocessing import scale
```

```
df2 = df.drop(['name', 'party', 'state'], axis=1)
```

```
df2.shape
```

```
(100, 15)
```

```
hopkins(df2, df2.shape[0])
```

```
0.17276909042290103
```

H is close to zero, which means that the dataset contains meaningful clusters.

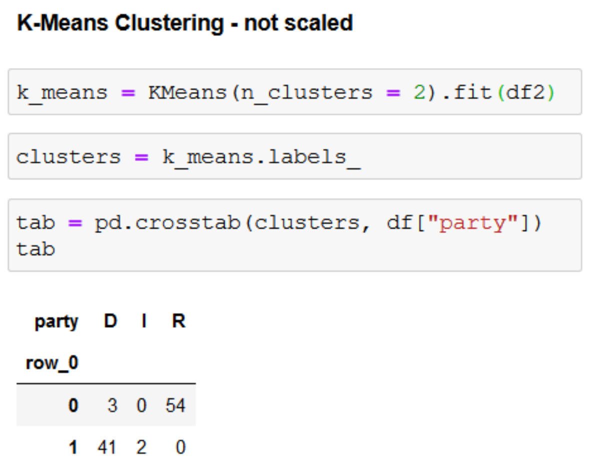
```
hopkins(scale(df2), df2.shape[0])
```

```
0.1377389495741034
```

Scaling caused an improvement in the Hopkins statistic.

K-Means Clustering

Applying the K-Means algorithm without scaling showed that the algorithm made a very precise prediction for the Republicans (and clustered all of them correctly). For the Democrats, most probably a few senators (around 3) did not vote according to the Party decisions and the algorithm assumed them as small cluster.



K-Means Clustering - not scaled

```
k_means = KMeans(n_clusters = 2).fit(df2)
```

```
clusters = k_means.labels_
```

```
tab = pd.crosstab(clusters, df["party"])
tab
```

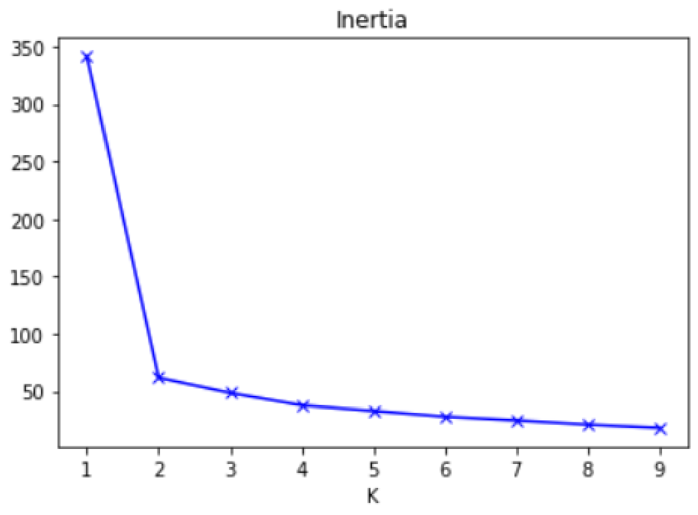
party	D	I	R
row_0			
0	3	0	54
1	41	2	0

Elbow Method

A crucial stage for any unsupervised algorithm will be to determine the optimal number of clusters into which the data may be clustered. Elbow Method is one of the most popular methods to determine the optimal value of k. Always considering the target, we are trying to figure out the number of clusters - number of clusters of senators who are voting as a group.

```
ssd = []
K = range(1,10)
for k in K:
    kmeans = KMeans(n_clusters = k).fit((df2))
    ssd.append(kmeans.inertia_)
```

```
plt.plot(K, ssd, "bx-")
plt.xlabel("K")
plt.title("Inertia");
```



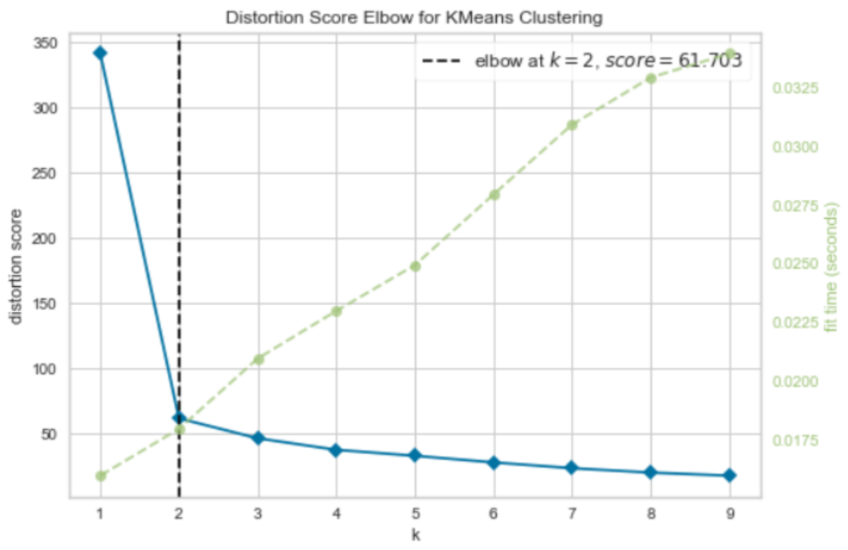
Considering the graph above, there are two clusters/groups, which was quite expected.

<u>Elbow Method – Yellowbrick Visualizing</u>

There are other codes used for estimating the number of clusters; Yellowbrick is the most popular. Yellowbrick gave the same result for the number of clusters, which is 2.

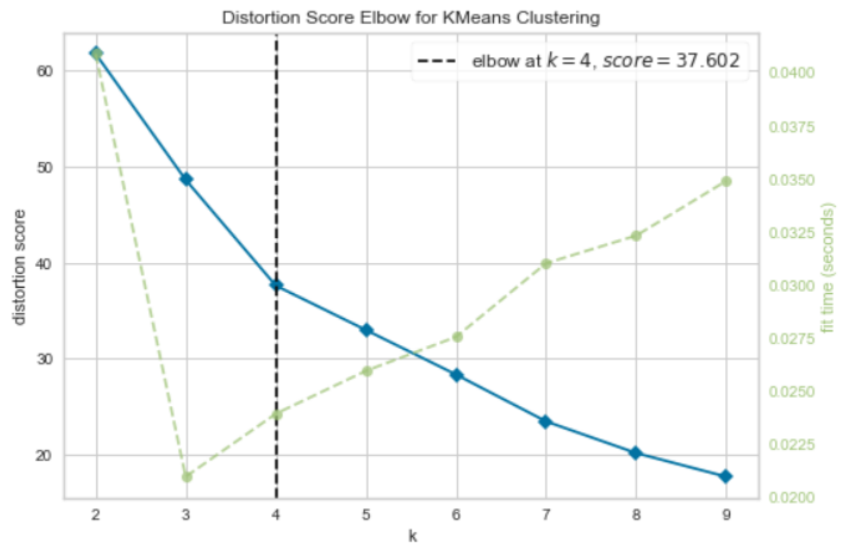
```
from yellowbrick.cluster import KElbowVisualizer
```

```
kmeans = KMeans()
visu = KElbowVisualizer(kmeans, k = (1,10))
visu.fit(df2)
visu.poof();
```

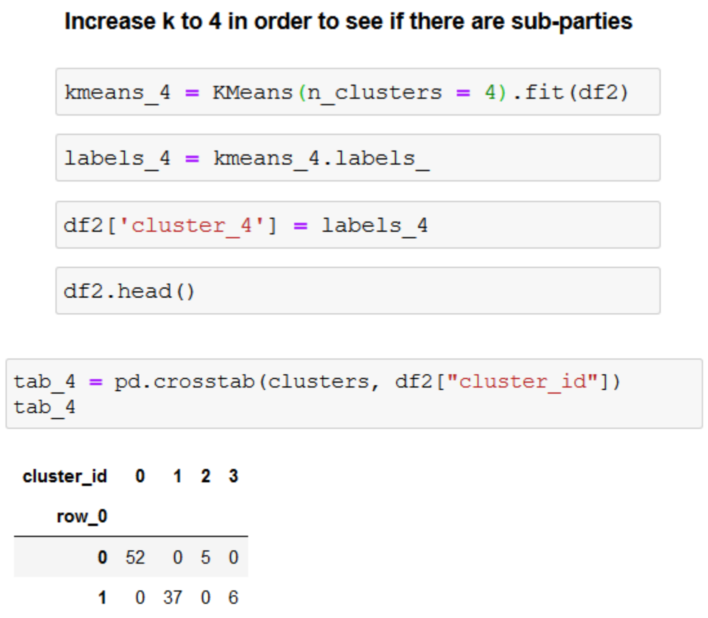


I run the Yellowbrick algorithm once again; this time using a possible cluster range between 2 and 10. Interestingly, Yellowbrick's advice was to use 4 clusters.

```
kmeans = KMeans()
visu = KElbowVisualizer(kmeans, k = (2,10))
visu.fit(df2)
visu.poof();
```



The results say that (when the k-search range is between 2 and 10), there are 4 clusters; although there are 2 parties and 2 independent senators. As I mentioned before, most probably, some senators do not always like taking orders while voting. Increasing the k size may give us a feeling about the sub-divisions in the parties (or in just one party).



Increase k to 4 in order to see if there are sub-parties

```
kmeans_4 = KMeans(n_clusters = 4).fit(df2)
```

```
labels_4 = kmeans_4.labels_
```

```
df2['cluster_4'] = labels_4
```

```
df2.head()
```

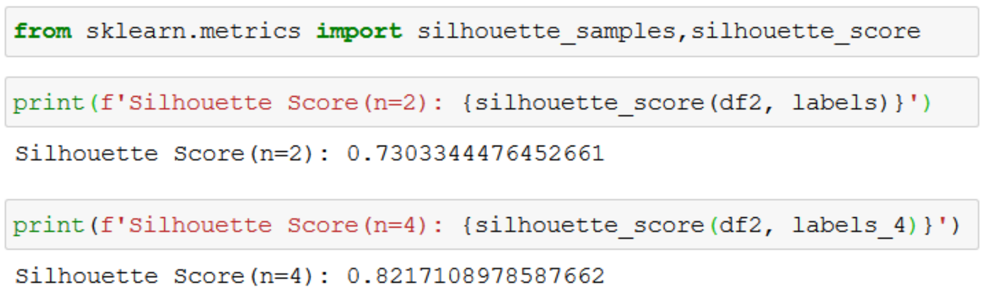
```
tab_4 = pd.crosstab(clusters, df2["cluster_id"])
tab_4
```

cluster_id	0	1	2	3
row_0				
0	52	0	5	0
1	0	37	0	6

Checking for 4 clusters (parties/groups) tell us that, Republican party senators are still a tight cluster, but Democrat party senators tend to vote as one big group and two smaller clusters.

Silhouette Scores

Silhouette is a method of interpretation and validation of consistency within clusters of data. Silhouette value is a measure of how similar an object is to its own cluster (intra-cluster distance) compared to other clusters (inter-cluster distance). The silhouette ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to the neighboring clusters.



```
from sklearn.metrics import silhouette_samples,silhouette_score
```

```
print(f'Silhouette Score(n=2): {silhouette_score(df2, labels)}')
```

```
Silhouette Score(n=2): 0.7303344476452661
```

```
print(f'Silhouette Score(n=4): {silhouette_score(df2, labels_4)}')
```

```
Silhouette Score(n=4): 0.8217108978587662
```

Well, here comes the interesting part. Silhouette score for two clusters/parties is 0.73 and 0.82 for four clusters/parties. Remembering the definition of the Silhouette score again, a higher value means that four clusters may be considered as a fact.

K-Means Clustering – Scaled

I used the scaled set in order to compare the results with the unscaled data. The results were quite similar. Most often than not, the literature says that, for K-Means algorithms, scaling provides (most often than not) more precise results. In cases of combining data of similar types, scaling may not be necessary; which is most probably our case.

```
hopkins(scale(df3), df3.shape[0])
```

```
0.13179016969095544
```

```
k_means_s = KMeans(n_clusters = 2).fit(scale(df3))
```

```
clusters_s = k_means_s.labels_
```

```
tab = pd.crosstab(clusters_s, df["party"])
tab
```

party	D	I	R
row_0			
0	3	0	54
1	41	2	0

Hierarchical Clustering

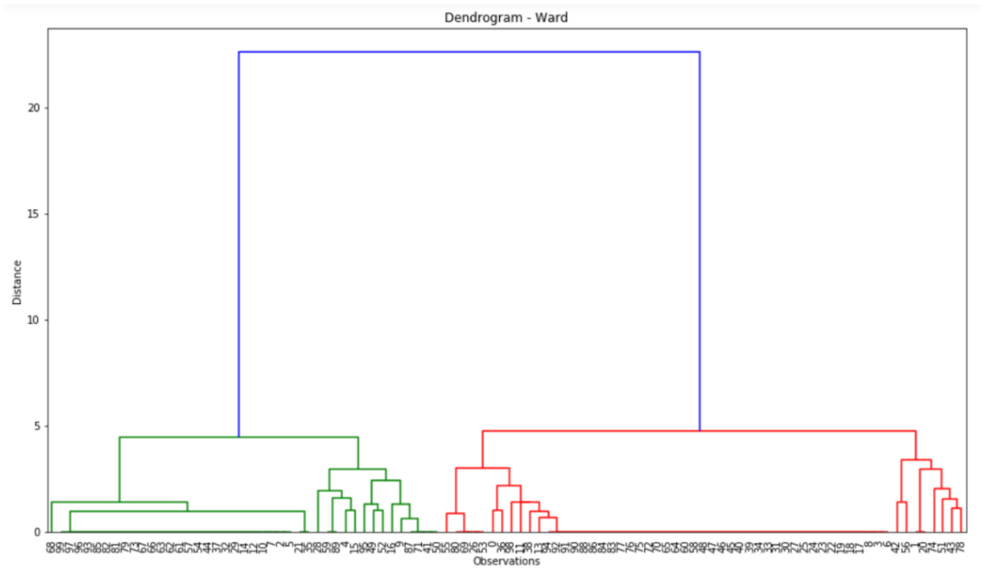
Hierarchical clustering is a method of cluster analysis which seeks to build a hierarchy of clusters. It is an unsupervised clustering algorithm and tries to group similar objects (in our case, senators) into groups called clusters.

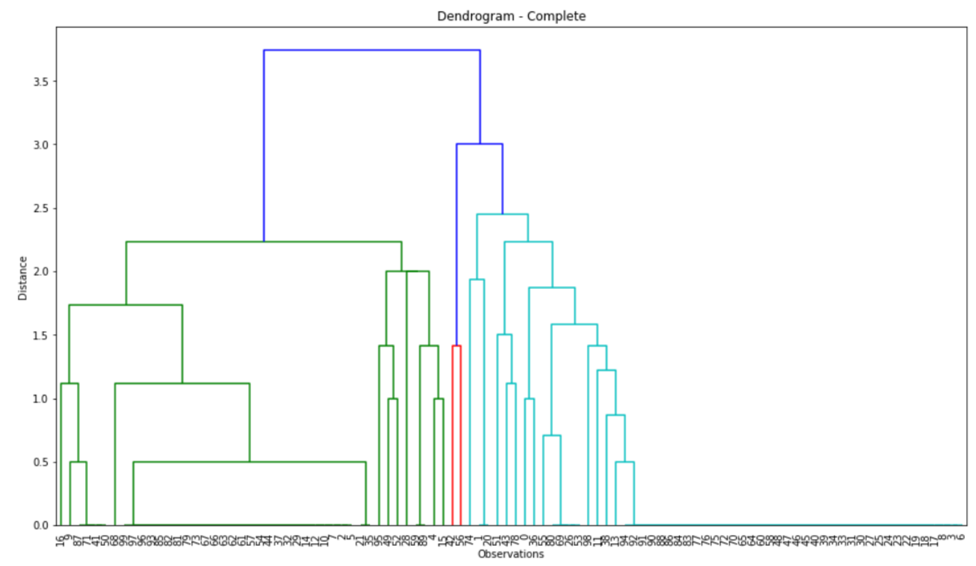
```
from scipy.cluster.hierarchy import linkage
```

```
df2 = df.drop(['name', 'party', 'state'], axis=1)
```

```
hc_complete = linkage(df2, 'complete')
hc_ward = linkage(df2, 'ward')
```

Once the above coding is used to run the algorithm, it is possible to get quite a visual diagram, which is called a dendrogram. Two different approaches, “Ward” and “complete”, were used in order to observe their effect on the clustering.



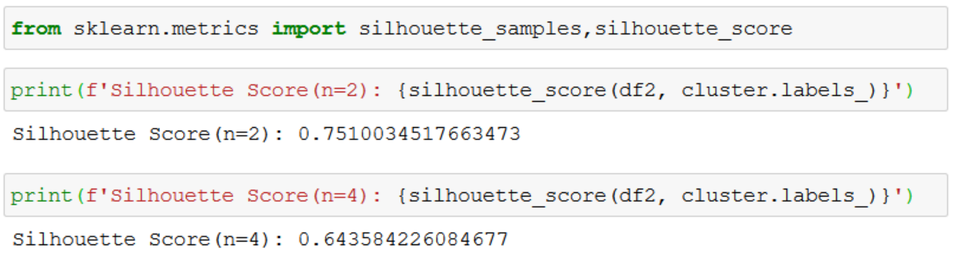


According to the dendrograms, 2 clusters were selected as the possible number of clusters when "complete" was used as the linkage parameter in the algorithm and 4 clusters were selected when "ward" was used ("ward" is also the default value of the linkage parameter).

Silhouette Scores

Remembering that a higher Silhouette Score value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters, two clusters/groups/party groups came out as the more probable result from the Hierarchical Clustering algorithm.

This result does not match with what the K-Means Clustering algorithm produced.



```
from sklearn.metrics import silhouette_samples,silhouette_score
```

```
print(f'Silhouette Score(n=2): {silhouette_score(df2, cluster.labels_)}')
```

```
Silhouette Score(n=2): 0.7510034517663473
```

```
print(f'Silhouette Score(n=4): {silhouette_score(df2, cluster.labels_)}')
```

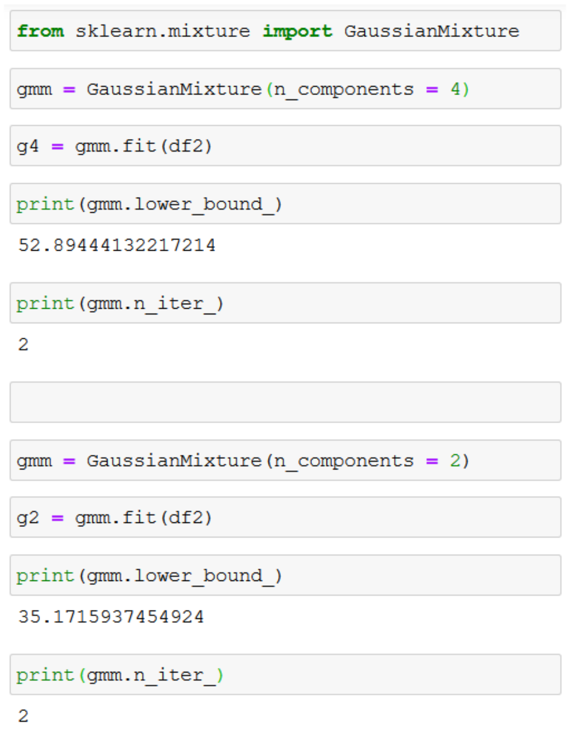
```
Silhouette Score(n=4): 0.643584226084677
```

Gaussian Mixture Models

As the final algorithm, Gaussian Mixture Model (GMM) was used. GMMs assume that there are a certain number of Gaussian distributions, and each of these distributions represent a cluster. Hence, a Gaussian Mixture Model tends to group the data points belonging to a single distribution together. GMMs are probabilistic models and use the soft clustering approach for distributing the points in different clusters. Unlike K-Means, Gaussian mixture models account for variance and returns the probability that a data point belongs to each of the *K* clusters.

It may not be the subject of this article, but there is a distinction between probability and likelihood: Probability attaches to possible results; likelihood attaches to hypotheses. In this analysis, the hypothesis was, the dataset is probably clustered into 2-4 groups and log-likelihood values were used to find out the stronger possibility.

Converged log-likelihood values and number of iterations needed for the model to converge for 2 and 4 clusters are as follows:



```
from sklearn.mixture import GaussianMixture
```

```
gmm = GaussianMixture(n_components = 4)
```

```
g4 = gmm.fit(df2)
```

```
print(gmm.lower_bound_)
```

```
52.89444132217214
```

```
print(gmm.n_iter_)
```

```
2
```

```
gmm = GaussianMixture(n_components = 2)
```

```
g2 = gmm.fit(df2)
```

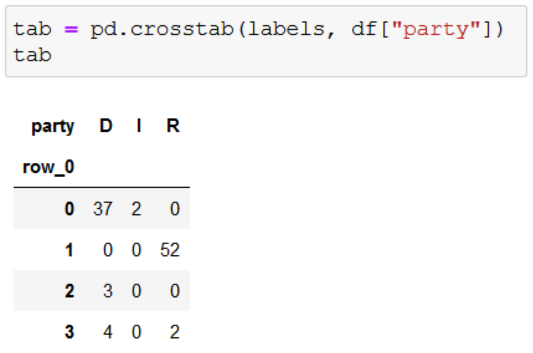
```
print(gmm.lower_bound_)
```

```
35.1715937454924
```

```
print(gmm.n_iter_)
```

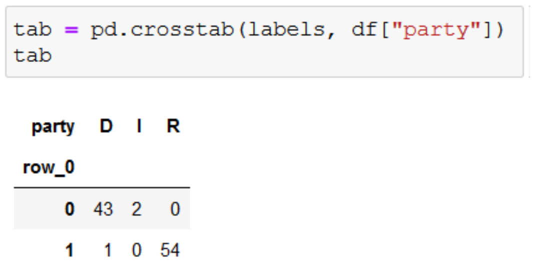
```
2
```

According to the GMM algorithm, the clusters and the details are:



```python
tab = pd.crosstab(labels, df["party"])
tab
```

party	D	I	R
row_0			
0	37	2	0
1	0	0	52
2	3	0	0
3	4	0	2



```python
tab = pd.crosstab(labels, df["party"])
tab
```

party	D	I	R
row_0			
0	43	2	0
1	1	0	54

Conclusion

In this article, I used three different Unsupervised Machine Learning (clustering) algorithms with the purpose of predicting the party membership of the senators according to their votes for 15 bills in the 114th US Congress. The analysis provides evidence that:

- Hopkins statistic showed that, the data was not uniformly distributed; on the contrary there were clusters.
- All three algorithms provided similar results showing that Republican Senators came out as a tight cluster, which means that their voting was quite similar.
- On the other hand, checking for more than 2 clusters proved that, for some bills, some Democrats had the tendency to vote with the opposing party.
- Scaling did not improve the results significantly for the K-Means Clustering.
- Silhouette scores for K-Means Clustering and Hierarchical Clustering methods did not match.

You can access to this article and similar ones here.

Photo by Darren Halstead on Unsplash