Codon Usage For Organism Clustering

IBM Unsupervised Machine Learning: Clustering

Course Assignment

Vladas Alesius

2021-06-27

Main Objective

The purpose of this study is to see how different clustering approaches work to cluster organisms using their codon information. Codons are triplets of nucleotides (organic units that compose RNA and DNA), and they encode information within genetic material. The primary interest is training a model that is capable of telling between different organism kingdoms - in other words, the focus is **clustering**.

We will examine three types of clustering algorithms:

- k-means
- Agglomerative Clustering
- Mean Shift

In all three cases, the best parameters will be searched. For k-means, it will be the number of clusters. Agglomerative Clustering model will be tuned using the number of clusters and the linkage type. Finally, we will search for the best bandwidth for Mean Shift.

These models differ in scalability - some are better for different numbers of features or samples. Also, they might have different usecases - for example, some models might perform better on datasets with uneven cluster sizes than others (such as Mean Shift). Data properties will be checked in below parts.

After fitting the models, their efficiency will be compared using several metrics. Calculating Adjusted Rand Score and Contingency Matrix will show how well the calculated clusters align with the "true" labels. However, Silhouette Coefficient and Calinski-Harabasz index do not require ground truth labels.

Additionally, Principal Component Analysis will be applied on the dataset to reduce dimensionality and avoid multicollinearity. The number of components will be selected to explain as much variance as possible, while keeping few dimensions. For PCA, data transformation and standardization are necessary, and will be performed in later parts.

As per the description, the dataset contains "codon usage frequencies in the genomic coding DNA of a large sample of diverse organisms from different taxa tabulated in the CUTG database." Please see Data Exploration part for further details.

The codon dataset was used in the following publications:

- *Khomtchouk BB: 'Codon usage bias levels predict taxonomic identity and genetic composition'. bioRxiv, 2020, doi: 10.1101/2020.10.26.356295*
- *Nakamura Y, Gojobori T, Ikemura T: 'Codon usage tabulated from international DNA sequence databases: status for the year 2000'. Nucleic Acids Research, 2000, 28:292.*

We will start with exploratory data analysis, to see how our variables are distributed and correlated, what transformations might be necessary.

Data Description

We will start with importing necessary modules:

In [1]:
```
import pandas as pd
import seaborn as sns
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import PowerTransformer, LabelEncoder
from sklearn.cluster import KMeans, AgglomerativeClustering, MeanShift, estimate_bandwidth
from sklearn.metrics import adjusted_rand_score, silhouette_score, calinski_harabasz_score
from matplotlib import pyplot as plt
```

The dataset contains more than 13000 entries, the first rows are presented below:

In [2]:
```
df = pd.read_csv(r"C:\Users\Lenovo\Desktop\ibm_machine_learning\4_unsupervised_machine_learning\week3\assignmen
```

In [3]:
```
df.shape
```

Out[3]:
```
(13028, 69)
```

In [4]:
```
df.head()
```

Out[4]:

	Kingdom	DNAtype	SpeciesID	Ncodons	SpeciesName	UUU	UUC	UUA	UUG	CUU	...	CGG	AGA	AGG	GAU
0	vrl	0	100217	1995	Epizootic haematopoietic necrosis virus	0.01654	0.01203	0.00050	0.00351	0.01203	...	0.00451	0.01303	0.03559	0.01003
1	vrl	0	100220	1474	Bohle iridovirus	0.02714	0.01357	0.00068	0.00678	0.00407	...	0.00136	0.01696	0.03596	0.01221
2	vrl	0	100755	4862	Sweet potato leaf curl virus	0.01974	0.0218	0.01357	0.01543	0.00782	...	0.00596	0.01974	0.02489	0.03126
3	vrl	0	100880	1915	Northern cereal mosaic virus	0.01775	0.02245	0.01619	0.00992	0.01567	...	0.00366	0.01410	0.01671	0.03760
4	vrl	0	100887	22831	Soil-borne cereal mosaic virus	0.02816	0.01371	0.00767	0.03679	0.01380	...	0.00604	0.01494	0.01734	0.04148

5 rows × 69 columns

The following features are provided in the dataset:

- Column 1: Kingdom
- Column 2: DNAtype
- Column 3: SpeciesID
- Column 4: Ncodons
- Column 5: SpeciesName
- Columns 6-69: codon (header: nucleotide bases; entries: frequency of usage (5 digit floating point number))

The 'Kingdom' is a 3-letter code corresponding to the group of organisms: 'arc'(archaea), 'bct'(bacteria), 'phg'(bacteriophage), 'plm' (plasmid), 'pln' (plant), 'inv' (invertebrate), 'vrt' (vertebrate), 'mam' (mammal), 'rod' (rodent), 'pri' (primate), and 'vrl'(virus) sequence entries.

The 'DNAtype' is denoted as an integer for the genomic composition in the species: 0-genomic, 1-mitochondrial, 2-chloroplast, 3-cyanelle, 4-plastid, 5-nucleomorph, 6-secondary_endosymbiont, 7-chromoplast, 8-leucoplast, 9-NA, 10-proplastid, 11-apicoplast, and 12-kinetoplast.

The species identifier ('SpeciesID') is an integer, which uniquely indicates the entries of an organism.

The number of codons ('Ncodons') is the algebraic sum of the numbers listed for the different codons in an entry. Codon frequencies are normalized to the total codon count, hence the number of occurrences divided by 'Ncodons' is the codon frequencies listed in the data file.

The species' name ('SpeciesName') is represented in strings purged of 'comma' (which are now replaced by 'space'). This is a descriptive label of the name of the species for data interpretations.

Lastly, the codon frequencies ('codon') including 'UUU', 'UUA', 'UUG', 'CUU', etc., are recorded as floats (with decimals in 5 digits).

Data Exploration

As stated above, the dataset contains data for 13 different DNA types. They might differ among samples from different kingdoms (e.g., chromoplast DNA entries will be related mostly to plants), so we will stick to type 0 (genomic DNA) only:

In [5]:
```
df = df[df.DNAtype == 0]
```

The reduced dataset is smaller by about a third:

In [6]:
```
df.shape
```

Out[6]:
```
(9267, 69)
```

The vast majority of all variables are numeric, either float or integer:

In [7]:
```
df.dtypes.value_counts()
```

Out[7]:
```
float64    62
object      4
int64       3
dtype: int64
```

Examining data types shows that some codon related variables are of object type, because of non-numeric entries:

In [8]:
```
df.dtypes[:10]
```

Out[8]:
```
Kingdom         object
DNAtype          int64
SpeciesID        int64
Ncodons          int64
SpeciesName     object
UUU             object
UUC             object
UUA            float64
UUG            float64
CUU            float64
dtype: object
```

All codon related variables will be made numeric below, entries with NA values will be dropped. Only 2 entries will be lost after dropping:

In [9]:
```
for col in df.columns[5:]:
    if df[col].dtype == 'object':
        df[col] = pd.to_numeric(df[col], errors='coerce')
```

In [10]:
```
df.dropna(inplace=True)
```

In [11]:
```
df.shape
```

Out[11]:
```
(9265, 69)
```

According to the dataset description, 'Kingdom' field contains information about the species classification. However, its entries might mean not only actual biological kingdom, but some lower taxonomic units as well (such as belonging to primates, rodents etc.). In order to be consistent, all entries will be assigned the actual biological kingdom they belong to:

In [12]:
```
df[['Kingdom']].value_counts()
```

Out[12]:
```
Kingdom
bct        2917
vrl        2831
pln        1523
inv         922
vrt         464
phg         220
arc         126
mam         102
pri          83
rod          59
plm          18
dtype: int64
```

In [13]:
```
df['Kingdom'].replace(['phg', 'plm', 'inv', 'vrt', 'mam', 'rod', 'pri', 'bct', 'arc'],
                      ['vrl', 'mon', 'anm', 'anm', 'anm', 'anm', 'anm', 'mon', 'mon'], inplace=True)
```

After the replace, only 4 kingdoms remain - moneros, viruses, animals and plants:

In [14]:
```
df[['Kingdom']].value_counts()
```

Out[14]:
```
Kingdom
mon        3061
vrl        3051
anm        1630
pln        1523
dtype: int64
```

Let's keep kingdom data in a separate series, so that it can be used later on to assess clustering efficiency:

In [15]:
```
kingdoms = df.Kingdom
```

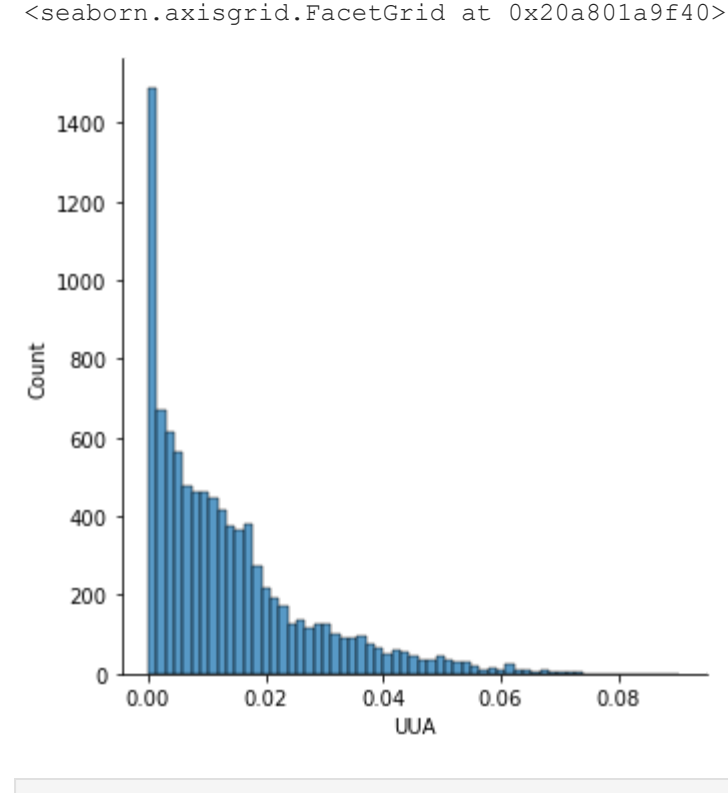
For clustering, only codon data will be used, so the rest of columns will be dropped:

In [16]:
```
df = df.iloc[:, 5:]
```

The histograms below show that some codon columns are skewed and/or have multiple modes. In other words, their distribution differs from normal, which is desirable in data analysis (such as Principal Component Analysis, which we will demonstrate below):

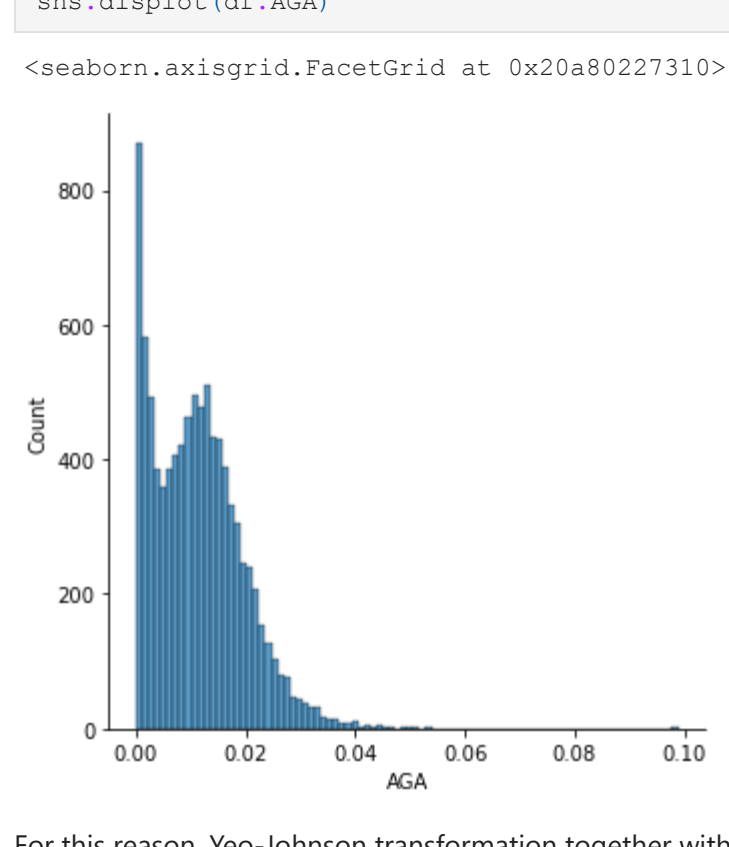
In [17]:
```
sns.displot(df.UUA)
```

Out[17]:
```
<seaborn.axisgrid.FacetGrid at 0x20a801a9f40>
```



In [18]:
```
sns.displot(df.AGA)
```

Out[18]:
```
<seaborn.axisgrid.FacetGrid at 0x20a80227310>
```



For this reason, Yeo-Johnson transformation together with standardization will be applied, in order to make codon frequency features as close to normal distribution as possible:

In [19]:
```
df_s = PowerTransformer().fit_transform(df)
df_s = pd.DataFrame(df_s, columns=df.columns)
df_s.head()
```

Out[19]:

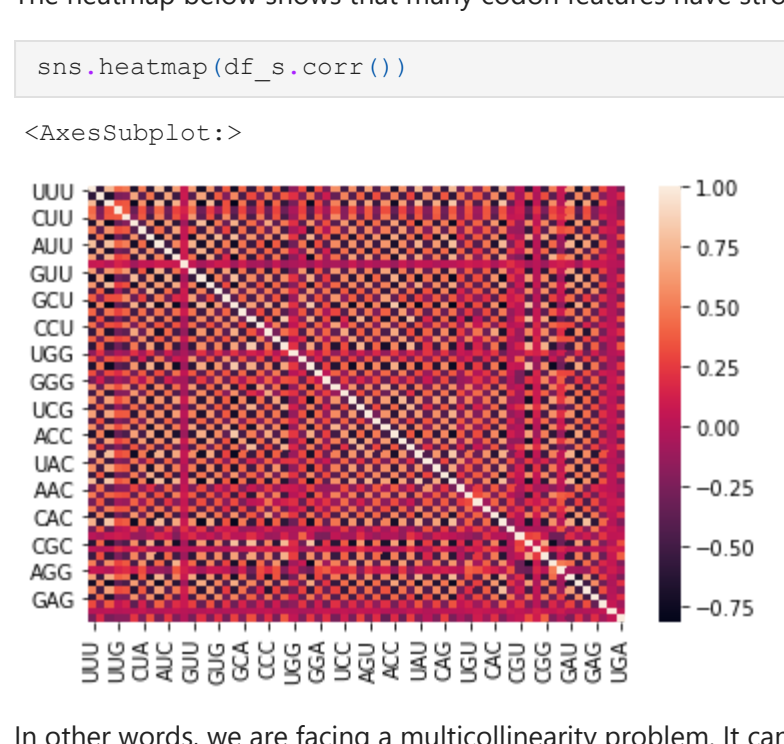
	UUU	UUC	UUA	UUG	CUU	CUC	CUA	CUG	AUU	AUC	...	CGG	AGA	AGG
0	-0.232325	-1.011920	-1.343770	-1.792038	-0.272402	1.454079	-1.532802	1.173442	-1.511870	-0.083982	...	-0.117761	0.359065	2.395750
1	0.722609	-0.807270	-1.313230	-1.213107	-1.601194	1.237002	-1.243567	1.311214	-0.867834	0.420071	...	-1.064644	0.807580	2.405832
2	0.073302	0.186284	0.309663	0.101207	-0.942562	-0.346413	0.604944	-0.517692	0.170147	-0.526307	...	0.219925	1.089774	1.964086
3	-0.114901	0.258069	0.533863	-0.701649	0.253551	-0.053495	0.440257	-0.070576	0.267899	0.162575	...	-0.342046	0.487330	1.345635
4	0.806221	-0.788975	-0.310672	2.332002	-0.010768	-1.126320	-0.558800	0.186773	0.516247	-1.449594	...	0.237047	0.584722	1.406396

5 rows × 64 columns

The heatmap below shows that many codon features have strong correlations, both positive and negative:

In [20]:
```
sns.heatmap(df_s.corr())
```

Out[20]:
```
<AxesSubplot:>
```



In other words, we are facing a multicollinearity problem. It can be solved with Principal Component Analysis (PCA). According to Sklearn tutorial, "PCA is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance." It will help to create uncorrelated variables (linear combinations of dataset features) and reduce their number. Since we have 64 features in total, we will try PCA with a number of components between 1 and 64, and see what percentage of variance each number explains:

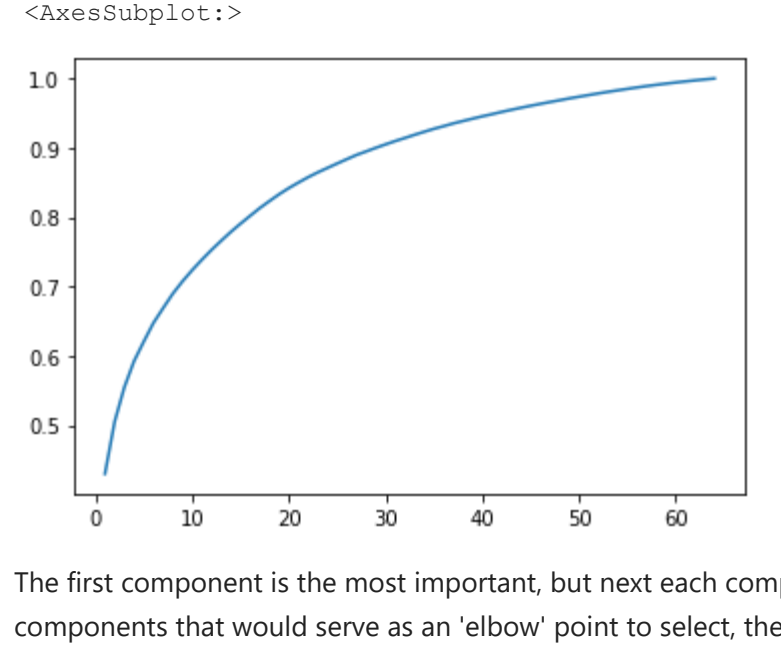
In [21]:
```
pca_list = list()

for n in range(1, 65):
    PCAmod = PCA(n_components=n)
    PCAmod.fit(df_s)
    pca_list.append(PCAmod.explained_variance_ratio_.sum())
```

As we can see from the plot below, it takes many components to achieve a high ratio of explained variance. For example, 90% of variance is explained by about 30 components:

In [23]:
```
sns.lineplot(x=range(1, 65), y=pca_list)
```

Out[23]:
```
<AxesSubplot:>
```



The first component is the most important, but next each component adds less and less explanation. As a result, there is no number of components that would serve as an 'elbow' point to select, the explained variance increases gradually. Taking 12 components helps explain 75% of variance, this number will be selected for our PCA.

In [24]:
```
pd.DataFrame({'n_comp':range(1, 65), 'pca':pca_list})[:15]
```

Out[24]:

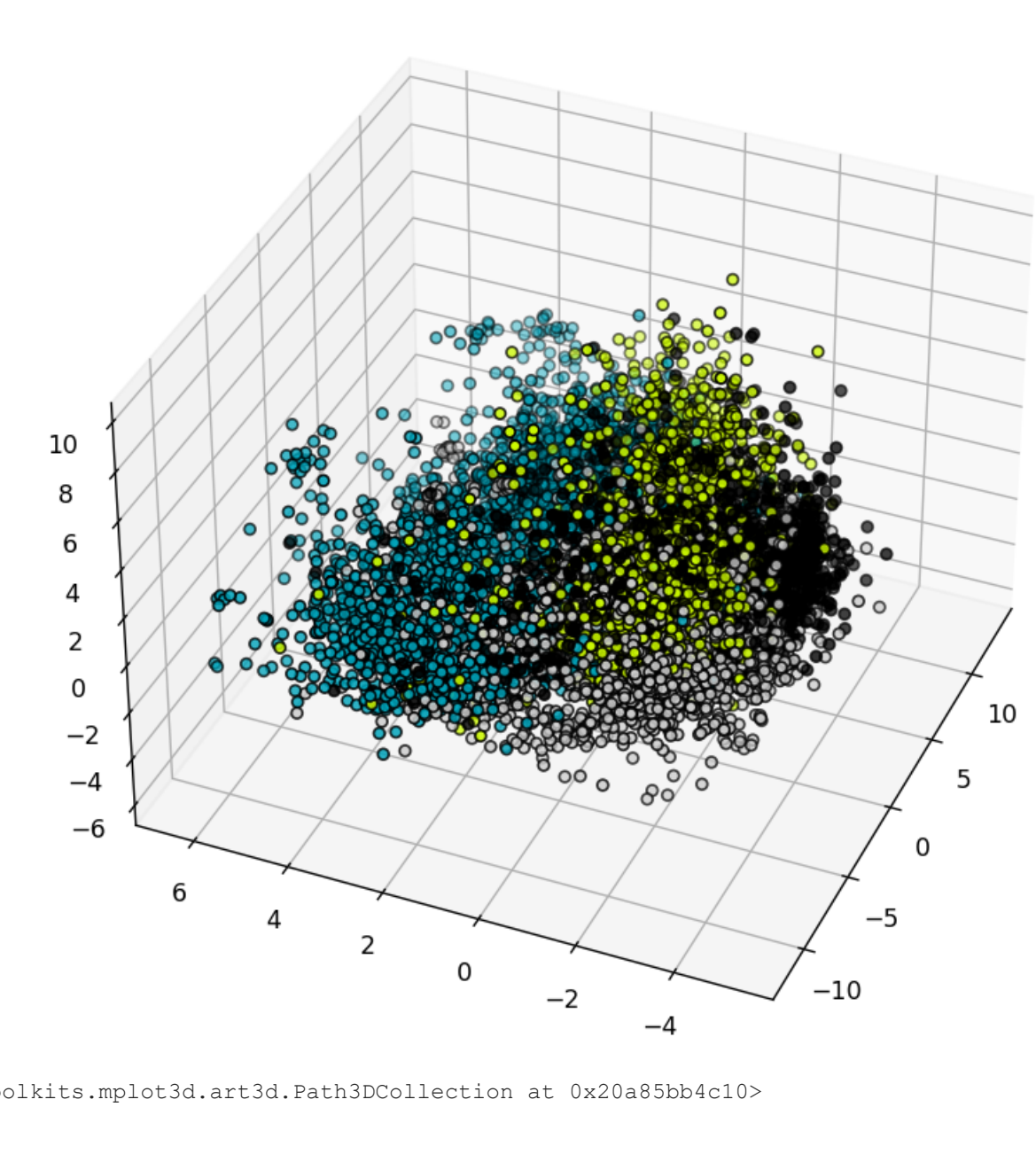
	n_comp	pca
0	1	0.429818
1	2	0.504948
2	3	0.554516
3	4	0.592152
4	5	0.620262
5	6	0.647223
6	7	0.668581
7	8	0.689410
8	9	0.707205
9	10	0.723359
10	11	0.738317
11	12	0.752541
12	13	0.765798
13	14	0.778616
14	15	0.790472

Before applying PCA to the dataset, let's visualize how well potential clusters are separated. We will select the first 3 principal components and visualize the dataset entries according to their kingdoms.

In [25]:
```
le = LabelEncoder()
kingdoms_le = le.fit_transform(kingdoms)
```

As we can see from the 3D chart below, the kingdoms tend to overlap, there are no clear boundaries between them. This can make it harder to cluster the organisms. However, we use only 3 first principal components to visualize the data that explain only about 55% of variance:

In [50]:
```
%matplotlib notebook
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
PCAmod = PCA(n_components=3)
X = PCAmod.fit_transform(df_s)
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=kingdoms_le, cmap=plt.cm.nipy_spectral,
           edgecolor='k')
```



Out[50]:
```
<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x20a85bb4c10>
```

Model Training

As described in the introduction, we will train three clustering models - k-means, Agglomerative Clustering and Mean Shift. We will assess their performance on telling between kingdoms of different organisms. Their performance will be assessed using four different metrics - Silhouette Coefficient, Calinski-Harabasz Index, Contingency Matrix and Adjusted Rand Score.

Among those metrics, Contingency Matrix and Adjusted Rand Score will require ground truth labels, for which we will use kingdom labels prepared above. Other metrics - Silhouette Coefficient and Calinski-Harabasz index - require no such labels. These metrics are shortly described below.

Adjusted Rand Score is proportional to the number of sample pairs whose labels are the same in both predicted and true labels, or are different in both. It can be between -1 and 1 (perfect labeling is scored 1), 0 meaning random label assignments.

Silhouette Coefficient is bounded between -1 for incorrect clustering and +1 for highly dense clustering. Scores around zero indicate overlapping clusters.

Calinski-Harabasz index is the ratio of the sum of between-clusters dispersion and of within-cluster dispersion for all clusters (where dispersion is defined as the sum of distances squared). A higher Calinski-Harabasz score relates to a model with better defined clusters.

Contingency matrix reports the intersection cardinality for every true/predicted cluster pair.

Now let's run PCA with 12 components, as described above. This processed dataset will be used for clustering below:

In [51]:
```
PCAmod = PCA(n_components=12, random_state=42)
df_s_pca = PCAmod.fit_transform(df_s)
```

In [52]:
```
df_s_pca = pd.DataFrame(df_s_pca)
df_s_pca.head()
```

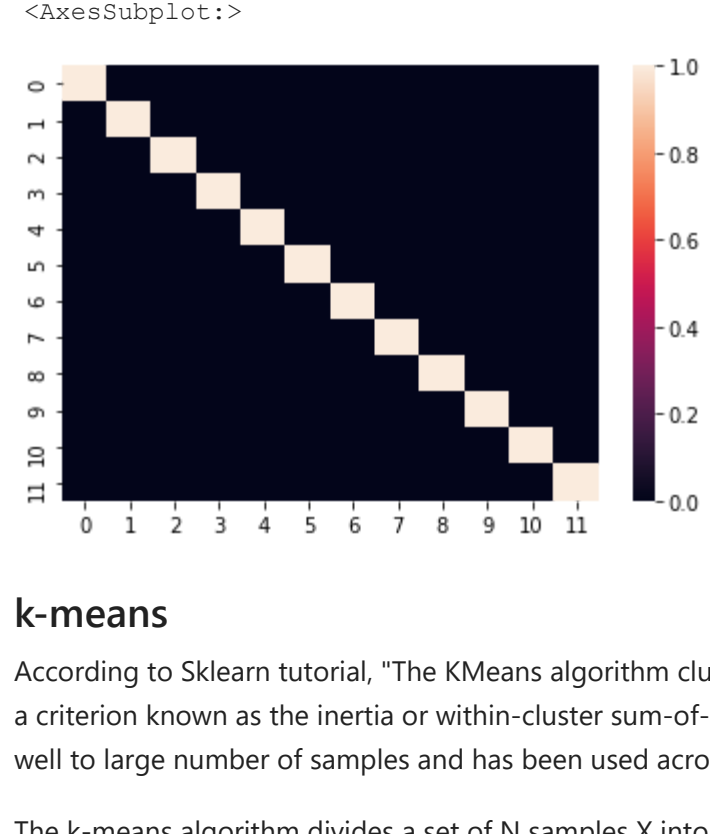
Out[52]:

	0	1	2	3	4	5	6	7	8	9	10	11
0	5.745618	-3.683023	0.129796	1.332753	0.738305	-0.308409	0.424477	0.252840	1.815368	-0.776459	-0.332412	-0.998918
1	5.032145	-3.757916	1.005059	3.411457	-0.261028	-0.211419	0.677138	-0.832573	0.808754	-0.836650	0.762782	-0.169716
2	-1.487814	-2.279710	-1.517634	-1.670556	-1.162737	0.860591	1.089175	0.308700	-0.282969	0.292758	-0.549317	-2.164097
3	-3.339949	-2.063569	-0.852803	1.091724	-0.406542	-1.665486	0.311130	1.345320	0.903403	-1.112769	0.999487	-0.422499
4	-2.575966	0.713268	-0.624448	-1.228539	-0.465571	-2.936104	0.557663	0.035491	-2.289824	-2.761067	-1.103762	-1.000001

As expected, the new variables are uncorrelated:

In [45]:
```
sns.heatmap(df_s_pca.corr())
```

Out[45]:
```
<AxesSubplot:>
```



k-means

According to Sklearn tutorial, "The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (...). This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields.

The k-means algorithm divides a set of N samples X into K disjoint clusters C, each described by the mean of the samples in the cluster. The means are commonly called the cluster "centroids"; note that they are not, in general, points from X, although they live in the same space."

We will run multiple k-means clustering models on the dataset processed with PCA above, with the number of clusters ranging from 1 to 10. Apart from the performance metrics described above, we will compute inertia (within-cluster sum-of-squares criterion - a measure of how internally coherent clusters are; lower values are better) for each cluster number.

In [53]:
```
km_lst = []

for c in range(1,11):
    km = KMeans(n_clusters=c, random_state=42)
    km = km.fit(df_s_pca)

    try:
        sil_score = silhouette_score(df_s_pca, km.labels_, metric='euclidean')
        ch_score = calinski_harabasz_score(df_s_pca, km.labels_)
    except:
        sil_score = 0
        ch_score = 0

    km_lst.append([c, round(km.inertia_, 0), adjusted_rand_score(kingdoms, km.labels_), sil_score, round(ch_sco

km_df = pd.DataFrame(km_lst, columns=['cluster_no', 'inertia', 'adj_rand', 'sil_score', 'ch_score'])
```

Inertia is the lowest for the maximum number of clusters (10). Adjusted Rand Score is the highest for a 5-cluster model (0.18), while Silhouette and CH scores are the highest for a 2-cluster model (0.32 and 5891).

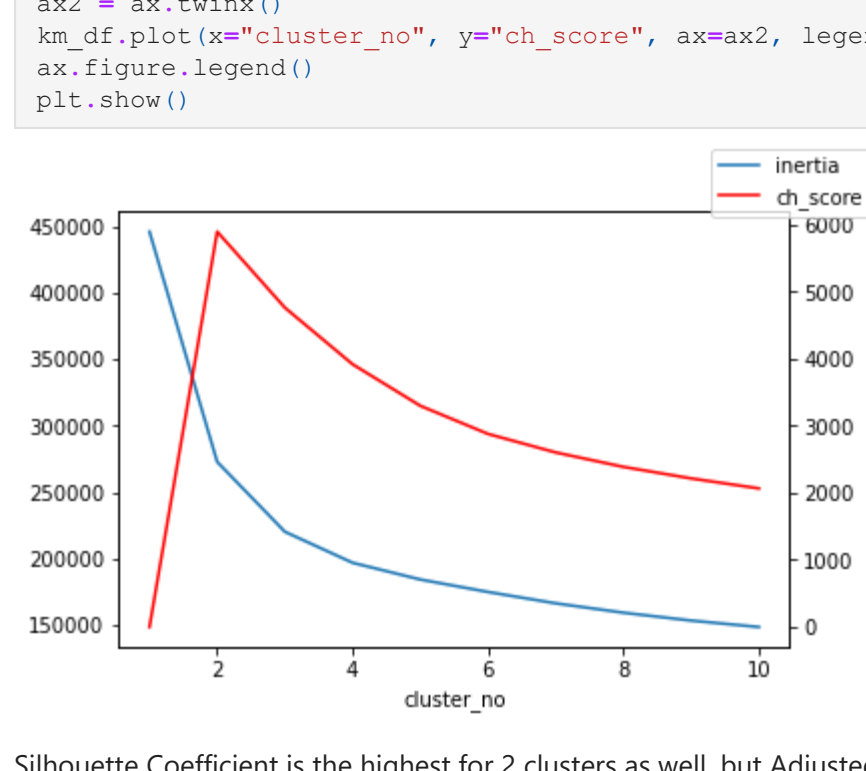
In [54]:
```
km_df
```

Out[54]:

	cluster_no	inertia	adj_rand	sil_score	ch_score
0	1	446227.0	0.000000	0.000000	0.0
1	2	272763.0	0.075393	0.322856	5891.0
2	3	220093.0	0.079800	0.237143	4758.0
3	4	196736.0	0.125795	0.192298	3915.0
4	5	184121.0	0.177546	0.201188	3296.0
5	6	174729.0	0.169620	0.168378	2877.0
6	7	166165.0	0.161284	0.165860	2601.0
7	8	159107.0	0.148920	0.168824	2386.0
8	9	153172.0	0.148758	0.169590	2214.0
9	10	148325.0	0.133827	0.153553	2065.0

As expected, inertia decreases together with a number of clusters. Calinski-Harabasz score is the highest for 2 clusters (the best number), and decreases for a higher number.

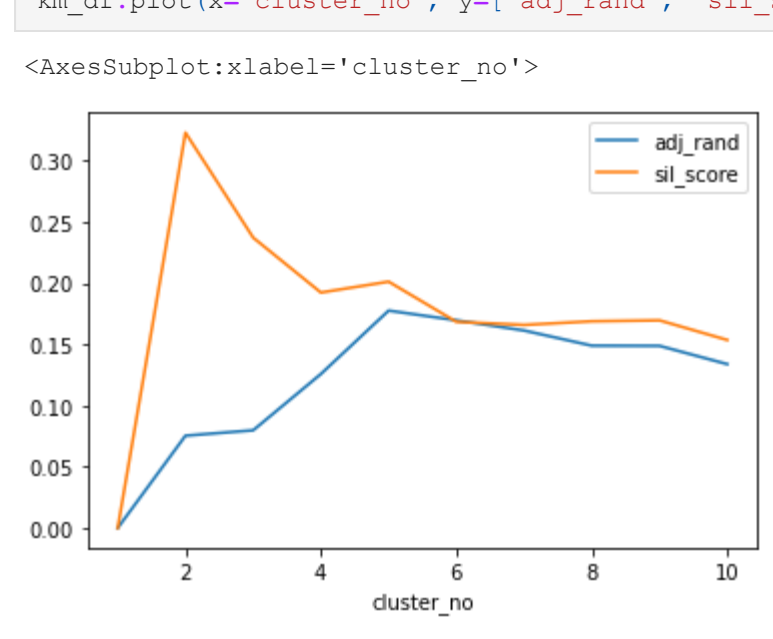
In [52]:
```
ax = km_df.plot(x="cluster_no", y="inertia", legend=False)
ax2 = ax.twinx()
km_df.plot(x="cluster_no", y="ch_score", ax=ax2, legend=False, color="r")
ax.figure.legend()
plt.show()
```



Silhouette Coefficient is the highest for 2 clusters as well, but Adjusted Rand Score is highest for a 5-cluster model. These scores get lower when the cluster number increases:

In [54]:
```
km_df.plot(x="cluster_no", y=["adj_rand", "sil_score"])
```

Out[54]:
```
<AxesSubplot:xlabel='cluster_no'>
```



Out of 3 metrics, 2 are the highest for the 2-cluster model. So we select this one as the best for k-means.

Cluster 0 is bigger than cluster 1, but they both contain thousands of entries:

In [73]:
```
km_2 = KMeans(n_clusters=2, random_state=42)
km_2 = km_2.fit(df_s_pca)
pd.DataFrame(km_2.labels_).value_counts()
```

Out[73]:
```
0    5742
1    3523
dtype: int64
```

Both clusters contain organisms from all kingdoms. Organisms from each kingdom are more or less equally separated into both clusters, except viruses (vrl) - cluster 0 contains almost 6 times more viruses than cluster 1:

In [74]:
```
pd.crosstab(kingdoms, km_2.labels_)
```

Out[74]:

col_0	0	1
Kingdom		
anm	786	844
mon	1488	1573
pln	867	656
vrl	2601	450

Agglomerative Clustering

The next model type we will examine is Agglomerative Clustering.

According to Sklearn tutorial, "Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample. (...) The AgglomerativeClustering object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:

Ward minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.

Maximum or complete linkage minimizes the maximum distance between observations of pairs of clusters.

Average linkage minimizes the average of the distances between all observations of pairs of clusters.

Single linkage minimizes the distance between the closest observations of pairs of clusters."

This algorithm will be tuned using two parameters - cluster number and linkage type. Cluster numbers will change from 1 to 10 (as for k-means), and all four linkage types will be tried. Adjusted Rand Score, Silhouette Coeffcient and Calinski-Harabasz score will be calculated.

In [55]:
```
linkages = ['ward', 'complete', 'average', 'single']
ag_lst = []
for l in linkages:
    for c in range(1, 11):
        ag = AgglomerativeClustering(n_clusters=c, linkage=l)
        ag = ag.fit(df_s)

        try:
            sil_score = silhouette_score(df_s_pca, ag.labels_, metric='euclidean')
            ch_score = calinski_harabasz_score(df_s_pca, ag.labels_)
        except:
            sil_score = 0
            ch_score = 0

        ag_lst.append([l, c, adjusted_rand_score(kingdoms, ag.labels_), sil_score, ch_score])

ag_df = pd.DataFrame(ag_lst, columns=['linkage', 'cluster_no', 'adj_rand_score', 'sil_score', 'ch_score'])
```

For each metric, different parameter selections give the best results.

Adjusted Rand Score is the highest for a 5-cluster ward algorithm (0.18). Silhouette Coefficient is the highest for a 2-cluster complete algorithm (0.33). Calinski-Harabasz score is the highest for a 2-cluster ward algorithm (4509).

In [56]:
```
max_rand = ag_df[ag_df.adj_rand_score == np.max(ag_df.adj_rand_score)]
```

In [57]:
```
max_sil = ag_df[ag_df.sil_score == np.max(ag_df.sil_score)]
```

In [58]:
```
max_ch = ag_df[ag_df.ch_score == np.max(ag_df.ch_score)]
```

In [59]:
```
max_ag = pd.concat([max_rand, max_sil, max_ch])
```

In [60]:
```
max_ag
```

Out[60]:

	linkage	cluster_no	adj_rand_score	sil_score	ch_score
4	ward	5	0.180124	0.153936	2709.721602
11	complete	2	0.034551	0.326619	4056.185164
1	ward	2	0.047456	0.252567	4509.298085

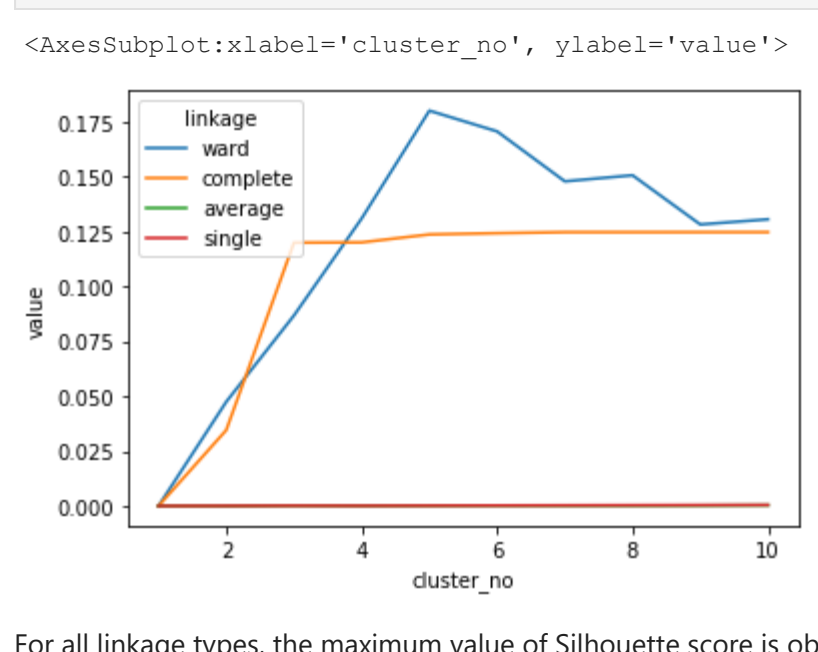
In [74]:
```
ag_df2 = pd.melt(ag_df, id_vars=["linkage", "cluster_no"])
```

The charts below show how each metric value changes for a linkage type and a number of clusters.

Adjusted Rand Score is the highest for a 5-cluster ward linkage model, and decreases for a bigger cluster number. However, in case of a complete linkage, its value rises until 3 clusters and remains almost the same for more.

In [79]:
```
sns.lineplot(x="cluster_no", y="value", hue="linkage", data=ag_df2[ag_df2.variable == "adj_rand_score"])
```

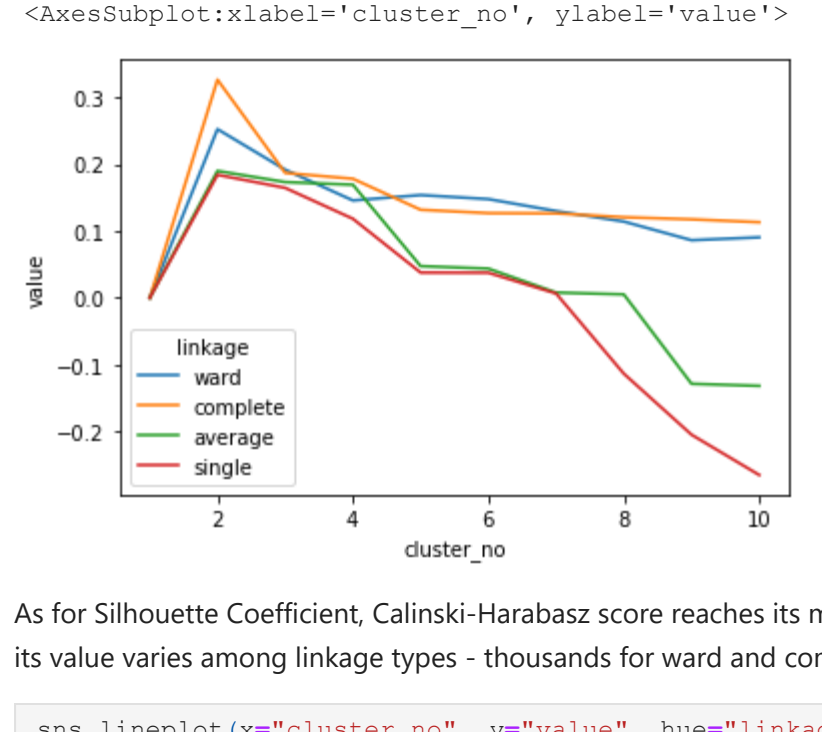
Out[79]:
```
<AxesSubplot:xlabel='cluster_no', ylabel='value'>
```



For all linkage types, the maximum value of Silhouette score is obtained with a 2-cluster model. For more clusters, the score value decreases for all linkages, but the pace of decrease is different. Average and single models even give negative score values for large cluster numbers.

In [80]:
```
sns.lineplot(x="cluster_no", y="value", hue="linkage", data=ag_df2[ag_df2.variable == "sil_score"])
```

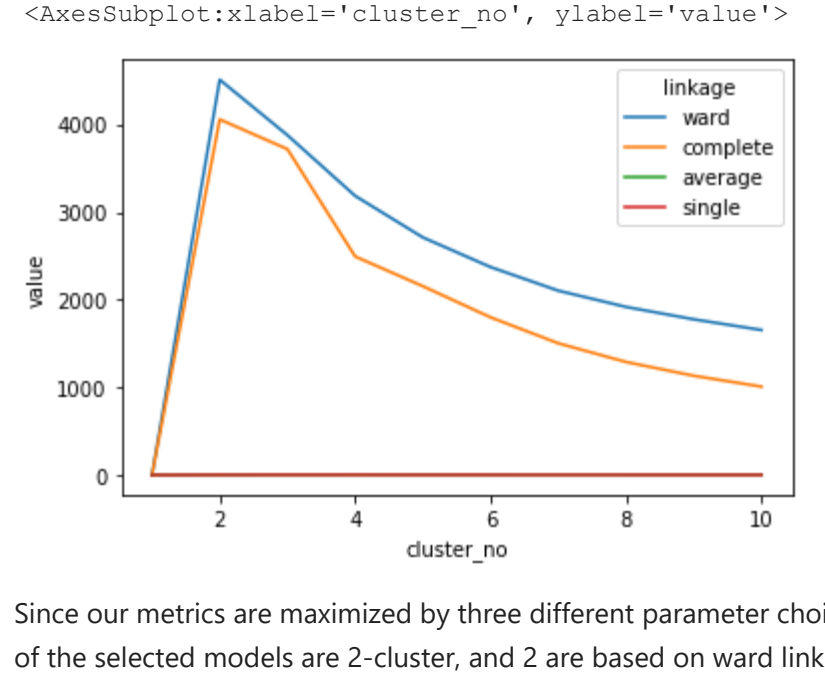
Out[80]:
```
<AxesSubplot:xlabel='cluster_no', ylabel='value'>
```



As for Silhouette Coefficient, Calinski-Harabasz score reaches its maximum value for 2-cluster models and decreases afterwards. However, its value varies among linkage types - thousands for ward and complete linkage vs close to zero for average and single linkage.

In [81]:
```
sns.lineplot(x="cluster_no", y="value", hue="linkage", data=ag_df2[ag_df2.variable == "ch_score"])
```

Out[81]:
```
<AxesSubplot:xlabel='cluster_no', ylabel='value'>
```



Since our metrics are maximized by three different parameter choices, we will select the best model based on parameter majority vote - 2 of the selected models are 2-cluster, and 2 are based on ward linkage. So we will select a 2-cluster ward linkage model, that provides the best Calinski-Harabasz score, and 2nd best values for Silhouette Coefficient and Adjusted Rand Score.

Both clusters are of similar size:

In [86]:
```
ag_ward_2 = AgglomerativeClustering(n_clusters=2, linkage="ward")
ag_ward_2 = ag_ward_2.fit(df_s)
pd.DataFrame(ag_ward_2.labels_).value_counts()
```

Out[86]:
```
0    4870
1    4395
dtype: int64
```

Cluster 0 contains more animals (anm) and moneros (mon), and cluster 1 - viruses (vrl):

In [87]:
```
pd.crosstab(kingdoms, ag_ward_2.labels_)
```

Out[87]:

col_0	0	1
Kingdom		
anm	1185	445
mon	1900	1161
pln	763	760
vrl	1022	2029

Mean Shift

The final algorithm is Mean Shift.

According to description in Sklearn tutorial, "MeanShift clustering aims to discover blobs in a smooth density of samples. It is a centroid based algorithm, which works by updating candidates for centroids to be the mean of the points within a given region. These candidates are then filtered in a post-processing stage to eliminate near-duplicates to form the final set of centroids. (...) The algorithm automatically sets the number of clusters, instead of relying on a parameter bandwidth, which dictates the size of the region to search through. This parameter can be set manually, but can be estimated using the provided estimate_bandwidth function, which is called if the bandwidth is not set."

Its bandwidth parameter will be estimated using a special function. It will be computed using different quantiles of pairwise distances between the dataset points. Experiments show that large enough quantiles fail to find any clusters in this case, so the quantiles do not exceed 0.3.

In [61]:
```
quantiles = np.linspace(0.02, 0.3, 15)
bandwidths = []
for q in quantiles:
    b = estimate_bandwidth(df_s_pca, quantile=q)
    bandwidths.append(b)
```

For each computed bandwidth value, Mean Shift model is trained. A number of clusters and performance metrics will be found for each model instance:

```
In [63]: ms_lst = []

for b in bandwidths:
    ms = MeanShift(bandwidth=b)
    ms.fit(df_s_pca)

    try:
        sil_score = silhouette_score(df_s_pca, ms.labels_, metric='euclidean')
        ch_score = calinski_harabasz_score(df_s_pca, ms.labels_)
    except:
        sil_score = 0
        ch_score = 0
    categs = pd.Series(ms.labels_).nunique()
    ms_lst.append([b, categs, adjusted_rand_score(kingdoms, ms.labels_), sil_score, ch_score])


ms_df = pd.DataFrame(ms_lst, columns=['bandwidth', 'clust_no', 'adj_rand_score', 'sil_score', 'ch_score'])
```

```
In [64]: ms_df["quantile"] = quantiles
```

Again, different bandwidth values optimize different metrics. Adjusted Rand Score is the highest for the model with 92 clusters (0.14) - this model would be hard to use and interpret, so we will pay more attention to other metrics. Silhouette Coefficient is the highest for models with 2 clusters (0.25), while Calinski-Harabasz score is the highest for a 7-cluster model (686).

```
In [66]: ms_df
```

Out[66]:

	bandwidth	clust_no	adj_rand_score	sil_score	ch_score	quantile
0	4.038823	92	0.142449	0.004683	196.385013	0.02
1	4.576521	38	0.095270	0.014293	265.823592	0.04
2	4.977182	19	0.080759	0.047918	433.492867	0.06
3	5.285271	14	0.066663	0.068446	546.211831	0.08
4	5.548612	7	0.014155	0.093321	686.239630	0.10
5	5.785990	7	0.014175	0.094213	684.974095	0.12
6	6.004099	5	0.000849	0.045111	180.613098	0.14
7	6.203698	5	0.000822	0.043329	175.593573	0.16
8	6.388517	3	-0.002696	0.126270	200.481904	0.18
9	6.562058	2	0.000438	0.251856	25.768293	0.20
10	6.727673	2	0.000438	0.251856	25.768293	0.22
11	6.884945	1	0.000000	0.000000	0.000000	0.24
12	7.037239	1	0.000000	0.000000	0.000000	0.26
13	7.185963	1	0.000000	0.000000	0.000000	0.28
14	7.330660	1	0.000000	0.000000	0.000000	0.30

```
In [67]: max_rand_ms = ms_df[ms_df.adj_rand_score == np.max(ms_df.adj_rand_score)]
```

```
In [68]: max_sil_ms = ms_df[ms_df.sil_score == np.max(ms_df.sil_score)]
```

```
In [69]: max_ch_ms = ms_df[ms_df.ch_score == np.max(ms_df.ch_score)]
```

```
In [70]: max_ms = pd.concat([max_rand_ms, max_sil_ms, max_ch_ms])
```

```
In [71]: max_ms
```

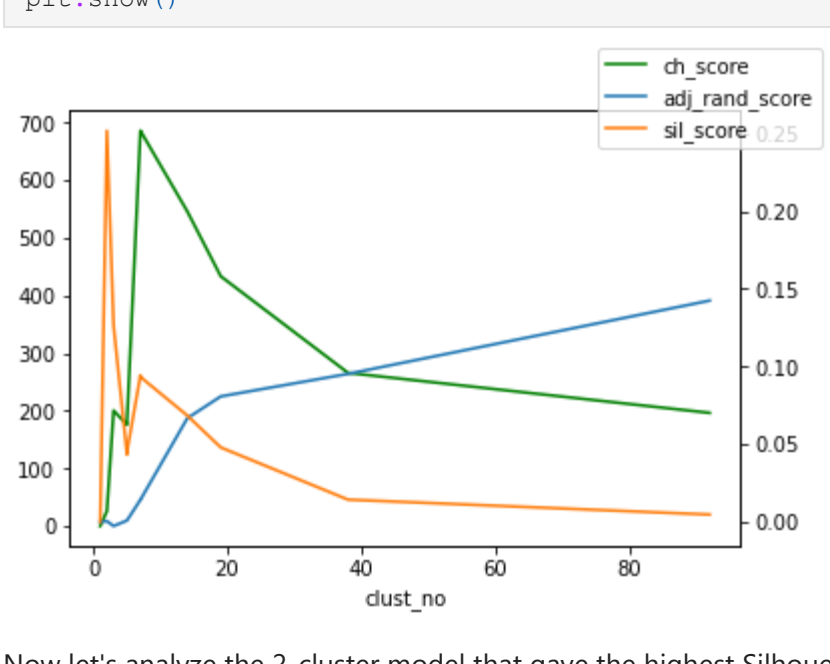
Out[71]:

	bandwidth	clust_no	adj_rand_score	sil_score	ch_score	quantile
0	4.038823	92	0.142449	0.004683	196.385013	0.02
9	6.562058	2	0.000438	0.251856	25.768293	0.20
10	6.727673	2	0.000438	0.251856	25.768293	0.22
4	5.548612	7	0.014155	0.093321	686.239630	0.10

We selected 2-cluster models for the previous algorithms, so we will choose it for Mean Shift as well for comparability.

The plot below shows that while Adjusted Rand Score increases with the number of clusters (which shows improvement), Calinski-Harabasz and Silhouette scores decrease (a sign of worse performance) after the peak cluster number:

```
In [112...: ax = ms_df.plot(x="clust_no", y="ch_score", legend=False, color="g")
ax2 = ax.twinx()
ms_df.plot(x="clust_no", y=["adj_rand_score", "sil_score"], ax=ax2, legend=False)
ax.figure.legend()
plt.show()
```



Now let's analyze the 2-cluster model that gave the highest Silhouette Coefficient:

```
In [98]: ms_02 = MeanShift(bandwidth=bandwidths[9], bin_seeding=False)
ms_02 = ms_02.fit(df_s_pca)
```

The vast majority of samples fall under cluster 0, only 25 of them are clustered separately. Both clusters contain organisms from all kingdoms:

```
In [99]: pd.DataFrame(ms_02.labels_).value_counts()
```

```
Out[99]: 0    9240
1      25
dtype: int64
```

```
In [101...: pd.crosstab(kingdoms, ms_02.labels_)
```

Out[101...:

col_0	0	1
Kingdom		
anm	1625	5
mon	3051	10
pln	1515	8
vrl	3049	2

Model Selection

In the analysis above, we tried three different clustering algorithms - k-means, Agglomerative Clustering and Mean Shift. Results for different parameters were compared for each algorithm and the best one (or their combination) selected.

For k-means, 2-cluster algorithm was the best selection, with Silhouette Coefficient of 0.32, Calinski-Harabasz score of 5891 and Adjusted Rand Score of 0.08.

For Agglomerative Clustering, 2-cluster ward linkage model was the best, with Silhouette Coefficient of 0.25, Calinski-Harabasz score of 4509 and Adjusted Rand Score of 0.05.

For Mean Shift, 2-cluster model was again selected. Its Silhouette Coefficient is 0.25, Calinski-Harabasz score - around 26, and Adjusted Rand Score - close to 0.

Among those three selections, k-means 2-cluster model provides the best results and is thus selected as the final one.

Key Findings And Insights

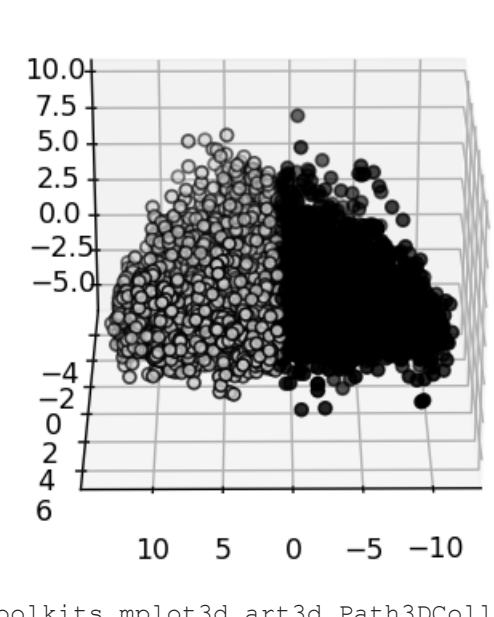
As discussed above, k-means algorithm is the most suitable for this analysis.

However, its efficiency is still low. Good values for Silhouette and Adjusted Rand scores would be close to 1. Low values demonstrate that the selected algorithm does a relatively poor job in finding clusters.

The plot below shows that there is no lower-density area that would separate the clusters obtained with k-means model:

```
In [75]: le = LabelEncoder()
labels_le = le.fit_transform(km_2.labels_)
```

```
In [77]: %matplotlib notebook
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(1, figsize=(4, 3))
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)
PCAmod = PCA(n_components=3)
X = PCAmod.fit_transform(df_s)
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=labels_le, cmap=plt.cm.nipy_spectral,
           edgecolor='k')
```




```
Out[77]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x20a81fc5c40>
```

This is in line with our previous observation of overlapping organism kingdoms.

Although clustering power is low, our clusters still make some difference between organism kingdoms. For example, k-means clusters have a very different number of viruses (2601 vs 450), while the counts for other kingdoms are very similar. A similar pattern can be observed for the Agglomerative Clustering model, where one cluster contains much more virus samples (2029 vs 1022), while the other cluster of the two has more animals (1185 vs 445) and moneros (1900 vs 1161).

Another similarity between the results for those two algorithms is a similar cluster size. k-means algorithm split the dataset into clusters of 5742 and 3523, while Agglomerative Clustering provides counts of 4870 and 4395. Mean Shift clusters almost all samples into one cluster, keeping only 25 entries separately (so it has done the worst job in clustering).

Suggestions

Relatively poor results of our clustering models might encourage us to say that codon information is not suitable for organism clustering. However, it is worth considering some possible model improvements, before making the final conclusion.

Firstly, we used only genomic DNA data, skipping other types. The analysis might be repeated with DNA data of all types. Also, it might be worth adding some other features, that might cluster organisms well together with codon frequencies.

Ground truth labels (in our case - kingdoms) is a result of a somewhat arbitrary choice. They were remade to keep taxonomic consistency, but we might have used the initial labels. After the initial analysis, we might regroup the samples into e.g. viruses and all the rest.

Also, we selected 12 PCA components (given 64 codon features). Changing the number of components might yield different results. Reducing the component number will make us lose some information, and using more components will make more susceptible to "the curse of dimensionality", so experimenting with several numbers might be useful.

Furthermore, using other clustering algorithms is also worth trying, depending on their properties. For example, Gaussian mixture models are not scalable, but given the size of our dataset (about 10000 entries - not too large), might be interesting to experiment with.