Maciej Zagórski
Computer Science, 3rd year

PROJECT REPORT

Assessing the operation and verifying the accuracy of selected clustering algorithms

1. Object of the project

The object of the project was to evaluate the performance of the following learned clustering algorithms:

- centroid algorithm (k-means),
- the density-based spatial clustering of applications with noise (DBSCAN) algorithm.

Additionally, a basic implementation of the hierarchical clustering algorithm (agglomerative clustering) was provided.

Furthermore, the object of the project was to verify the accuracy of the algorithms in question (results of their application).

The above was done on the basis of the sample datasets "iris2D.csv" and "irisORG.csv", both of which contain data on flowers belonging to the genus iris. The application of grouping algorithms in this case is aimed at combining flowers into groups – their division into species, i.e. assigning individual flowers to successively extracted groups (corresponding to flower species of the iris genus).

2. Project implementation

2.1. Analysis and preparation of datasets

Each of the aforementioned datasets – "iris2D.csv" and "irisORG.csv" – contains information on 150 flowers belonging to the genus iris. The "irisORG.csv" dataset contains (raw) information about the parameters of each flower: sepal length and width and petal length and width (the dataset does not contain information on the units in which these values are expressed).

On the other hand, the "iris2D.csv" dataset also contains the above data (for each of the 150 flowers), but in the form compressed to two dimensions. This form of the data in question was obtained by first standardising the data (*StandardScaler* function from the *sklearn.preprocessing* module, from the *scikit-learn* library) and then compressing it, using *principal component analysis,* PCA (*PCA* function from the *sklearn. decomposition* module, from the aforementioned library).

```
stdsc = StandardScaler()
mms = MinMaxScaler()
pca = PCA(n_components=2)

data_standardized_uncompressed = stdsc.fit_transform(data_uncompressed.iloc[:, :-1])
data_standardized_compressed = pca.fit_transform(data_standardized_uncompressed)

data_standardized_compressed = pd.DataFrame(data_standardized_compressed,
                                            columns=['"PC1"', '"PC2"'])

print(data_standardized_compressed)
```

```
     sepal.length  sepal.width  petal.length  petal.width     variety
0             5.1          3.5           1.4          0.2      Setosa
1             4.9          3.0           1.4          0.2      Setosa
2             4.7          3.2           1.3          0.2      Setosa
3             4.6          3.1           1.5          0.2      Setosa
4             5.0          3.6           1.4          0.2      Setosa
..            ...          ...           ...          ...         ...
145           6.7          3.0           5.2          2.3   Virginica
146           6.3          2.5           5.0          1.9   Virginica
147           6.5          3.0           5.2          2.0   Virginica
148           6.2          3.4           5.4          2.3   Virginica
149           5.9          3.0           5.1          1.8   Virginica

[150 rows x 5 columns]

        "PC1"     "PC2"
0   -2.264703  0.480027
1   -2.080961 -0.674134
2   -2.364229 -0.341908
3   -2.299384 -0.597395
4   -2.389842  0.646835
..        ...       ...
145  1.870503  0.386966
146  1.564580 -0.896687
147  1.521170  0.269069
148  1.372788  1.011254
149  0.960656 -0.024332

[150 rows x 2 columns]
```

The "irisORG.csv" dataset furthermore contains in the variety column information on to which species of iris the specified flower (with the parameters specified in the dataset) belongs. This information can take one of three values: *Setosa*, *Versicolor* or *Virginica* (the distribution of species is even, i.e. 50 samples belong to each species). This means that each of the flowers whose data is included in the dataset belongs to one of these three iris species. The column with this information from the "irisORG.csv" collection will be used to verify the correctness of the clustering algorithms within the project (results of their application). For this reason, said column has been extracted as a subset of the data, to be used at the later stage of the project (see sec. **Error! Reference source not found.**).

```
data_iris_results = pd.read_csv('../data/report_2_clustering_algorithms_iris_org.csv')

y_true = data_iris_results.iloc[:, -1]
```

The "iris2D.csv" dataset, which will be used to test how the clustering algorithms work, contains ordinal data – in the form of a first, unnamed column which contains the ordinal numbers of individual records. It has been removed in the first instance to ensure that such data does not inappropriately affect the effects of the application of the algorithms.

```
data_iris = pd.read_csv('../data/report_2_clustering_algorithms_iris_2d.csv')

data_iris.drop([data_iris.columns[0]], axis=1, inplace=True)
```

The data contained in the "irisORG.csv" dataset are correct in the sense that for none of the flowers the length or width of the sepal or petal is of a zero value (which would mean that no measurement has been made in this respect). As already indicated, the "iris2D.csv" dataset is the result of appropriate compression of the data in the former dataset. Also, the variety column in the "irisORG.csv" dataset, which contains

information on the species of each flower, is not empty in any case, i.e. each flower is assigned to one of the three iris species mentioned earlier.

2.2. Selection and presentation of datasets

As already indicated, in order to carry out the project, firstly, the "iris2D.csv" dataset was imported into the programme and the column containing the ordinal numbers of the individual records was removed (using the *pandas* library for doing this).

To assess the performance of the selected clustering algorithms, the imported data was copied and prepared in three forms:

- "raw data", i.e. the data as it is presented in the "iris2D.csv" dataset,
- normalised data, i.e. the above data, which were normalised using the *MinMaxScaler* function from the *sklearn.preprocessing* module of the *scikit-learn* library,
- standardised data, i.e. data from the "iris2D.csv" dataset, which were standardised by reusing the *StandardScaler* function from the *sklearn.preprocessing* module.

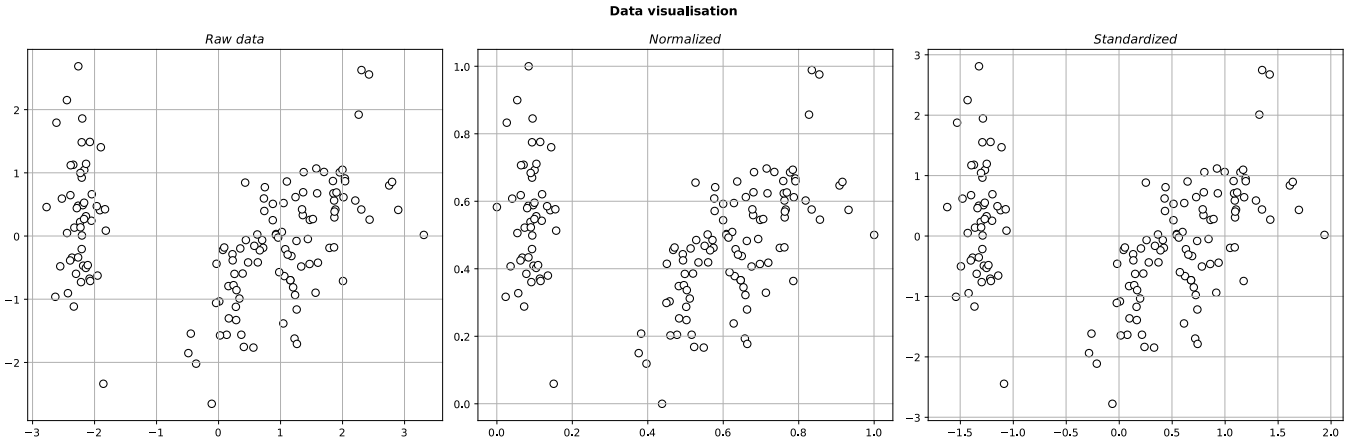
A dictionary named *X_data_sets* was then created from the above datasets, also containing information on the name (type) of each dataset.

```
stdsc = StandardScaler()
mms = MinMaxScaler()
# [...]

X = np.array(data_iris)
X_norm = mms.fit_transform(data_iris)
X_stand = stdsc.fit_transform(data_iris)

X_data_sets = {
    "Raw data": X,
    "Normalized": X_norm,
    "Standardized": X_stand,
}
```

In the next step, the above data were visualised as a scatter plot – using the *matplotlib.pyplot* module from the *matplotlib* library and the defined *plot_data* and *plot_data_loop* functions. Each point in the plot is a separate flower (iris – a record, row, sample in the dataset).

```
def plot_data(data_set, axis, data_set_name):
    axis.scatter(data_set[:, 0], data_set[:, 1], c='white', marker='o',
                 edgecolor='black', s=50)
    axis.set_title(data_set_name, style="italic")
    axis.grid()


def plot_data_loop(data_sets, plot_title):
    f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))
    axes = [ax1, ax2, ax3]
    i = 0

    for data_set_name, data_set in data_sets.items():
        plot_data(data_set, axes[i], data_set_name)
        i += 1
```

```
    f.suptitle(plot_title, weight='bold')
    plt.tight_layout()
    # [...]


plot_data_loop(X_data_sets, "Data visualisation")
# [...]

plt.show()
```



2.3. Application of clustering algorithms

2.3.1. General comments

The clustering algorithms indicated in sec. **Error! Reference source not found.** were applied to the above-mentioned datasets – each algorithm, with different parameters, separately for the "raw", normalised and standardised data. The results of their application – clustering of records from the dataset – are presented in the form of scatter plots, separately for each algorithm (with its different parameters) and the type of dataset. To this end, arrays were defined with the names of the colours and markers used in the plots (which will be plotted using the *matplotlib.pyplot* module from the *matplotlib* library) and the defined *plot_fit_data* and *plot_fit_data_loop* functions.

```
colors = ['red', 'green', 'orange', 'blue', 'yellow', 'indigo', 'violet', 'cyan',
          'sienna', 'teal', 'purple', 'olive', 'lime', 'crimson', 'aqua', 'pink']
markers = ['^', 's', 'p', '>', 'o', 'h', 'v', 'd', '*', '<', '8', 'P', 'H', 'X', 'D']


def plot_fit_data(data_set, axis, data_set_name, algorithm, results):
    if algorithm == 'km':
        y = km.fit_predict(data_set)
    elif algorithm == 'db':
        y = db.fit_predict(data_set)
    else:
        y = ac.fit_predict(data_set)

    for i in range(min(y), max(y) + 1):
        if i == -1:
            lb = "Noise"
        else:
            lb = f'Cluster {i + 1}'

        axis.scatter(data_set[y == i, 0],
                     data_set[y == i, 1],
```

```
                    s=50, c=colors[i],
                    marker=markers[i], edgecolor='black',
                    label=lb)

    if algorithm == 'km':
        axis.scatter(km.cluster_centers_[:, 0],
                    km.cluster_centers_[:, 1],
                    s=250, marker='*',
                    c='grey', edgecolor='black',
                    label='Centroids')

    axis.set_title(data_set_name, style="italic")
    axis.legend(scatterpoints=1)
    axis.grid()

    results.append(y)


def plot_fit_data_loop(data_sets, plot_title, algorithm):
    f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(18, 6))
    axes = [ax1, ax2, ax3]
    i = 0
    results = []

    for data_set_name, data_set in data_sets.items():
        plot_fit_data(data_set, axes[i], data_set_name, algorithm, results)
        i += 1

    f.suptitle(plot_title, weight='bold')
    plt.tight_layout()
    # [...]

    return results
```

2.3.2. Centroid algorithm (k-means)

The k-means algorithm was applied in the programme using the *KMeans* function from the *sklearn.cluster* module (from the *scikit-learn* library). The value of the parameter k (*n_clusters*) – the number of clusters, groups, into which the records in the dataset will be divided – was set as 3 – given that the flowers included in the dataset can belong to one of the three iris species (*Setosa*, *Versicolor* or *Virginica*).

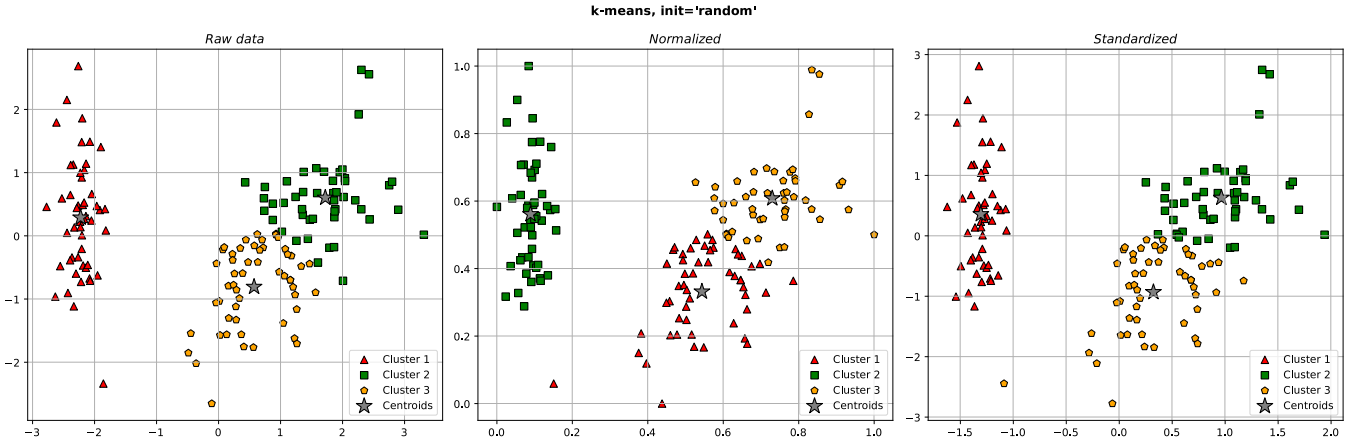
Moreover, the *init* parameter – which determines how the initial centroids are selected – was first specified as *random* – a random selection of initial centroids – and then as *k-means++* – the initial centroids are spaced as far apart as possible.

The other parameters of the algorithm – *n_init*, *max_iter*, *tol*, *random_state* – were specified in default values, according to the relevant *scikit-learn* library documentation[1].

- Application of the k-means algorithm with the *init* parameter specified as *random*:

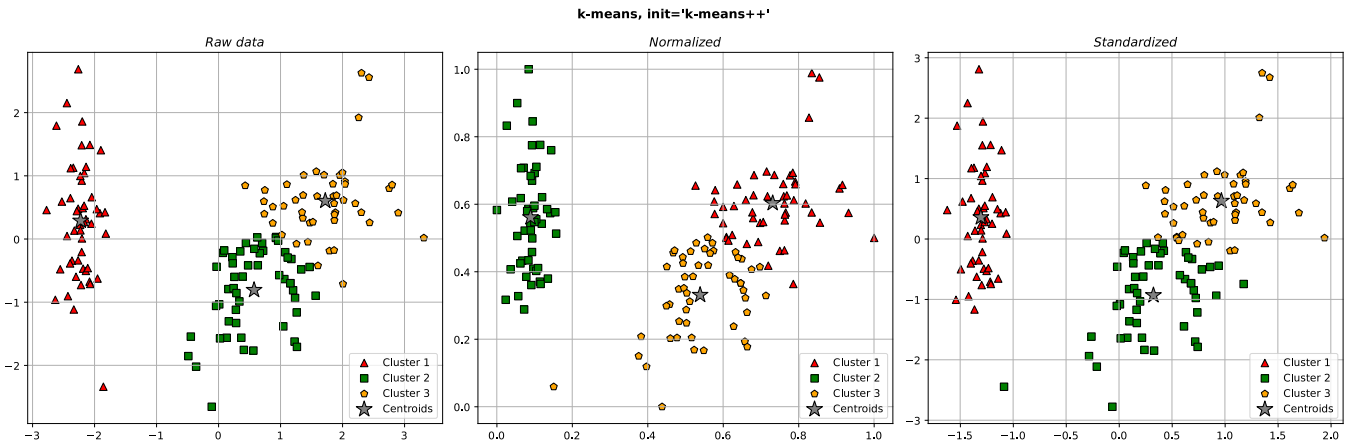
```
km = KMeans(n_clusters=3, init='random', n_init=10, max_iter=300, tol=1e-04,
            random_state=0)
km_results = plot_fit_data_loop(X_data_sets, "k-means, init=random", "km")
# [...]

plt.show()
```

[1] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html [accessed: 06.05.2023].



- Application of the k-means algorithm with the *init* parameter specified as *k-means++*:

```
km.set_params(init='k-means++')
km_pp_results = plot_fit_data_loop(X_data_sets, "k-means, init=k-means++", "km")
# [...]

plt.show()
```



2.3.3. Density-based spatial clustering of applications with noise (DBSCAN) algorithm

In the DBSCAN algorithm, the clustering (cluster analysis) of points (samples, records) is carried out based on the density areas of these points – on the basis of this density, each point is assigned a specific cluster label (it is assigned to a specific group; in the implemented project: to a specific iris species).

The density in this case is understood as the number of points within a certain radius ε (from the selected point). If, within the specified radius from the selected point, there is also a specified minimum number of other points, such a selected point is considered as a core point. If, within the specified radius from the selected point, there are smaller number of points than the specified minimum number of other points, but the point itself is within the radios of the core point, it is referred to as a border point. If, on the other hand, the point is neither a core point nor a border point, it is considered to be noise (a noise point).

Based on such classification of the points, the DBSCAN algorithm extracts a cluster (creates a cluster) for each core point (or group of connected core points, i.e. located in relation to each other within a radius ε). To the clusters formed in this way, boundary points are appropriately assigned – boundary relative to the core points located in these clusters. The remaining points, on the other hand, are treated as noise (they are not assigned to any of the clusters).

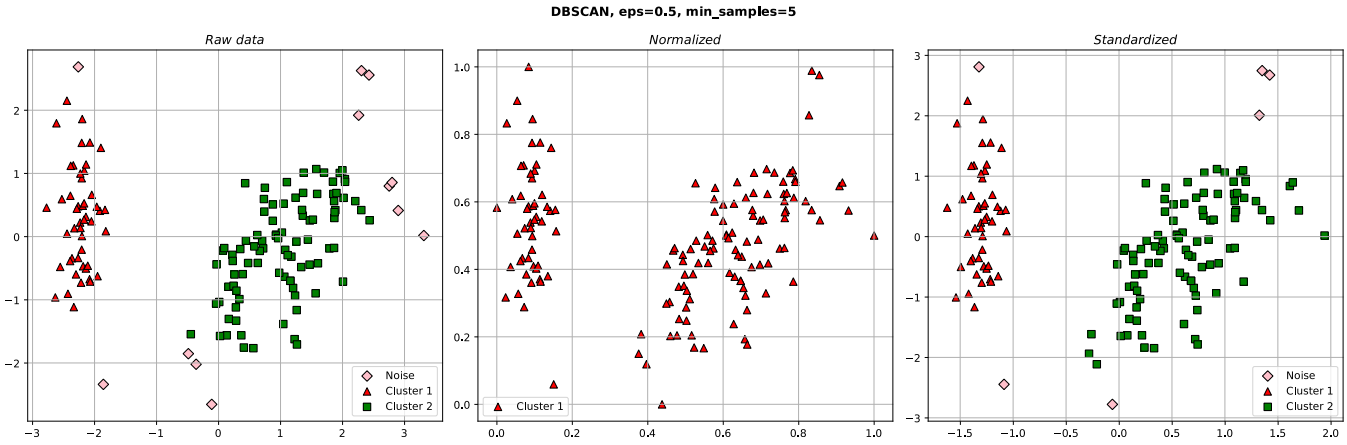
In the case of the DBSCAN algorithm, there are no assumptions regarding the sphericity of the clusters (as is the case of, for example, the k-means algorithm) – the clusters extracted in this algorithm, visualised using a two-dimensional scatter plot, can take on shapes other than the spherical or elliptical ones.

The DBSCAN algorithm was applied in the programme using the *DBSCAN* function, also derived from the *sklearn.cluster* module (from the *scikit-learn* library). In doing so, it was necessary to specify the length of the radius ε – parameter *eps* – and the minimum number of points based on which a core point is extracted – parameter *min_samples*. Moreover, the (default) Euclidean metric (*metric='euclidean'*) was used to calculate the distance between points.

In the case of the DBSCAN algorithm, it is not predetermined (as a parameter for its application) how many groups the records in the dataset are to be divided into – this is determined automatically, as a part of the algorithm's operation – based on the density of the points (samples).

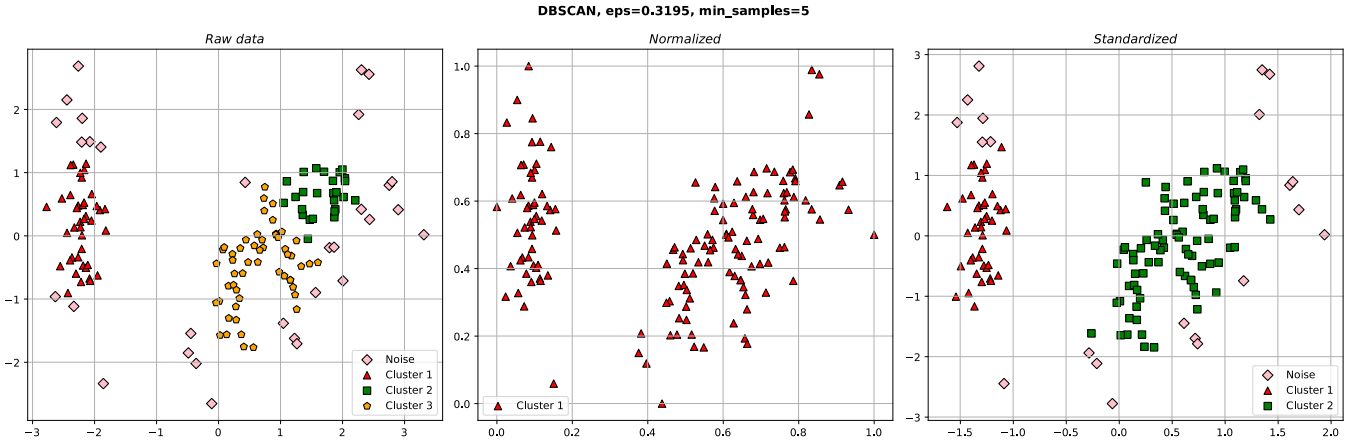
- Application of the DBSCAN algorithm with parameter *eps* and *min_samples* with default values, i.e. (respectively) 0.5 and 5:

```
db = DBSCAN(eps=0.5, min_samples=5, metric='euclidean')
db_results = plot_fit_data_loop(X_data_sets, "DBSCAN, eps=0.5, min_samples=5", "db")
# [...]

plt.show()
```



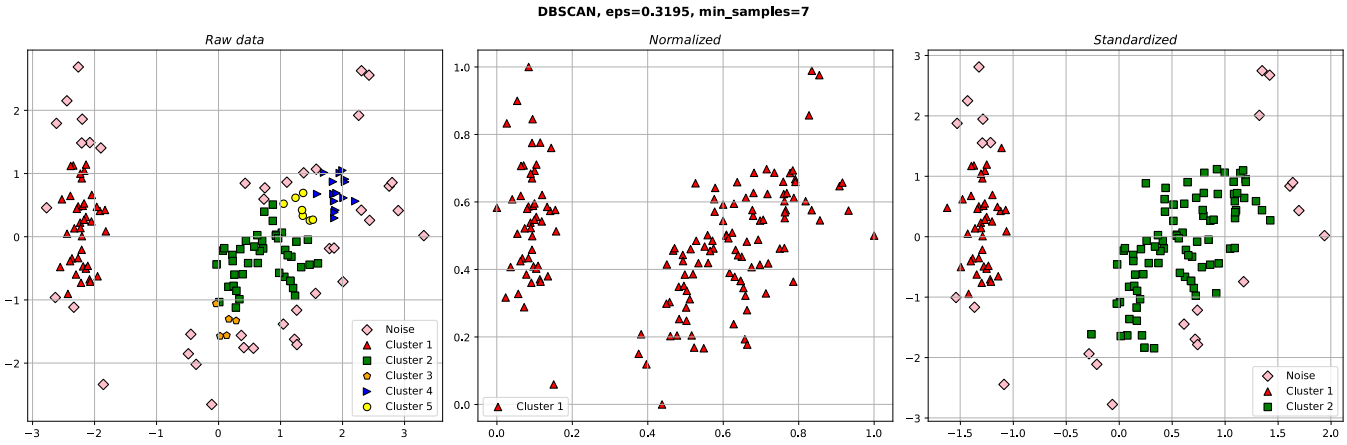
- Application of the DBSCAN algorithm with parameter eps of 0.3195 and *min_samples* equal to 5:

```
db.set_params(eps=0.3195)
db_eps_results = plot_fit_data_loop(X_data_sets, "DBSCAN, eps=0.3195, min_samples=5",
                                    "db")
# [...]

plt.show()
```



- Application of the DBSCAN algorithm with parameter eps of 0.3195 and min_samples equal to 7:

```
db.set_params(eps=0.3195, min_samples=7)
db_eps_min_samp_results = plot_fit_data_loop(X_data_sets, "DBSCAN, eps=0.3195, "
                                                          "min_samples=7", "db")
# [...]

plt.show()
```

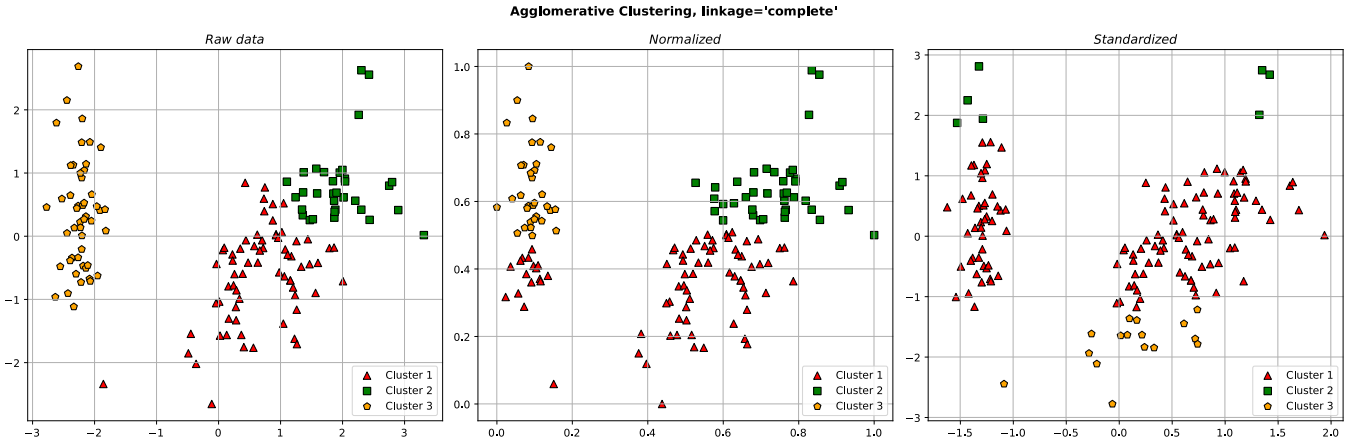


2.3.1. Hierarchical algorithm – agglomerative clustering

Additionally, the agglomerative clustering algorithm was implemented in the project, using the *AgglomerativeClustering* function, also from the *sklearn.cluster* module (from the *scikit-learn* library). As with the k-means algorithm, the value of the parameter *n_cluster* – the number of clusters to be extracted – was defined as 3. The Euclidean metric (*metric='euclidean'*) was used to measure the distance between points, and the full linkage method (*linkage='complete'*) was chosen as the method for agglomerating the clusters.

```
ac = AgglomerativeClustering(n_clusters=3, metric='euclidean', linkage='complete')
ac_results = plot_fit_data_loop(X_data_sets, "Agglomerative Clustering, "
                                             "linkage='complete'", "ac")
```

```
plt.show()
```



2.4. Verifying the accuracy of selected clustering algorithms

2.4.1. General comments

Verification of the accuracy of the clustering algorithms (results of their application) can be done from two perspectives:

- an internal one – internal evaluation – where the verification of the clustering made by the algorithm is done on the basis of the clustered data itself (without the use of "external" data),
- an external one – external evaluation – in which the clustering made is assessed on the basis of external data, e.g. information on the clusters to which the individual samples should have been assigned.

The project verified the accuracy of the k-means, DBSCAN and agglomerative clustering algorithms from the external perspective, using the "irisORG.csv" dataset – the information contained in this dataset (in the *variety* column) about to which species of iris a particular flower belongs (see sec. **Error! Reference source not found.**). The verification performed was therefore the external evaluation.

Based on the data in question (and the clustering made), the following were assessed:

- "Purity" of the clusters resulting from the performed clustering – i.e. an assessment of the extent to which the extracted clusters contain samples belonging to a single group.
- Consistency of the clusters with the true classes (*rand index*) – i.e. an assessment of the extent to which the clusters extracted by the algorithms are similar to the benchmark classification.

2.4.2. Calculation of the purity index

In order to calculate the purity index for each of the extracted clusters (as a result of applying the clustering algorithm), the highest number of points belonging to one group should be counted (based on external data) – the number of points belonging to the group most frequently occurring in a given cluster. Such values should then be summed for all clusters and divided by the total number of samples.

The above was done within the project by using the *contingency_matrix* function from the *sklearn.metrics.cluster* module (from the *scikit-learn* library). The indicated function accepts as parameters external

data (indicating the correct clustering of samples) and data (clustering) obtained as a result of the application of the clustering algorithm. Information about how many samples from the specified ("true") groups (indicated in the rows) fall into the respective clusters (listed in the columns) is obtained as a result of its application.

```
matrix = contingency_matrix(labels_true, labels_pred)
print(matrix)
```

```
[[50  0  0]
 [ 0 11 39]
 [ 0 36 14]]
```

The above matrix shows that: the first extracted cluster contains 50 samples belonging to one group; the second extracted cluster contains 11 samples belonging to the second group and 36 samples belonging to the third group; the third extracted cluster contains 39 samples belonging to the second group and 14 samples belonging to the third group (total number of samples: 150).

Then, using the defined *purity_score* function, the number of points occurring most frequently in each cluster (in the example above: 50 + 36 + 39) was summed and divided by the total number of samples (150).

```
def purity_score(labels_true, labels_pred):
    matrix = contingency_matrix(labels_true, labels_pred)
    return np.sum(np.amax(matrix, axis=0)) / np.sum(matrix)
```

2.4.3. Calculation of the consistency of the clusters with the true classes (*rand index*)

The assessment of the consistency of the clusters with the true classes is made by counting the *true positive* and *true negative* cases and the ratio of their number to all cases combined (i.e. also *false positive* and *false negative* in addition to those mentioned).

Individual cases are assessed based on pairs of points (samples): if two samples belonging to one group according to external (benchmark) data have been assigned to one cluster as a result of the application of the algorithm, this is a *true positive* case; similarly, if samples belonging to different groups have been assigned by the algorithm to different clusters – this is a *true negative* case. It does not matter to which clusters individual samples are assigned – what is important is the assessment of their clustering in relation to each other (within individual pairs of samples – whether they belong to the same cluster or to the different clusters).

To calculate the *rand index*, the *rand_score* function from the *sklearn.metrics* module (from the *scikit-learn* library) was used. Like *contingency_matrix,* said function accepts as parameters the external data indicating the correct clustering of samples and the (clustering) data obtained by the application of the clustering algorithm.

```
rand_score(labels_true, labels_pred)
```

3. Presentation of results

The defined function *score_loop* was used to calculate the above-described values:

```
def score_loop(data_sets, score_title, labels_true, results):
    i = 0
    for data_set_name in data_sets:
        print(score_title + ": " + data_set_name + ": Purity [%%]: %.2f" %
              (purity_score(labels_true, results[i]) * 100))
        print(score_title + ": " + data_set_name + ": Rand index [%%]: %.2f" %
              (rand_score(labels_true, results[i]) * 100))
        print()
        i += 1


# [...]
score_loop(X_data_sets, "K-means, init='random'", y_true, km_results)
score_loop(X_data_sets, "K-means, init='k-means++'", y_true, km_pp_results)
score_loop(X_data_sets, "DBSCAN, eps=0.5, min_samples=5", y_true, db_results)
score_loop(X_data_sets, "DBSCAN, eps=0.3195, min_samples=5",  y_true, db_eps_results)
score_loop(X_data_sets, "DBSCAN, eps=0.3195, min_samples=7",  y_true,
           db_eps_min_samp_results)
score_loop(X_data_sets, "Agglomerative Clustering, linkage='complete'",  y_true,
           ac_results)
```

The resulting *purity* and *rand index* values (rounded to two decimal places) are shown in the table below – separately for each clustering algorithm (k-means, DBSCAN, both with appropriate parameters*, and agglomerative clustering),* as well as the data types subjected to the algorithms ("raw data", normalised and standardised).

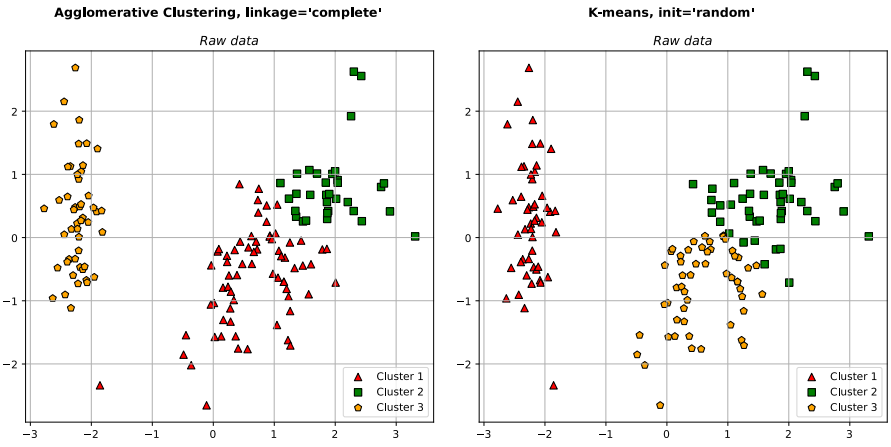
	Algorithm	***Parameters***	***Data***	***Purity [%]***	***Rand index [%]***
1.	*k-means*	*init='random'*	*Raw data*	83.33	83.22
2.			*Normalized*	82.67	82.33
3.			*Standardized*	81.33	81.47
4.		*init='k-means++'*	*Raw data*	83.33	83.22
5.			*Normalized*	83.33	82.79
6.			*Standardized*	81.33	81.47
7.	*DBSCAN*	*eps=0.5, min_samples=5*	*Raw data*	68.00	76.73
8.			*Normalized*	33.33	32.89
9.			*Standardized*	66.67	76.73
10.	*DBSCAN (cd.)*	*eps=0.3195, min_samples=5*	*Raw data*	76.67	76.64
11.			*Normalized*	33.33	32.89
12.			*Standardized*	64.67	74.20
13.		*eps=0.3195, min_samples=7*	*Raw data*	74.00	73.41
14.			*Normalized*	33.33	32.89

	Algorithm	***Parameters***	***Data***	***Purity [%]***	***Rand index [%]***
15.			*Standardized*	64.00	73.34
16.	*Agglomerative clustering*	*linkage='complete'*	*Raw data*	84.00	83.11
17.			*Normalized*	68.67	67.06
18.			*Standardized*	40.67	42.88

4. Comments and conclusions

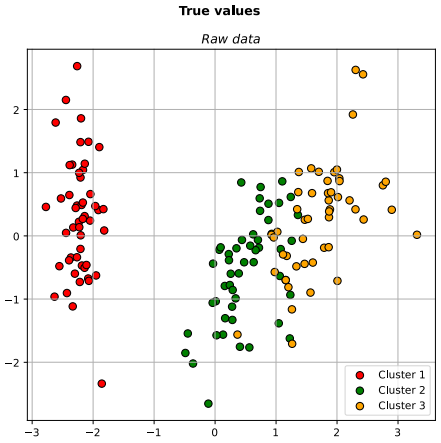
Based on the results obtained, it can be indicated that:

- **The best results were achieved as a result of the application of the k-means algorithm** (both with *init='random'* and *init='k-means++'* parameter) **and the agglomerative clustering algorithm,** in both cases with regard to the "raw data":
  - for the k-means algorithm, the *rand index* of 83.22% (and the *purity* of 83.33%) was achieved,
  - for the agglomerative clustering algorithm, the *purity* of 84% (and the *rand index* of 83.11%) was achieved.
- Based on the scatter plots generated for the application of these algorithms, it can be indicated that the agglomerative clustering algorithm assigned more points (samples) to cluster number 1 (which is "merged" with cluster number 2), while the k-means algorithm divided the samples between these clusters more evenly. This may correspond with the more "pure" application of the agglomerative clustering algorithm, and, at the same time, with the lower *rand index* value (too many samples, which should be in different clusters, being assigned to the same cluster).



- **The worst results (values) were achieved in the case of the application of the DBSCAN algorithm –** using parameters *eps* of 0.3195, and *min_samples* of 5 (with respect to the "raw data"), it was possible to achieve at most the *purity* of 76.67% and the *rand index* of 76.64%. The indicated results may have been affected by the fact that the individual samples are in a high density, there is no clear boundary between two of the clusters (of the three), and the third cluster is much further away from the other two. These circumstances are illustrated in the scatter plot below, drawn up using

information from the *variety* column of the "irisORG.csv" collection – indicating to which species of iris a particular flower belongs.



- The above results were not affected by changes of the parameters of the application of the DBSCAN algorithm – *eps* or *min_samples*. Increasing or decreasing them led either to extracting too few clusters (this was the case with *eps=0.5* and *min_samples=5*) or to extracting too many of them (so with *eps=0.3195* and *min_samples=7*)[2] .
- **The normalisation and standardisation of data from the "iris2D.csv" dataset did not lead to better results** than the direct application of the values from this dataset. On the contrary: the use of normalised or standardised data usually led to worse results than with "raw data" (this was the case of the DBSCAN and agglomerative clustering algorithms) or, at most, to similar results (k-means algorithm). The reason for this may have been that the "raw data" used in the project had already been standardised (prior to compression); subsequent normalisation or (subsequent) standardisation may have degraded the "quality" of the data in the sense that it became more difficult to be interpreted for the clustering algorithms ("overdone").
- Applying the DBSCAN algorithm with the parameters adopted in the project in the case of the normalised data did not allow more than one cluster to be extracted at all. For these data – i.e. data in which the coordinates of each sample range from 0 to a maximum of 1 – the adopted values of the *eps* parameter, although fractional anyway, may have been too large. Only by changing this parameter to a value of, for example, 0.1, a larger number of clusters could be extracted.
- **Assessing the performance of clustering algorithms from the "external" perspective –** i.e. using the external data indicating the correct clustering of samples to which the clustering algorithms have been applied – **can be questionable.** After all, clustering is an unsupervised learning technique – and in a situation where we have information about the correct clustering of specific samples, applying clustering algorithms to them may be pointless. Indeed, it may be questionable to cluster something that has already been clustered (of the clustering of which we already have knowledge)[3] .

[2] As S. Raschka and V. Mirjalili indicate, *"Finding a good combination of MinPts and ε can be problematic if the density differences in the dataset are relatively large"* (*Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2,* 3rd ed., Gliwice 2021, p. 367).
[3] *"Additionally, from a knowledge discovery point of view, the reproduction of known knowledge may not necessarily be the intended result"* ([n.a.], *Cluster analysis – Wikipedia,* https://en.wikipedia.org/wiki/Cluster_analysis#External_evaluation [accessed: 08.05.2023].

- However, the purposefulness of such an action may be sought, for example, in a situation when there is a very large number of samples, and we have information on correct clustering only in relation to a part of them – in such a case, the application of specific algorithms, with specific parameters, may be verified on such a part of the samples. Although the selection of such "verified" samples itself and their representativeness for the whole dataset may also result in further problems.
- Based on the final scatter plot presented above, illustrating the proper clustering of the samples to which the clustering algorithms were applied within the project, it can be seen that **none of the algorithms applied achieved the "right" clustering –** despite the fact that each of them achieved high (above 80%) values of the *purity* and *rand index*, as it may seem. This is particularly evident when splitting two "contacting each other" clusters, between which the boundary runs along the Y-axis (as a result of each algorithm used, the boundary ran diagonally).

5. Sources

[n.a.], *Cluster analysis – Wikipedia,* https://en.wikipedia.org/wiki/Cluster_analysis [accessed: 08.05.2023]

[n.a.], *Matplotlib – Visualization with Python*, https://matplotlib.org/, [accessed: 08.05.2023]

[n.a.], *pandas – Python Data Analysis Library*, https://pandas.pydata.org/, [accessed: 08.05.2023]

[n.a.], *scikit-learn: machine learning in Python – scikit-learn 1.2.2 documentation*, https://scikit-learn.org/stable/, [accessed: 08.05.2023]

S. Raschka, V. Mirjalili, *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2,* 3rd ed., Gliwice 2021.