# Algorithms for Data Science
# Practical Report (Part 1, Clustering)

---

**Mariem Nsiri**
MSc in Applied Data Science and Analytics
Technological University Dublin

**Declaration:** I herby certify that this material, which I now submit for assessment on the programme of study leading to the award of MSc in Computing in Applied Data Science and Analytics, TU Dublin, is entirely my own work except where otherwise stated, and has not been submitted for assessment for an academic purpose at this or any other academic institution other than in partial fullfilment of the requirements of that stated above.

Mariem Nsiri

October 30, 2024

This report is an adaptation of the LaTeX code generated from a local Jupyter Notebook used to prepare the clustering labs practical assessment notes. The clustering dataset is automatically generated, which means there is no specific business context or attribute descriptions, as mentioned in the course documents. The necessary Python libraries and their API (Application Programming Interface) functions are included in the appropriate code snippets presented in the document, rather than including them all at the beginning, ensuring clarity and improving the document's organization.

The report is divided into the following main contents:

- The nature of the generated data attributes is explained, and additional details about key concepts are provided for better understanding.
- Insights on data and initial observations about the clusters are presented through relevant plots.
- The data is prepared for use with the clustering algorithms, with details about the scaling and normalization processes.
- Two algorithms are chosen for clustering the dataset, and the reasons for their selection are justified.
- Analyses are provided on how to determine if the clustering results are correct in the absence of class labels, while following the guidelines outlined in the course material.

# 1 Import, explore and visualize the data to gain insights

The data is first loaded into a `DataFrame` data structure from the `assessment_cluster_dataset.csv` file using the *Pandas* library. It is useful for working with table data like in spreadsheets or databases. It helps to explore, clean, and process the data.

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np, pandas as pd
# Load the data in DataFrame
data = pd.read_csv('data/assessment_cluster_dataset.csv', sep=',')
data.head()
```

```
   att1      att2      att3
0 -3.218423 -0.575000 -3.881158
1  5.801795 -1.340335 -0.201488
2  1.394402 -1.112765 -4.646334
3  4.767334 -2.324507  4.026738
4  4.590624 -1.456617 -1.873809
```

## 1.1 Exploring data information

The output information about the dataset, provided by `data.info()`, includes the number of rows, the number of columns, the names (data attributes) of the columns, types, and how many entries in each column are not missing.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1500 entries, 0 to 1499
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   att1    1500 non-null   float64
 1   att2    1500 non-null   float64
 2   att3    1500 non-null   float64
dtypes: float64(3)
memory usage: 35.3 KB
```

The data imported contains **1500** entries, numbered from **0** to **1499**. There are **3 columns** corresponding to **3 attributes** in total (**att1**, **att2**, and **att3**). Each column has **1500 non-null values** with the type `float64`.

```
data.describe()
```

```
              att1         att2         att3
count  1500.000000  1500.000000  1500.000000
mean      1.590267    -0.031389    -0.420662
std       2.963958     1.750105     3.087797
min      -4.661216    -3.896381    -6.361849
25%      -1.310032    -1.036573    -3.325445
50%       1.726168    -0.360621    -1.101806
75%       4.239675     1.106294     2.619525
max       7.470099     3.763953     6.651459
```

The information displayed by `data.describe()` are the following:

- **count**: the number of non-null entries in a column used for the statistics.
- **mean**: the **average of the values** in a column.
- **std**: the **standard deviation** in a column.
- **min**: the smallest value in a column.
- **25%**: the **first quartile** or **25th percentile** in a column.
- **50%**: the **second quartile** or **50th percentile/median** in a column.
- **75%**: the **third quartile** or **75th percentile** in a column.
- **max**: the largest value in a column.

The **standard deviation** of a column in a dataset measures **how much the values in that column differ from the average (or mean) value**. The standard deviation shows how spread out the values are around the mean, where a **low standard deviation** means the values are **close to the mean with less difference**, and a **high standard deviation** means the values are **more spread out with more differences**. For a dataset column $x$ with values $x_1, x_2, ..., x_n$ and mean $\bar{x}$, the standard deviation $\sigma_x$ is defined by $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$.

A **percentile** measures **how values are spread in a column**. It shows the value below which a certain percentage of entries fall. **Quartiles split the column into four equal parts**. For example, the **second quartile** (also known as the 50th percentile or median) is the **value at which 50% of the data points fall below**. This helps to understand how the data is distributed, showing the lower, middle, and upper parts of the values, as well as whether most scores are close to the mean or vary widely.

### 1.2 Visualizing the data

The data will now be visualized to **help understand its patterns, clusters and trends**. By using **2D and 3D plots**, important insights can be gained about the data. Some of these visualizations are useful to see how the values relate to each other within distinct clusters to the naked eye.

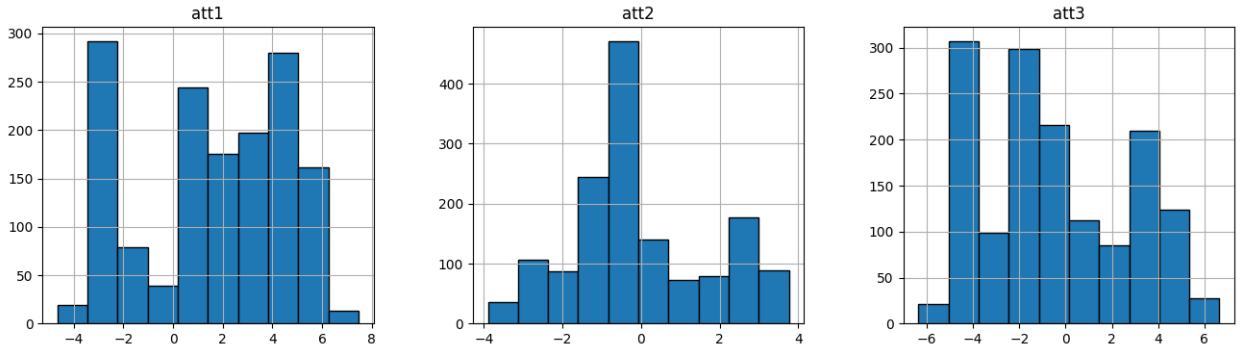


Figure 1: Attribute distribution histograms.

While these **histograms are not effective for identifying clusters**, as they display each attribute's data independently, they **clearly show how the data is spread around the mean value**, helping us understand the overall distribution and variation in the data values for each attribute (see Figure 1).

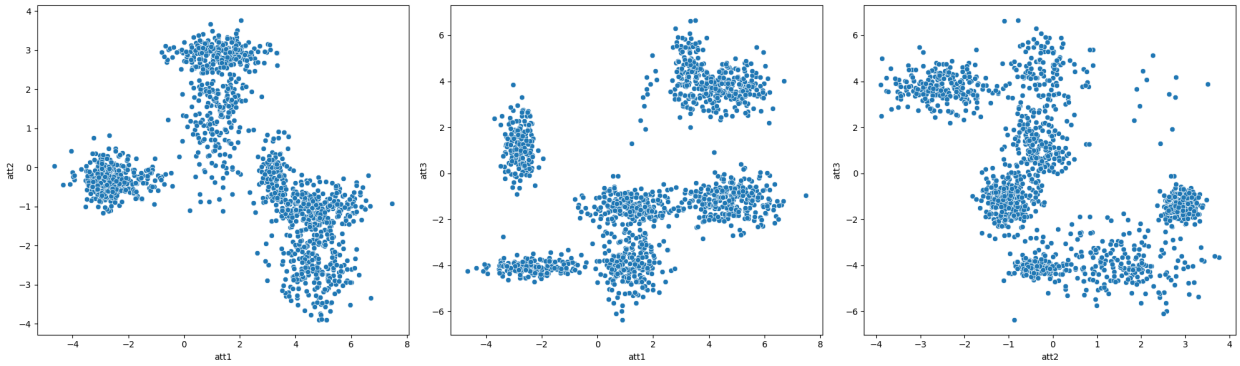


Figure 2: 2D scatter plots for all possible paired combinations of attributes.

```
import matplotlib.pyplot as plt, seaborn as sns
from itertools import combinations
cols = data.columns
indices_combinations = list(combinations(range(len(cols)), 2))
plt.figure(figsize=(20, 6))
for i, pair in enumerate(indices_combinations):
  plt.subplot(1, len(indices_combinations), i + 1) # one row, three columns
  sns.scatterplot(data, x=cols[pair[0]], y=cols[pair[1]])
plt.tight_layout()
plt.show()
```

The 2D scatter plots, shown in Figure 2, offer valuable insights about the number of clusters present in the data. While they help visualize how the data points group together, accurately estimating the exact number of clusters **can still be non trivial**. These plots can **show patterns and relationships between attributes**, making it easier to see how they connect with one another. Overall, these 2D plots are useful for **understanding paired data correlations**, even if they **don't provide an accurate visualization** of the number of clusters.

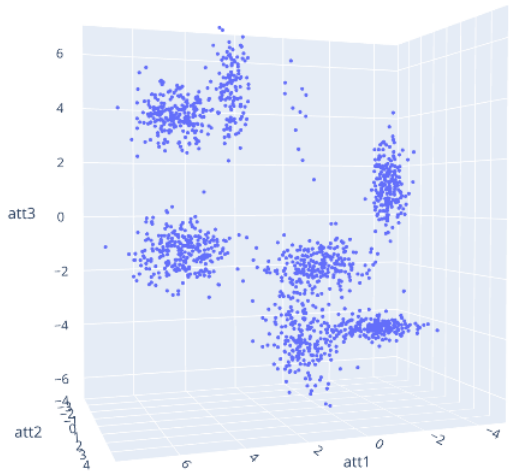


Figure 3: 3D scatter plot (visually **7 clusters are identifiable**).

The 3D scatter plots provide **clear insights into the number of clusters** and how the three attributes are connected with each other. They make it easier to see the relationships and patterns among the data points in three dimensions. These are the main characteristics:

- There are **7 distinct globular clusters** of different sizes.
- Most clusters are **dense in the center** and **more sparse at the edges**.
- **Tens** of scattered dots appear to be **noise** in the data.
- Some clusters are **close to one another**, while others **are well separated**.

## 2 Data preparation for clustering algorithms

Clustering algorithms don't perform well when input numerical attributes have **very different scales**. Two common methods are commonly used for **attributes scaling**: **min-max** and **standardization**.

**Min-max scaling, often called normalization**, is straightforward: each attribute's values are adjusted to range from 0 to 1 by subtracting the minimum and dividing by the range (the difference between the maximum and the minimum values). **Standardization** works differently: it first subtracts the mean from each value (resulting in a zero mean) and then divides by the standard deviation. Unlike min-max scaling, standardization **doesn't limit values to a set range**, making it **less sensitive to outliers** (Géron, 2022).

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# Train the scaler on data
scaler.fit(data)
# Apply to data
data_transformed = scaler.transform(data)
# Create a new DataFrame with the scaled data
data_scaled = pd.DataFrame(data_transformed, columns=data.columns)
data_scaled.head()
```

```
       att1      att2      att3
0 -1.622929 -0.310720 -1.121074
1  1.421388 -0.748173  0.071004
2 -0.066104 -0.618098 -1.368963
3  1.072258 -1.310712  1.440795
4  1.012618 -0.814639 -0.470767
```

# 3 Data clustering

In this section, the selected algorithms, K-Means and DBScan, are presented and applied to the scaled data. The agglomerative hierarchical clustering was discarded, and the reasons will be provided later.

## 3.1 K-Means clustering

K-means is a popular clustering method that starts by selecting $k$ random rows as initial cluster centers. Each object is assigned to the nearest cluster based on distance, and new centers are calculated for each cluster. This process **repeats** until **no objects change clusters** or **the distances to the centers stabilize**. K-means performs effectively when natural **clusters are compact and clearly separated**, as is almost the case here, and it is **efficient for high-dimensional data**. However, it does not work well when the chosen $k$ **does not match the actual number of clusters**. In addition, it is **sensitive to noise and outliers**, and **performs poorly with non-globular or differently-sized clusters**.

For the data studied here, there is **strong confidence in the number (7) of clusters**, and most of them are **fairly elliptically compact** and **well-separated**, except for a few. Therefore, K-Means is deemed a suitable algorithm for this case.

In the following, two initialization techniques for K-Means, namely **`random`** and **`k-means++`**, will be applied.

### 3.1.1 Initialization using `random` and `kmeans++` with the other hyperparameters set to their default values

```
from sklearn.cluster import KMeans
# Create the KMeans clustering model with 7 clusters and random initialization
kmeans_model = KMeans(n_clusters=7, init='random').set_output(transform='pandas')
# or create the KMeans clustering model with 7 clusters and kmeans++ initialization
kmeans_model = KMeans(n_clusters=7, init='k-means++').set_output(transform='pandas')
# Train and transform the scaled data
results_kmeans = kmeans_model.fit_transform(data_scaled)
# Add the new clusters class labels as a new column
kmeans_data = data_scaled.copy()
kmeans_data['cluster'] = kmeans_model.labels_
kmeans_centroids = pd.DataFrame(kmeans_model.cluster_centers_, columns=data_scaled.columns)
kmeans_centroids['cluster'] = ['Centroid 0','Centroid 1','Centroid 2','Centroid 3',
                               'Centroid 4','Centroid 5','Centroid 6']
# Import parallel_coordinates plotting class from pandas.plotting library
from pandas.plotting import parallel_coordinates
# Plot the centroids across all attributes
parallel_coordinates(kmeans_centroids, 'cluster', marker='o')
```

After training the model several times with both initialization techniques, it was observed that all the **centroids are stable, different and do not overlap**. The visualizations of centroids helps to illustrate how each cluster is formed and how they are positioned in relation to the data. The centroids of both the `random` and `kmeans++` initialization techniques are quasi-similar (see Figure 4, bottom).

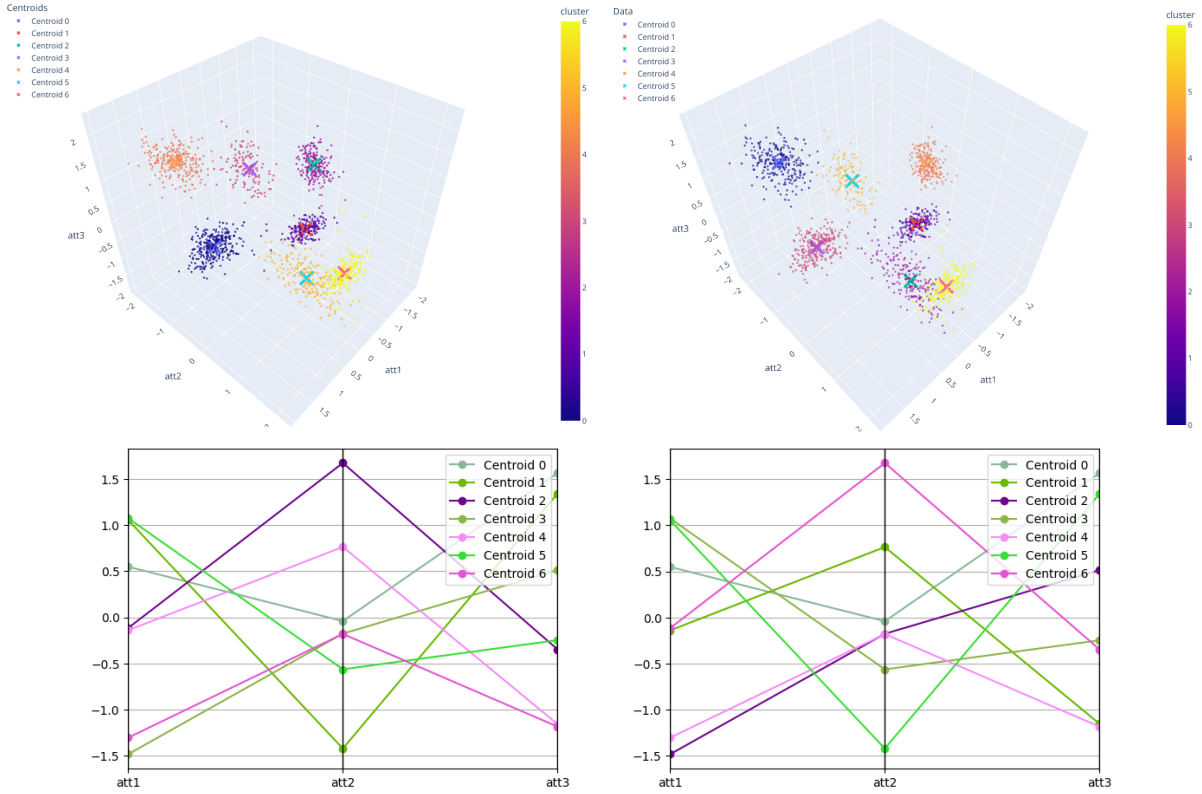


Figure 4: Centroids of the `random` model (bottom left); Centroids of the `kmean++` model (bottom right); 3D scatter plot with centroids [k=7, init=`random`, default for other hyperparameters] (top left); 3D scatter plot with centroids [k=7, init=`k-means++`, default for other hyperparameters] (top right).

According to the 3D plotting, it is clear that the **seven clusters are well identified** and **the centroids are well positioned in the center of the different clusters**. This indicates that the random initialization of the K-Means clustering is **effective and demonstrates a high level of stability** (see Figure 4, top).

### 3.1.2 Searching for the optimal configuration

```
from sklearn.model_selection import GridSearchCV
# Define the range of values to try out for each hyperparameter
param_grid = {
  'n_clusters': range(6, 9),
  'init': ['k-means++', 'random'],
  'n_init': [5, 10, 15],
  'max_iter': [100, 200, 300, 400, 500],
  'random_state': [1, 16, 34, 42, 57]
}
# Create the kmeans model
kmeans = KMeans()
# Use the grid search to try out all possible combinations of
# hyperparameter values from the grid above and fit a new model for each
grid_search = GridSearchCV(kmeans, param_grid)
grid_search.fit(data_scaled)
# output the hyperparameter values of the model that achieved highest performance
grid_search.best_params_
```

```
'init': 'k-means++',
'max_iter': 100,
'n_clusters': 8,
'n_init': 5,
'random_state': 42
```

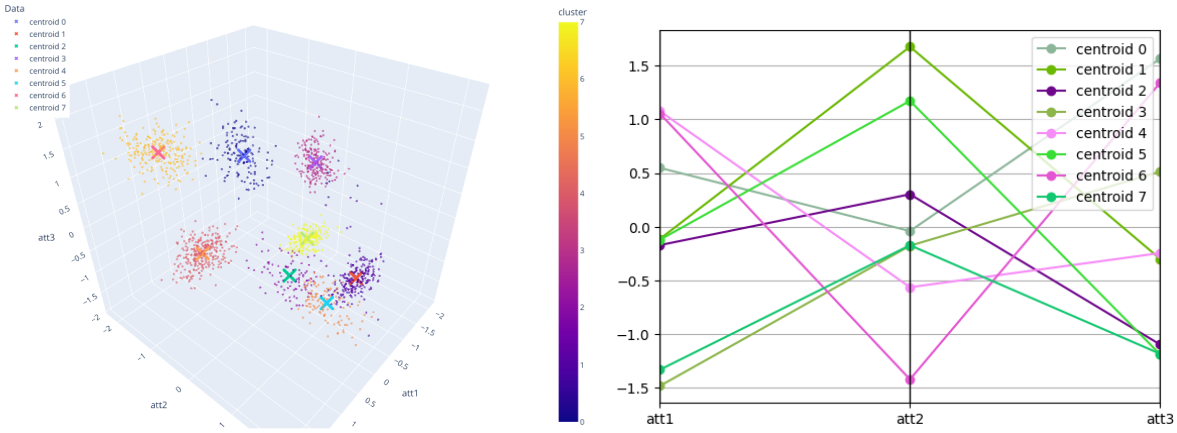


Figure 5: 3D scatter plot with centroid (left); Centroids (right) [optimal `GridSearchCV` hyperparameters].

In Figure 4 (top), the **lower cluster close to the ground plan** (`att1`, `att2`) is spread out elliptically across `att2`. It **splits into two clusters under the optimal configuration** found by `GridSearchCV` (8 clusters) and shown in Figure 5. **The seven clusters visually identified may actually be more accurate**. This split in the optimal configuration could be due to the following weaknesses of K-Means:

1. K-Means works well when clusters are **compact**, **globular**, and **well-separated**. However, it **performs poorly when clusters are not distinct or vary in size**, especially around the sparse (`att1`, `att2`) ground cluster and its neighboring clusters.

2. K-Means is sensitive to **noise and outliers, which are present here**.

Subjective analyses were already provided gradually in the previous content. An **objective evaluation** of K-Means clustering hyperparameter fine-tuning will help confirm our subjective decision regarding the appropriate number of clusters.

### 3.1.3 K-Means clustering objective evaluation

```
# Elbow method
inertias = []
for k in range(2, 11):
   # Only the n_clusters hyperparameter will vary, while the
   # others will remain fixed according to the GridSearch results
   kmeans = KMeans(n_clusters=k, init='k-means++', n_init=5,
                   max_iter=100, random_state=42)
   kmeans.fit(data_scaled)
   inertias.append(kmeans.inertia_)
plt.plot(range(2, 11), inertias, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
```

```
# Davies-Bouldin and Silhouette indexes
from sklearn.metrics import silhouette_score, davies_bouldin_score
db_scores = []
sil_scores = []
for k in range(2, 11):
   kmeans = KMeans(n_clusters=k, init='k-means++', n_init=5,
                   max_iter=100, random_state=42)
   labels = kmeans.fit_predict(data_scaled)
   db_scores.append(davies_bouldin_score(data_scaled, labels))
   sil_scores.append(silhouette_score(data_scaled, labels))
plt.subplots()
plt.plot(range(2, 11), db_scores, marker='o')
plt.plot(range(2, 11), sil_scores, marker='x')
plt.xlabel('Number of clusters')
plt.ylabel('Score')
plt.legend(['Davies-Bouldin', 'Silhouette'])
```

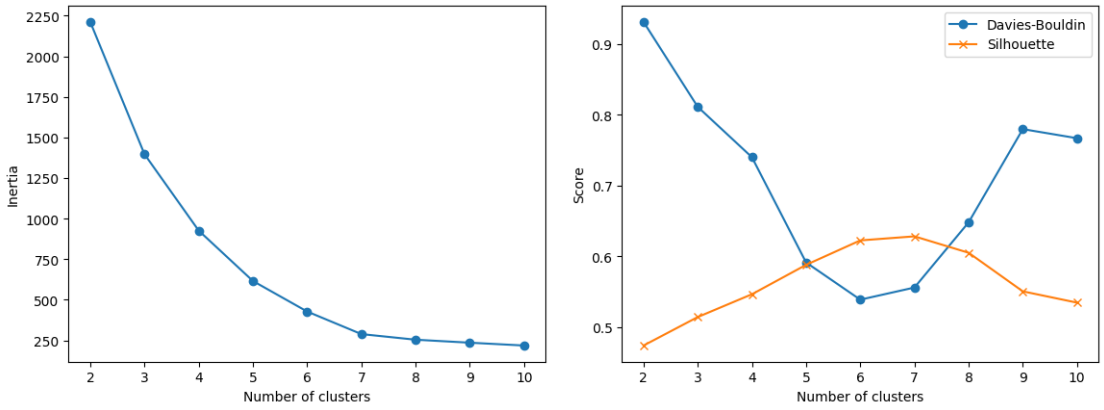


Figure 6: Elbow method output (left); Davies-Bouldin and Silhouette indexes (right).

Given that the Elbow method diagram **doesn't accurately reflect the number of clusters**, as the convergence starts at 7 clusters, it doesn't help to **decide about the right number of clusters** (see Figure 6, left). However, the Davies-Bouldin (DBI) and Silhouette (SI) curves provide us with clear, accurate indications (see Figure 6, right):

- The **higher SI value at 7 indicates the best clustering schema and confirms the subjective observations**.
- The **intersections after the lower DBI values and the higher SI value** occur between 7 and 8 clusters, **confirming the tradeoff between these two options**.

### 3.2 DBScan (density-based) clustering

DBScan was selected as the second algorithm along with K-Means because it has some special advantages. Unlike K-Means, which works best with round clusters of similar sizes, **DBScan can find clusters of different shapes and sizes**. This means it can discover clusters that K-Means might miss. Also, DBScan is **good at ignoring noise points and border points**, so it **works well with data that has outliers**. However, DBScan can have trouble with clusters that have different densities since it uses fixed values for the `eps` and `min_samples` parameters.

Overall, **DBScan's capability to manage complex diverse clusters with noise makes it useful choice** alongside K-Means.

```
from sklearn.cluster import DBScan
dbscan_model = DBSCAN()
dbscan_labels = dbscan_model.fit_predict(data_scaled)
dbscan_data = data_scaled.copy()
dbscan_data['cluster'] = dbscan_model.labels_
dbscan_data.groupby('cluster')['cluster'].value_counts()
```

```
cluster
-1       2
 0    1290
 1     197
 2      11
Name: count, dtype: int64
```

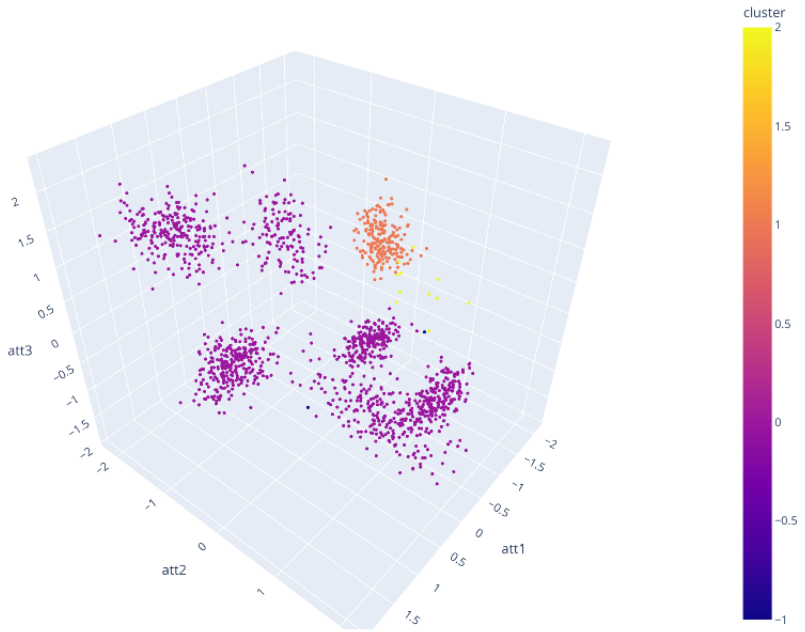


Figure 7: 3D scatter plot with centroid for DBScan (default hyperparameters)

Using the default hyperparameters (`min_samples=5` and `eps=0.5`) **gave poor results**, identifying only 3 clusters and 2 noise points (cluster -1), which **is much less than the expected 7 clusters**. This is mainly because the `min_samples` value is not set correctly; it should be at least the dimensionality of the dataset plus 1 (3 + 1). Additionally, the default `eps=0.5` led to too few noise points, while the actual data has around hundreds of noise points, as can be seen visually. The other hyperparameters are better left at their default values (see Figure 7).

That's why we need to find the correct values for `eps` and `min_samples` to achieve at least 7 clusters, along with an accurate count of noise and border points. To achieve this, we need to create a k-distance plot following the instructions given in the lab sheet.

```
from sklearn.neighbors import NearestNeighbors

def plot_k_distance(k):
   nbrs = NearestNeighbors(n_neighbors=k).fit(data_scaled)
   # Get distances to the k-th nearest neighbors
   distances, indices = nbrs.kneighbors(data_scaled)
   # Sort the distances to the k-th neighbor
   distances = np.sort(distances[:, k - 1])
   plt.figure(figsize=(10, 6))
   plt.plot(distances, marker='o', markersize=2.5)
   plt.ylabel('k-distance')
   plt.xlabel('Points sorted by distance to k-th nearest neighbor')
   plt.title(f'k-Distance Plot for k={k}')
   plt.grid(True)
   plt.show()

k = 10
plot_k_distance(k)
```

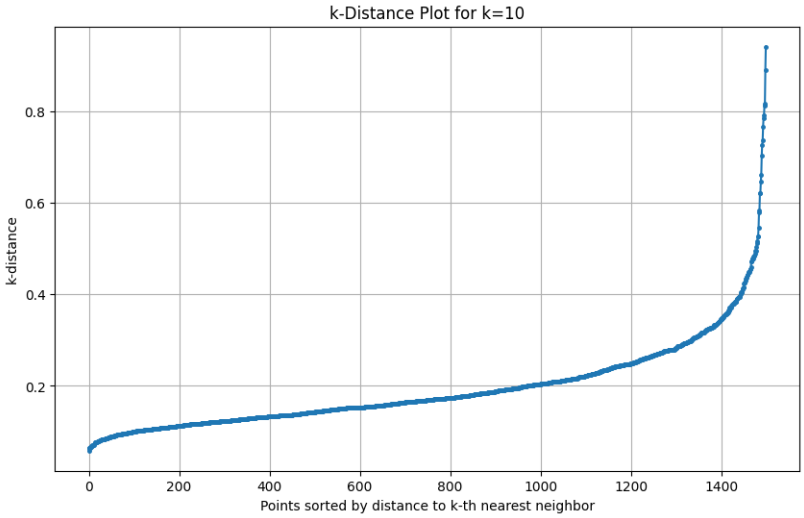


Figure 8: 10-distance plot

We can now evaluate different combinations of `min_samples` (in the range [4, 10]) and `eps` (in the range from 0.2 to 1 exclusively, using a step of 0.01) using a grid search.

```
from sklearn.metrics import silhouette_score
param_grid = {
  'eps': np.arange(0.2, 1, 0.01),
  'min_samples': range(4, 10)
}
dbscan = DBSCAN()
grid_search = GridSearchCV(dbscan, param_grid, scoring=silhouette_score)
grid_search.fit(data_scaled)
grid_search.best_params_
```

```
'eps': 0.2, 'min_samples': 4
```

The results indicate that the optimal values are `eps=0.2` and `min_samples=4`. **An `eps` value of `2.3` yields the best outcome of 7 clusters while accurately identifying a mostly perfect cloud of noise and border points** (see Figures 8 and 9).

```
dbscan_model = DBSCAN(eps=0.23, min_samples=4)
dbscan_labels = dbscan_model.fit_predict(data_scaled)
dbscan_data = data_scaled.copy()
dbscan_data['cluster'] = dbscan_model.labels_
dbscan_data.groupby('cluster')['cluster'].value_counts()
```

```
cluster
-1     73
 0    200
 1    265
 2    230
 3    194
 4    124
 5    195
 6    219
Name: count, dtype: int64
```

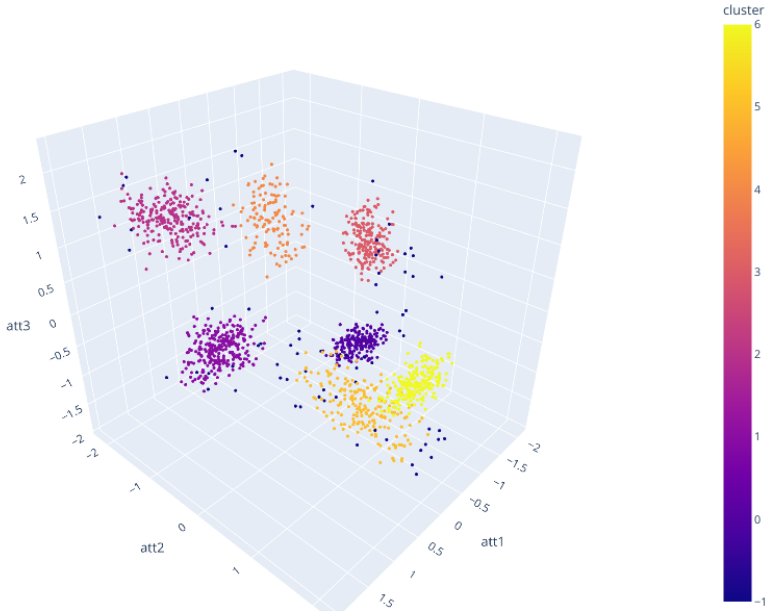


Figure 9: 3D scatter plot for DBScan with the optimal hyperparameters `eps=0.23` and `min_samples=4`.

# 4 Conclusion

Both K-Means and DBScan can identify clusters in the dataset, but each has unique strengths and weaknesses. K-Means produced stable centroids using both `random` and `kmeans++` initialization techniques, with seven clusters. DBScan, with its default settings, didn't find the expected clusters but results were improved after adjusting `eps` and `min_samples` values. The Elbow method was not clear on the best number of clusters, but the Silhouette and Davies-Bouldin Index pointed to seven clusters as the most reliable option. Overall, tuning both models helped achieve clear identifiable clusters and confirm that seven clusters best fit the data.

Agglomerative hierarchical clustering may be less suited to this data due to its **irreversible merging**, which **limits flexibility**, especially with noisy data, and its high computational cost. **Pre-clustering with partitioning methods can help**, but K-Means and DBScan, after tuning, provide clearer, faster, and more efficient clustering results.

## References

Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* O'Reilly Media, Inc., 2022.