MLII - Unsupervised Learning, Agents

Before diving into our report here is our the work has been divided :

- Meven : Part 1, 2 and 5
- Yoann : Part 3 and 4

PART 1 :

1. Let X represent the weight of a person in a population, measured in kilograms, and Y represent the height of the person, measured in meters. We can assume that the weight of the people in the population follows a normal distribution with mean 75 kg and standard deviation 10 kg, and that the height of the people in the population follows a normal distribution with mean 1.75 m and standard deviation 0.1 m.

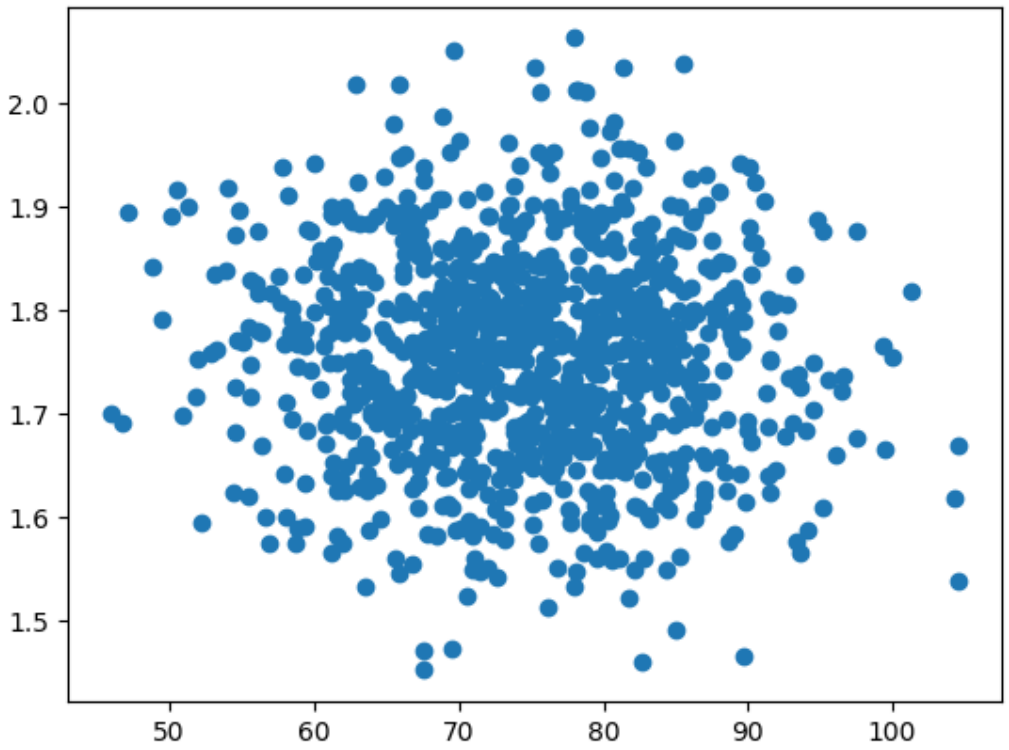
   The expected value of Z can be computed as follows:

   E[Z] = E[(X, Y)] = (E[X], E[Y]) = (75, 1.75)

   Since both X and Y are continuous random variables with finite means, the expected value of Z is also finite.

2. To sample n points from the law of Z, we can use a random number generator to sample n values for X and n values for Y, according to the normal distributions that we defined for them earlier. Here is some example code in Python that demonstrates how this could be done:

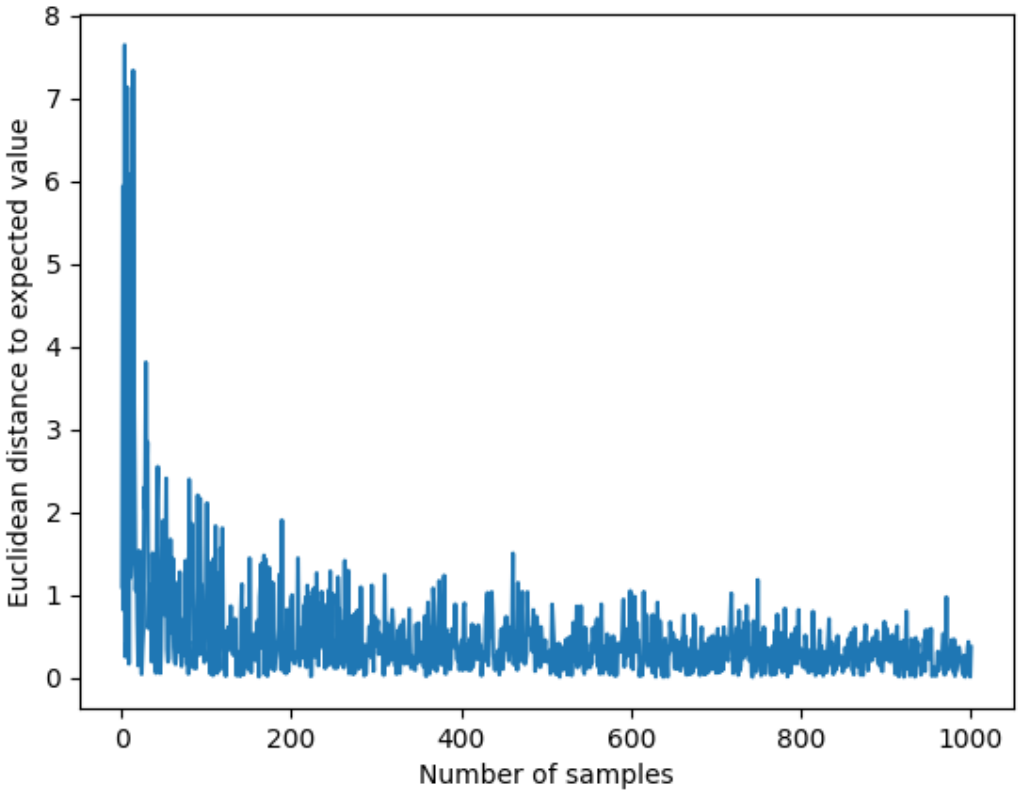
```
import numpy as np

# Set the number of points to sample
n = 1000

# Set the mean and standard deviation for X and Y
mean_x = 75
std_x = 10
mean_y = 1.75
std_y = 0.1

# Sample n values for X and Y
x = np.random.normal(mean_x, std_x, n)
y = np.random.normal(mean_y, std_y, n)

# Plot the points in a 2-dimensional figure
import matplotlib.pyplot as plt
plt.scatter(x, y)
plt.show()
```



This will generate a scatter plot with n points, each representing a person in the population with a certain weight and height. The points should be distributed around the mean values of 75 kg and 1.75 m, respectively, according to the normal distributions of X and Y.

3. To compute the empirical average of the first n samples and plot the euclidean distance to the expected value as a function of n, we can modify the example code that I provided earlier as follows:

```python
import numpy as np

# Set the number of points to sample
n_max = 1000

# Set the mean and standard deviation for X and Y
mean_x = 75
std_x = 10
mean_y = 1.75
std_y = 0.1

# Initialize arrays to store the samples and the empirical averages
x = np.empty(n_max)
y = np.empty(n_max)
empirical_averages = np.empty((n_max, 2))

# Set the expected value of Z
expected_value = [mean_x, mean_y]

# Initialize an array to store the euclidean distances
euclidean_distances = np.empty(n_max)

# Sample the first n points and compute the empirical average for each n
for n in range(1, n_max+1):
  x[:n] = np.random.normal(mean_x, std_x, n)
  y[:n] = np.random.normal(mean_y, std_y, n)
  empirical_averages[n-1] = [x[:n].mean(), y[:n].mean()]
  euclidean_distances[n-1] = np.linalg.norm(empirical_averages[n-1] -
expected_value)

# Plot the euclidean distances as a function of n
import matplotlib.pyplot as plt
plt.plot(range(1, n_max+1), euclidean_distances)
plt.xlabel('Number of samples')
plt.ylabel('Euclidean distance to expected value')
plt.show()
```



This will generate a plot showing how the euclidean distance between the empirical average of the first n samples and the expected value of Z changes as a function of n. As n increases, the empirical average should converge to the expected value, and the euclidean distance should decrease.

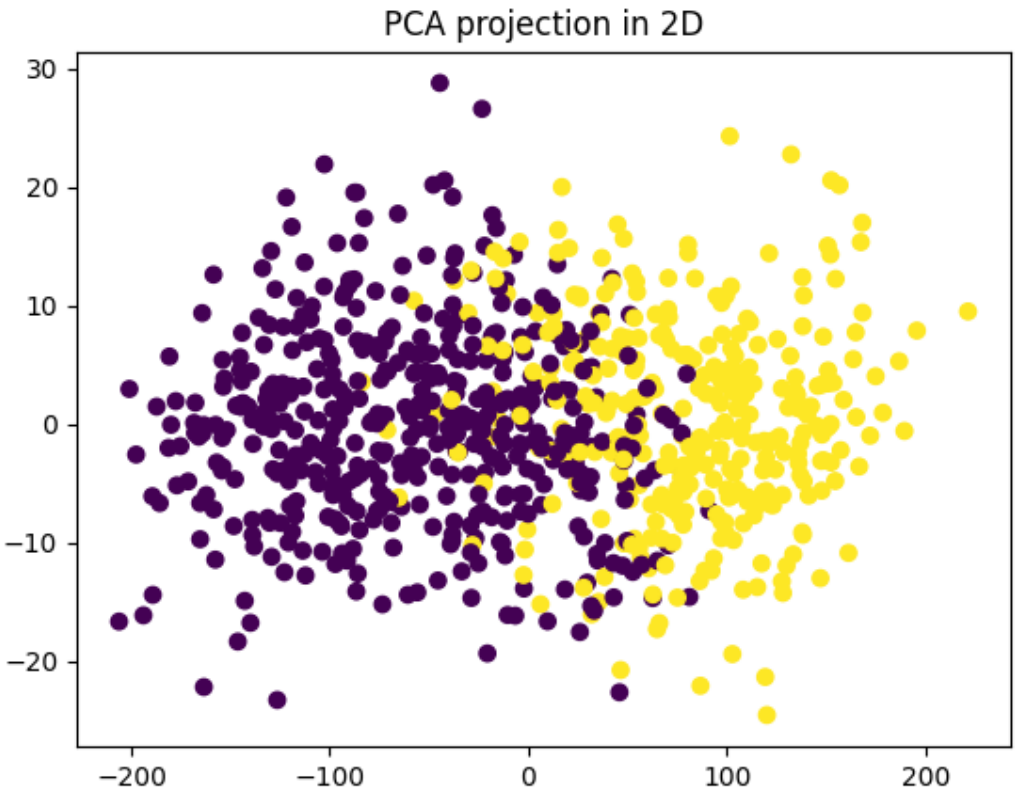
PART 2 :

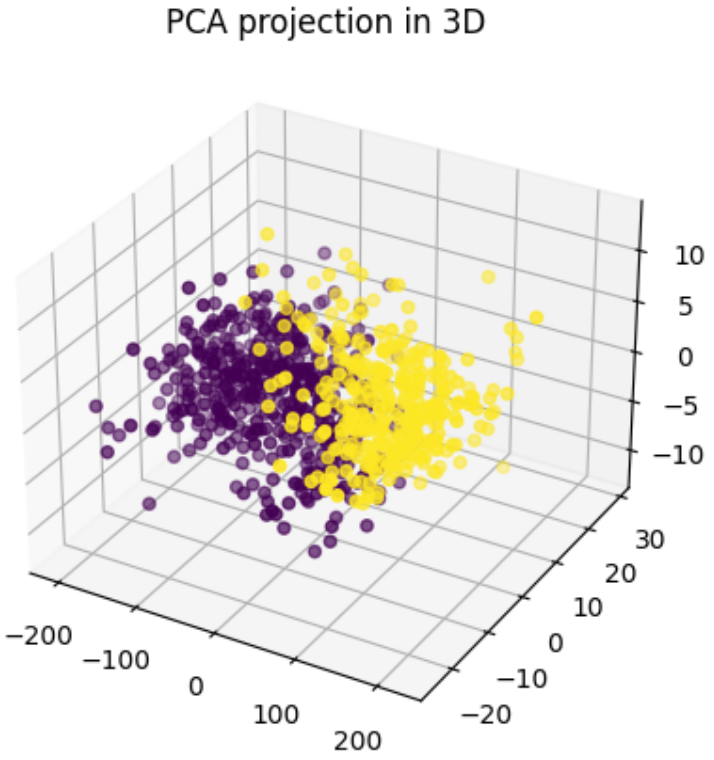
To perform a dimensionality reduction of the data to a lower dimension, you can use techniques such as principal component analysis (PCA) or t-SNE (t-distributed stochastic neighbor embedding).

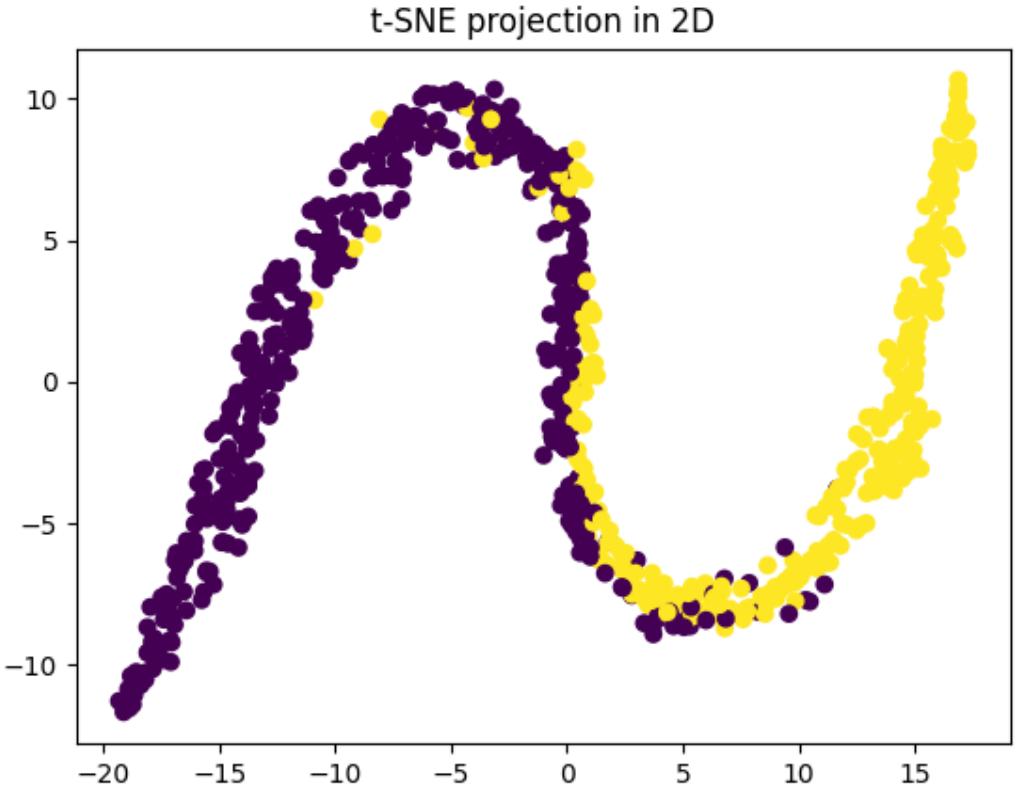
PCA is a linear dimensionality reduction technique that seeks to find the directions of maximum variance in high-dimensional data and projects the data onto a lower-dimensional space while preserving as much of the variance as possible. To plot the projected data onto a scatter plot in 2D or 3D, you can use the first two or three principal components as the x and y or x, y, and z axes, respectively. You can then color the projected samples according to their label.

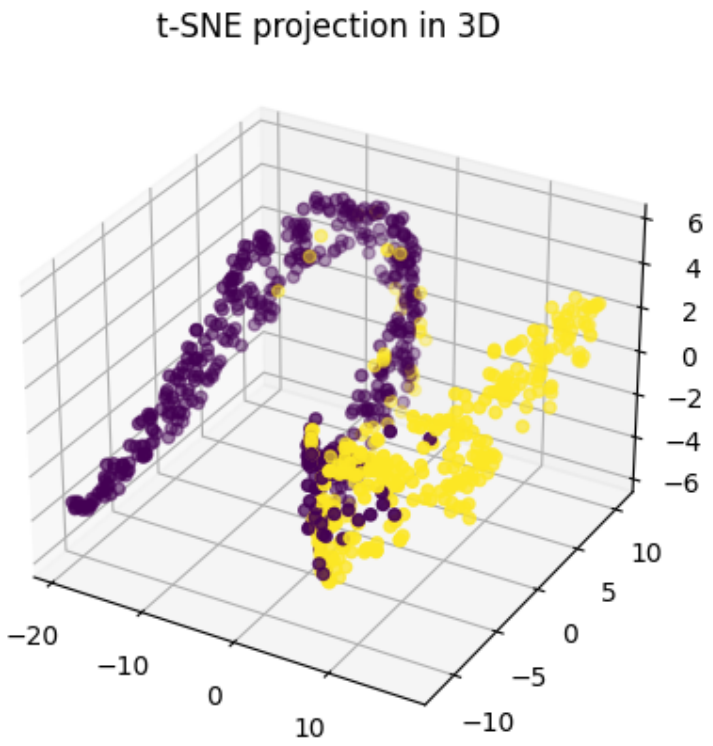
```
import numpy as np
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

# Assume that X is a numpy array with the original data and y is a
numpy array with the labels
X = np.load("data.npy")
y = np.load("labels.npy")

# Perform PCA on the data
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X)

# Perform t-SNE on the data
tsne = TSNE(n_components=3)
X_tsne = tsne.fit_transform(X)

# Plot the PCA projection in 2D
plt.figure()
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y)
plt.title("PCA projection in 2D")

# Plot the PCA projection in 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_pca[:, 0], X_pca[:, 1], X_pca[:, 2], c=y)
plt.title("PCA projection in 3D")

# Plot the t-SNE projection in 2D
plt.figure()
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y)
plt.title("t-SNE projection in 2D")

# Plot the t-SNE projection in 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_tsne[:, 0], X_tsne[:, 1], X_tsne[:, 2], c=y)
plt.title("t-SNE projection in 3D")

plt.show()
```

This code will perform PCA and t-SNE on the data in X and plot the projections onto scatter plots in 2D and 3D, coloring the samples according to their labels in y. You can then compare the separation of the different classes in the scatter plots to determine which dimensionality allows for better prediction of the label.




PCA projection in 3D
10
5
0
−5
−10
30
20
10
0
−10
−20
−200
−100
0
100
200


t-SNE projection in 2D
10
5
0
−5
−10
−20
−15
−10
−5
0
5
10
15



PART 3 :

1. K-Means clustering

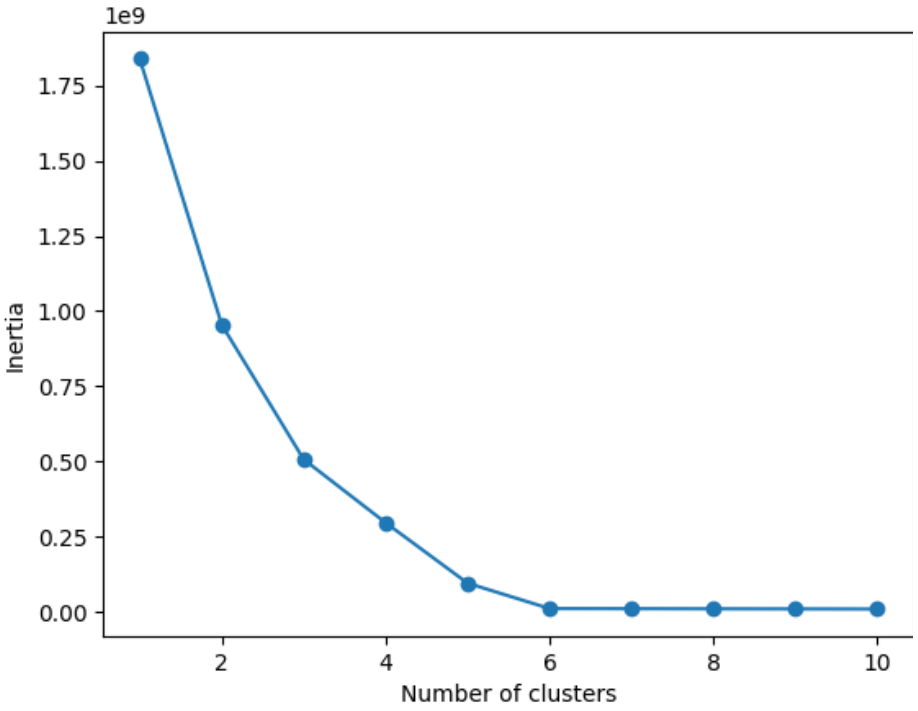
Before going into the K-Means let's define what were my parameters for this algorithm.

```
KMeans(n_clusters=k, random_state=1, n_init=10)
```

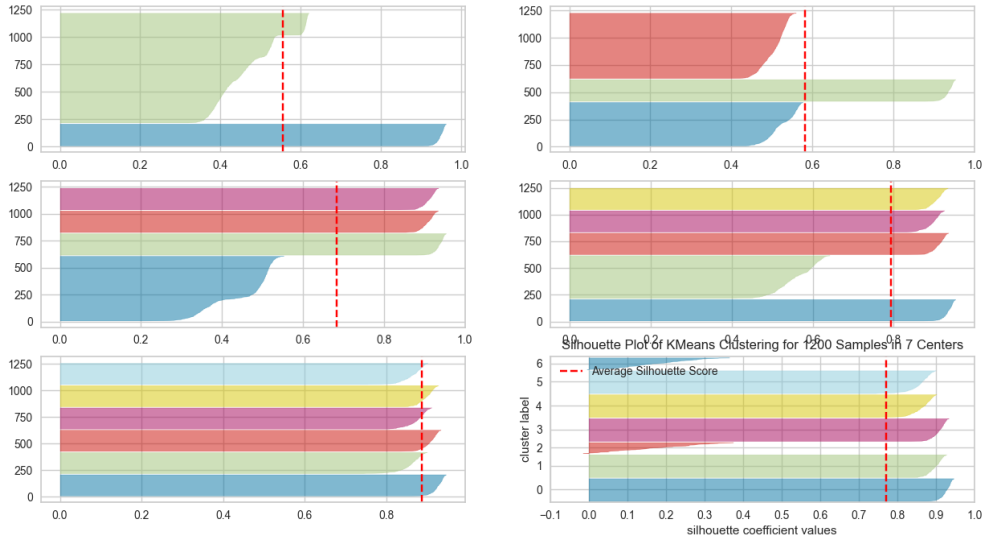
n_clusters is k because the numbers will be different, i set a seed to my K-Means to avoid randomness and i had to specify the n_init value to avoid a warning so i set it to the default value which is 10. Time to dive into the K-Means.

1.1. Elbow method on K-Means

With the elbow graph we can see that the elbow is between 5 and 6. I think it's 6 but the difference between 5 and 6 is not big enough to make a clear statement. Let's see what the silhouette method have to say.



1.2. Silhouette method on K-Means

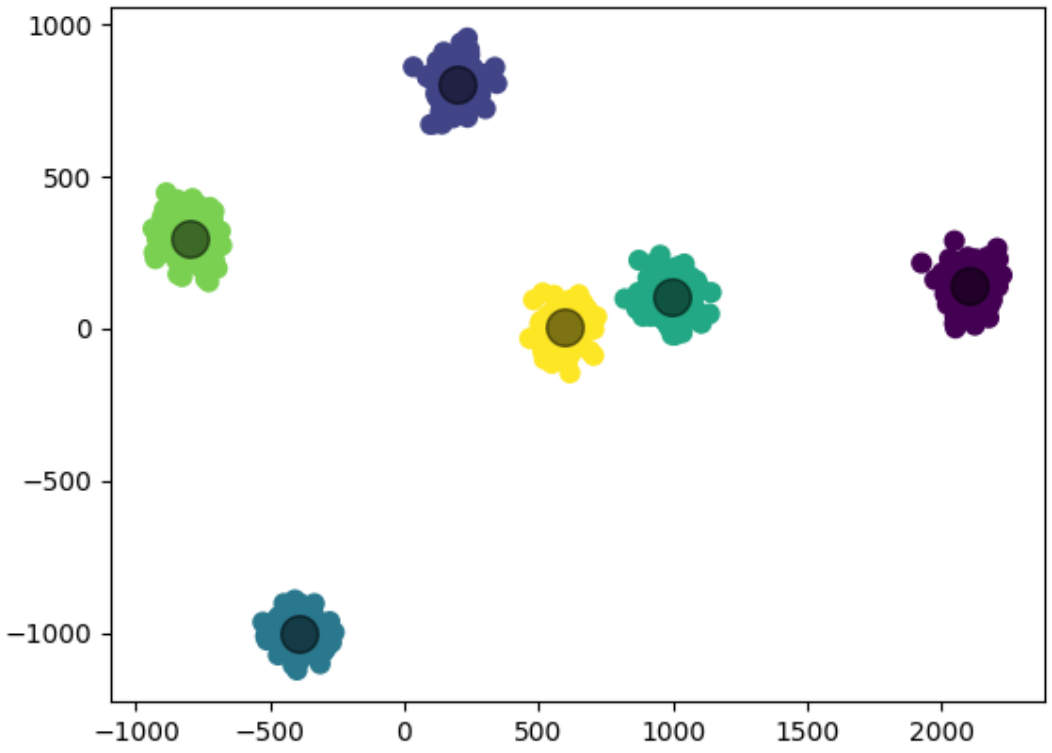


I used the SilhouetteVisualize from the Yellowbrick package to have this image. I did from 2 to 7 clusters. And this time we can clearly see that the most optimized amount of clusters is 6. Because the silhouette coefficient is the closest to 1. And if you want to have the numbers here they are :

For n_clusters = 5 The average silhouette_score is : 0.792549276476918
For n_clusters = 6 The average silhouette_score is : 0.887139316265711
For n_clusters = 7 The average silhouette_score is : 0.770241014068024

1.3. K-Means

So let's try the K-Means algorithm with 6 as the amount of clusters.



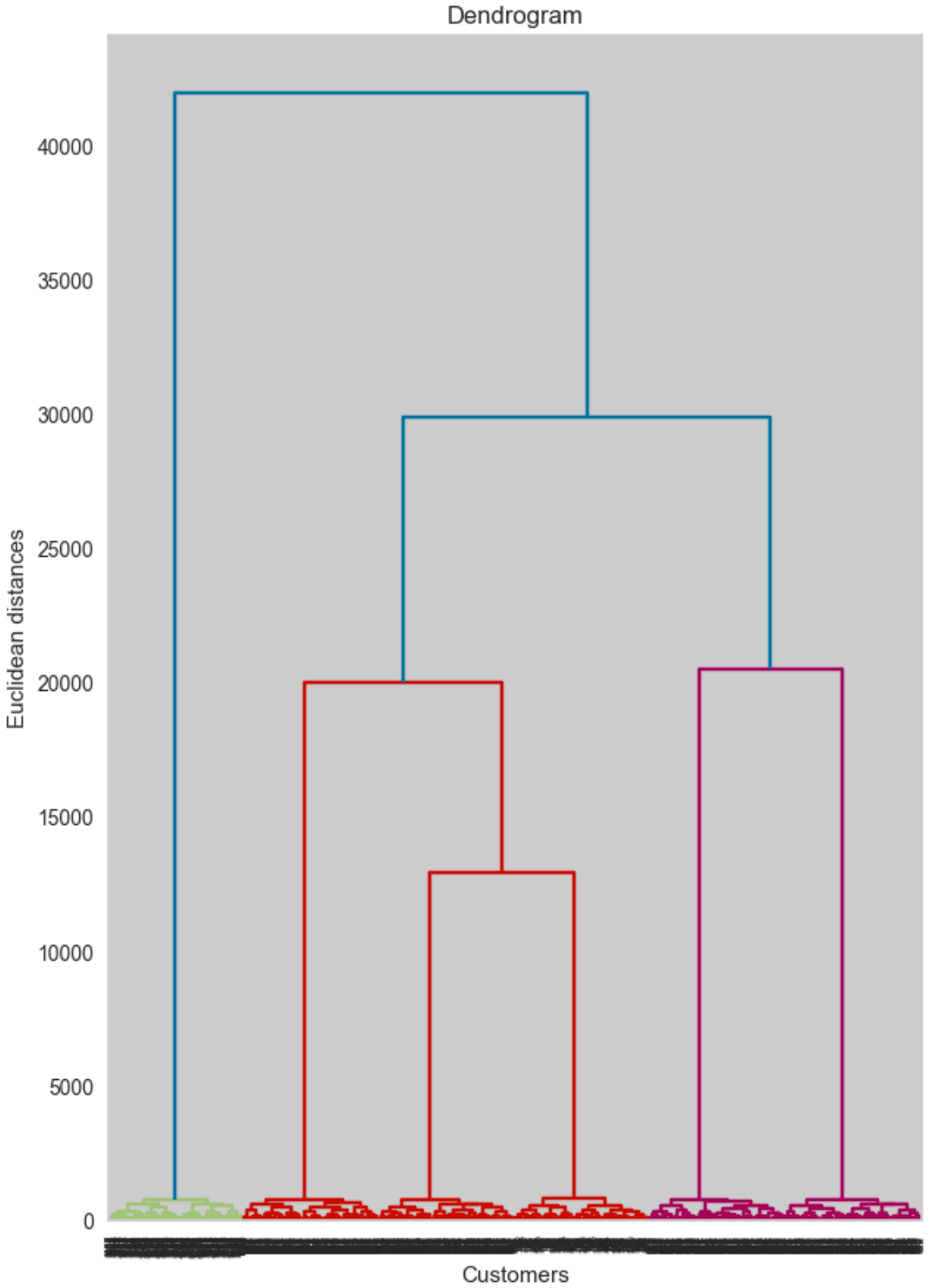
So we were right 6 is the right amount of clusters for our dataset.

2. Hierarchical Clustering

For the hierarchical clustering I decided to use the dendrogram and the silhouette analysis.

2.1. Dendrogram

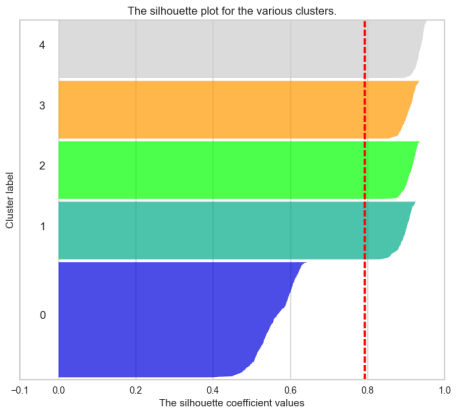
So to make this dendrogram I used the scipy.cluster.hierarchy and here is the result. For a dendrogram to have an idea of the right amount of clusters we have to find the longest distance and we can see it on the rightest node. And so if we divide it we can find that the right amount seems to be 6 clusters.

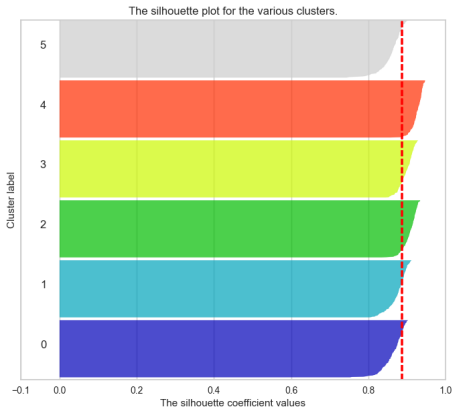


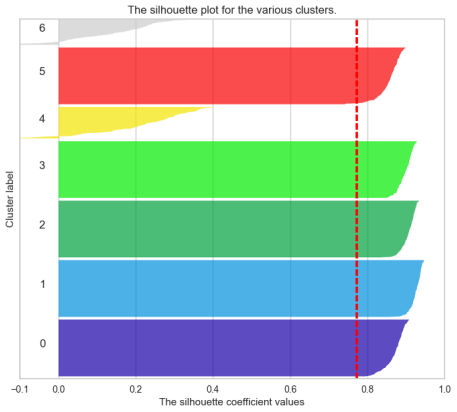
2.2. Silhouette analysis

So the dendrogram makes us think that the right amount of clusters seems to be around 6, so I made a silhouette analysis of 5, 6 and 7 clusters. Let's discuss the results.
Here are the numbers :
For n_clusters = 5 The average silhouette_score is : 0.792549276476918
For n_clusters = 6 The average silhouette_score is : 0.887139316265711

For n_clusters = 7 The average silhouette_score is : 0.771826478879976
And here are the graphs made by matplotlib this time.

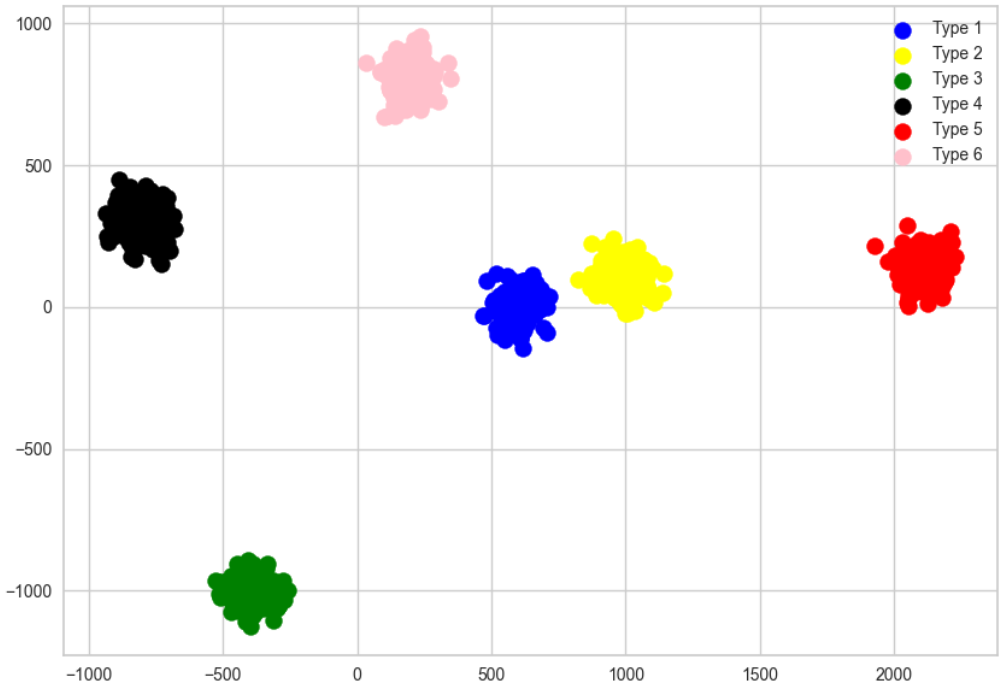






So again the right amount with the silhouette analysis is clearly 6.

2.3. Hierarchical Clustering



3. Discussion

So first of all I think that both algorithms worked really well with the silhouette and I think it's the most accurate way of finding the right amount of clusters. The score it gives seems really precise, whereas the dendrogram can give an idea but only using a dendrogram is having a high percentage of failure. And the elbow/knee method kind of gave me the same vibe, but I think this method is a bit more precise. Using both in combination is a good idea though, using different heuristics to kind of merge multiple results. So to conclude i think the best is the silhouette but using multiple heuristics at the same time is good way to have the most precise answer possible;

PART 4 :

So before starting to write my own policy I try to display every single variable to have a better knowledge of the exercise.

To really start the policy I created the node class. This class is composed of three variables : left, none, right which are 3 floats. And also 4 functions : throw, update_left, update_right and update_none.

1. My node Class

1.1. The constructor

```python
class Node:
    def __init__(self, left = 1/3, none = 1/3, right = 1/3) -> None:
        self.left = left
        self.none = none
        self.right = right
```

I set every variable by default to ⅓ to be able to create a special node for the first position and the last position. Since in those positions left or right are equal to 0 because it should never happen.

1.2. The throw function

```python
def throw(self) -> str:
    rand = random.random()
    if rand > 0 and rand <= self.left:
        return 'left'
    elif rand > self.left and rand <= (self.none + self.left):
        return 'none'
    else:
        return 'right'
```

This is to reproduce randomness and I used the native random module.

1.3. The update_left function

```python
def update_left(self, newValue):
    newLeftThreshold = 1 - newValue

    if (self.left > 0.80 or self.left < 0.20):
        return

    if (newValue > 0.80):
        newValue = 0.80
        newLeftThreshold = 0.20
    elif (newValue < 0.20):
        newValue = 0.20
        newLeftThreshold = 0.80

    if (self.right == 0):
        self.none = newLeftThreshold * (self.none + self.left)
        self.left = newValue
    else:
        self.none = newLeftThreshold * (self.none + self.left / 2)
        self.left = newValue
        self.right = newLeftThreshold * (self.right + self.left / 2)
```

This is where math kicks in. The new value represents the new percentage that the left variable should have. I set some rules to avoid values going over 1 and under 0. It's to avoid scenarios where one value is really close to 1 and then it's going to stay there for a really long time and for an AI it's a really bad scenario. So after some tests i set the limit from 0.20 to 0.80.

To compute my new variables I set the threshold which represents the remaining maximum variable.
And to compute the new none and right i use this as a maximum value and a little cross product creates our new value. The self.left / 2 you see on the left side is to be able to have both right and none under the maximum value of 1. Because without that both values added are not equal to 1 and it's quite difficult to make an accurate cross product with that.
Let's say we have 0.56 for left, 0.32 for none and 0.12 for right.
Our new value would be 0.67
Our threshold is going to be 0.33.
The calculation of our new none would look like this : 0.33 * (0.32 + (0.56 / 2)) = 0.198
The calculation of our new right would look like this : 0.33 * (0.12 + (0.56 / 2)) = 0.132
And our three new variables would be : 0.67 for left, 0.198for none and 0.132 for right.

Moving on to update_right and update_none which work the same way

1.4. The update_right function

```
def update_right(self, newValue):
    newRightThreshold = 1 - newValue

    if (self.right > 0.80 or self.right < 0.20):
        return

    if (newValue > 0.80):
        newValue = 0.80
        newRightThreshold = 0.20
    elif (newValue < 0.20):
        newValue = 0.20
        newRightThreshold = 0.80
    if (self.left == 0):
        self.none = newRightThreshold * (self.none + self.right)
        self.right = newValue
    else:
        self.none = newRightThreshold * (self.none + self.right / 2)
        self.right = newValue
        self.left = newRightThreshold * (self.left + self.right / 2)
```

1.5. The update_none function

```
def update_none(self, newValue):
    newNoneThreshold = 1 - newValue

    if (self.none > 0.80 and newValue > self.none) or (self.none < 0.20 and newValue < self.none):
        return

    if (newValue > 0.80):
        newValue = 0.80
        newNoneThreshold = 0.20
    elif (newValue < 0.20):
        newValue = 0.20
        newNoneThreshold = 0.80
    if (self.right == 0):
        self.left = newNoneThreshold * (self.left + self.none)
        self.none = newValue
    elif (self.left == 0):
        self.right = newNoneThreshold * (self.right + self.none)
        self.none = newValue
    else:
        self.left = newNoneThreshold * (self.left + self.none / 2)
        self.right = newNoneThreshold * (self.right + self.none / 2)
        self.none = newValue
```

The only thing I would add is why did I check whether right or left are equal to 0 because in those situations the threshold is basically the remaining since it's 2 variables and not 3.

That was my node class. Now it's time to dive into the dmz_policy.

2. The DMZ policy

2.1. The dmz_policy function

```
def dmz_policy(agent: Agent) -> str:
    global new
    global nodes

    if new == True:
        new = False
        nodes = [Node(0,1/2,1/2),
                 Node(),
                 Node(),
                 Node(),
                 Node(),
                 Node(),
                 Node(),
                 Node(1/2,1/2,0),]
    update_values(agent, (nodes[agent.position]))
    move = nodes[agent.position].throw()
    return move
```

I have 2 global values : new and nodes. New is a boolean which is here to know whether we should initiate our nodes or not and the node variables are all my different nodes. When I initiate my nodes I create 8 of them because we have 8 different positions. I set values or not

to initiate each variable attached to each action. If we are not in the initiation process we are doing the regular things. Which are updating all the values we will talk about a bit later and throwing our next movement.

2.2. The update_values function

```
def update_values(agent: Agent, node: Node):
    rewardValue = agent.known_rewards[agent.position]
    if rewardValue >= 30:
        increase = 1 + (rewardValue * 30 / 100) / 100
        node.update_none(node.none * increase)
    else:
        node.update_none(node.none * (1-(0.15 - rewardValue / 2 / 100)))
    if agent.position > 0 and agent.position < 7 and agent.known_rewards[agent.position - 1] > agent.known_rewards[agent.position + 1]:
        diff = agent.known_rewards[agent.position - 1] - agent.known_rewards[agent.position + 1]
        diffPer = 1 + (diff * 30 / 100) / 100
        node.update_left(node.left * diffPer)
    elif agent.position > 0 and agent.position < 7 and agent.known_rewards[agent.position - 1] < agent.known_rewards[agent.position + 1]:
        diff = agent.known_rewards[agent.position + 1] - agent.known_rewards[agent.position - 1]
        diffPer = 1 + (diff * 30 / 100) / 100
        node.update_right(node.right * diffPer)
```

If our reward is greater than 30 we are going to increase the none value of the node where we are and if not we are going to decrease the none value of our node.

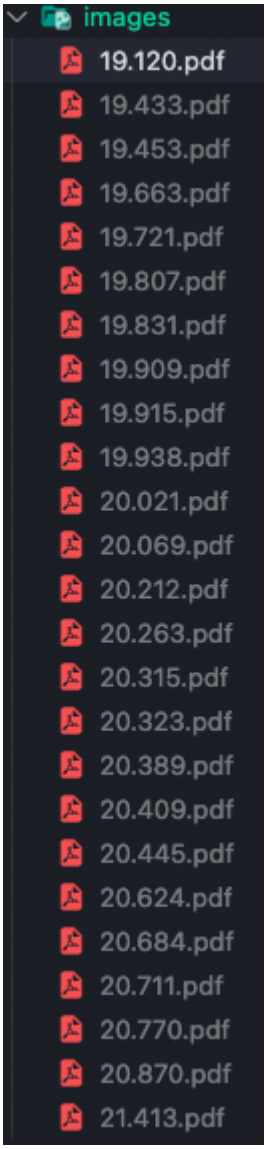
The calculations are pretty simple:

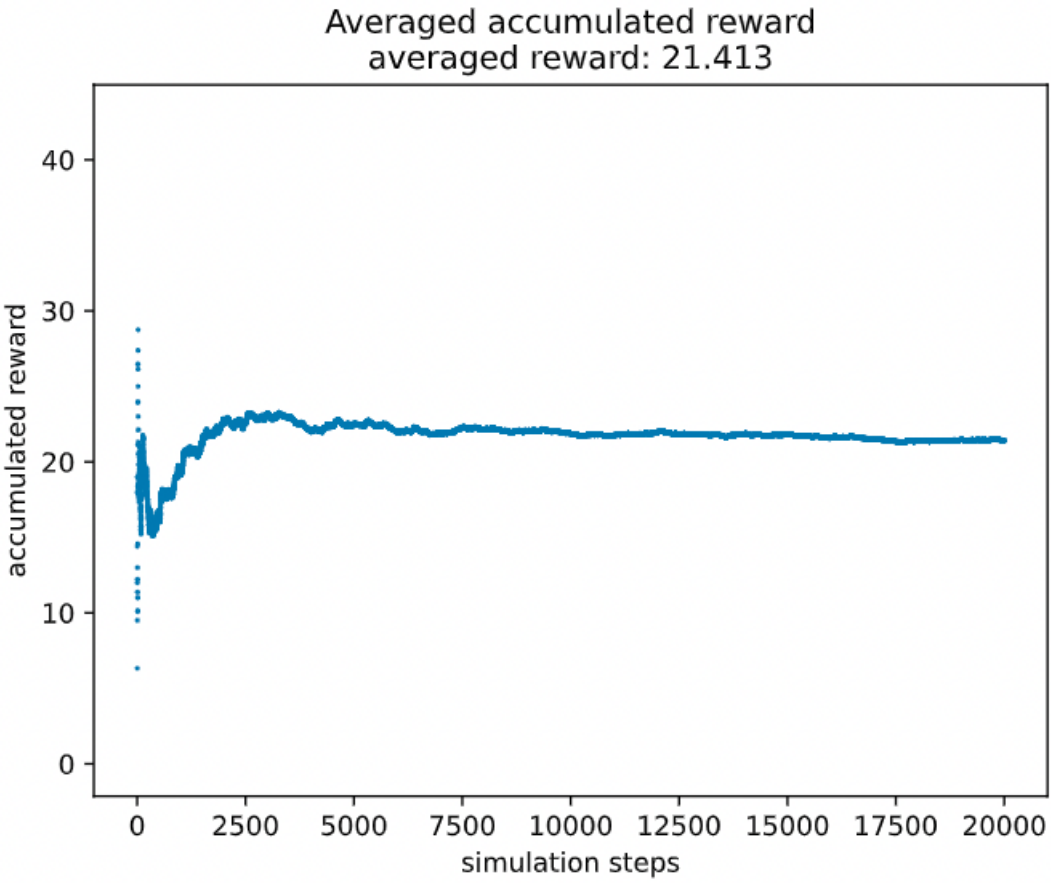
- For the increase I make a cross product so that the increase is going up to 30%.
    - Which mean for a reward value of 100 it's going to increase at a 30% rate
- For the decrease I make a cross product so that the decrease is going up to 15%
    - Which mean for a reward value of 15 it's going to decrease at a 7.5% rate

After that I will modify left or right depending on which holds the biggest reward. It's going to increase once again up to 30% depending on the difference between both.

3. Results

I did the simulation 25 times to have a bit more accurate view of my ai and here are the results :





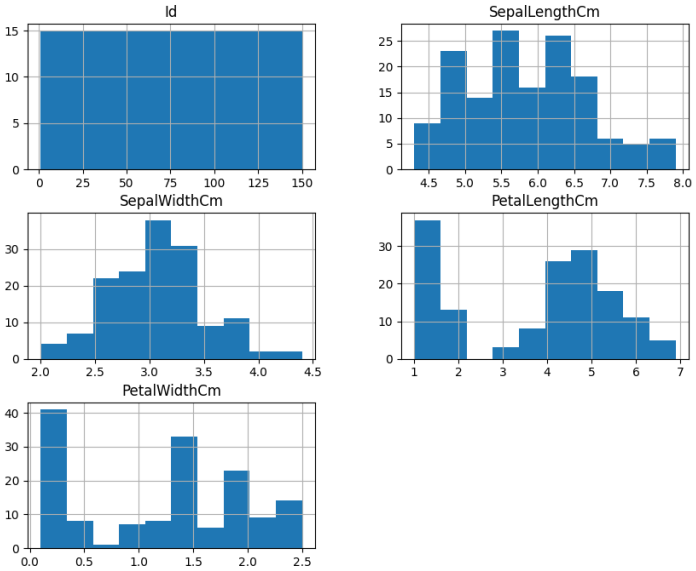
So on the left are the results I get after the 20 simulations and on the right it's the graph created by the highest averaged accumulated reward.

PART 5 :

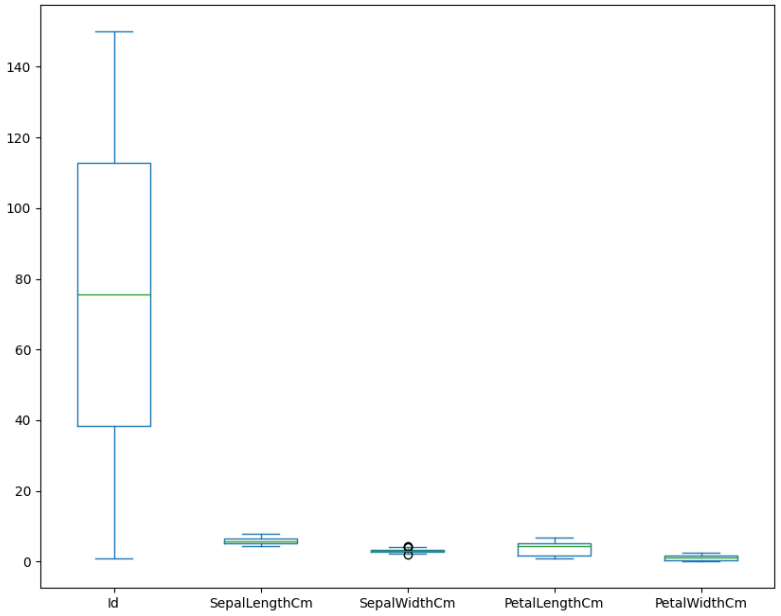
I will use the "Iris" dataset which is a popular dataset for classification tasks in machine learning. It contains 150 samples of iris flowers with 4 attributes: sepal length, sepal width, petal length, and petal width. The first column is a unique id, which satisfies the constraints of the prompt.

First we start with an analysis,

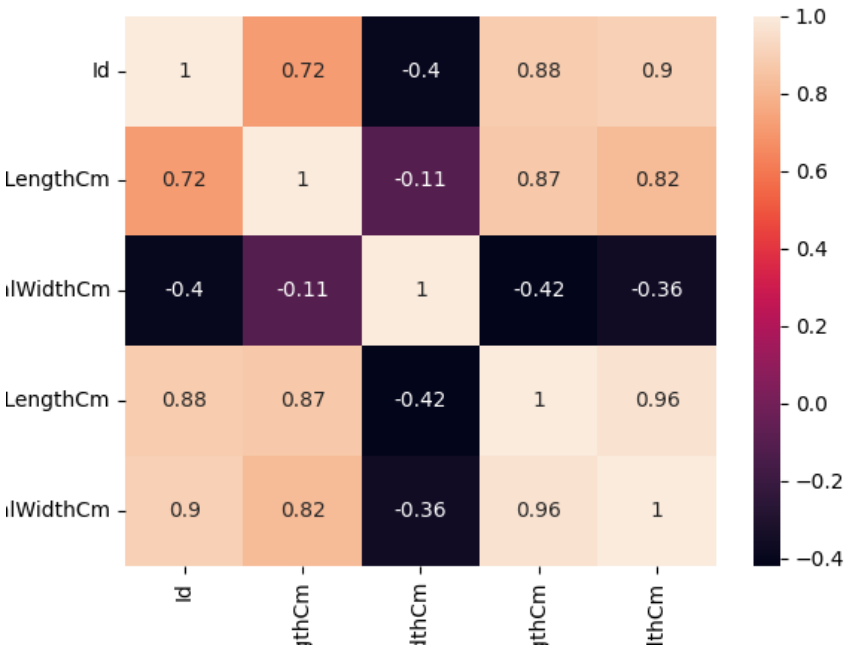
```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
df = pd.read_csv("Iris.csv")

# Histograms of quantitative variables
df.hist(figsize=(10, 8))
plt.show()

# Comment on important statistical aspects
print(df.describe())

# Study of potential outliers
df.plot(kind="box", figsize=(10, 8))
plt.show()

# Correlation matrix
sns.heatmap(df.corr(), annot=True)
plt.show()

# Study of categorical data
print(df["species"].value_counts())
```



	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	75.500000	5.843333	3.054000	3.758667	1.198667
std	43.445368	0.828066	0.433594	1.764420	0.763161
min	1.000000	4.300000	2.000000	1.000000	0.100000
25%	38.250000	5.100000	2.800000	1.600000	0.300000
50%	75.500000	5.800000	3.000000	4.350000	1.300000
75%	112.750000	6.400000	3.300000	5.100000	1.800000
max	150.000000	7.900000	4.400000	6.900000	2.500000




```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: Species, dtype: int64
```

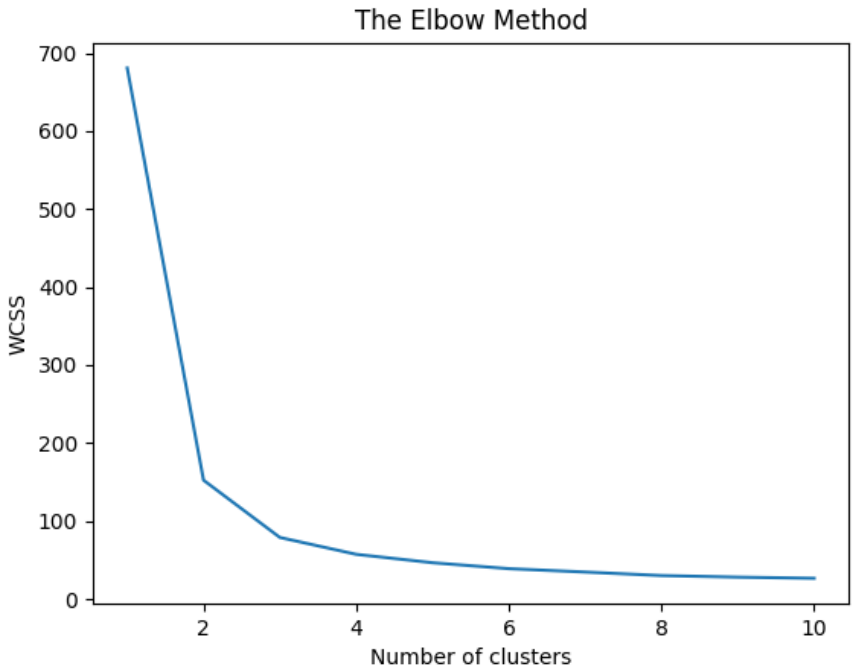


This analysis will give us a general understanding of the distribution of the quantitative variables, the statistical summary of the dataset, any potential outliers, the correlation between variables, and the distribution of the categorical variable (species of iris flower in this case).

Next, we can perform unsupervised learning on this dataset. One approach could be to use clustering to group the samples into different clusters based on their attributes. We can use k-means clustering for this purpose and try to find the optimal number of clusters using the elbow method.

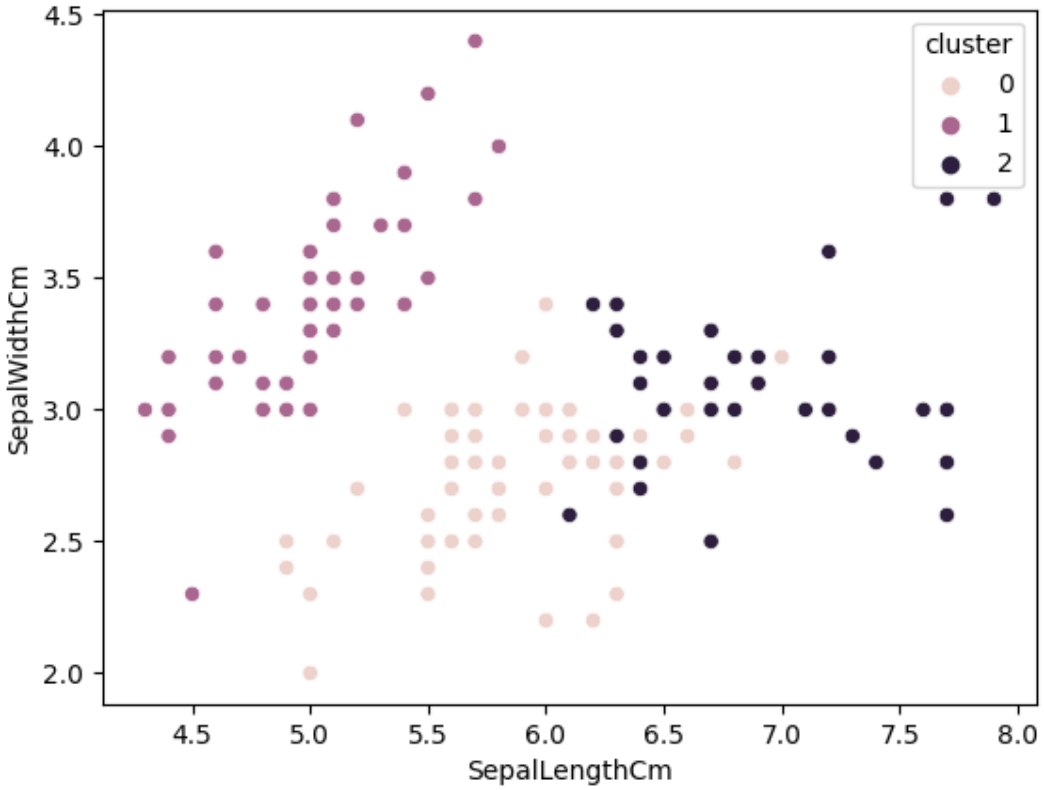
```
from sklearn.cluster import KMeans

# Select only the numeric columns for clustering
X = df.iloc[:, 1:-1]

# Use the elbow method to find the optimal number of clusters
wcss = []
for i in range(1, 11):
      kmeans = KMeans(n_clusters=i, init="k-means++", random_state=42)
      kmeans.fit(X)
      wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss)
```

```
plt.title("The Elbow Method")
plt.xlabel("Number of clusters")
plt.ylabel("WCSS")
plt.show()
```



Based on the elbow plot, we can choose the optimal number of clusters as 3. We can then fit the k-means model on the dataset and predict the cluster labels for each sample.

```
# Fit the k-means model and predict the cluster labels
kmeans = KMeans(n_clusters=3, init="k-means++", random_state=42)
pred_labels = kmeans.fit_predict(X)

# Add the predicted cluster labels to the dataframe
df["cluster"] = pred_labels

# Visualize the clusters
sns.scatterplot(x="SepalLengthCm", y="SepalWidthCm", hue="cluster",
data=df)
plt.show()

score = silhouette_score(X, pred_labels)
print('Silhouette score: {:.3f}'.format(score))
```




```
Silhouette score: 0.553
```

With this unsupervised learning the silhouette score at the end is 0.553.