LDATA2010 Exercise Session - Clustering

> Code online : https://colab.research.google.com/drive/1eRTqcVbUlpRvERDXvEnjUj6I_a8tuz90?usp=sharing

Clustering aims at automatically discovering groups of similar samples among data sets. This statement briefly introduces different clustering algorithms. The exercises can be realized using the software of your preference (Python, R, Matlab, etc.). The dataset we will use in this session is based on transcriptomics data. As you can see when importing the dataset, the first columns are 300 different gene expressions, and the last column ("cell_type") contains an identifier for the type of the cell.

Each row represents the analysis of *one* cell, and each of the "gene" columns contains (in layman's terms and not wholly accurately) the expression of the gene at one location in the sequence.

Tools and libraries

To help you implement and test the various clustering algorithms presented during this session, we provide a (non-exhaustive) list of libraries that can be useful. Don't hesitate to look at the documentation for each of these libraries, to get a feel of what they can offer you.

In particular, the `pca_decomposition` function implements PCA, a dimensionality reduction method you will see more thoroughly during the course. We will use it now for 2D visualization, because although we might want to test clustering algorithms in 2D, they start being especially useful with high dimensional data.

> Is it better to first reduce the dimensionality of the dataset and then do clustering or first do clustering and then visualize in a lower dimension ?

```
# Importing the data and nice libraries
import matplotlib.pyplot as plt # https://matplotlib.org/
import numpy as np # https://numpy.org/doc/1.26/user/index.html#user
import pandas as pd # https://pandas.pydata.org/
import seaborn as sns # https://seaborn.pydata.org/
from sklearn.decomposition import PCA # https://scikit-
learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
from sklearn.preprocessing import StandardScaler # https://scikit-
learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.ht
ml
from IPython.display import display, clear_output #
https://ipython.readthedocs.io/en/stable/api/generated/IPython.display.htm
l
```

```
from sklearn.datasets import make_circles

def pca_decomposition(df, components=2):
    """Perform PCA on a dataset.

    PCA is a dimensionality reduction method, which we use to go from 300
dimensions down to 2.
    """
    pca = PCA(n_components=components)
    df2 = pca.fit_transform(df)
    return df2, pca
```

Dataset

> **Exercise 1**
> Create a scatter plot of X after PCA in a 2-D space. How many clusters do you intuitively observe ? Do you think these clusters are present in the 300-dimension space ?

```
dataframe =
pd.read_csv("https://gist.githubusercontent.com/victorjoos/34b43873616f253
451af610ca8460ddf/raw/ad9b89f72c6c67318e08de13871c34d56fbae259/transcripto
mics_data.csv")
y = dataframe["cell_type"]
X = dataframe[[col for col in dataframe.columns if
col.startswith("gene_")]].to_numpy() # Only keep gene expressions

X_pca, pca = pca_decomposition(X) # Compute PCA DR

print("dataset shape before PCA:", X.shape)
print("dataset shape after PCA:", X_pca.shape)
```

```
dataset shape before PCA: (4765, 300)
dataset shape after PCA: (4765, 2)
```

```
# Your code here
```

K-means

The K-means algorithm may be summarized as follows : given,

- a set $D = \{x_1, \ldots, x_N\}$ of N data points with dimension H,
- a distance function d,
- a number K of clusters,

it defines a set $C = c_1, \ldots, c_K$ of K centroids with dimension H. After initializing the centroids according to some heuristic (e.g. randomly in the data space, randomly in D, etc.), the centroid coordinates are updated according to the following scheme:

1. For each $i \in \{1, \ldots, K\}$, compute $D_i = \{x_j \in D | i = \text{argmin}_{l \in 1, \ldots, K} d(c_l, x_j)\}$
2. For each $i \in \{1, \ldots, K\}$, set $c_i \leftarrow \frac{1}{|D_i|} \sum_{x \in D_i} x$

Computing steps 1 and 2 corresponds to performing *one* iteration. Iterations are repeated until convergence of the centroids or after a maximum number has been reached. The clusters are then defined as the sets D_i for $i \in \{1, \ldots, K\}$

> **Exercise 2** Implement the K-means algorithm with Euclidian distance and apply it to X with K = 3 clusters and with the centroids chosen at random from the dataset. After each iteration, show the new clusters and centroids. How many iterations are needed before convergence ? Does the result seem intuitive ?
>
> What happens when you change K ?
>
> Do all the cells with the same cell type go in the same cluster ? How would you visualize this ?
>
> Does K-Means work even when the initial centroids are all in the same cluster ?
>
> Do you get the same result when doing the clustering on `X_pca` ?

```
def kmeans_iteration(X, centroids):
    # 1. compute the distance to the closest centroid
    clusters = np.random.randint(len(centroids), size=len(X)).astype(int)
# CHANGME : The ID of the closest centroid for each element in the dataset
    # 2. update the centroids
    centroids = X[:3] # CHANGME : the updated centroids
    return clusters, centroids

max_iter = 10 # CHANGME ?
K = 3 # CHANGME ?
centroids = X[:3] # CHANGME : Pick initial centroids randomly

## Visualization

old_centroids = np.zeros_like(centroids)
i = 0
while not np.allclose(centroids, old_centroids) and i<max_iter: # Test if
has converged
    fig, ax = plt.subplots(1)
    i += 1
    old_centroids = centroids
    clusters, centroids = kmeans_iteration(X, centroids)
    ax.cla()
```

```
    sns.scatterplot(x=X_pca[:,0], y=X_pca[:,1], hue=clusters,
palette="deep", ax=ax)
    C_pca = pca.transform(centroids)
    ax.scatter(C_pca[:,0], C_pca[:,1], c="black", linewidth=3, marker="P")
    ax.set_title(f"iteration {i}")
    plt.show()
```

Nearest neighbor clustering

Given a set $D = \{x_1, \dots, x_N\}$ of N data points, a distance function d and a threshold t, nearest neighbor clustering defines clusters in D in the following way :

- Define $C_1 = \{x_1\}, C = \{C_1\}$ and $K = 1$;
- For $i = 2, \dots, N$,

1. Compute $j = \text{argmin}_{l \in \{1,\dots,i-1\}} d(x_l, x_i)$
2. If $d(x_j, x_i) \geq t$ set $K \leftarrow K + 1$, define $C_K = x_i$ and set $C \leftarrow C \cup \{C_K\}$
3. Otherwise, identify $k \in 1, \dots, K$ such that $x_j \in C_k$ and set $C_k \leftarrow C_k \cup \{x_i\}$

Whereas K-means needs the user to specify the desired number K of clusters, nearest neighbor clustering instead requires to define a threshold t for the minimum distance between two clusters.

> **Exercise 3** Implement nearest neighor clustering with the Euclidian distance and apply it to X, using a threshold t = 5. Do you obtain the same result as when using K-means ? Does the order of the dataset matter ?
>
> How does the algorithm change when using a different threshold ? What is the threshold range where you get 3 clusters ?

```
def nneighbor_iteration(X, i, clusters, threshold):
    # 1. compute the index of the closest neighbor
    j = i-1 # CHANGME
    # 2. update the clusters
    clusters[i] = clusters[j] # CHANGME : take care of the threshold
    return clusters

threshold = 50 # CHANGME ?
clusters = np.ones(len(X), dtype=int) * -1 # init all nodes to to a
negative index ("unassigned")
clusters[0] = 0 # create the first cluster

##  Visualization
display_every_n = 500 # CHANGME ?


for i in range(1, len(X)):
```

```
    clusters = nneighbor_iteration(X, i, clusters, threshold)
    if (i % display_every_n) == 0 or i == len(X)-1:
      fig, ax = plt.subplots(1)
      sns.scatterplot(x=X_pca[:,0], y=X_pca[:,1], hue=clusters[:],
palette="deep", ax=ax)
      ax.set_title(f"node {i}")
      # Alternative, to display only assigned nodes :
      # sns.scatterplot(x=X_pca[:i,0], y=X_pca[:i,1], hue=clusters[:i],
palette="deep", ax=ax)
      plt.show()
```

DBSCAN

DBSCAN, for "Density-Based Spatial Clustering of Applications with Noise" allows the unsupervised clustering of data points by defining clusters as groups of points of high density separated by zones of low point density. Taking into account the density of data points allows the algorithm to be more robust to outliers, and to identify clusters of more diverse shapes than K-means. More information can be found on scikit-learn description, wikipedia article and more detailed publication.

DBSCAN does not require the user to set a number K of clusters. Instead, the algorithm relies on ϵ, the radius of the hypersphere around each point used to determine if other points are linked to it, and $min\_samples$, the minimum number of neighbours in the hypersphere for a point to be considered a core sample (see links above).

> **Exercise 4** Complete the code below to visualise the 2-dimensional circle dataset using the true labels, labels determined by KMeans (use $K = 4$ for this dataset), and labels obtained with scikit-learn's implementation of the DBSCAN algorithm.
>
> 1. Load the 2D circles and observe the data using the true labels in a 2D scatterplot (use the plot_with_labels(X, Y) function).
> 2. Apply k-means on it, using $K = 4$. Observe the results. Does the assigned clusters make sense considering the data set?
> 3. After reading about the DBSCAN algorithm, why could one expect it to be more appropriate that an isotropic prototype-based algorithm such as K-means, on this data set?
> 4. Apply the DBSCAN algorithm on the data set using the default hyperparameters proposed by sci-kit learn. Are the results the expected results?
> 5. Fix $min\_samples = 4$, and try multiple hyperparameter values for ϵ, how do the results change? Can you build intuition on how the number of found clusters changes with changes in ϵ values?
> 6. Think of protocols or heuristics that could help find an appropriate value for ϵ. Try them with some code.

```
# 2D plot of the data
def plot_with_labels(X, Y):
  N, M = X.shape
  X2d = X
  if M < 2:
    print("too low dimensionality")
    return
  elif M > 2:
    X2d = PCA(n_components=2).fit_transform(X)
  plt.scatter(X2d[:, 0], X2d[:, 1], c = Y)

# returns the number of unique values in the 1D numpy array Y
# can be useful in order to determine the number of clusters that were
found by a clustering algorithm
def N_unique_values(Y):
  return np.unique(Y).shape[0]

# building the dataset
circles_1, true_labels_1 = make_circles(n_samples = 1000, noise = 0.02,
factor=0.75)
circles_2, true_labels_2 = make_circles(n_samples = 500, noise = 0.04,
factor=0.5)
circles     = np.vstack((circles_1*1.2, circles_2 * 0.6))
true_labels = np.hstack((true_labels_1, true_labels_2+2))
```

```
# show the dataset using the true labels
# plot_with_labels(circles, true_labels)

# Your code here !
```

```
# apply K-means on the circles dataset and visualise the results
# With your own implementation of K-means or
# from sklearn.cluster import KMeans

# Your code here !
```

```
# apply the DBSCAN algorithm on the circles dataset and visualise the
results. Try using different hyperparameter values
from sklearn.cluster import DBSCAN # https://scikit-
learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

# Your code here !
```

Agglomerative hierarchical clustering

Given a data set and some distance function d, agglomerative hierarchical clustering initially considers every example as a cluster of its own. Then at each iteration, the two closest clusters are merged. The algorithm terminates as soon as there is only a single cluster remaining, containing the whole data set. In order to determine the two closest clusters at each iteration, the distance between two clusters C_1 and C_2 is defined as:

- $min_{x \in C_1, y \in C_2} d(x, y)$ in single link agglomerative clustering
- $max_{x \in C_1, y \in C_2} d(x, y)$ in complete-link agglomerative clustering
- $\frac{1}{|C_1|.|C_2|} \sum_{x \in C_1, y \in C_2} d(x, y)$ in average-link agglomerative clustering
  -$d(\frac{\sum_{x \in C_1} x}{|C_1|}, \frac{\sum_{x \in C_2} x}{|C_2|})$ in centroid agglomerative clustering

When applying agglomerative clustering, a hierarchy of clusters is defined and can be represented through a dendrogram.

> **Exercise 5** Implement the algorithm, with the four variants. Apply it on the `data_2d` data set and visualise the hierarchy using a dendrogram. Does the results differ from one variant to another?
>
> What is the time complexity of the algorithm? Compare the computation time when applying it to the `data_2d` data set and `X`, the data set from exercice 1. If a hierarchy of clusters was desired on a large, real-world data set, how would you mitigate the effect of the high temporal complexity of this algorithm?

```
data_2d = np.array([[4., 9.5], [2., 5.], [8., 6.], [5., 8.], [7., 7.],
[7., 6.], [1., 4.], [4.5, 9.]])
plt.scatter(data_2d[:, 0], data_2d[:, 1])
```

```
# Your code here !
```