INF283 - Exercise - 8

(**Exercise 8 deadline: 16th of November, 23.59**).
Submission details: Computer written or scanned pages.
Deliver here: https://mitt.uib.no/courses/12791/assignments

Weekly exercises are a compulsory part of the course. You will need to complete at least half of them. Weekly exercises give a total of 16 points to the final grade, of the total 8 exercises each then gives 2 points to you final grade, as long as you upload your answer to MittUIB.no/assignments before 23.59 on Fridays. They will then be reviewed, and points are added to your total grade score (If we see you have made an effort. So no score loss if your answer had some error in the calculation etc). If you have completed only a fraction of the tasks then you will get a fraction of 2 points.

If you follow these exercises and ask for help when you need it during the group sessions, it will help you a lot, especially through the more difficult parts of the course.

In this exercise we will learn clustering: the Lloyd algorithm of K-means, Mixture of Gaussians, and how we can use it in applications. You can use your output plots as answers for most of these questions.

K-means

***k*-means clustering** is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. *k*-means clustering aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. In machine learning is categorized as an unsupervised method.

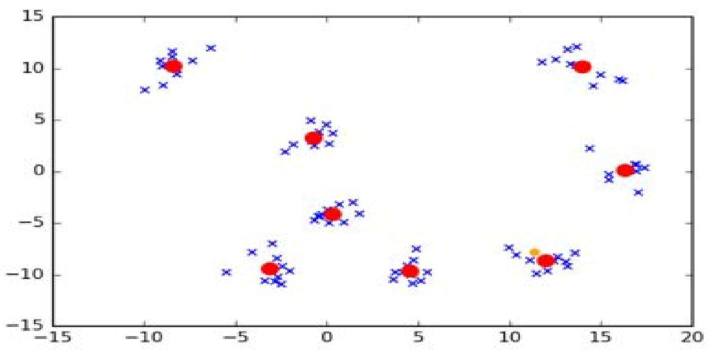

Figure 1: A clustering of K = 7, it finds the group by making the mean distance between the red circles and it's predicted group (the blue x's around the red circle) as small as possible.

Lloyd’s algorithm

Finding the solution of *k*-means clustering is unfortunately NP hard. Nevertheless, an iterative method known as Lloyd’s algorithm exists that converges (albeit to a local minimum) in few steps.

1.1 Implement Lloyds algorithm from scratch (it is not that hard):
HINT: python example with almost correct implementation here. You can make a nice dataset with sklearn.datasets.make_blobs(n_samples=n_samples, random_state=170).

1.2 Compare your model with a known implementation like: sklearn.cluster.KMeans
Note: this function can use Lloyd’s algorithm with KMeans(algorithm = “full”), it can also use a faster version called “elkan” that uses Elkan’s algorithm.

```
import numpy as np; import matplotlib.pyplot as plt
from sklearn import cluster, datasets, mixture

np.random.seed(0)
# ============
# Generate datasets
# ============
n_samples = 1500
X, y = datasets.make_blobs(n_samples=n_samples, random_state=170)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X, y = (np.dot(X, transformation), y)

# your method ->
centers = kmeans(X, 3)
colors = [] # colors for plot
new_centers = [] # reshape, the old way
for i in range(len(centers)):
  colors.extend([i] * len(centers[i]))
  new_centers.extend(centers[i])

# sklearn version
centersSKL = cluster.MiniBatchKMeans(n_clusters=3)
centersSKL.fit(X)
y_pred = centersSKL.predict(X)

# ============
# Set up figure settings
# ============
plt.figure(figsize=(9 * 2 + 3, 12.5))
x = [ x[0] for x in new_centers]
y = [ x[1] for x in new_centers]

plt.subplot(1, 2, 1) # your subplot
plt.scatter(x, y, c=colors)
plt.xticks(());plt.yticks(())

plt.subplot(1, 2, 2) # sklearn subplot
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.xticks(());plt.yticks(())
plt.show()
```

1.3 Explain why it could work to use random initialization of the centroids (the red circles in figure 1).
NOTE: Normally in the algorithm, there is some clever trick to find good start positions, like the Forgy and the Random Partition method.

Mixture of Gaussians

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. One can think of mixture models as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

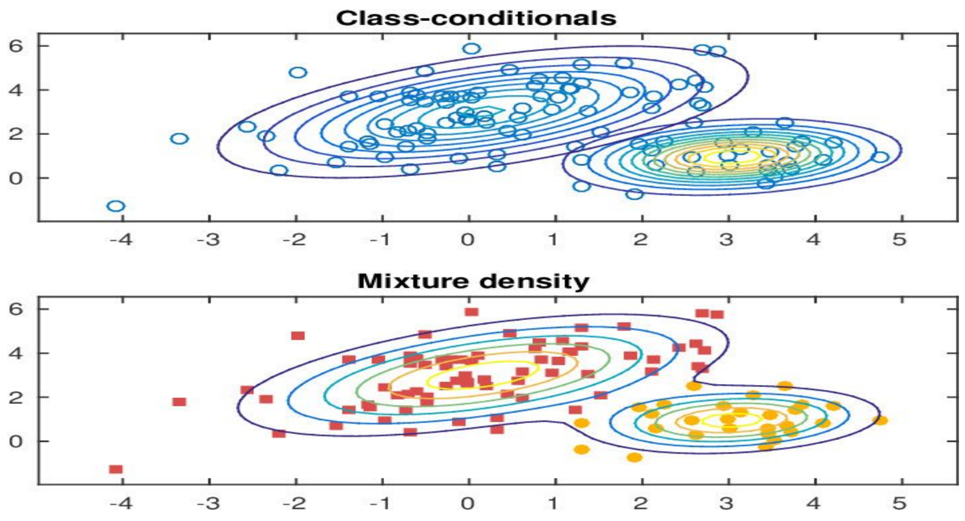

Figure 2: A representation of gaussian mixture with 2 classes.

2.1 Use a known implementation of mixture of gaussians: For example: sklearn.mixture.GaussianMixture and discuss when performs it better than sklearn.cluster.KMeans.
Python Example (all needed code):

```
import numpy as np; import matplotlib.pyplot as plt
from sklearn import cluster, datasets, mixture
from sklearn.preprocessing import StandardScaler

np.random.seed(0)
# ============
# Generate datasets
# ============
n_samples = 1500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=.5,
                                      noise=.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None
```

```
# Anisotropicly distributed data
X, y = datasets.make_blobs(n_samples=n_samples, random_state=170)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
aniso = (np.dot(X, transformation), y)

# The datasets, shown as rows
datasets = [
   (noisy_circles, {'damping': .77, 'preference': -240,
                    'quantile': .2, 'n_clusters': 2}),
   (noisy_moons, {'damping': .75, 'preference': -220, 'n_clusters': 2}),
   (aniso, {'eps': .15, 'n_neighbors': 2}),
   (blobs, {}),
   (no_structure, {})]

# ============
# Create cluster objects
# ============
two_means = cluster.MiniBatchKMeans(n_clusters=3)
gmm = mixture.GaussianMixture(
   n_components=3, covariance_type='full')
# the algorithms shown as colors
clustering_algorithms = (
   ('MiniBatchKMeans', two_means), ('GaussianMixture', gmm)
)
# ============
# Set up figure settings
# ============
plt.figure(figsize=(9 * 2 + 3, 12.5)); plot_num = 1
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

for i_dataset, (dataset, algo_params) in enumerate(datasets):
   # split in data and classes
   X, y = dataset
   # normalize dataset for easier parameter selection
   X = StandardScaler().fit_transform(X)

   for name, algorithm in clustering_algorithms:
       algorithm.fit(X)

       y_pred = algorithm.predict(X)

       # make plot fancy
       plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
       if i_dataset == 0:
           plt.title(name, size=18)
       plt.scatter(X[:, 0], X[:, 1], s=10, c=y_pred)
       plt.xlim(-2.5, 2.5);plt.ylim(-2.5, 2.5)
       plt.xticks(());plt.yticks(())
       plot_num += 1
plt.show()
```

Hierarchical Clustering

In data mining and statistics, hierarchical clustering (also called **hierarchical cluster analysis** or **HCA**) is a method of cluster analysis which seeks to build a hierarchy of clusters.

Biological hierarchical example

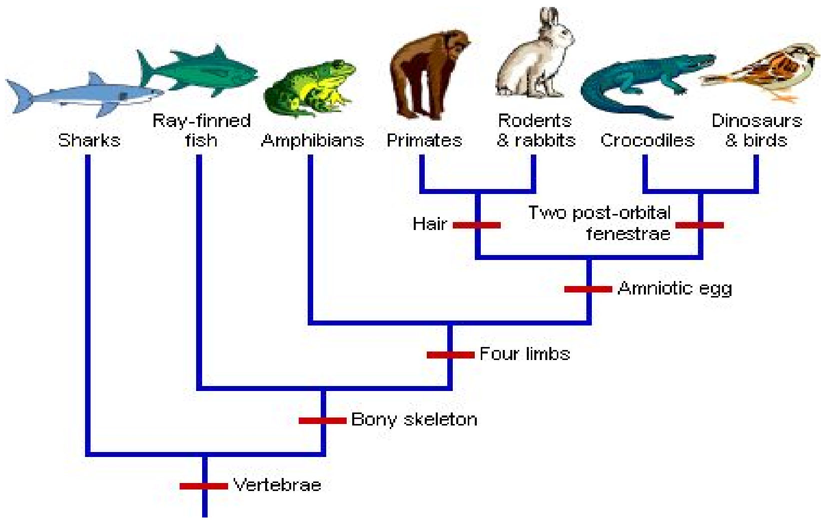

Figure 3: hierarchical representation of evolutionary relatedness between species.

The gene ATP5A1 is a very important gene used in energy production in all species shown in figure 3. A gene consists of DNA, an alphabet of A,T,C or G. For example:
ATG AGG ATT TGA AGA TTC CAA

Even though all the species in figure 2 have this gene, they do not have the same dna sequence of the gene, because of natural selection (mutations in the DNA increasing fitness).

For example:
Primates: ACG AGG G
Rodents: ACG AGG A

Sharks: ATG AAA A

Here you see that primates and rodents are more similar, than primates and sharks etc.

***3.1** Use a* hierarchical clustering algorithm like **scipy.cluster.hierarchy** , and show the plot of the resulting tree based on the code below.
Hint: hierarchical clustering algorithms uses numbers for calculating distance, therefore we can not use the dna strings in “data” directly, we need to make a matrix of integers, where each integer is the number of differences between each word. This is what your phone does,

when it tries to guess what you try to spell. The problem is called edit distance, and the most famous algorithm is called the Levenshtein distance algorithm.

```
data = ["ATGTAAA", "ATGAAAA", "ACGTGAA", "ACGAGGG", "ACGAGGA", "ACGAGTC", "ACGAGCC"]
labels = ["shark", "ray-finned fish", "amphibians", "primates", "rodents", "crocodiles", "dinosaurs"]
```

Procedure; There are 3 steps in hierarchical clustering:

1. Choosing metric: We will use levenshtein edit distance as metric
2. clustering method: called linkage method, scikit default is “ward”
3. print dendrogram: output the clustering as a dendrogram (tree structure)

```
####################################
# code: imports and data ->
import scipy.cluster.hierarchy as hier
from matplotlib import pyplot as plt
from scipy.spatial.distance import pdist
import numpy as np
import distance # <- you most likely need pip install distance here
data = ?
labels = ?

# The levenshtein distance matrix ->
mat = np.zeros((len(data), len(data)),  dtype=int)
for i in range(0, len(data)):
  for j in range(0, len(data)):
     mat[i][j] = distance.levenshtein(data[i], data[j])
     print(mat[i][j], end="")
  print("\n")
mat = pdist(mat) # make an upper triangle matrix

# The hierarchy clustering ->
# here run scipy.cluster.hierarchy.linkage() on triangle matrix
z = ?
fig = plt.figure(figsize=(25, 10))
# here run scipy.cluster.hierarchy.dendrogram() with the linkage z and labels = labels
dn =?
plt.show()
```

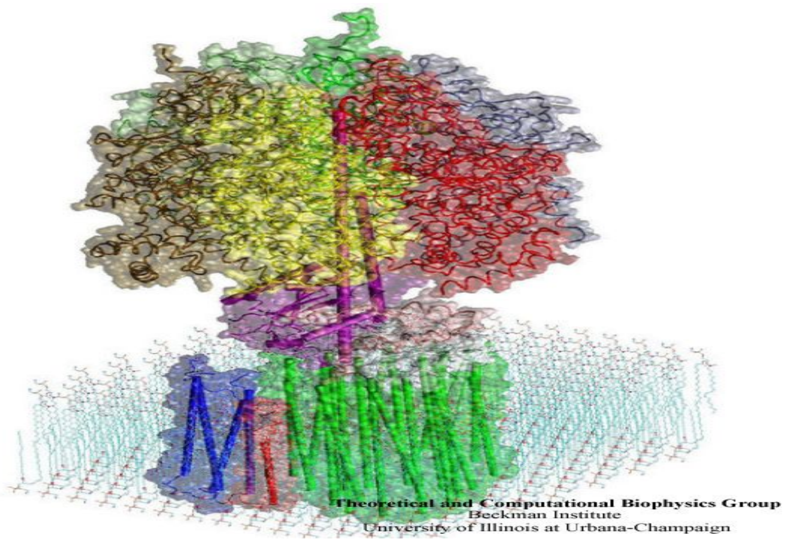
Theoretical and Computational Biophysics Group
Beckman Institute
University of Illinois at Urbana-Champaign

Figure 4: The resulting protein complex of ATPase. The information in DNA is read to create RNA, which is read to create the protein. ATP5A1 is only a subunit of ATPase (the yellow part). ATPase is a highly efficient proton pump, creating energy from degraded food.