Techniques of Clustering with Python Programming

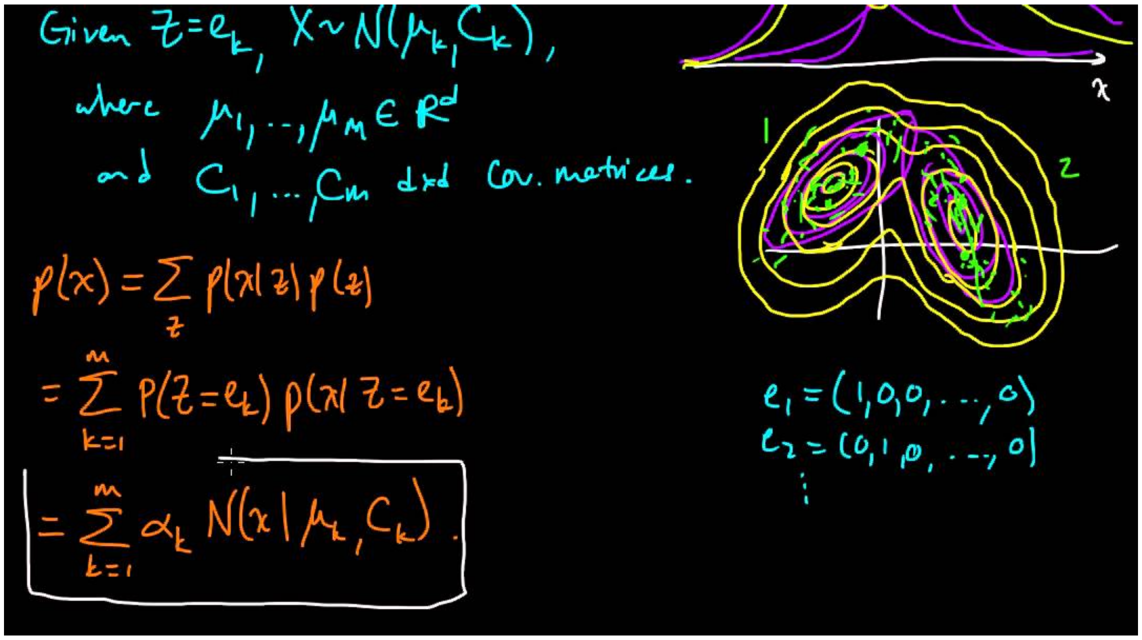
Chrysovalantou Kalaitzidou

April 2017

University of Crete Faculty of Medicine
Bioinformatics M.Sc.

Methods in Bioinformatics

Second Assignment



Given $z = e_k$, $X \sim N(\mu_k, C_k)$,

where $\mu_1, \ldots, \mu_M \in \mathbb{R}^d$

and $C_1, \ldots, C_m$ $d \times d$ cov. matrices.

$$p(x) = \sum_{z} p(x|z)\,p(z)$$

$$= \sum_{k=1}^{m} P(z = e_k)\,p(x|z = e_k)$$

$$\boxed{= \sum_{k=1}^{m} \alpha_k N(x|\mu_k, C_k).}$$

$e_1 = (1, 0, 0, \ldots, 0)$
$e_2 = (0, 1, 0, \ldots, 0)$
$\vdots$

Contents

1 Introduction . . . . **iii**

2 Theoretical Background **1**
2.1 Clustering 1
2.1.1 Distance Measure 1
2.1.2 Clustering Algorithms 3
2.2 K-Means Algorithm 4
2.3 Mixtures of Gaussians Algorithm 5
2.3.1 EM Algorithm 7

3 Analysis **9**
3.1 Theoretical Problem 9
3.2 Practical Problem 16

Appendices **21**

A Python Codes **21**

B Suggested Bibliography **37**

1 Introduction

The current report is part of the second assignment in class *Methods in Bioinformatics* and refers to ***Clustering*** and, in particular, two of its methods ***K Means*** and ***Mixture of Gaussians***. Certain data sets have been provided for analysis with each method implemented in Python programming language. In the following Chapters the reader can find some Theoretical Notes for the methods and their algorithms for the implementation, the Results derived from data sets' analysis along with some discussion and, at last, the Python Code.
The data set that has been given for the *Practical Part* of analysis, acquired from *NCBI* database, refers to organism *Mus musculus* and a particular study of livers of $C57BL/6J$ *mice* fed a high fat diet for up to 24 weeks. Significant body weight gain was observed after 4 weeks. Their results provide insight into the effect of high fat diets on metabolism in the liver. For more information, please visit https://www.ncbi.nlm.nih.gov/sites/GDSbrowser?acc=GDS6248

Chrysovalantou Kalaitzidou

2 Theoretical Background

2.1 Clustering

Cluster Analysis or ***Clustering*** is the task of grouping a set of objects in such a way that objects in the same group, ***cluster*** are more similar, in some sense or another, to each other than to those in other groups (clusters). It is a main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, bioinformatics, data compression, and computer graphics.

Cluster analysis itself is not one specific algorithm, but the general task to be solved. It can be achieved by various algorithms that differ significantly in their notion of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances among the cluster members, dense areas of the data space, intervals or particular statistical distributions. Clustering can therefore be formulated as a *multi-objective optimization problem.* The appropriate clustering algorithm and parameter settings, including values such as the distance function to use, a density threshold or the number of expected clusters, depend on the individual data set and intended use of the results. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.
As we mentioned above, a cluster is therefore a collection of objects which are **similar** to each other and are **dissimilar** to the objects belonging to other clusters. Cluster analysis is also used to form descriptive statistics to ascertain whether or not the data consists of a set distinct subgroups, each group representing objects with substantially different properties. The latter goal requires an assessment of the degree of difference between the objects assigned to the respective clusters. Central to clustering is to decide what constitutes a good clustering. This can only come from subject matter considerations and there is no absolute "best" criterion which would be independent of the final aim of the clustering. For example, we could be interested in finding representatives for homogeneous groups (data reduction), in finding "natural clusters" and describe their unknown properties ("natural" data types), in finding useful and suitable groupings ("useful" data classes) or in finding unusual data objects (outlier detection).
Two important components of cluster analysis are the similarity **distance** measure between two data samples and the **clustering algorithm.**

2.1.1 Distance Measure

Different formula in defining the distance between two data points can lead to different classification results. Domain knowledge must be used to guide the formulation of a suitable distance measure for each particular application.
For high dimensional data, a popular measure is the ***Minkowski Metric***:

$$d(x_i, x_j) = \left(\sum_{k=1}^{D} |x_{i,k} - x_{j,k}|^p \right)^{\frac{1}{p}}$$

where D is the dimensionality of the data. Special cases:

- $p = 2$: *Euclidean Distance*
- $p = 1$: *Manhattan Distance*
- $p = \infty$: and taking a limit, we gain *Chebyshev Distance* :

$$d(x_i, x_j) = max_i \Big(|x_i - x_j| \Big), \forall i \in [1, D]$$

Another important measure is ***Mahalanobis Distance***:

$$D_M(\vec{x}) = \sqrt{(\vec{x} - \vec{\mu})^T S^{-1} (\vec{x} - \vec{\mu}))}$$

which is the distance of an observation $\vec{x} = (x_1, ..., x_n)^T$ from a set of observations with mean $\vec{\mu} = (\mu_1, ..., \mu_n)^T$ and covariance matrix S.
The Mahalanobis distance is a measure of the distance between a point P and a distribution D. It is a multi-dimensional generalization of the idea of measuring how many standard deviations away P is from the mean of D. This distance is zero if P is at the mean of D, and grows as P moves away from the mean. Along each principal component axis, it measures the number of standard deviations from P to the mean of D. If each of these axes is rescaled to have unit variance, then Mahalanobis distance corresponds to standard *Euclidean Distance* in the transformed space. Mahalanobis distance is thus unitless and scale-invariant, and takes into account the correlations of the data set. Defined as a dissimilarity measure between two random vectors of the same distributio:

$$d(x_i, x_j) = \sqrt{(x_i - x_j)^T S^{-1} (x_i - x_j)}$$

where S is the covariance matrix and the $x_{i,j}$ are vectors of size $Dx1$. If the covariance matrix is the identity matrix, the Mahalanobis distance reduces to the *Euclidean distance.* If the covariance matrix is diagonal, then the resulting distance measure is called a *normalized Euclidean distance*:

$$d(x_i, y_i) = \sqrt{\sum_{i=1}^{N} \frac{(x_i - y_i)^2}{s_i^2}}$$

where s_i is the standard deviation of the x_i and y_i over the sample set. Mahalanobis distance is preserved under full-rank linear transformations of the space spanned by the data. This means that if the data has a nontrivial nullspace, Mahalanobis distance can be computed after projecting the data (non-degenerately) down onto any space of the appropriate dimension for the data. Mahalanobis distance is widely used in cluster analysis and classification techniques. In order to classify a test point as belonging to one of N classes, one first estimates the covariance matrix of each class, usually based on samples known to belong to each class. Then, given a test sample, one computes the Mahalanobis distance to each class, and classifies the test point as belonging to that class for which the Mahalanobis distance is minimal.

2.1.2 Clustering Algorithms

Clustering algorithms may be classified as listed below:

- Exclusive Clustering :
  In exclusive clustering data are grouped in an exclusive way, so that a certain datum belongs to only one definite cluster. **K-means clustering** is one example of the exclusive clustering algorithms.

- Overlapping Clustering :
  The overlapping clustering uses fuzzy sets to cluster data, so that each point may belong to two or more clusters with different degrees of membership.

- Hierarchical Clustering :
  Hierarchical clustering algorithm has two versions: Agglomerative clustering and Divisive clustering. Agglomerative clustering is based on the union between the two nearest clusters. The beginning condition is realized by setting every datum as a cluster. After a few iterations it reaches the final clusters wanted. Basically, this is a bottom-up version.
  Divisive clustering starts from one cluster containing all data items. At each step,clusters are successively split into smaller clusters according to some dissimilarity. Basically this is a top-down version.

- Probabilistic Clustering :
  Probabilistic clustering, e.g. **Mixture of Gaussian**, uses a completely probabilistic approach.

All the clustering analysis methods introduced above are examples of **unsupervised learning algorithms**. A learning method is considered unsupervised if it learns in the absence of a teacher signal that provides prior knowledge of the correct answer. Supervised learning has a substantial advantage over unsupervised learning. In particular, supervised learning allows us to take advantage of our own knowledge about the classification problem we are trying to solve. Instead of just letting the algorithm work out for itself what the classes should be, we can tell it what we know about the classes: how many there are and what examples of each one look like. The supervised learning algorithm's job is then to find the features in the examples that are most useful in predicting the classes.

In this current project, we make use of two specific Clustering Algorithms, ***K Means*** and ***Mixture of Gaussians***.

2.2 K-Means Algorithm

K-means is one of the simplest unsupervised learning algorithms that solve the well known clustering problem. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed a priori. The main idea is to define k centroids, one for each cluster. These centroids should be placed in a cunning way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done. In other words centroids do not move any more. Finally, this algorithm aims at minimizing an *objective function*, in this case a squared error function. The objective function

$$J = \sum_{j=1}^{k} \sum_{i=1}^{N} \|x_{i,j} - c_j\|^2$$

where $\|x_{i,j} - c_j\|^2$ is a chosen distance measure between a data point $x_{i,j}$ and the cluster centre c_j, is an indicator of the distance of the n data points from their respective cluster centres.

The algorithm is composed of the following steps:

- Place K points into the space represented by the objects that are being clustered. These points represent initial group centroids.
- Assign each object to the group (cluster) that has the closest centroid.
- When all objects have been assigned, recalculate the positions of the K centroids.This is done by taking the mean of all data points assigned to that centroid's cluster.
- The algorithm iterates between steps two and three until a stopping criteria is met: either the centroids no longer move, no data points change clusters, the sum of the distances is minimized, or some maximum number of iterations is reached. This produces a separation of the objects into clusters from which the metric to be minimized can be calculated.

Although it can be proved that the procedure will always terminate, the **K Means** algorithm does not necessarily find the most optimal configuration, corresponding to the global objective function minimum. In addition to that, outliers can cause considerable trouble. The algorithm is also significantly sensitive to the initial randomly selected cluster centres. Meaning that assessing more than one run of the algorithm with randomized starting centroids may give a better outcome.

Initialization methods:

Commonly used initialization methods are $Forgy$ and *Random Partition*. The Forgy method randomly chooses k observations from the data set and uses these as the initial means. The Random Partition method first randomly assigns a cluster to each observation and then proceeds to the update step, thus computing the initial mean to be the centroid of the cluster's randomly assigned points. The Forgy method tends to spread the initial means out, while Random Partition places all of them close to the center of the data set. The Random Partition method is generally preferable for algorithms such as the *K-harmonic means* and *fuzzy K-means*. For $Expectation\ Maximization$ and *standard k-means* algorithms, the Forgy method of initialization is preferable.

2.3 Mixtures of Gaussians Algorithm

The ***Mixture of Gaussians*** is among the most enduring, well weathered models of applied statistics. A widespread belief in its fundamental importance has made it the object of close theoretical and experimental study for over a century. In a typical application, sample data are thought of as originating from various possible sources, and the data from each particular source is modelled by a Gaussian. This choice of distribution is common in the physical sciences and finds theoretical corroboration in the central limit theorem. Given mixed and unlabelled data from a weighted combination of these sources, the goal is to identify the generating mixture of Gaussians, that is, the nature of each Gaussian source, its mean and covariance, and also the ratio in which each source is present, known as its 'mixing weight'.
Modern methods delegate the bulk of the work to computers, and amongst them the most popular appears to be the Expectation Magimization Algorithm formalized by Dempster, Laird, and Rubin (1977). EM is a local search heuristic of appealing simplicity. Its principal goal is convergence to a local maximum in the space of Gaussian mixtures ranked by likelihood.

The Gaussian mixture distribution can be written as a linear superposition of Gaussians in the form:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \tag{1}$$

For a D dimensional vector x_n each Gaussian density $\mathcal{N}(x|\mu_k, \Sigma_k)$ is called a *component* of the mixture and has its own mean μ_k ,which is a D dimensional vector, and DxD covariance matrix Σ_k :

$$\mathcal{N}(x|\mu_k, \Sigma_k) = \frac{1}{\sqrt{(2\pi)^D |\Sigma_k|}} exp\Bigg\{ -\frac{1}{2}(x_n - \mu_k)^T \Sigma_k^{-1} (x_n - \mu_k) \Bigg\} \tag{2}$$

The parameters π_k are called *mixing coefficients.* If we integrate both sides of with respect to x_n, and note that both $p(x)$ and the individual Gaussian components are normalized, we obtain

$$\sum_{k=1}^{K} \pi_k = 1 \tag{3}$$

. Also, the requirement that $p(x) \geq 0$, together with $\mathcal{N}(x|\mu_k, \Sigma_k) \geq 0$, implies that $\pi_k \geq 0$ for all k. Combining all these together we obtain $0 \leq \pi_k \leq 1$

We therefore see that the mixing coefficients satisfy the requirements to be probabilities. From the sum and product rules, the marginal density is given by

$$p(x) = \sum_{k=1}^{K} p(k)p(x|k) \tag{4}$$

which is equivalent to (1) in which we can view $\pi_k = p(k)$ as the prior probability of picking the k_{th} component, and the density $\mathcal{N}(x|\mu_k, \Sigma_k) = p(x|k)$ as the probability of x conditioned on k. Of great importance though, is the posterior probabilities $p(k|x)$, which are also known as responsibilities. From Bayes' theorem these are given by

$$\gamma_k(x) \equiv p(k|x) = \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(x|\mu_j, \Sigma_j)} \tag{5}$$

The form of the Gaussian mixture distribution is governed by the parameters π, μ and Σ, where we have used the notation $\pi = (\pi_1, ..., \pi_K)$, $\mu = (\mu_1, ...\mu_K)$ and $\Sigma = (\Sigma_1, ...\Sigma_K)$. One way to set the values of these parameters is to use *maximum likelihood* . From (1) the log of the likelihood function is given by

$$lnp(X|\pi, \mu, \Sigma) = \sum_{n=1}^{N} ln\left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \right\} \tag{6}$$

where $X = (x_1 ... x_N)$.
We turn now to a formulation of Gaussian mixtures in terms of discrete *latent* variables. This will provide us with a deeper insight into this important distribution, and will also serve to motivate the expectation maximization algorithm. Let us introduce a K dimensional binary random variable z having a $1-of-K$ representation in which a particular element z_k is equal to 1 and all other elements are equal to 0. The values of z_k therefore satisfy $z_k \in \{0, 1\}$ and $\sum_k z_k = 1$, and we see that there are K possible states for the vector z according to which element is nonzero. We shall define the *joint distribution* $p(x, z)$ in terms of a *marginal distribution p(z)* and a *conditional distribution* $p(x|z)$. The marginal distribution over z is specified in terms of the mixing coefficients π_k , such that

$$p(z_k = 1) = \pi_k \tag{7}$$

where the parameters $\{\pi_k\}$ must sutisfy $0 \leq \pi_k \leq 1$ together with (5) in order to be valid probabilities. Because z uses a $1 - of - K$ representation, we can also write this distribution in the form

$$p(z) = \prod_{k=1}^{K} \pi_k^{z_k} \tag{8}$$

Similarly, the conditional distribution of x given a particular value for z is a Gaussian

$$p(x|z_k = 1) = \mathcal{N}(x|\mu_k, \Sigma_k)$$

which can also be written in the form

$$p(x|z) = \prod_{k=1}^{K} \mathcal{N}(x|\mu_k, \Sigma_k)^{z_k} \tag{9}$$

The joint distribution is given by $p(z)p(x|z)$, and the marginal distribution is then obtained by summing the joint distribution over all possible states of z to give

$$p(x) = \sum_z p(z)p(x|z) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \tag{10}$$

where we have made use of (8) and (9). Thus the marginal distribution of x is a *Gaussian Mixture* of the form (1). If we have several observations $x_1, ... x_N$ then, because we have represented the marginal distribution in the form $p(x) = \sum_z p(x, z)$ it follows that for every observed data point x_n there is a corresponding latent variable z_n.
We have therefore found an equivalent formulation of the Gaussian mixture involving an explicit latent variable. We are now able to work with the joint distribution $p(x, z)$ instead of the marginal distribution $p(x)$, and this will lead to significant simplifications, most notably through the introduction of the *Expectation Maximization* (*EM*) Algorithm.

Another quantity that will play an important role is the *conditional probability* of z given x. We shall use $\gamma(z_k)$ to denote $p(z_k = 1|x)$, whose value can be found using Bayes' theorem:

$$\gamma(z_k) \equiv p(z_k = 1|x) = \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_j \pi_j \mathcal{N}(x|\mu_j, \Sigma_j)} \tag{11}$$

We shall view π_k as the prior probability of $z_k = 1$, and the quantity $\gamma(z_k)$ as the corresponding posterior probability once we have observed x.
$\gamma(z_k)$ can also be viewed as the *responsibility* that component k takes for 'explaining' the observation x.

2.3.1 EM Algorithm

Suppose we have a data set of observations $\{x_1 \ldots x_N\}$ and we wish to model this data using a mixture of Gaussians. We can represent this data set as an $D \times N$ matrix X in which the n_{th} column is given by x_n. Similarly, the corresponding latent variables will be denoted by an $K \times N$ matrix Z with columns z_n. From (6) the *log likelihood* function is given by

$$lnp(X|\pi, \mu, \Sigma) = \sum_{n=1}^{N} ln\left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \right\}$$

Maximizing the log likelihood function for a Gaussian mixture model turns out to be a more complex problem than for the case of a single Gaussian. The difficulty arises from the presence of the summation over k that appears inside the logarithm, so that the logarithm function no longer acts directly on the Gaussian. If we set the derivatives of the log likelihood to zero, we will no longer obtain a closed form solution

An elegant and powerful method for finding maximum likelihood solutions for models with latent variables is called the ***expectation-maximization algorithm, EM algorithm***. Let us begin by writing down the conditions that must be satisfied at a maximum of the likelihood function. Setting the derivatives of $lnp(X|\pi, \mu, \Sigma)$ with respect to the means μ_k , the covariance matrices Σ_k and the mixture coefficients π_k of the Gaussian components to zero, we obtain respectively:

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n \tag{12}$$

where we have defined

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \tag{13}$$

We can interpret N_k as the effective number of points assigned to cluster k.
We see that the mean μ_k for the k_{th} Gaussian component is obtained by taking a *weighted mean* of all of the points in the data set, in which the weighting factor for data point x_n is given by the posterior probability $\gamma(z_{nk})$ so that component k was responsible for generating x_n .

If we set the derivative with respect to Σ_k to zero, we have:

$$\Sigma_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k)(x_n - \mu_k)^T \tag{14}$$

Finally, maximizing with respect to the mixing coefficients π_k and taking account of (5) :

$$\pi_k = \frac{N_k}{N} \tag{15}$$

so that the mixing coefficient for the k_{th} component is given by the *average responsibility* which that component takes for explaining the data points.

It is worth emphasizing that the results (12), (14), and (15) do not constitute a closed-form solution for the parameters of the mixture model because the responsibilities $\gamma(z_{nk})$ depend on those parameters in a complex way through (11). However, these results do suggest a simple iterative scheme for finding a solution to the maximum likelihood problem, which as we shall see turns out to be an instance of the EM algorithm for the particular case of the Gaussian mixture model.
We first choose some initial values for the means, covariances, and mixing coefficients. Then we alternate between the following two updates that we shall call the *E step* and the *M step*, for reasons that will become apparent shortly. In the expectation step, or E step, we use the current values for the parameters to evaluate the posterior probabilities, or responsibilities, given by (11).

We then use these probabilities in the maximization step, or M step, to re-estimate the means, covariances, and mixing coefficients using the results (12), (14), and (15). Note that in so doing we first evaluate the new means using (12) and then use these new values to find the covariances using (14), in keeping with the corresponding result for a single Gaussian distribution. We shall show that each update to the parameters resulting from an E step followed by an M step is guaranteed to increase the log likelihood function. In practice, the algorithm is deemed to have converged when the change in the log likelihood function, or alternatively in the parameters, falls below some threshold.

Note that the *EM* algorithm takes many more iterations to reach (approximate) convergence compared with the *K-means algorithm*, and that each cycle requires significantly more computation. It is therefore common to run the K means algorithm in order to find a suitable initialization for a Gaussian mixture model that is subsequently adapted using EM. The covariance matrices can conveniently be initialized to the sample covariances of the clusters found by the K means algorithm, and the mixing coefficients can be set to the fractions of data points assigned to the respective clusters.

EM Algorithm for Gaussians Mixture

Given a Gaussian mixture model, the goal is to maximize the likelihood function with respect to the parameters (comprising the means and covariances of the components and the mixing coefficients).

- Initialize the means μ_k , covariances Σ_k and mixing coefficients π_k, and evaluate the initial value of the log likelihood.
- **E Step**
  Evaluate the responsibilities using the current parameter values

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_n|\mu_j, \Sigma_j)} \tag{16}$$

- **M Step**
  Re-estimate the parameters using the current responsibilities:

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n \tag{17}$$

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k^{new})(x_n - \mu_k^{new})^T \tag{18}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{19}$$

  where

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \tag{20}$$

- Evaluate the log likelihood

$$lnp(X|\pi, \mu, \Sigma) = \sum_{n=1}^{N} ln \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k) \right\} \tag{21}$$

  and check for convergence of either the parameters or the log likelihood. If the convergence criterion is not satisfied return to step two.

3 Analysis

3.1 Theoretical Problem

In this part we work with a set of random normally distributed data, of $D = 2$ dimensions and $N = 500$ observations. The first 220 observations are generated from a multivariate Gaussian with $\mu_1 = [1 \quad 1]^T$ and $\Sigma_1 = 0.5I$, while the last 280 observations come from a Gaussian with $\mu_2 = [-1 \quad -1]$ and $\Sigma_2 = 0.75I$. We do not have any labels.

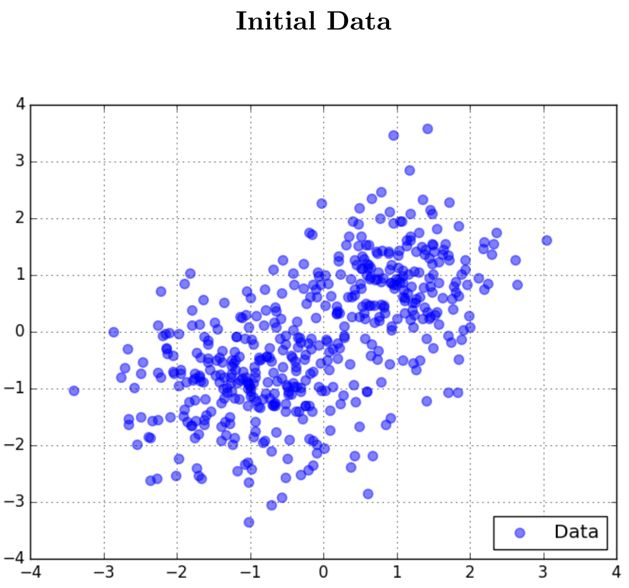


Figure 1: Random normally distributed

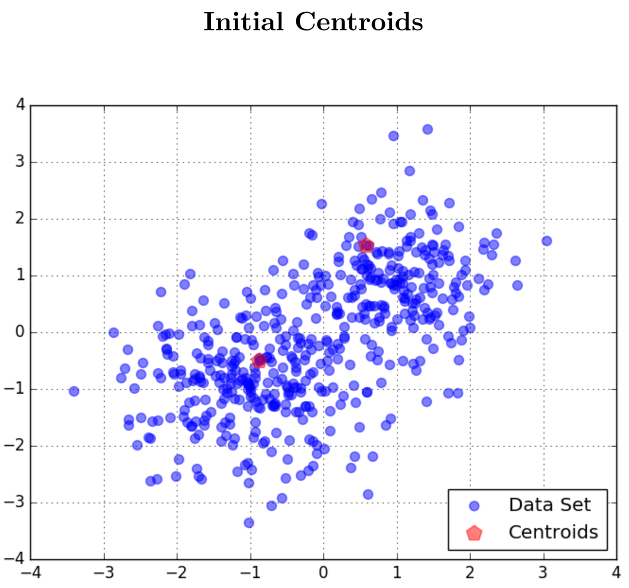


Figure 2: Initial Centroids randomly chosen from data set

We make use of **K Means Algorithm** and **Mixture of Gaussians** to separate our data in k clusters. For the K Means, we use three distance measures *Euclidean, Manhattan and Mihalanobis.*

<u>K Means</u>

As we mentioned above, this algorithm follows an easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed *a priori*. So, we expect to see the data separated in the specific number of clusters we initially "enforced" the algorithm to do so. Here are some examples, with different number of k, as well as with different distance measure for its time:

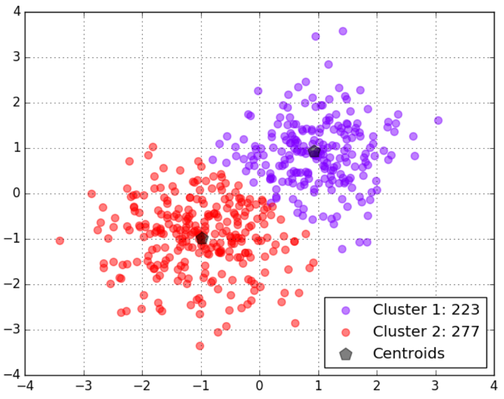


Figure 3: $k = 2$ with Euclidean Distance

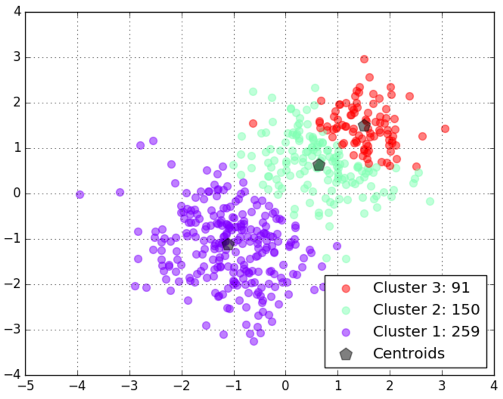


Figure 4: k = 3 with Manhattan Distance

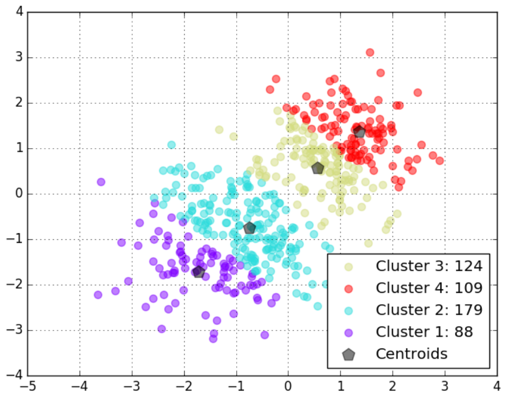


Figure 5: $k = 4$ with Mahalanobis Distance

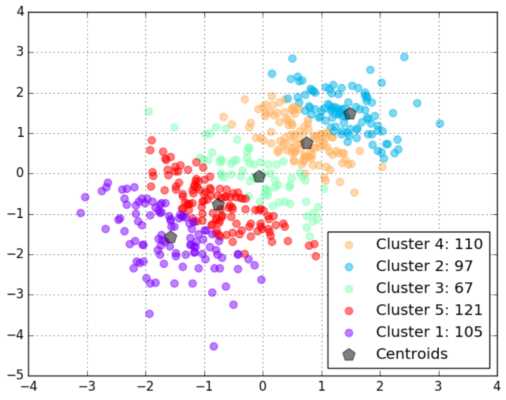


Figure 6: $k = 5$ with Euclidean Distance

All these plot figures gained after 5 replications of the algorithm, with the convergence achieved at different number of iterations each time, with the maximum limit of 30 iterations. It is clear that K $Means$ succeeds to form the clustering of our desire.

<u>Gaussian Mixture</u>

On the other hand, it is very interesting to see how *Mixture of Gaussians* behave. We follow the same initial steps here, by giving initial values to the parameters and a specific number k of the clustering we wish to obtain at each run. Since we refer to a probabilistic algorithm we expect to see the *probability* of each observation to belong in certain clusters.

k = 3 at different replications

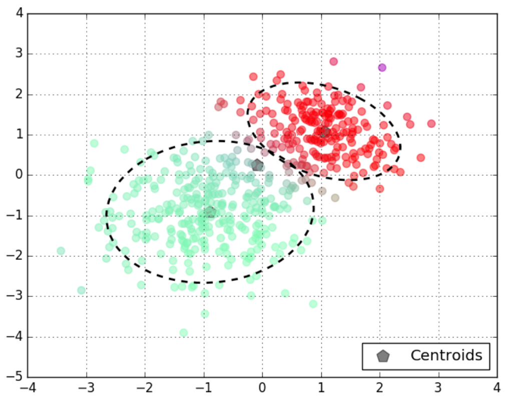


Figure 7: After one replication

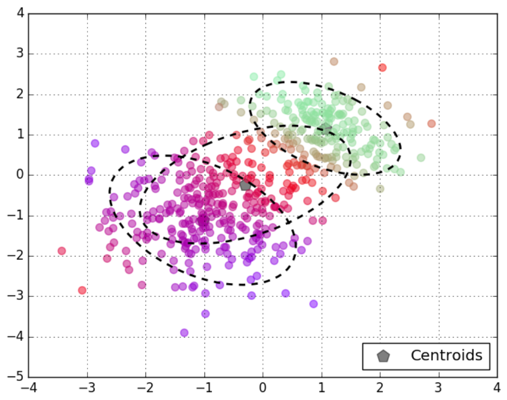


Figure 8: After three replications

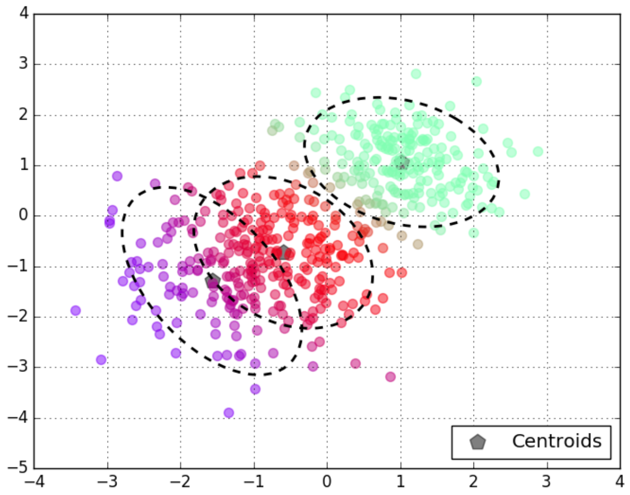


Figure 9: After five replications

It is clear that there are some observations with clean colouring while others, found between two or even three clusters, gain intermediate colour. What we can also observe is the 'tendency' to have 2 clusters, despite the fact that we asked for more. This becomes more certain as we try for bigger k:

k = 4 at different replications

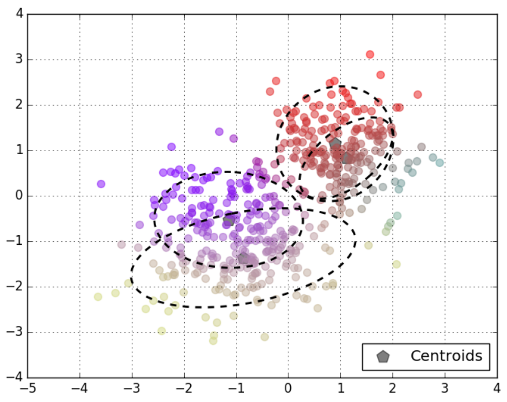


Figure 10: After one replication

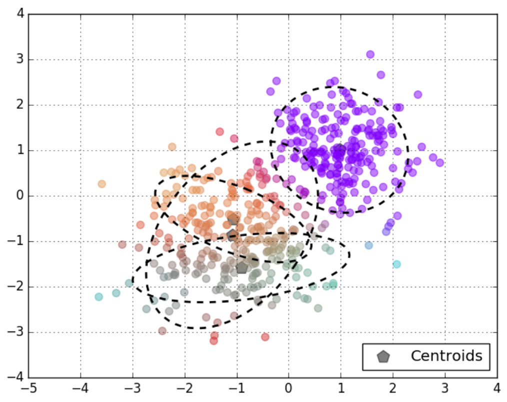


Figure 11: After two replications

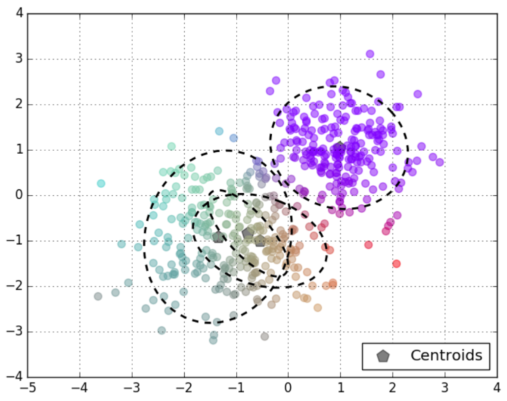


Figure 12: After three replications

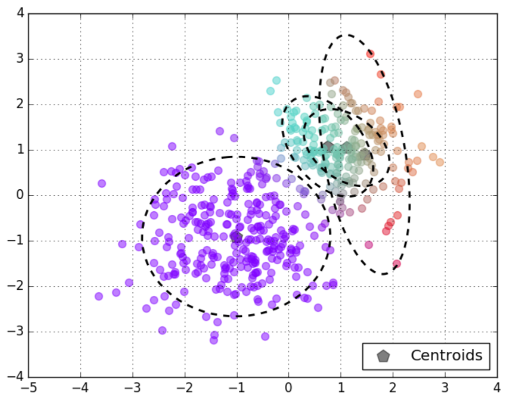


Figure 13: After four replications

k = 4, Final Clustering after 5 replications

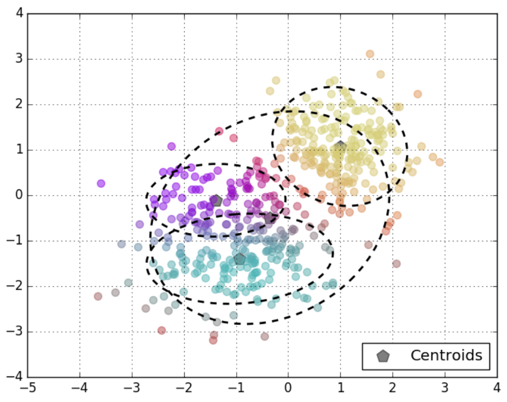


Figure 14: Responsibilities Plot

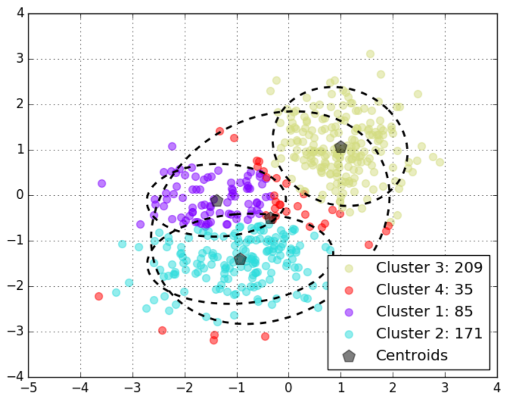


Figure 15: Labeled Plot

k = 5 at different replications

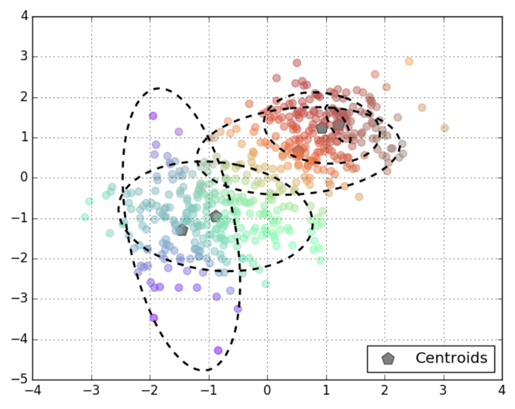


Figure 16: After one replication

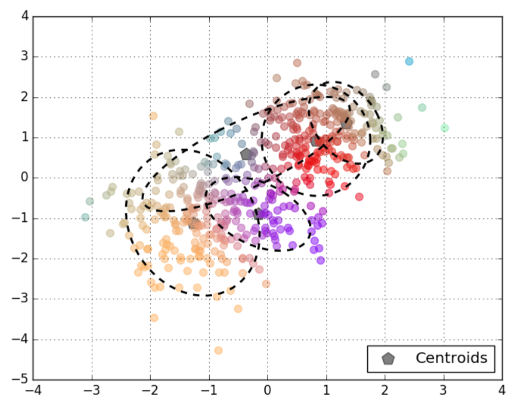


Figure 17: After two replications

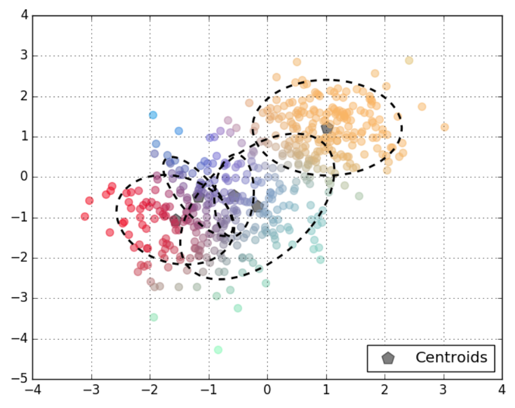


Figure 18: After three replications

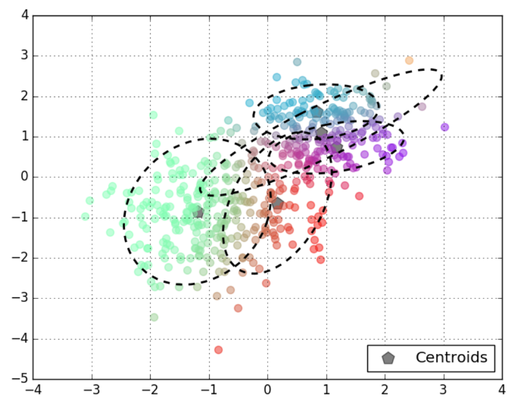


Figure 19: After four replications

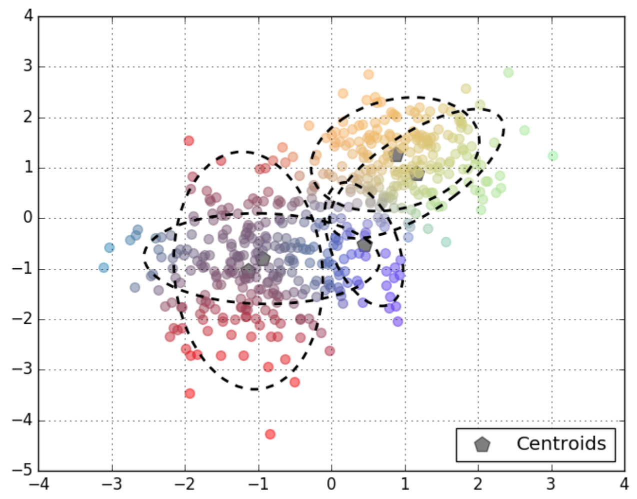


Figure 20: After five replications

In this case we should note the efficiency of the algorithm. During each replication we ask from the algorithm to keep the parameters for which we gain the clusters with the minimum summation of their inner distances. Here we see that final clustering corresponds to the one after fourth replication (Figure 14)

k = 5, Final Clustering after 5 replications

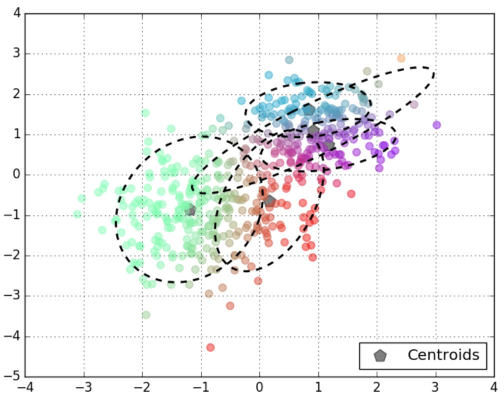


Figure 21: Responsibilities Plot

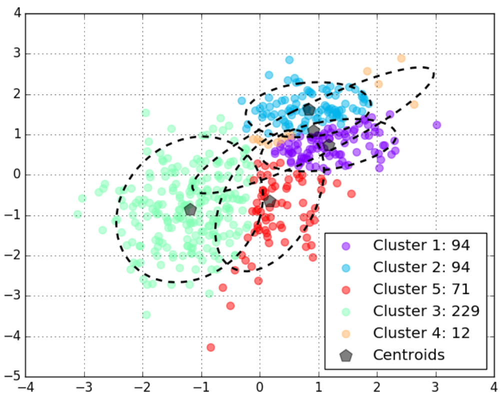


Figure 22: Labeled Plot

Another way to measure the performance of each algorithm and decide whether the cluastering result is satisfying or not is to make use of ***Silhouette Score***. The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from -1 to 1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is appropriate. If many points have a low or negative value, then the clustering configuration may have too many or too few clusters.The silhouette can be calculated with any distance metric, such as the $Euclidean$ distance or the $Manhattan$ distance.

Assume the data have been clustered via any technique, into k clusters. For each datum i, let $a(i)$ be the average dissimilarity of i with all other data within the same cluster. We can interpret $a(i)$ as how well i is assigned to its cluster (the smaller the value, the better the assignment). We then define the average dissimilarity of point i to a cluster c as the average of the distance from i to all points in c.

Let $b(i)$ be the lowest average dissimilarity of i to any other cluster, of which i is not a member. The cluster with this lowest average dissimilarity is said to be the "neighbouring cluster" of i because it is the next best fit cluster for point i. We now define a silhouette:

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}$$

and $-1 \leq s(i) \leq 1$. For $s(i)$ to be close to 1 we require $a(i) \ll b(i)$. As $a(i)$ is a measure of how dissimilar i is to its own cluster, a small value means it is well matched. Furthermore, a large $b(i)$ implies that i is badly matched to its neighbouring cluster. Thus an $s(i)$ close to one means that the data is appropriately clustered. If $s(i)$ is close to negative one, then by the same logic we see that i would be more appropriate if it was clustered in its neighbouring cluster. An $s(i)$ near zero means that the datum is on the border of two natural clusters.

In this project, we have calculated the *Silhouette Score* using the *sklearn.metrics.silhouette score* for the initial data along with the predicted labels for each sample, at each replication.

Clusters	k:2	k:3	k:4	k:5
Euclidean	0.5270127542952221	0.38054663668407035	0.29607094484376956	0.2463873373022683
Manhattan	0.5484922910881146	0.39041247008414437	0.23276416231882785	0.16641533680499246
Mahalanobis	0.40594759330911223	0.2398355556193275	0.1398853646243387	0.11842311977613763
Gaussian	0.5197012854573158	0.3165759671054599	0.2470913120568375	0.28200674826112915

In the table above, we present the silhouette scores for the final clustering, after 5 replications for both *K Means*, with each one of the three distance metrics, and *Gaussian Mixtures*. Typically, there is not significant difference between the two algorithms, for the specific data set. What we should note though is that better score we gain with $k = 2$, as we expected.

Algorithms Performance

An other issue we should discuss is the performance, meaning the time needed for each algorithm to perform a complete procedure. We present below a second table comparing the two algorithms speed, counting the time in seconds for 5 replications and maximum 30 iterations :

Clusters	k:2	k:3	k:4	k:5
Euclidean	2.7692768573760986	7.017005205154419	10.389461755752563	11.16935682296753
Manhattan	8.052412986755371	11.031578540802002	14.005500555038452	17.496265172958374
Mahalanobis	9.137458086013794	70.88409328460693	73.53448510169983	92.87273740768433
Gaussian	96.16514754295349	130.60655546188354	179.98293089866638	218.1177258491516

As we can see *Mixture of Gaussians* needs much more time, something that we expected to see due to the computational complexity of the algorithm.

3.2 Practical Problem

We attempt to perform clustering on real data set in this section. The analysis of livers of $C57BL/6J$ mice fed a high fat diet for up to 24 weeks has shown significant body weight gain was observed after 4 weeks. The results provide insight into the effect of high-fat diets on metabolism in the liver. We have 51 observations divided in 3 categories: $Baseline$, $Normal\ Diet$ and $High-\ Fat\ Diet$. The aim is to see if the implemeented $K\ Means$ and $Mixture\ of\ Gaussians$ are good approach for this data set.

The number of the observations, as mentinoed above, is 51, while the number of dimensions is 45281. At first, we perform $Principal\ Component\ Analysis$, using, in the first case, the $EM\ Algorithm$, for $\sigma^2 > 0$ and, in the second case, $Kernel\ PCA$. Afterwards, we either use the matrix of the projected data to perform $K\ Means$ and take the centroids derived from the clustering as initial μ_k components of the $Gaussian\ Mixture$ for the $EM\ Algorithm$, or we perform directly the EM on the projected matrix, taking arbitrary randomly k observations from the data set and use these as the initial means.

Preprocess: Probabilistic PCA

First we perform $PPCA$, with $\sigma^2 > 0$,to take the projection in 2 dimensions for the data set. Here is the representation of the projected data by the first two principal components.

PPCA

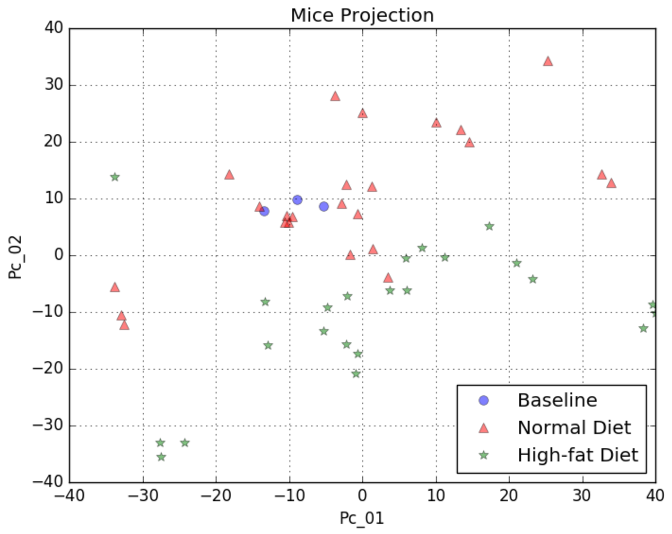


Figure 23: Data separated in three groups

Then, we perform EM for the $Gaussian\ Mixture$ algorithm, taking as initial μ_k from the data set two, $k = 2$, centroids. It is remarkable to see that even if we set $k = 2$ centroids, the algorithm pointed three separate groups. After 5 replications, with maximum iterations number 30, of the algorithm we take the following probabilistic projection which follows exactly the pattern that the observations follow above in $PPCA$ projection:

Mixture of Gaussians

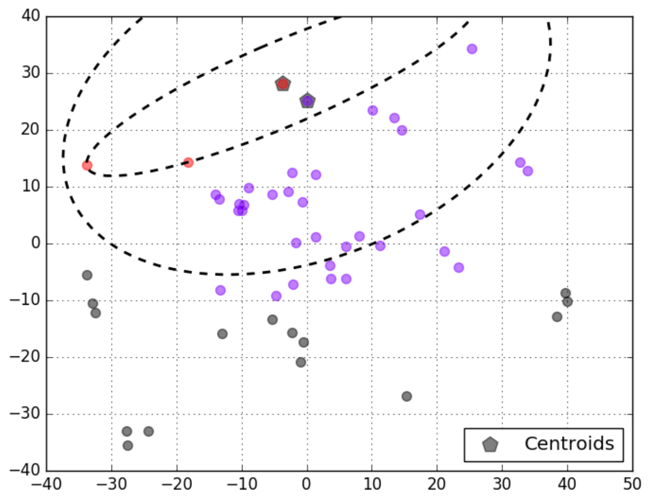


Figure 24: Responsibilities Plot

An other way, as we already discussed, is to initialise the $Gaussians'$ components μ_k as the centroids of the final clustering by $K\ Means$ algorithm. Again, for 5 replications and maximum 30 iterations, for $k = 2$ we take the following plots:

k = 2, PPCA and K Means preprocess

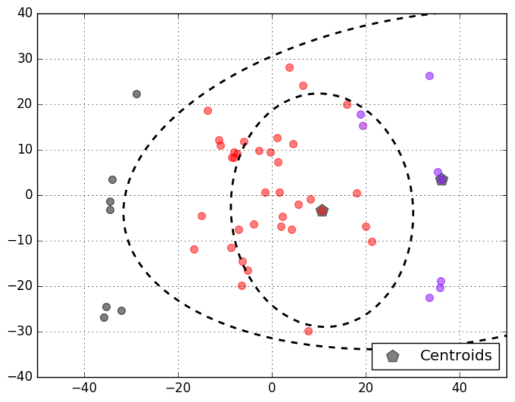


Figure 25: Plot after second replication

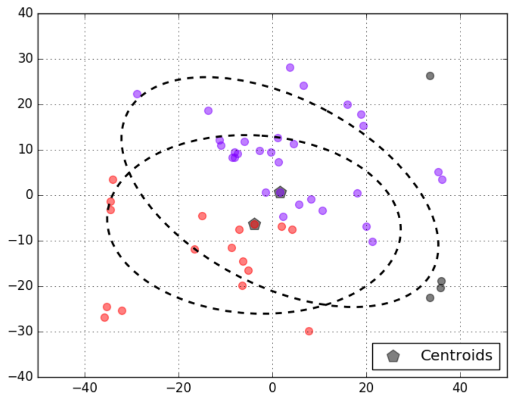


Figure 26: Plot after fifth replication

The plots above represent the probabilities of each observation to belong to a certain cluster. Once again, we see that despite initialization of $k = 2$, these probabilistic methods, $PPCA$ and $Gaussian$ $Mixture$, combined give a relatively sufficient assumption for this specific data set clustering. In the next two plots we have the $K\ Means$ clustering representation along with the Gaussian, but using the final fixed labels this time and not their responsibilities:

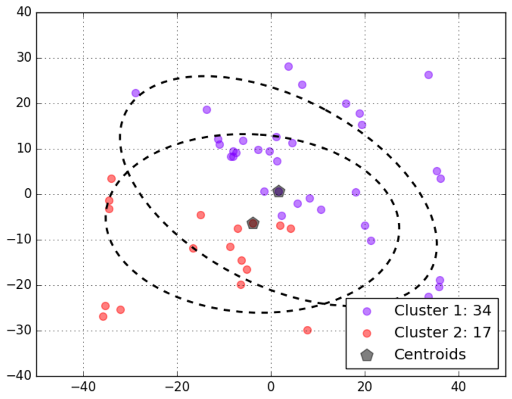


Figure 27: Labeled Plot

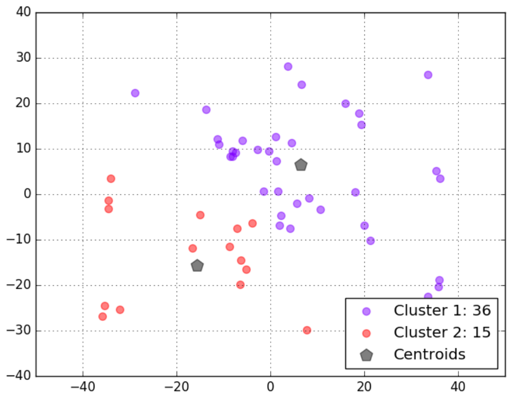


Figure 28: K Means clustering

Preprocess: Kernel PCA

We shall continue our analysis by performing *Kernel PCA* this time as preprocess step. Specifically, we choose as Kernel function the *Hyperbolic Tangent Kernel*

$$K(x_i, x_j) = tanh(x_i x_j + \delta)$$

with $\delta = 3$ and project our data set in $D = 2$ dimensions. Here is the representation of the projected data by the first two principal components:

Kernel Tangent $\delta = 3$

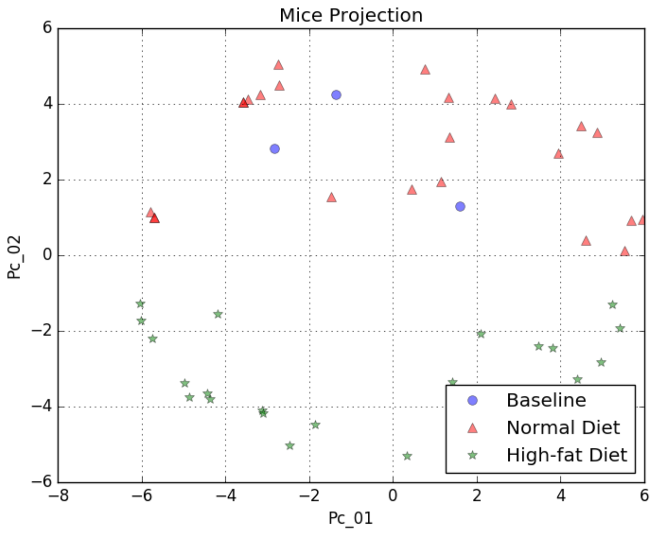


Figure 29: Data separated in three groups

Now, we operate Gaussian Mixture with arbitrary chosen initial μ_k from the data set, exactly as we did before. And we get the following clustering:

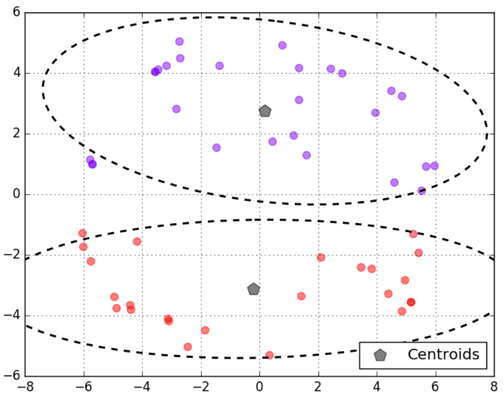


Figure 30: After third replication

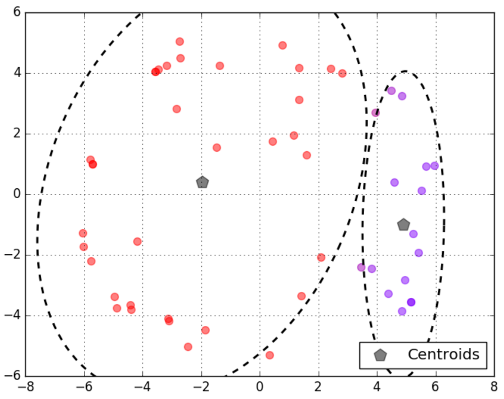


Figure 31: After fifth replication

And here is the final clustering, using the responsibilities matrix $\gamma(z_{kn})$ itself at the first plot and the labels defined by it in the second plot:

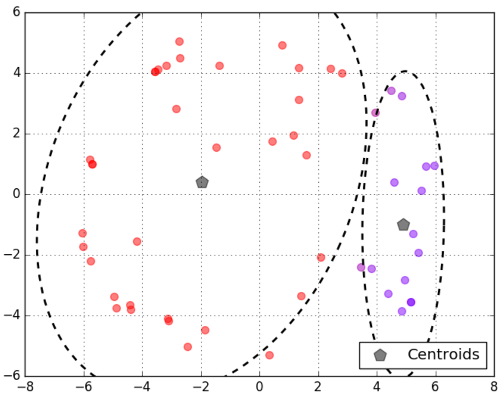


Figure 32: Responsibilities plot

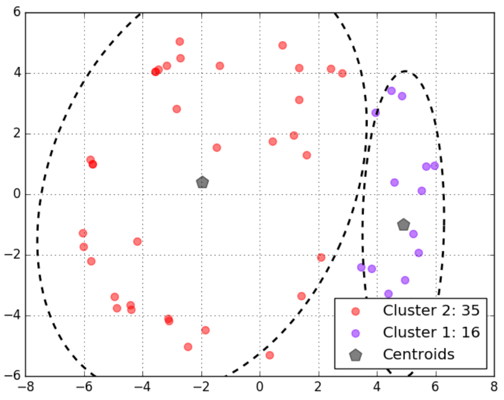


Figure 33: Labeled plot

What we should note here is that the probabilistic plot is exact the same with the Labeled one. What we expected to see, however, is that data points in between the represented clusters should have intermediate colour in the probabilistic plot. In contrast, the clustering that algorithm predicts here is divided in two separate groups. In addition, comparing the reprentation to the one of $Kernel$ above, oservations follow the same pattern, only rotated at some replications.

Finally, we initialise the $Gaussians'$ components μ_k as the centroids of the final clustering by K $Means$ algorithm. Again, for 5 replications and maximum 30 iterations, for $k = 2$ we take the following plots. We add again the initial projection of Kernel so as to evaluate the results:

k = 2, Kernel PCA and K Means preprocess

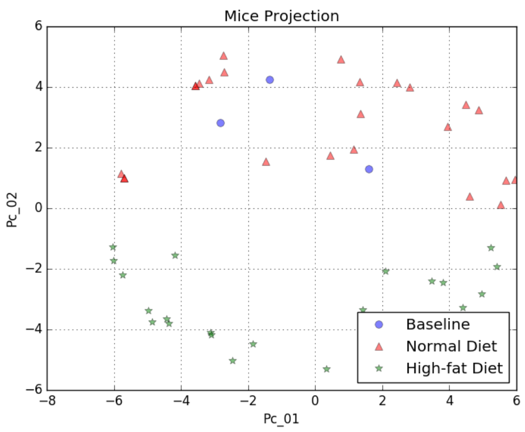


Figure 34: Kernel Projection

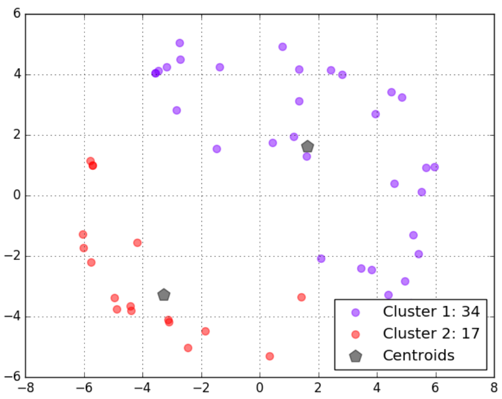


Figure 35: K Means Clustering

Final Mixture of Gaussians Clustering

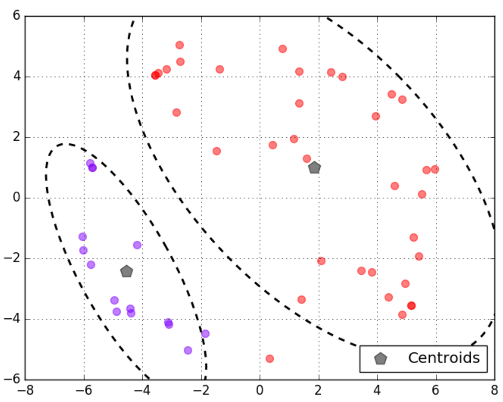


Figure 36: Responsibilities Clustering

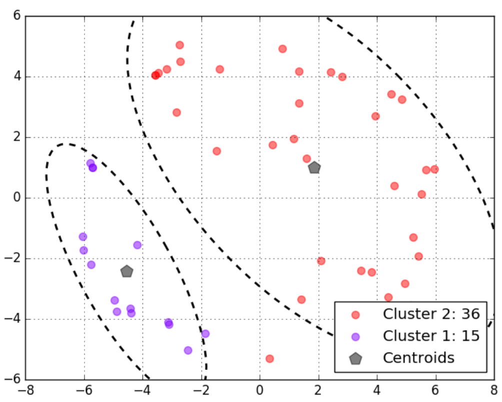


Figure 37: Labels Clustering

Observations follow the same pattern in each plot, either projected in 2 dimensions or clustered in $k = 2$ clusters. Once again, we should note the efficiency of the probabilistic model, *Gaussian*, which gives a remarkable prediction for the clustering that initially *Principal Component Analysis* indicates.

Appendices

A Python Codes

File: main.py

```python
import time
import os
from functions import*
from KMeans import*
from Principal_Methods import*
from Gauss import*

Problem = input("Choose either Theoretical or Practical Problem.\n Enter A or B for Theoretical or
     Practical respectively:")

if Problem == 'A':
  print("##........Theoretical Problem has been chosen.........##\n")

  Method = input("Choose Method: Enter K for K means Algorithm ,\n G for Mixture of Gaussians: ")

  Distance = input("Choose Distance Function\n E for Euklidean\n M for Manhattan\n H for Mahalanobis:
      ")
  n_iter = int(input("Choose number of iterations:  "))
  n_reps = int(input("Choose number of replications: "))

  #### Constructing normally distributed data:  D =2 dimensions, N=500 observations (220 and 280). No
      labels.

  X = Data(220,280,1.0,-1.0,2)

  if Method == 'K':
    print("\n##........K means Algorithm........##\n")

    k = int(input("Choose number of k clusters: "))


    K_means(X,k,Distance,n_iter,n_reps)


  else:
    print("\n##........Mixture of Gaussians Algorithm........##\n")

    k = int(input("Choose number of k clusters: "))

    Mixture_of_Gaussians(X,k,Distance,n_iter,n_reps)


else:
  print("##........Practical Problem has been chosen.........##\n")


  print("\nData set should be downloaded automatically and the process shall begin.\n")


  if not os.path.exists("Final.txt"):

    Filename = os.system('wget
        ftp://ftp.ncbi.nlm.nih.gov/geo/datasets/GDS6nnn/GDS6248/soft/GDS6248.soft.gz')

    os.system('gunzip <GDS6248.soft.gz> Data_set.txt')
    os.system('grep -i ILMN Data_set.txt > Data.txt')
    os.system('cut -f3- Data.txt > Final.txt')
  else:
    print('Skipping file download, Data file exists...')

  #### Constructing the Data set Array:

  X = np.loadtxt("Final.txt")
  print(X.shape)

  q_01 = input("\nPrincipal Component Analysis: Y or N: ")

  if q_01=='Y':
    y = [0]
    Dimension = int(input("Give Dimensionality of Projection:"))
    M = Dimension
    q_02 = input("\nPlease enter EM for Probabilistic PCA or K for Kernel PCA: ")

    if q_02=='EM':
      Y = PPCA(X,y,M)
    else:
      Y = KERNEL(X,y,M)

    Method = input("Choose Method: Enter K for K means Algorithm ,\n G for Mixture of Gaussians: ")

    Distance = input("Choose Distance Function\n E for Euklidean\n M for Manhattan\n H for Mahalanobis: ")
    n_iter = int(input("Choose number of iterations:  "))
    n_reps = int(input("Choose number of replications: "))

    if Method == 'K':
      print("\n##........K means Algorithm........##\n")

      k = int(input("Choose number of k clusters: "))

      K_means(Y,k,Distance,n_iter,n_reps)


    else:
      print("\n##........Mixture of Gaussians Algorithm........##\n")

      k = int(input("Choose number of k clusters: "))

      Mixture_of_Gaussians(Y,k,Distance,n_iter,n_reps)

  else:
    Method = input("Choose Method: Enter K for K means Algorithm ,\n G for Mixture of Gaussians: ")

    Distance = input("Choose Distance Function\n E for Euklidean\n M for Manhattan\n H for Mahalanobis: ")
    n_iter = int(input("Choose number of iterations:  "))
    n_reps = int(input("Choose number of replications: "))

    if Method == 'K':
      print("\n##........K means Algorithm........##\n")

      k = int(input("Choose number of k clusters: "))


      K_means(X,k,Distance,n_iter,n_reps)


    else:
      print("\n##........Mixture of Gaussians Algorithm........##\n")

      k = int(input("Choose number of k clusters: "))

      Mixture_of_Gaussians(X,k,Distance,n_iter,n_reps)
```

File: K Means Algorithm

```python
##### ............File including Methods for Clustering : K means ............#####

from functions import*

############.................... K_Means Algorithm..............####################

def K_means(Arr,k,Distance,n_iter,n_reps):

  start = time.time()

  N = Arr.shape[1] #;print(N)
  D = Arr.shape[0] #;print(D)

  Centroids = Arr[:,np.random.choice(range(N),k,replace=False)] #;print(Centroids) #;
      print(Centroids[:,0])

  ##Plot of data set with the initial centroids
  Plot_Cen(Arr,Centroids)

  # Dictionary in which we will keep the labels (the cluster in which every input points belongs)
  labels = {}
  Dist_old = math.inf
  scores = []

  # We build in a failsafe mechanism: do at most 'n_iter' iterations, if convergence is not achieved.
  # If convergence is achieved, we break the loop manually.

  for i in range(n_reps):
    print("\nIn {} replication: \n".format(i))

    # Randomly choosing initial centroids  from the sample
    Centroids = Arr[:,np.random.choice(range(N),k,replace=False)] #;print(Centroids) #;
        print(Centroids[:,0])

    iterations = 0
    counter = 0   # setting a counter to check the number of iterations needed before convergence
        achieved.
    while iterations < n_iter:
      iterations += 1
      counter +=1

      # partitions contains the partitioned data (it's a list of k lists, each one corresponding to a
          cluster)
      partitions = []
      # initialized with k empty lists
      for i in range(k):
        partitions.append([])

      # Iterate for each input point: we have to find to which cluster it belongs
      for i in range(N):
        min_dist = dist_function(Arr,Distance,Arr[:,i], Centroids[:,0])     # initially, suppose that
            the minimum distance is achieved for the first cluster
        idx = 0                                 # keep the index of the cluster (we need it)
        for cluster in range(1,k):                        # check all clusters from 1 to k
          td = dist_function(Arr,Distance,Arr[:,i], Centroids[:,cluster])   # compute the distance of
              the point to the centroid
          if min_dist > td:                           # if distance is less than the (current)
              minimum...
            min_dist = td                             # ...update the current minimum...
            idx = cluster                             # ...ans also the index at which we found it.


        # At this point, min_dist contains the minimum distance, and idx is the index of the cluster
            for which we achieved it
        # min_dist = min { dist(Arr[:,i], Centroids[:,j]) }, 0 <= j <= k, idx is the j for which we
            achieve that
        partitions[idx].append(Arr[:,i].tolist())

        # idx is also the label that we assign to the point i
        labels[i] = idx;

      # Compute new centroids
      NCentroids = np.empty((D,k))
      for i in range(len(partitions)):
        NCentroids[:,i] = np.mean(np.vstack([partitions[i]]))

      # If the new centroids are too close to the old ones, there is no need to continue.

      dist =0
      for cluster in range(k):
        dist += dist_function(Arr,Distance,Centroids[:,cluster],NCentroids[:,cluster])

      if dist > 10**(-10):
        Centroids = NCentroids
      else:
        print("Convergence achieved!")
        break

  ## End of maximum iterations ,achieving convergence or not.
  #print(Centroids)
  print('Number of iterations until convergence achieved: ' + str(counter))

  ## Plot of the Clusters at each replication

  # colors = cm.rainbow(np.linspace(0, 1, k))
  # label_added = False
  # for i in range(k):
  #   data = partitions[i]
  #   if len(data) > 0:
  #     ddata = np.vstack(data).T
  #     plt.scatter(ddata[0,:], ddata[1,:], color=colors[i], s=50,alpha=0.5)

  #   if not label_added:
  #     plt.scatter(Centroids[0,i], Centroids[1,i], color='black', alpha=0.5, marker=(5,0), s=150,label='Centroids')
  #     label_added = True
  #   else:
  #     plt.scatter(Centroids[0,i], Centroids[1,i], color='black', alpha=0.5,marker=(5,0),s=150)

  # plt.grid(True)
  # plt.legend( scatterpoints =1 ,loc='lower right')
  # plt.savefig("Final Clusters")
  # plt.show()
  # plt.close()

  ##...Silhouette score at each replication to value how good the clustering is...##
  l = np.array(list(labels.values()))
  s=Silhouette(Arr.T,l,Distance)
  scores.append(s)


  ## We calculate the distances inside each of the clusters at every replication...##
  ##...if the sum of the distances is smaller, we keep the centroids and the labels...##
  ##...calculated at the specific replication this achieved.##
  Dist_new = 0
  for i in range(k):
   data = partitions[i]
   if len(data) > 0:
    ddata = distance(Distance,(np.vstack(data).T))
   Dist_new += ddata
  if (Dist_new < Dist_old) or (math.isnan(Dist_new)):
   Labels = labels.copy()
   Centroids_F = Centroids.copy()

  Dist_old = Dist_new

 ## End of replications. we take the final Labels and Centroids and a final plot of the clusters.

 end = time.time()

 y = np.array(list(Labels.values()))   # Labels
 Plot_Clusters(Arr,y,Centroids_F,k)    # Plot

 ## Silhouette score for the final clustering (using the final labels)
 f = Silhouette(Arr.T,y,Distance)



 elapsed = (end - start)
 print("Time {}".format(elapsed))
 print("Scores at each replication: {}".format(scores))
 print("K Means....Final Silhouette score: {} with distance metric: {}".format(f,Distance))


 return(Centroids_F,Labels)


# X = Data(220,280,1.0,-1.0,2)

# Centroids_F, Labels = K_means(X,3,'M',50,5)
```

File: Mixture of Gaussians

```python
#########................. Mixture of Gaussians ..........##########
from functions import*
from KMeans import*


##### ...................Gaussian Density Function.................#####
def N_gauss(x,y,S,D):
  # N = Arr.shape[1]
  # D = Arr.shape[0]

  a_01 = np.sqrt(((2*np.pi)**D)*np.linalg.det(S))
  a_02 = (((x-y).reshape(1,D)).dot(np.linalg.inv(S))).dot((x-y).reshape(D,1))

  return (1/a_01)*np.exp((-1/2)*a_02)

######.................Mixture of Gaussians Algorithm.................######


def Mixture_of_Gaussians(Arr,k,Distance,n_iter,n_reps):

  start = time.time()

  N = Arr.shape[1] #;print(N)
  D = Arr.shape[0] #;print(D)

  ## We have the option to perform K Means first, so as to take initial Centroids

  q_01 = input("\nRun K_Means first to take initial Centroids, Y/n: ")

  # if q_01 == 'Y':
  #   Distance = input("\nChoose distance. Enter E for Euclidean, M for Manhattan and H for Mahalanobis: ")

  scores=[]
  Dist_old = math.inf

  # We build in a failsafe mechanism: do at most 'n_iter' iterations, if convergence is not achieved.
  # If convergence is achieved, we break the loop manually.

  for i in range(n_reps):
    print("\nIn {} replication: \n".format(i))

    if q_01 == 'Y':
      #Distance = input("\nChoose distance. Enter E for Euclidean, M for Manhattan and H for Mahalanobis: ")
      Centroids = K_means(Arr,k,Distance,n_iter,n_reps)
    else:
      Centroids = Arr[:,np.random.choice(range(N),k,replace=False)]

    ### Initializing Parameters:

     _old   = np.random.uniform(size=k)
     _old   /=   _old  .sum()

     _old   =Arr[:,np.random.choice(range(N),k,replace=False)]
    # _old   = Centroids   ## Typically the means of each Gaussian are clusters' centroids.
    #print( _old  .shape)

    Nk = np.zeros(k)

     k_old   = np.zeros(k,dtype=object)
    for cluster in range(k):
       k_old  [cluster] = np.eye((D))

    Pz = np.zeros(k)     ## Pz : array of k values ,each for every Gaussian
     _nk_new   = np.zeros((k,N))

    L_old = 0
    Diff_Log = 1
    #L_new = 0



    iterations = 0
    counter = 0

    while iterations < n_iter:
      iterations += 1
      counter +=1

      L_new =0
      Athr =0
       _nk   = np.zeros((k,N))          # gama_nk a kxN array

       _new    = np.zeros((D,k))      # mean - centroids : Dxk
       k_new    = np.zeros(k,dtype=object) # Sk : k objects /arrays of DxD dimension
       _new   = np.zeros(k)

      Pz_new=np.zeros(k)


      ##.................. E step..................##
      for i in range(N):
        for cluster in range(k):
          Pz[cluster] =  _old [cluster]*N_gauss(Arr[:,i], _old [:,cluster], k_old [cluster],D)
        s = np.sum(Pz)
         _nk [:,i] = Pz/s


      Nk_new = np.sum( _nk ,axis=1)


      ##.................. M step..................##

      #  _new
      for cluster in range(k):
        for i in range(N):
           _new [:,cluster] += (1/Nk_new[cluster])* _nk [cluster,i]*Arr[:,i]
      # print( _new )

      #  k_new
      for cluster in range(k):
         k_new [cluster] = np.zeros((D,D))

      for cluster in range(k):
        for i in range(N):
          product = (((Arr[:,i] -  _new [:,cluster]).reshape(D,1)).dot((Arr[:,i] -
                _new [:,cluster]).reshape(1,D)))
           k_new [cluster] += ((1/Nk_new[cluster])*( _nk [cluster,i]))*product
      # print( k_new )


      #  _new
      for cluster in range(k):
         _new [cluster] = Nk_new[cluster]/N
      # print( _new )

      ##............. Log-Likelihood................##

      for i in range(N):
        Athr =0
        for cluster in range(k):
          Athr +=  _new [cluster]*N_gauss(Arr[:,i], _new [:,cluster], k_new [cluster],D)
        L_new += math.log(Athr)

      Diff_Log = (abs(L_old-L_new))**2

      sys.stdout.write("\rDiff_Log:  {}".format(Diff_Log))
      sys.stdout.flush()

      L_old = L_new
      # print(Diff_Log)
      if Diff_Log > 10**(-5):     ## Convergence criteria
         _old  =  _new .copy()
         k_old  =  k_new .copy()
         _old  =  _new .copy()
        Pz = Pz_new.copy()
        Nk = Nk_new.copy()

      else:
        print("\n\nConvergence achieved!")
        break

    print('\n\nNumber of iterations until convergence achieved: ' + str(counter))


    ## End of maximum iterations ,achieving convergence or not.

    ## Calculating the responsibilities after each replication:
    for i in range(N):
      for cluster in range(k):
        Pz[cluster] =  _old [cluster]*N_gauss(Arr[:,i], _old [:,cluster], k_old [cluster],D)
      s = np.sum(Pz)
       _nk_new [:,i] = Pz/s
    #print("density:{}\n".format( _nk_new ))

    y = np.argmax( _nk_new ,axis=0)     ## Labels from the responsibilities array

    Clusters = np.zeros(k,dtype=object)   ## Forming the clusters using the labels

    for i in range(k):
      Clusters[i] = np.zeros((D,int(Nk[i])))

    for i in range(k):
      Clusters[i] = Arr[:,y== i]
```

```python


  ## Plot of responsibilities at each replication:

  Plot_Prob(Arr,  _old  ,  _nk_new  ,Clusters,k)


  s=Silhouette(Arr.T,y,Distance)
  scores.append(s)

  Dist_new = 0
  for i in range(k):
   #if Clusters[i].size() >2:
   Dist = distance(Distance,Clusters[i])
   Dist_new += Dist


  if (Dist_new < Dist_old) or (math.isnan(Dist_new)):
   Labels = y.copy()
   Centroids_F =   _old  .copy()
   Clusters_F  = Clusters.copy()
   Respons =   _nk_new  .copy()

  Dist_old = Dist_new

 ## End of replications. we take the final Labels,Centroids,Clusters and responsibilities...##
 ##... and a final plot of the clusters.##

 end = time.time()

 ## Final plot using Labels (discrete clusters)

 colors = cm.rainbow(np.linspace(0, 1, k))
 label_added = [False]*k
 for i,j in zip(range(N),Labels):
  if not label_added[j]:
   plt.scatter(Arr[0,i], Arr[1,i], color= colors[j],s=50,alpha=0.5,label='Cluster {}:
     {}'.format(j+1,np.count_nonzero(Labels == j)))
   label_added[j] = True
  else:
   plt.scatter(Arr[0,i], Arr[1,i], color= colors[j],s=50,alpha=0.5)

 for i in range(k):
   plot_cov_ellipse(np.cov(Clusters_F[i]), Centroids_F[:,i], nstd=2, ax=None)

 label_added = False
 for i in range(k):
  if not label_added:
   plt.scatter(Centroids_F[0,i], Centroids_F[1,i], color='black', alpha=0.5, marker=(5,0),
     s=150,label='Centroids')
   label_added = True
  else:
   plt.scatter(Centroids_F[0,i], Centroids_F[1,i], color='black', alpha=0.5,marker=(5,0),s=150)


 plt.grid(True)
 plt.legend( scatterpoints =1 ,loc='lower right')
 #plt.title("Final Clusters_MoG")
 plt.savefig("MoG Labels Final Clusters")
 plt.show()
 plt.close()


 ## Final plot using final responsibilities:
 Plot_Prob(Arr,Centroids_F,Respons,Clusters_F,k)

 ## Silhouette score for the final clustering (using the final labels)
 f = Silhouette(Arr.T,Labels,Distance)

 elapsed = (end - start)
 print("Time {}".format(elapsed))
 print("Scores at each replication: {}".format(scores))
 print("Mixture of Gaussians....Final Silhouette score: {} with distance metric:
   {}".format(f,Distance))



 return(Centroids_F,Labels)
################################


# X = Data(220,280,1.0,-1.0,2)


# Centroids = Mixture_of_Gaussians(X,3,'H',50,5)
```

File: General Functions

```python
###.................... Additional functions and libraries for the basic Algorithms ............###
import sys
import time
import math
from math import sqrt
import numpy as np
import scipy.spatial.distance
import sklearn.metrics
from scipy import linalg
from matplotlib import pyplot as plt
from matplotlib import cm as cm
from matplotlib.patches import Ellipse


#####.................Function returning the Silhouette Score (details in report).............#####

def Silhouette(Arr,lab,dist):
  if dist == 'E':
    return(sklearn.metrics.silhouette_score(Arr, lab, metric='euclidean'))
  elif dist == 'M':
    return(sklearn.metrics.silhouette_score(Arr, lab, metric='cityblock'))
  else:
    return(sklearn.metrics.silhouette_score(Arr, lab, metric='mahalanobis'))


#####.................Function that constructs the data set for Theoretical Part................#####

def Data(N1,N2,m1,m2,D):

  #epsilon = np.random.uniform(-1,1,D)

  mean_01 = m1*np.ones(D) #;print(mean_01.shape)   # mu: Dx1 cov: DxD
  mean_02 = m2*np.ones(D) #;print(mean_02.shape)

  cov_01  = 0.5*np.eye(D) #+ epsilon
  cov_02  = 0.75*np.eye(D) #+ epsilon

  X_01 = np.random.multivariate_normal(mean_01,cov_01,N1).T
  X_02 = np.random.multivariate_normal(mean_02,cov_02,N2).T

  X = np.concatenate((X_01,X_02),axis=1)  #; print(X.shape)  ## should be DxN

  ### Initial Data Plot:

  plt.scatter(X[0,:], X[1,:], color='blue',marker='o',alpha=0.5,s=50,label='Data')
  plt.grid(True)
  plt.legend( scatterpoints =1 ,loc='lower right')
  plt.savefig("Initial plot")
  plt.show()

  return(X)

####...................Basic Metric Distances Functions.................#####

def euclidean_distance(x, y):
  return sqrt(np.sum((x-y)**2))

def Manhattan_Distance(x,y):
  return np.sum(np.abs(x-y))

def Mahalanobis_Distance(Arr,x,y):
  D = Arr.shape[0]
  S = np.cov(Arr)
  return np.sqrt((x-y).reshape(1,D).dot(np.linalg.inv(S)).dot((x-y).reshape(D,1)))

## Change distance function here according to needs

def dist_function(Arr,Distance,x, y):
  if Distance == 'E':
    return euclidean_distance(x, y)
  elif Distance == 'M':
    return Manhattan_Distance(x,y)
  else:
    return Mahalanobis_Distance(Arr,x,y)


## using built in functions for practical reasons in specific stages of the algorithm.

def distance(Distance,Arr):
  if Distance == 'E':
    return scipy.spatial.distance.pdist(Arr,metric='euclidean')
  elif Distance == 'M':
    return scipy.spatial.distance.pdist(Arr,metric='cityblock')
  else:
    S = np.linalg.inv(np.cov(Arr))
    return scipy.spatial.distance.pdist(Arr,metric='mahalanobis',VI = S)

```

```


#####......................Plots..........................#####


####....Adding Ellipses in plots....Credits to Dimitris :).....####

def plot_cov_ellipse(cov, pos, nstd=2, ax=None, **kwargs):
    """
    Plots an `nstd` sigma error ellipse based on the specified covariance
    matrix (`cov`). Additional keyword arguments are passed on to the
    ellipse patch artist.

    Parameters
    ----------
        cov : The 2x2 covariance matrix to base the ellipse on
        pos : The location of the center of the ellipse. Expects a 2-element
            sequence of [x0, y0].
        nstd : The radius of the ellipse in numbers of standard deviations.
            Defaults to 2 standard deviations.
        ax : The axis that the ellipse will be plotted on. Defaults to the
            current axis.
        Additional keyword arguments are pass on to the ellipse patch.

    Returns
    -------
        A matplotlib ellipse artist
    """
    def eigsorted(cov):
        vals, vecs = np.linalg.eigh(cov)
        order = vals.argsort()[::-1]
        return vals[order], vecs[:,order]

    if ax is None:
        ax = plt.gca()

    vals, vecs = eigsorted(cov)
    theta = np.degrees(np.arctan2(*vecs[:,0][::-1]))


    width, height = 2 * nstd * np.sqrt(vals)
    ellip = Ellipse(xy=pos, width=width, height=height, angle=theta,lw=2,fill=False,ls='--')

    ax.add_artist(ellip)
    return ellip
    # Width and height are "full" widths, not radius
    # for i in range(nstd):
    #   s = i+1
    #   width, height = 2 * s * np.sqrt(vals)
    #   ellip = Ellipse(xy=pos, width=width, height=height, angle=theta,lw=2, fill=False,ls='--')
    #   ax.add_artist(ellip)
    # return ellip


def Plot_Cen(Arr,Cen):
  plt.scatter(Arr[0,:], Arr[1,:], color='blue',marker='o',alpha=0.5,s=50,label='Data Set')
  plt.scatter(Cen[0,:],Cen[1,:],color='red', marker=(5,0),alpha=0.5,s=150,label='Centroids')
  plt.grid(True)
  plt.legend( scatterpoints =1 ,loc='lower right')
  plt.savefig("Initial Plot of Centroids")
  plt.show()

def Plot_Clusters(Arr,lab,Cen,k):   ## Plot of final clusters, K means

  N = Arr.shape[1]
  colors = cm.rainbow(np.linspace(0, 1, k))
  label_added = [False]*k

  for i,j in zip(range(N),lab):
    if not label_added[j]:
      plt.scatter(Arr[0,i], Arr[1,i], color= colors[j],s=50,alpha=0.5,label='Cluster {}:
          {}'.format(j+1,np.count_nonzero(lab == j)))
      label_added[j] = True
    else:
      plt.scatter(Arr[0,i], Arr[1,i], color= colors[j],s=50,alpha=0.5)

  label_added = False
  for i in range(k):
    if not label_added:
      plt.scatter(Cen[0,i], Cen[1,i], color='black', alpha=0.5, marker=(5,0), s=150,label='Centroids')
      label_added = True
    else:
      plt.scatter(Cen[0,i], Cen[1,i], color='black', alpha=0.5,marker=(5,0),s=150)

  plt.grid(True)
  plt.legend( scatterpoints =1 ,loc='lower right')
  #plt.title("Final Clusters")
  plt.savefig("K means Final Clusters")
```

```python
  plt.show()
  plt.close()

def Plot_Prob(Arr,mu,gamma,cluster,k):    ## Plot based on Probabilities ,MoG

  N = Arr.shape[1]
  colors = cm.rainbow(np.linspace(0, 1, k))

  label_added = False
  for i in range(k):
    if not label_added:
      plt.scatter(mu[0,i], mu[1,i], color='black', alpha=0.5, marker=(5,0), s=150,label='Centroids')
      label_added = True
    else:
      plt.scatter(mu[0,i], mu[1,i], color='black', alpha=0.5,marker=(5,0),s=150)

  for i in range(N):
    col = np.multiply(gamma[0,i],colors[0])
    for j in range(1,k):
      col = col + np.multiply(gamma[j,i],colors[j])

    plt.scatter(Arr[0,i], Arr[1,i], color=np.asarray(col.astype(np.float32)),s=50, alpha=0.5)


  for i in range(k):
    plot_cov_ellipse(np.cov(cluster[i]), mu[:,i], nstd=2, ax=None)


  plt.grid(True)
  plt.legend( scatterpoints =1 ,loc='lower right')
  plt.savefig("MoG Final Clusters")
  plt.show()
  plt.close()
```

File: PCA Methods

```python
##### File including All Methods for Principal Component Analysis

import numpy as np
from random import random
from scipy import linalg
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d import proj3d
from sklearn.datasets import make_circles
from sklearn.metrics import mean_squared_error


########## Principal Component Analysis #########

def PCA(Arr,y,M):
  print("\n######___Principal Component Analysis___######\n")

  D = Arr.shape[0]
  N = Arr.shape[1]

  ## Mean Vector:

  mean_vector = np.empty([D,1])
  for i in range(D):
    mean_vector[i] = np.mean(Arr[i,:])
  # print(mean_vector.shape)

  mean_d = np.repeat (mean_vector,N,axis=1)      # The mean vector with DxN shape
  X = Arr- mean_d                    # Normalize the Data matrix, X: DxN

  ## Scatter Matrix: is used to estimate the Covariance matrix of a multivariate normal distribution

  Scatter_Matrix = np.empty([D,D])
  for i in range(X.shape[1]):
      Scatter_Matrix += (X[:,i].reshape(D,1)).dot((X[:,i].reshape(D,1)).T)
      #Scatter_Matrix += (X[:,i].reshape(D,1) - mean_vector).dot((X[:,i].reshape(D,1) -
          mean_vector).T)
  print('Scatter Matrix:\n', Scatter_Matrix)


  ## Eigen Vectors/ Values:
  eig_val_sc, eig_vec_sc  = np.linalg.eig(Scatter_Matrix)

  ## Check if Su= u :
  for i in range(len(eig_val_sc)):
      eigv = eig_vec_sc[:,i].reshape(1,D).T
      np.testing.assert_array_almost_equal(Scatter_Matrix.dot(eigv), eig_val_sc[i]*eigv,decimal=6,
          err_msg='The eigenvector eigenvalue calculation is NOT correct.', verbose=True)


  ## Rank the eigenvectors from highest to lowest corresponding eigenvalue and choose the top k
      eigenvectors.

  # Make a list of (eigenvalue, eigenvector) tuples
  Pairs = [(np.abs(eig_val_sc[i]), eig_vec_sc[:,i]) for i in range(len(eig_val_sc))]

  # Sort the (eigenvalue, eigenvector) tuples from high to low __ Using lambda function :)
  Pairs.sort(key=lambda x: x[0], reverse=True)
  print(len(Pairs))
  # Checking that the list is correctly sorted
  # for i in Pairs:
  #     print(i[0])
  #print(len(Pairs))

  ## Construction of  eigenvector matrix U.

  q = input("Please enter S if you wish to use SVD to calculate array U or E for Eigendecomposition:
      ")

  if q == 'S':
    # (i) SVD:
    U,S,V = np.linalg.svd(X, full_matrices=False)
  else:
    # (ii) Eigendecomposition:
    U = np.empty([D,M])
    for i in range(M):
      U[:,i] = Pairs[i][1]#.reshape(D,1)

  print('Matrix U:\n', U)
  print(U.shape)


  ## Transforming the samples onto the new subspace with M- Dimension:

  Projected_Data = U.T.dot(X)
  print("shape {}".format(Projected_Data.shape))      ## Should be DxM
  ## Plots

  if len(y) == 1:
    if M == 2:
      Practical_Plots(Projected_Data)
    else:
      D_Plots(Projected_Data)
      Practical_Plots(Projected_Data)
  else:
    if M == 1:
      one_Plots(Projected_Data)
    elif M == 2:
      one_Plots(Projected_Data)
      Theoretical_Plots(Projected_Data,y)
    else:
      one_Plots(Projected_Data)
      Theoretical_Plots(Projected_Data,y)
      T_D_Plots(Projected_Data,y)


########### Probabilistic Principal Component Analysis ###########

def PPCA(Arr,y,K):
  print("\n######___Probabilistic Principal Component Analysis___######\n")
  sigma_sq = int(input("Please enter 0 if s_square is zero: "))

  ## Mean Vector:
  D = Arr.shape[0]
  N = Arr.shape[1]

  mean_vector = np.empty((D,1))
  for i in range(D):
    mean_vector[i] = np.mean(Arr[i,:])

  mean_d = np.repeat(mean_vector,N,axis=1)       # The mean vector with DxN shape
  X = Arr- mean_d                      # X = x - mean(x)   X: Array of the Data  DxN

  ## Initializing the parameters:

  #sigma_sq = rand(0,1)
  W_old = np.array(np.random.rand(D,K))          # W_old: DxK
  #print("The matrix W is {}".format(W_old))
  print("The shape of matrix W is :{}".format(W_old.shape))
  W_new = np.empty((D,K))

  RMSE = mean_squared_error(W_old, W_new)**0.5       # Root Mean Squared Error (RMSE)
  #print(RMSE)

  ###### EM Algorithm:
  Times = int(input("\nPlease assign the number of iterations:"))

  # Limit case of s^2 -> 0
  if sigma_sq == 0:
    W_all = np.empty((D,K))
    for i in range(Times):
      RMSEdiff = 1
      while RMSEdiff > 10**(-7):
        RMSEold = RMSE
        Omega = (linalg.inv((W_old.T).dot(W_old))).dot((W_old.T).dot(X))         # E step
        #print(Omega.shape)
        W_new = (X.dot(Omega.T)).dot(linalg.inv(Omega.dot(Omega.T)))           # M step
        #print(W_new.shape)
        RMSE = mean_squared_error(W_old, W_new)**0.5
        W_old = W_new

        RMSEdiff = abs(RMSE - RMSEold)

        print(RMSEdiff)

      W_all+= W_new

  # s^2 != 0
  else:
    W_all = np.empty((D,K))
    for i in range(Times):
      sigma_sq = random() #; print(sigma_sq)
      sigma_sq_new = random()
      dif_sigma=1

      RMSEdiff = 1
      while RMSEdiff > 10**(-7) or dif_sigma > 10**(-8):
        RMSEold = RMSE
        M = (W_old.T).dot(W_old) + sigma_sq*(np.eye(K))
        E_Zn  = ((linalg.inv(M)).dot(W_old.T)).dot(X)
        E_Zn_ZnT = sigma_sq*linalg.inv(M) + E_Zn.dot(E_Zn.T)

        W_new = X.dot(E_Zn.T).dot(linalg.inv(E_Zn_ZnT)) #;print(W_new.shape)


        for i in range(N):
          Trace = np.trace(E_Zn_ZnT.dot((W_new).reshape(K,D)).dot(W_new))
          a_01 = (np.linalg.norm(X[:,i].reshape(1,D)))**2
          a_02 = ((E_Zn[:,i].reshape(1,K)).dot(W_new.T).dot(X[:,i].reshape(D,1)))
          sigma_sq_new +=  np.sum(a_01 -2*a_02 +Trace)

        sigma_sq_new = sigma_sq_new /(N*D)

        RMSE = mean_squared_error(W_old, W_new)**0.5
        dif_sigma = abs(sigma_sq - sigma_sq_new)

        W_old = W_new
        sigma_sq = sigma_sq_new

        RMSEdiff = abs(RMSE - RMSEold)
        #print(RMSE)
        print(RMSEdiff)
        print(dif_sigma)
        print("\n")

      W_all+= W_new

  W_mean = W_all/(Times)

  ### SVD and Projection

  U,S,V = np.linalg.svd(W_mean, full_matrices=False)

  Projected_Data = U.T.dot(X)
  print(Projected_Data.shape)


  if len(y) == 1:
    if K == 2:
      Practical_Plots(Projected_Data)
    else:
      D_Plots(Projected_Data)
      Practical_Plots(Projected_Data)
  else:
    if K == 1:
      one_Plots(Projected_Data)
    elif K == 2:
      one_Plots(Projected_Data)
      Theoretical_Plots(Projected_Data,y)
    else:
      one_Plots(Projected_Data)
      Theoretical_Plots(Projected_Data,y)
      T_D_Plots(Projected_Data,y)


  return (Projected_Data)
########### Kernel Method ###########

def KERNEL(Arr,y,M):

  kernel = input("Construct the Kernel Matrix:\n Press G for Gaussian, P for Polynomial and T for
       Tanget: ")

  D = Arr.shape[0]
  N = Arr.shape[1]

  ## Mean Vector:

  mean_vector = np.empty([D,1])
  for i in range(D):
    mean_vector[i] = np.mean(Arr[i,:])
  # print(mean_vector.shape)

  mean_d = np.repeat (mean_vector,N,axis=1)     # The mean vector with DxN shape
  X = Arr- mean_d                     # Normalize the Data matrix, X: DxN

  N = X.shape[1]
  K = np.empty((N,N)) #; print(K.shape)


  ### Constructing the Kernels:

  def Polynomial(Arr,p):
    for i in range(N):
      for j in range(N):
        K[i,j] = (1 + np.inner(Arr[:,i],Arr[:,j]))**p

  def Tanget(Arr,delta):
    for i in range(N):
      for j in range(N):
        K[i,j] = np.tanh(np.inner(Arr[:,i],Arr[:,j]) + delta)

  def Gaussian_Kernel(Arr,gama):
    for i in range(N):
      for j in range(N):
        K[i,j] = np.exp(-gama*((((Arr[:,i] - Arr[:,j]).T).dot(Arr[:,i] - Arr[:,j]))**2))

  ## Kernels:

  if kernel == 'G':
    gama = float(input("\nGive value for gama: "))    #gama = int(input("\nGive value for gama: "))
    Gaussian_Kernel(X,gama)
  elif kernel == 'P':
    p = float(input("\nGive value for p: "))      #p = int(input("\nGive value for p: "))
    Polynomial(X,p)
  else:
    delta = float(input("\nGive value for delta: "))  #delta = int(input("\nGive value for delta: "))
    Tanget(X,delta)

  One_N = np.empty((N,N))
  for i in range(N):
    One_N[i] = 1/N

  ## Method:

  K_bar = K-(One_N.dot(K)) - (K.dot(One_N)) + ((One_N.dot(K)).dot(One_N))

  eig_values, eig_vectors  = np.linalg.eig(K_bar)
  Pairs = [(np.abs(eig_values[i]), eig_vectors[:,i]) for i in range(len(eig_values))]

  # Sort the (eigenvalue, eigenvector) tuples from high to low __ Using lambda function :)
  Pairs.sort(key=lambda x: x[0], reverse=True)
  print("lenght {}".format(len(Pairs)))

  U = np.empty([N,M])
  for i in range(M):
    U[:,i] = Pairs[i][1]#.reshape(D,1)

  print('Matrix U:\n', U)
  print(U.shape)

  Projected_Data = U.T.dot(K_bar)
  print(Projected_Data.shape)

  if len(y) == 1:
    if M == 2:
      Practical_Plots(Projected_Data)
    else:
      D_Plots(Projected_Data)
      Practical_Plots(Projected_Data)
  else:
    if M == 1:
      one_Plots(Projected_Data)
    elif M == 2:
      one_Plots(Projected_Data)
      Theoretical_Plots(Projected_Data,y)
    else:
      one_Plots(Projected_Data)
      Theoretical_Plots(Projected_Data,y)
      T_D_Plots(Projected_Data,y)

  return (Projected_Data)

########### Function for Subquestion (ii) of Theoretical Exercise ##########

def Theoretical_II(Arr01,Arr02):
  print(Arr02.shape)
  def Eigenvalues(Arr):
    ## Mean Vector:
    D = Arr.shape[0]
    N = Arr.shape[1]

    mean_vector = np.empty([D,1])
    for i in range(D):
      mean_vector[i] = np.mean(Arr[i,:])
    mean_d = np.repeat (mean_vector,N,axis=1)      # The mean vector with 3x80 shape DxN
    X = Arr- mean_d                    # Normalize the join matrix X DxN

    ## Scatter Matrix: is used to estimate the Covariance matrix of a multivariate normal distribution
    Scatter_Matrix = np.empty([D,D])
    for i in range(X.shape[1]):
      Scatter_Matrix += (X[:,i].reshape(D,1)).dot((X[:,i].reshape(D,1)).T)

    ## Eigen Vectors/ Values of Scatter Matrix:
    eig_values, eig_vectors  = np.linalg.eig(Scatter_Matrix)

    val = np.ndarray.tolist(eig_values)
    sort_val = sorted(val,reverse =True)

    return sort_val
```

```python



  sort_val = Eigenvalues(Arr01)
  sort_val2= Eigenvalues(Arr02)
  print("Eig 10:{}".format(sort_val))
  print("Eig 5:{}".format(sort_val2))
  maxx = max([max(sort_val),max(sort_val2)])
  x_val =[i for i in range(1,len(sort_val)+1)]

  fig, ax = plt.subplots()

  ax.scatter(x_val,sort_val,marker='o', color='blue', alpha=0.5, label='N=10')
  ax.scatter(x_val,sort_val2,marker='*',color='red',alpha=0.5,label='N=5')
  ax.set_ylim([0,maxx+10])

  plt.grid(True)
  plt.legend( numpoints=1 ,loc='upper right')
  plt.title('Eigenvalues_Plot')
  plt.savefig("Eigenvalues_Plot.png")
  plt.show()



########### Plot Functions ##########

## Theoretical Part:

def Theoretical_Plots(Arr,y):

  plt.scatter(Arr[0,y==0], Arr[1,y==0],color='red',marker='^',alpha=0.5,label='Circle_01')
  plt.scatter(Arr[0,y==1], Arr[1,y==1],color='blue',marker='o',alpha=0.5,label='Circle_02')
  plt.grid(True)
  plt.xlabel('Pca_01')
  plt.ylabel('Pca_02')
  plt.legend(numpoints =1,loc='lower right')
  plt.title('Projection')
  plt.savefig("Theoretical_01.png")
  plt.show()

def T_D_Plots(Arr,y):

  fig = plt.figure(figsize=(8,8))
  ax  = fig.add_subplot(111, projection='3d')
  plt.rcParams['legend.fontsize'] = 10

  ax.scatter(Arr[0,y==0], Arr[1,y==0],Arr[2,y==0],color='red',marker='^',alpha=0.5,label='Circle_01')
  ax.scatter(Arr[0,y==1], Arr[1,y==1],Arr[2,y==1],color='blue',marker='o',alpha=0.5,label='Circle_02')
  ax.grid(True)

  ax.set_xlabel('Pc_01')
  ax.set_ylabel('Pc_02')
  ax.set_zlabel('Pc_03')

  plt.title('Projected in 3D')
  ax.legend(numpoints=1,loc='lower right')
  plt.savefig("3d Plot.png")
  plt.show()
  plt.close()

def one_Plots(Arr):
  y = [2 for i in range(500)]
  plt.plot(Arr[0,0:500],y,'o', markersize=7, color='blue', alpha=0.5, label='Circle_01')
  plt.plot(Arr[0,500:1000],y,'^', markersize=7, color='red', alpha=0.5, label='Circle_02')
  plt.grid(True)
  plt.xlabel('Pc_01')
  plt.ylabel('Pc_02')
  #plt.ylim([-10,10])
  plt.legend(numpoints=1,loc='lower right')
  plt.title('Projection in 1d')
  plt.savefig("1d Plot.png")
  plt.show()


## Practical:

def Practical_Plots(Arr):

  plt.plot(Arr[0,0:3], Arr[1,0:3], 'o', markersize=7, color='blue', alpha=0.5, label='Baseline')
  plt.plot(Arr[0,3:27], Arr[1,3:27], '^', markersize=7, color='red', alpha=0.5, label='Normal Diet')
  plt.plot(Arr[0,27:51],Arr[1,27:51],'*', markersize=7,color='green',alpha=0.5,label='High-fat Diet')
  plt.grid(True)
  plt.xlabel('Pc_01')
  plt.ylabel('Pc_02')

  plt.legend(numpoints=1,loc='lower right')
  plt.title('Mice Projection')
  plt.savefig("Mice Projection.png")
  plt.show()


def D_Plots(Arr):

  fig = plt.figure(figsize=(8,8))
  ax  = fig.add_subplot(111, projection='3d')
  plt.rcParams['legend.fontsize'] = 10

  ax.plot(Arr[0,0:3],Arr[1,0:3],Arr[2,0:3], 'o', markersize=7, color='blue', alpha=0.5,
       label='Baseline')
  ax.plot(Arr[0,3:27],Arr[1,3:27],Arr[2,3:27], '^', markersize=7, color='red', alpha=0.5,
       label='Normal Diet')
  ax.plot(Arr[0,27:51],Arr[1,27:51],Arr[2,27:51],'*',
       markersize=7,color='green',alpha=0.5,label='High-fat Diet')

  ax.grid(True)
  ax.set_xlabel("Pc_01")
  ax.set_ylabel("Pc_02")
  ax.set_zlabel("Pc_03")
  plt.title('Projected Mice 3D')
  ax.legend(numpoints=1,loc='lower right')
  plt.savefig("Mice 3d Plot.png")
  plt.show()
  plt.close()
```

B Suggested Bibliography

References

[1] Christopher M. Bishop *Pattern Recognition and Machine Learning*

[2] Sebastian Raschka *Python Machine Learning*

[3] Gazi N. Ali, Pei Ju Chiang, Aravind K. Mikkilineni, George T. Chiu, Edward J. Delp, and Jan P. Allebach *Application of Principal Components Analysis and Gaussian Mixture Models to Printer Identification*
https://www.cerias.purdue.edu/assets/pdf/bibtex_archive/nip04-ali.pdf

[4] Sanjoy Dasgupta, University of California, Berkeley *Learning Mixtures of Gaussians*
http://cseweb.ucsd.edu/~dasgupta/papers/mog.pdf