# Principal Component Analysis with Python Programming

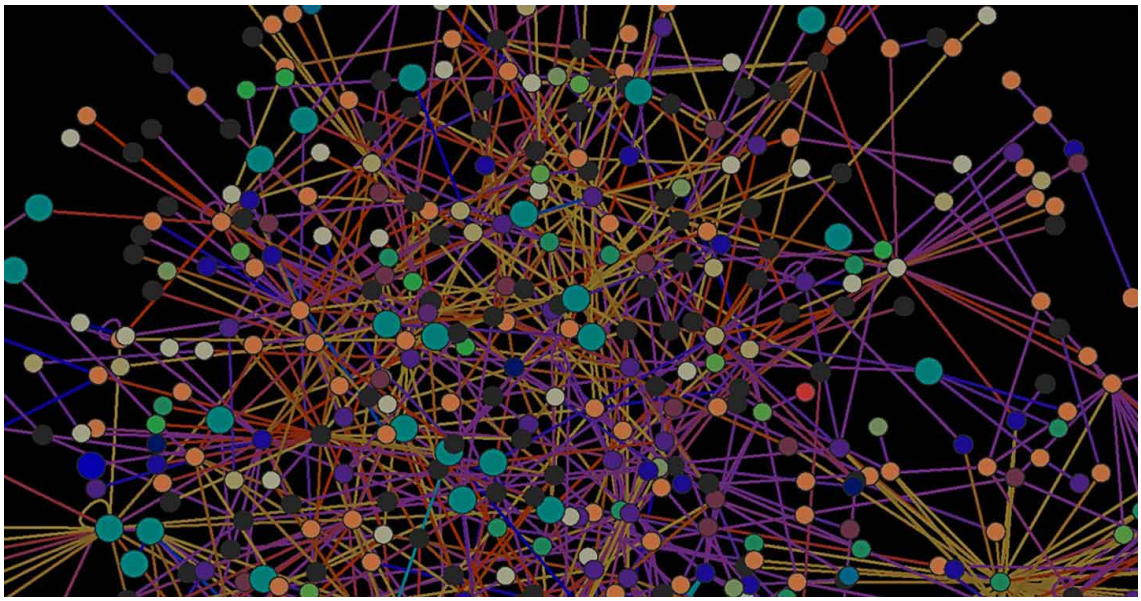
**Chrysovalantou Kalaitzidou**

**March 2017**

**University of Crete Faculty of Medicine**
**Bioinformatics M.Sc.**

**Methods in Bioinformatics**

**First Assignment**



# Contents

# 1 Introduction

The current report is part of the first assignment in class *Methods in Bioinformatics* and refers to ***Principal Component Analysis*** , with respect to its methods and algorithms. 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 Principal Component Analysis

***Principal component analysis, (PCA)*** is probably the oldest and best known of the techniques of *Multivariate Analysis*. It was first introduced by Pearson (1901), and developed independently by Hotelling (1933). Like many multivariate methods, it was not widely used until the advent of electronic computers, but it is now well entrenched in virtually every statistical computer package.
The central idea of principal component analysis is to reduce the dimensionality of a data set in which there are a large number of interrelated variables, while retaining as much as possible of the variation present in the data set. This reduction is achieved by transforming to a new set of variables, the *Principal Components*, which are uncorrelated, and which are ordered so that the first few retain most of the variation present in all of the original variables. Computation of the Principal Components reduces to the solution of an eigenvalue-eigenvector problem for a positive-semidefinite symmetric matrix. Thus, the definition and computation of Principal Components are straightforward but, as will be seen, this apparently simple technique has a wide variety of different applications, as well as a number of different derivations.
There are two commonly used definitions of $PCA$ that give rise to the same algorithm. PCA can be defined as the orthogonal projection of the data onto a lower dimensional linear space, known as the *Principal Subspace*, such that the variance of the projected data is *maximized* (Hotelling, 1933). Equivalently, it can be defined as the linear projection that *minimizes* the average projection cost, defined as the mean squared distance between the data points and their projections. In the current project we discuss the $Maximum\ variance\ formulation$ only.
At last, we should cite that $PCA$ is sometimes used as a preliminary to, or in conjunction with, other statistical techniques, especially with three well-known multivariate techniques, namely *Discriminant Analysis*, *Cluster Analysis* and *Canonical Correlation Analysis*.

### 2.1.1 Maximun Variance Formulation

Consider a data set of observations $x_n$ where $n = 1, ..., N$ , and $x_n$ is an Euclidean variable with dimensionality $D$. Our goal is to project the data onto a space having dimensionality $M < D$ while maximizing the variance of the projected data. To begin with, consider the projection onto a one-dimensional space ($M = 1$). We can define the direction of this space using a D-dimensional vector $u_1$, which for convenience (and without loss of generality) we shall choose to be a unit vector so that

$$u_1^T u_1 = 1$$

(note that we are only interested in the direction defined by $u_1$, not in the magnitude of $u_1$ itself). Each data point $x_n$ is then projected onto a scalar value

$$u_1^T x_n$$

The mean of the projected data is $u_1 \bar{x}$ where $\bar{x}$ is the sample set mean given by

$$\bar{x} = \sum_{n=1}^{N} \frac{x_n}{N}$$

and the variance of the projected data is given by

$$\frac{1}{N} \sum_{n=1}^{N} (u_1^T x_n - u_1^T \bar{x})^2 = u_1^T S u_1$$

where $S$ is the data covariance matrix defined by

$$S = \frac{1}{N} \sum_{n=1}^{N} (x_n - \bar{x})(x_n - \bar{x})^T$$

We now maximize the projected variance $u_1^T S u_1$ with respect to $u_1$. Clearly, this has to be a constrained maximization to prevent $\|u_1\| \to \infty$. The appropriate constraint comes from the normalization condition $u_1^T u_1 = 1$. To enforce this constraint, we introduce a *Lagrange multiplier* that we shall denote by $\lambda_1$ , and then make an unconstrained maximization of

$$u_1^T S u_1 + \lambda_1(1 - u_1^T u_1)$$

By setting the derivative with respect to $u_1$ equal to zero, we see that this quantity will have a stationary point when

$$S u_1 = \lambda_1 u_1$$

which says that $u_1$ must be an *eigenvector* of $S$. If we left-multiply by $u_1^T$ and make use of $u_1^T u_1 = 1$, we see that the variance is given by

$$u_1^T S u_1 = \lambda_1$$

and so the variance will be a *maximum when we set* $u_1$ *equal to the eigenvector having the largest eigenvalue* $\lambda_1$ . This eigenvector is known as the ***First Principal Component***.
We can define additional Principal Components in an incremental fashion by choosing each new direction to be that which maximizes the projected variance amongst all possible directions orthogonal to those already considered. If we consider the general case of an $M$ -dimensional projection space, the optimal linear projection for which the variance of the projected data is maximized is now defined by the $M$ eigenvectors $u_1, ..., u_M$ of the data *Covariance matrix* $S$ *corresponding to the* $M$ *largest eigenvalues* $\lambda_1, ..., \lambda_M$. This is easily shown using proof by induction.

To summarize, Principal Component Analysis involves:

- Evaluating the mean $\bar{x}$ of the data set.
- Evaluating the Covariance matrix $S$ of the data set.
- Finding the $M$ eigenvectors of $S$ corresponding to the $M$ largest eigenvalues.

Note that the computational cost of computing the full eigenvector decomposition for a matrix of size $D \times D$ is $O(D^3)$.If we plan to project our data onto the first $M$ Principal Components, then we only need to find the first $M$ eigenvalues and eigenvectors. This can be done with more efficient techniques or alternatively we can make use of the ***EM algorithm***.

## 2.2 Probabilistic PCA

The formulation of $PCA$ discussed in the previous section was based on a linear projection of the data onto a subspace of lower dimensionality than the original data space. We now show that PCA can also be expressed as the **Maximum Likelihood** solution of a *Probabilistic* latent variable model. This reformulation of PCA, known as ***Probabilistic PCA***, brings several advantages compared with conventional PCA:

- Probabilistic PCA represents a constrained form of the Gaussian distribution in which the number of free parameters can be restricted while still allowing the model to capture the dominant correlations in a data set.
- We can derive an *EM Algorithm* for PCA that is computationally efficient in situations where only a few leading eigenvectors are required and that avoids having to evaluate the data covariance matrix as an intermediate step.
- The combination of a probabilistic model and EM allows us to deal with missing values in the data set.
- Mixtures of probabilistic PCA models can be formulated in a principled way and trained using the EM Algorithm.

*Probabilistic PCA* is a simple example of the linear-Gaussian framework, in which all of the marginal and conditional distributions are *Gaussian*. We can formulate Probabilistic PCA by first introducing an explicit latent variable $z$ corresponding to the Principal Component Subspace. Next we define a Gaussian prior distribution $p(z)$ over the latent variable, together with a Gaussian conditional distribution $p(x|z)$ for the observed variable $x$ conditioned on the value of the latent variable. Specifically, the prior distribution over $z$ is given by a zero-mean unit-covariance Gaussian

$$p(z) = \mathcal{N}(z|0, I)$$

Similarly, the Conditional Distribution of the observed variable $x$, conditioned on the value of the latent variable $z$, is again Gaussian, of the form

$$p(x|z) = \mathcal{N}(x|Wz + \mu, \sigma^2 I)$$

in which the mean of $x$ is a general linear function of $z$ governed by the $D \times M$ matrix $W$ and the $D$ dimensional vector $\mu$. Note that this factorizes with respect to the elements of $x$, in other words this is an example of the naive *Bayes model*. **The columns of W span a linear subspace within the data space that corresponds to the principal subspace**.
The other parameter in this model is the scalar $\sigma^2$ governing the $Variance$ of the Conditional Distribution. Note that there is no loss of generality in assuming a zero mean, unit Covariance Gaussian for the latent Distribution $p(z)$ because a more general Gaussian Distribution would give rise to an equivalent Probabilistic Model. We can view the Probabilistic PCA model from a generative viewpoint in which a sampled value of the observed variable is obtained by first choosing a value for the latent variable and then sampling the observed variable conditioned on this latent value. Specifically, the $D$ dimensional observed variable $x$ is defined by a linear transformation of the $M$ dimensional latent variable $z$ plus additive Gaussian $'noise'$, so that

$$x = Wz + \mu + \epsilon$$

where $z$ is an $M$ dimensional Gaussian latent variable, and $\epsilon$ is an $D$ dimensional zero$-$ mean Gaussian distributed *noise* variable with $Covariance$ $\sigma^2 I$ :

$$z \sim \mathcal{N}(0, I)$$

and

$$\epsilon \sim \mathcal{N}(0, \sigma^2 I)$$

The introduction of a probabilistic model facilitates the use of the ***Expectation Maximization (EM) Algorithm*** (Dempster et al. 1977) to estimate the latent variables. We first present the $PPCA-EM$ Algorithm for complete data set. This framework was established by Tipping Bishop (1999) using an iterated, two-step process.

- **The Expectation (E) step:**
  the hidden (unknown) variables are estimated from the observations and the current values of the parameters. Statistical moments of the latent variables, $z_n$ and $z_n z_n^T$, are estimated using $p(z_n|x_n, W, \sigma^2)$, the conditional probability density of $z_n$ given the observations $x_n$ and the current values of $W$ and $\sigma^2$, (Little and Rubin 1987). Following the probability assumptions for $z$ and $\epsilon$, one obtains,
  $$E[z_n] = (W^T W + \sigma^2 I)^{-1} W^T (x_n - \bar{x})$$
  and
  $$E[z_n z_n^T] = \sigma^2 M^{-1} + E[z_n] E[z_n]^T$$
  where $M$ is the $M \times M$ matrix
  $$M = W^T W + \sigma^2 I$$

- **The Maximization (M) step:**
  new estimates of the parameters, $W$ and $\sigma^2$, are computed by maximizing the conditional expectation of the log-likelihood, $\ell$, with respect to the conditional probability density of the unknown variables $z_n$ given the known variables $x_n$, $p(z_n|x_n, W, \sigma^2)$. The log-likelihood is defined in terms of the joint probability of the observed variables and the latent variables:
  $$\ell = \sum_{n=1}^{N} lnp(x_n, z_n)$$
  So, we obtain the $M$- step equations
  $$W_{new} = \left( \sum_{n=1}^{N} (x_n - \bar{x}) E[z_n]^T \right) \left( \sum_{n=1}^{N} E[z_n z_n^T] \right)^{-1}$$
  and
  $$\sigma^2_{new} = \frac{1}{ND} \sum_{n=1}^{N} \left( \|x_n - \bar{x}\|^2 - 2E[z_n]^T W_{new}^T (x_n - \bar{x}) + Tr(E[z_n z_n^T] W_{new}^T W_{new}) \right)$$

The $EM$ Algorithm for Probabilistic PCA proceeds by initializing the parameters and then alternately computing the sufficient statistics of the latent space posterior distribution using the $E-$ *step* equations and revising the parameter values using the equations in the $M-$ *step*.

Please consider that the calculations from which these equations arise are beyond the purpose of this report, so they are not further discussed. The reader can find details in references and the suggested textbooks.

Another elegant feature of the $EM$ approach is that we can take the limit $\sigma^2 \to 0$, corresponding to standard PCA, and still obtain a valid EM-like algorithm (Roweis,1998). From $E$ and $M$ steps above, we see that the only quantity we need to compute in the $E$ step is $E[z_n]$. Furthermore, the $M$ step is simplified because $M = W^T W$. To emphasize the simplicity of the algorithm, let us define $\tilde{X}$ to be a matrix of size $D \times N$ whose $n^{th}$ row is given by the vector $x_n - \bar{x}$ and similarly define $\Omega$ to be a matrix of size $D \times M$ whose $n^{th}$ row is given by the vector $E[z_n]$.
The **E step** of the EM Algorithm for PCA then becomes

$$\Omega = (W_{old}^T W_{old})^{-1} W_{old}^T \tilde{X}$$

and the **M step** takes the form

$$W_{new} = \tilde{X}^T \Omega^T (\Omega \Omega^T)^{-1}$$

These equations have a simple interpretation as follows. The E step involves an orthogonal projection of the data points onto the current estimate for the principal subspace. Correspondingly, the M step represents a re-estimation of the principal subspace to minimize the squared reconstruction error in which the projections are fixed.

In both cases, $\sigma^2 \to 0$ *or not* , the $E$ and $M$ steps are repeated until a suitable *convergence criterion* is satisfied.

## 2.3 Kernel PCA

Many linear parametric models can be re-cast into an equivalent 'dual represen- tation' in which the predictions are also based on linear combinations of a *Kernel Function* evaluated at the training data points. For models which are based on a fixed *nonlinear feature space* mapping $\phi(x)$, the ***Kernel Function*** is given by the relation

$$k(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

From this definitionwe see that the Kernel is a symmetric function of its arguments so that $k(x_i, x_j) = k(x_j, x_i)$.Given any algorithm that can be expressed solely in terms of dot products, this *trick* allows us to construct different nonlinear versions of it
The concept of a Kernel formulated as an inner product in a feature space allows us to build interesting extensions of many well-known algorithms by making use of the *Kernel Trick*, also known as *Kernel substitution*. The general idea is that, if we have an algorithm formulated in such a way that the input vector $x$ enters only in the form of scalar products, then we can replace that scalar product with some other choice of Kernel.

**Popular Kernels:**

- Gaussian Kernel:
$$K(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2)$$
- Polynomial Kernel:
$$K(x_i, x_j) = (1 + x_i x_j)^p$$
- Hyperbolic Tangent:
$$K(x_i, x_j) = tanh(x_i x_j + \delta)$$

When this technique of kernel substitution is applied to principal component analysis, we obtain a nonlinear generalization called *Kernel PCA* (Schölkopf et al., 1998).
Consider a data set $x_n$ of observations, where $n = 1, ..., N$, in a space of dimensionality $D$. In order to keep the notation uncluttered, we shall assume that we have already subtracted the sample mean from each of the vectors $x_n$ , so that $\sum_{n=1}^{N} x_n = 0$. The first step is to express conventional PCA in such a form that the data vectors $x_n$ appear only in the form of the scalar products $x_n^T x_m$. Recall that the principal components are defined by the eigenvectors $u_i$ of the covariance matrix $Su_i =_i u_i$ where $i = 1, ..., D$. Here $D \times D$ Covariance matrix $S$ is defined by

$$S = \frac{1}{N} \sum_{n=1}^{N} x_n x_n^T$$

and the eigenvectors are normalized such that $u_i^T u_i = 1$.

Now consider a nonlinear transformation $\phi(x)$ into an $M-$ dimensional feature space, so that each data point $x_n$ is thereby projected onto a point $\phi(x_n)$. We can now perform standard PCA in the feature space, which implicitly defines a *nonlinear Principal Component* model in the original data space. The $M \times M$ sample covariance matrix in feature space is given by

$$C = \frac{1}{N} \sum_{n=1}^{N} \phi(x_n) \phi(x_n)^T$$

and its eigenvector expansion is defined by $Cv_i = \lambda_i v_i$ , $i = 1, ..., M$.
Our goal is to solve this eigenvalue problem without having to work explicitly in the feature space.

From the definition of $C$, the eigenvector equations tells us that $v_i$ satisfies

$$\frac{1}{N}\sum_{n=1}^{N}\phi(x_n)(\phi(x_n^T)v_i = \lambda_i v_i$$

and so we see that (provided $\lambda_i > 0$ ) the vector $v_i$ is given by a linear combination of the $\phi(x_n)$ and so can be written in the form

$$v_i = \sum_{n=1}^{N}\alpha_{in}\phi(x_n)$$

Substituting this expansion back into the eigenvector equation, we obtain

$$\frac{1}{N}\sum_{n=1}^{N}\phi(x_n)\phi(x_n)^T\sum_{m=1}^{N}\alpha_{im}\phi(x_m) = \lambda_i\sum_{n=1}^{N}\alpha_{in}\phi(x_n)$$

The key step is now to express everything in terms of the kernel function $k(x_n, x_m) = \phi(x_n^T)\phi(x_m)$ , which we do by multiplying both sides by $\phi(x_l)^T$ to give

$$\frac{1}{N}\sum_{n=1}^{N}k(x_l, x_n)\sum_{m=1}^{M}\alpha_{im}k(x_n, x_m) = \lambda_i\sum_{n=1}^{N}\alpha_{in}k(x_l, x_m)$$

This can be written in matrix notation as $K^2a_i = \lambda_i NKa_i$. where $a_i$ is an $N-$ dimensional column vector with elements $a_{ni}$ for $n = 1, ...N$. We can find solutions for $a_i$ by solving the following eigenvalue problem: $Ka_i = \lambda_i Na_i$. The normalization condition for the coefficients $a_i$ is obtained by requiring that the eigenvectors in feature space be normalized ($1 = v_i^T v_i$). Having solved the eigenvector problem, the resulting principal component projections can then also be cast in terms of the kernel function so that, the projection of a point $x$ onto eigenvector $i$ is given by:

$$y_i(x) = \phi(x)^T v_i = \sum_{n=1}^{N}\alpha_{in}\phi(x)^T\phi(x_n) = \sum_{n=1}^{N}\alpha_{in}k(x, x_n)$$

and so again is expressed in terms of the kernel function. In the original $D-$ dimensional $x$ space there are $D$ orthogonal eigenvectors and hence we can find at most $D$ linear principal components. The dimensionality $M$ of the feature space, however, can be much larger than $D$ (even infinite), and thus we can find a number of nonlinear principal components that can exceed D. Note,however, that the number of nonzero eigenvalues cannot exceed the number $N$ of data points, because (even if $M > N$ ) the covariance matrix in feature space has rank at most equal to N . This is reflected in the fact that kernel PCA involves the eigenvector expansion of the $N \times N$ matrix $K$. So far we have assumed that the projected data set given by $\phi(x_n)$has zero mean, which in general will not be the case. We cannot simply compute and then subtract off the mean, since we wish to avoid working directly in feature space, and so again, we formulate the algorithm purely in terms of the kernel function. The projected data points after centralizing, denoted $\bar{\phi}(x_n)$ are given by

$$\bar{\phi}(x_n) = \phi(x_n) - \frac{1}{N}\sum_{l=1}^{N}\phi(x_l)$$

and the corresponding elements of the Gram matrix,expressed in matrix notation, are given by:

$$\bar{K} = K - 1_N K - K1_N + 1_N K1_N$$

where $1_N$ denotes the$N \times N$ matrix in which every element takes the value $\frac{1}{N}$. Thus we can evaluate $\bar{K}$ using only the Kernel function and then use $\bar{K}$ to determine the eigenvalues and eigenvectors.
One obvious disadvantage of kernel PCA is that it involves finding the eigenvectors of the $N \times N$ matrix $\bar{K}$ rather than the $D \times D$ matrix $S$ of conventional linear PCA, and so in practice for large data sets approximations are often used. Finally, we note that in standard linear PCA, we often retain some reduced number $L < D$ of eigenvectors and then approximate a data vector $x_n$ by its projection onto the $L-$ dimensional principal subspace, defined by

$$\hat{x} = \sum_{i=1}^{L}(x_n^T u_i)u_i$$

# 3 Analysis

## 3.1 Theoretical Problem

### 3.1.1 Part (i): Synthetic Data

```
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=1000, factor=0.3, noise=0.05)
```

We consider here the nonlinear problem *concentric circles* . We assume two class problem where the triangle shapes represent one class and the circle shapes represent another class,respecticely:

We shall perform PCA and PCa Kernel Methods and compare the results.

**PCA method**, either with *Eigendecomposition* or $SVD$ Algorithms, and **Probabilistic PCA**, either with $\sigma^2 \neq 0$ or $\sigma^2 \to 0$ : None of these techniques is able to produce results suitable for training a linear classifier.

**Initial plot of the** *concentric cicles*

```
from sklearn.datasets import make_circles
from matplotlib import pyplot as plt

X, y = make_circles(n_samples=1000, factor=0.3, noise=0.05)

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

plt.legend(numpoints =1,loc='lower right')
plt.title('Concentric circles')
plt.savefig("Initial plot")
plt.show()
```

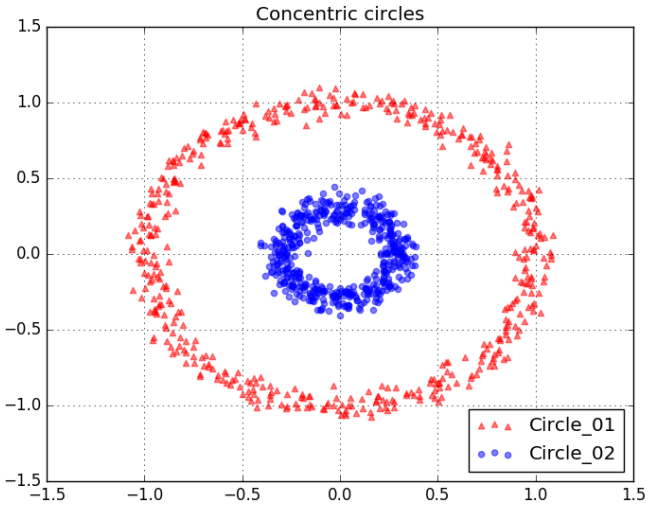



Figure 1: Concentric Circles Initial Plot

**PCA with Eigendecomposition**

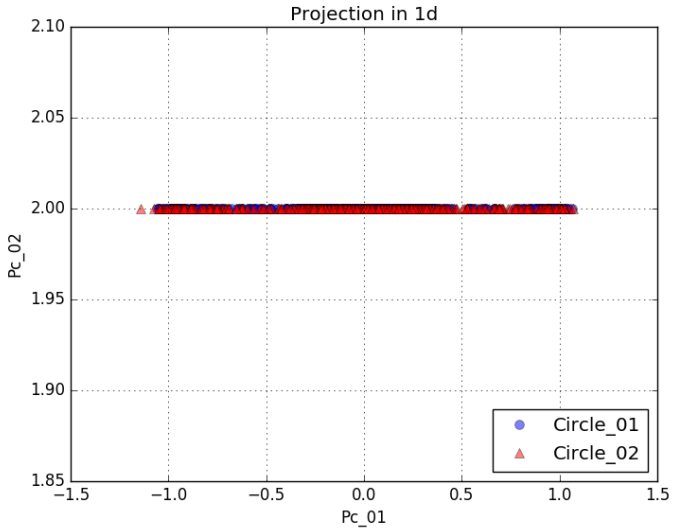



Figure 2: (a) Projection in 1d

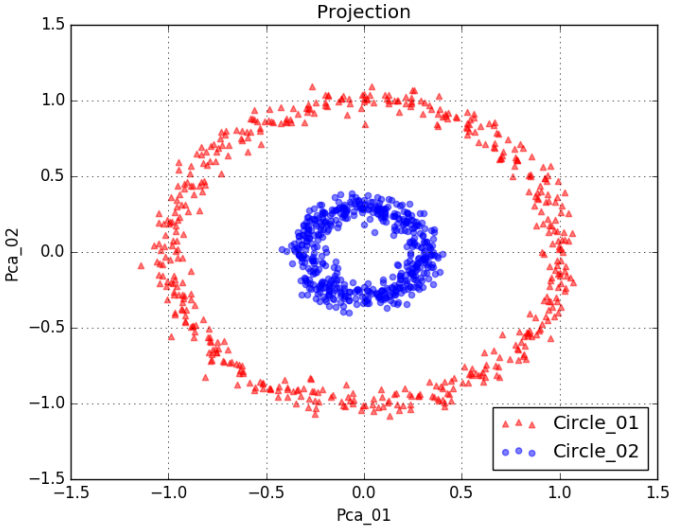



Figure 3: (b) Projection in 2d

**PCA with SVD**

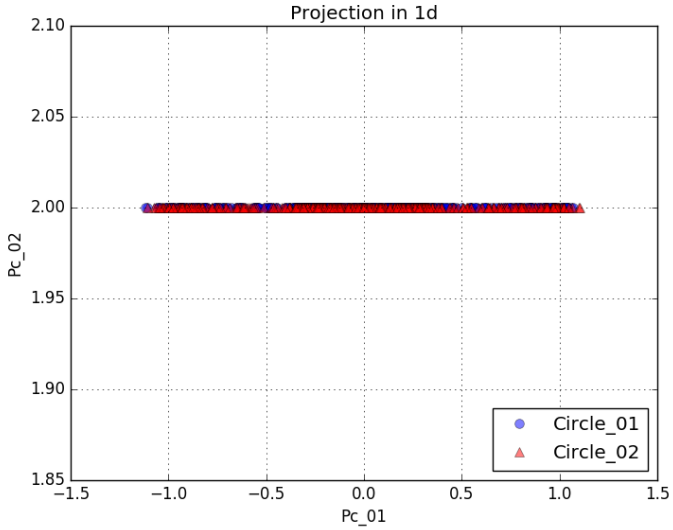



Figure 4: (a) Projection in 1d

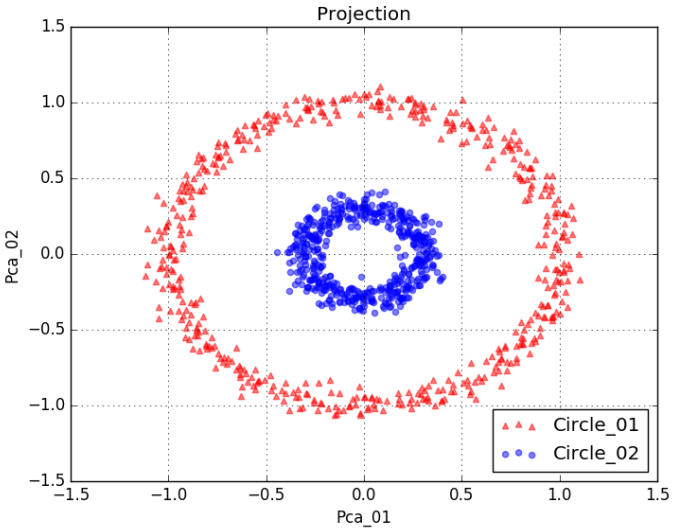



Figure 5: (b) Projection in 2d

**PPCA with $\sigma^2 = 0$**

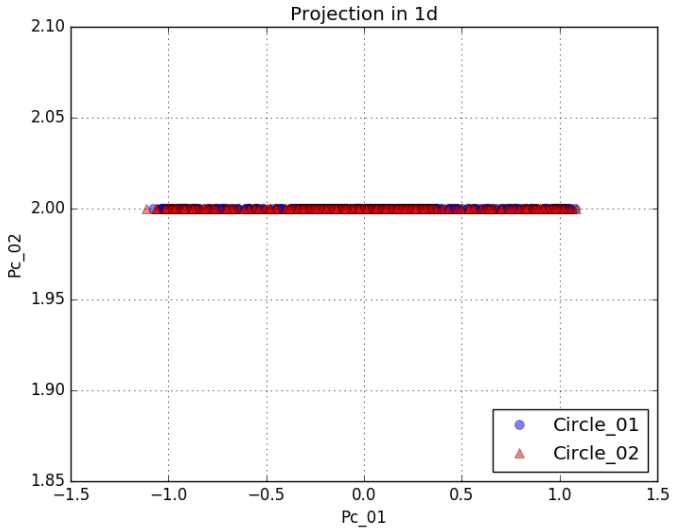



Figure 6: (a) Projection in 1d

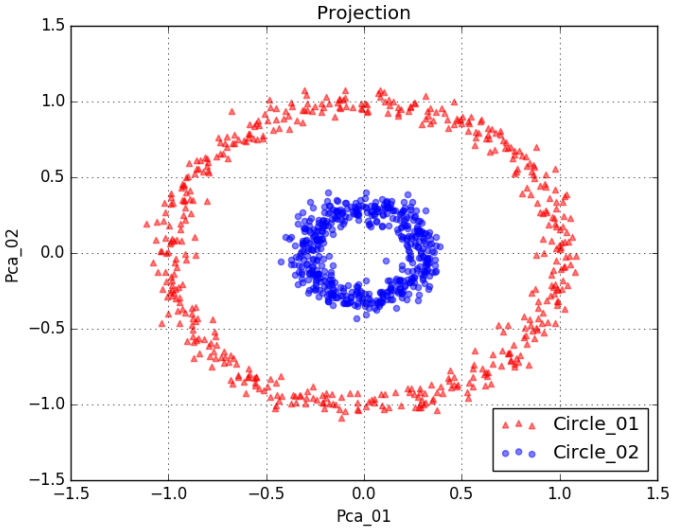



Figure 7: (b) Projection in 2d

**PPCA with $\sigma^2 \neq 0$**

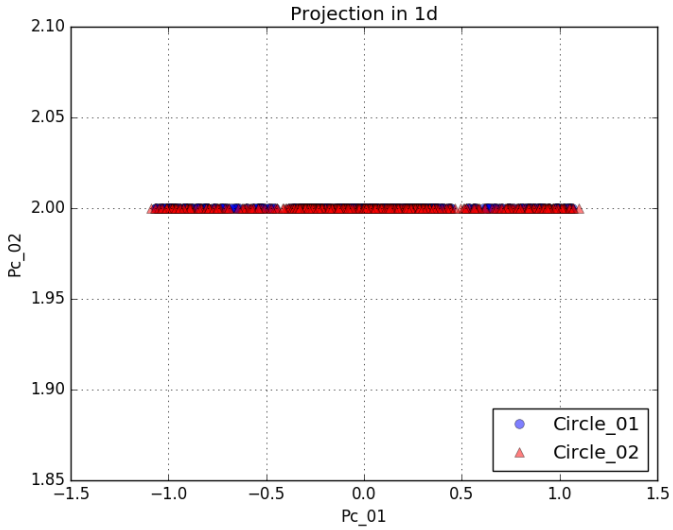



Figure 8: (a) Projection in 1d

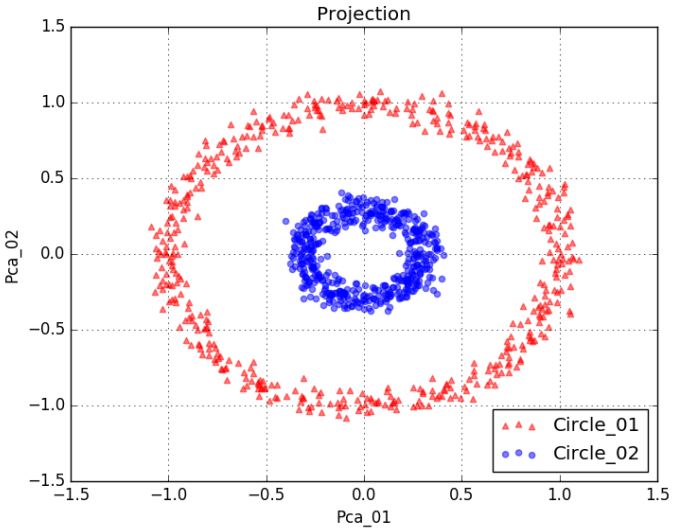



Figure 9: (b) Projection in 2d

**Kernel methods**

As we already discussed, there is a variety of different Kernel Functions. Choosing the most appropriate kernel highly depends on the problem at hand – and fine tuning its parameters can easily become a tedious and cumbersome task. The choice of a Kernel depends on the problem at hand because it depends on what we are trying to model. A polynomial kernel, for example, allows us to model feature conjunctions up to the order of the polynomial. On the other hand, Radial basis functions, such as Gaussian Kernel, allow us to pick out circles (or hyperspheres). The motivation behind the choice of a particular kernel can be very intuitive and straightforward depending on what kind of information we are expecting to extract about the data.

Here, we perform Kernel PCA for the three Kernels *Gaussian*, *Polynomial* and *Hyperbolic Tangent*.

Gaussian Kernel PCA, as it was expected, projected the data onto a new subspace where the two classes become linearly separable:

**Kernel PCA with Gaussian Kernel**

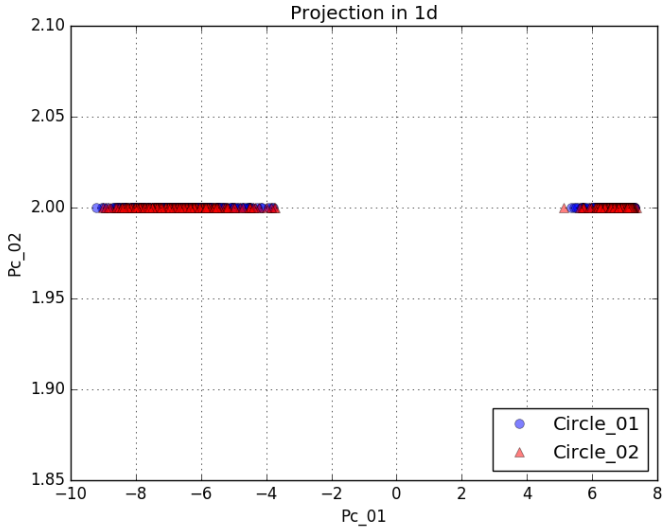



Figure 10: (a) Projection in 1d $\gamma = 5$

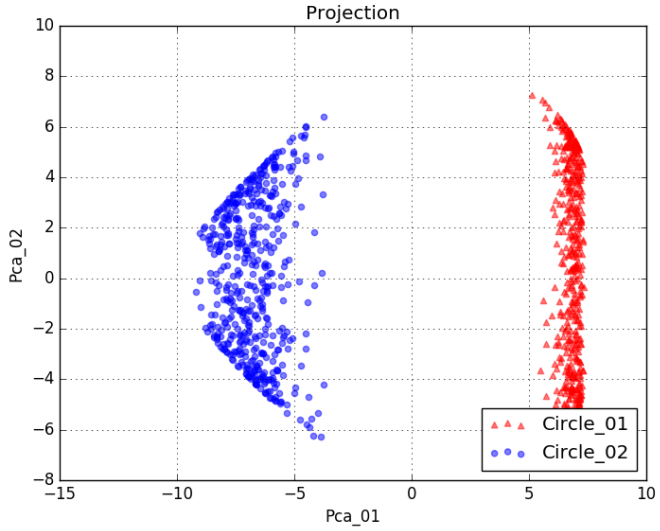



Figure 11: (b) Projection in 2d $\gamma = 5$

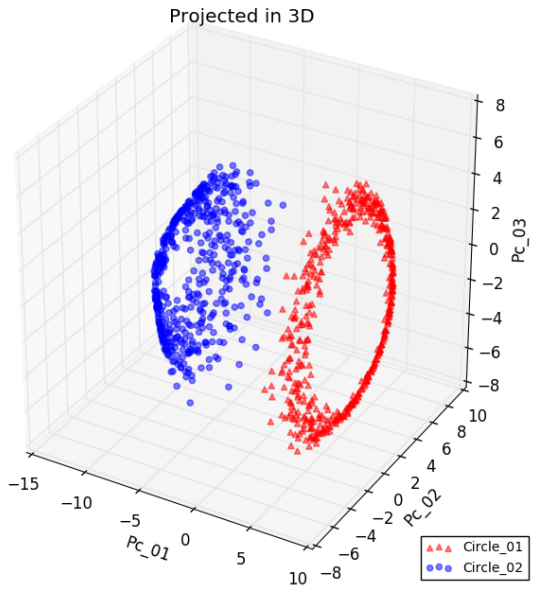



Figure 12: (c) Projection in 3d $\gamma = 5$

However, the other two Kernel functions *Polynomial* and *Hyperbolic Tangent* do not behave in the same way:

**Kernel PCA with Polynomial Kernel**

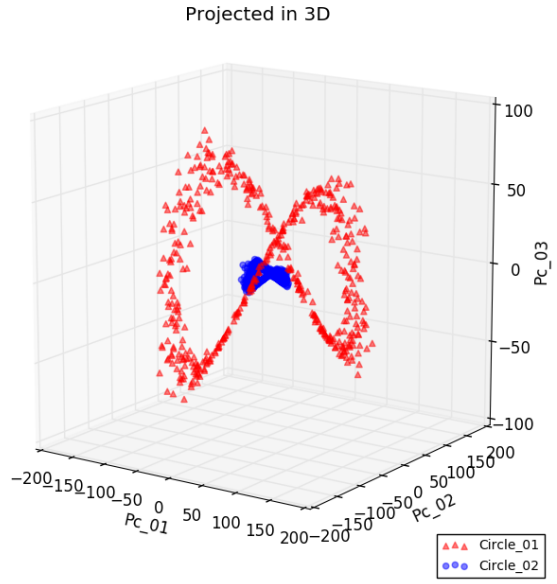



Figure 13: Projection in 3d and $p = 4$

**Kernel PCA with Tangent Kernel**

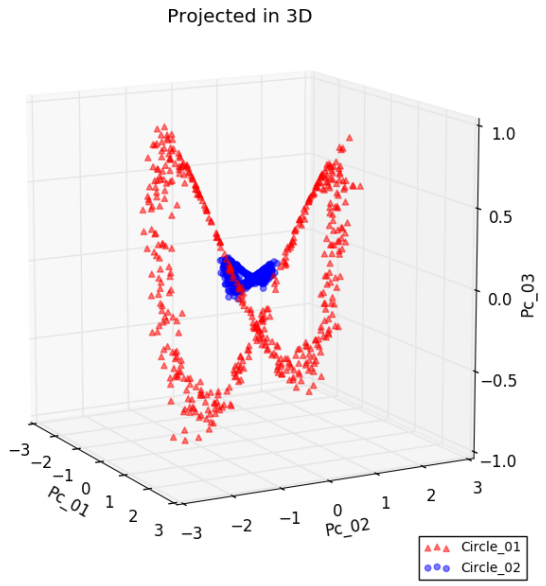



Figure 14: Projection in 3d and $\delta = 2$

Now, we change the *noise* parameter in the concentric circles to $noise = 3$ and perform *Gaussian Kernel* :

**Kernel PCA with Gaussian Kernel**

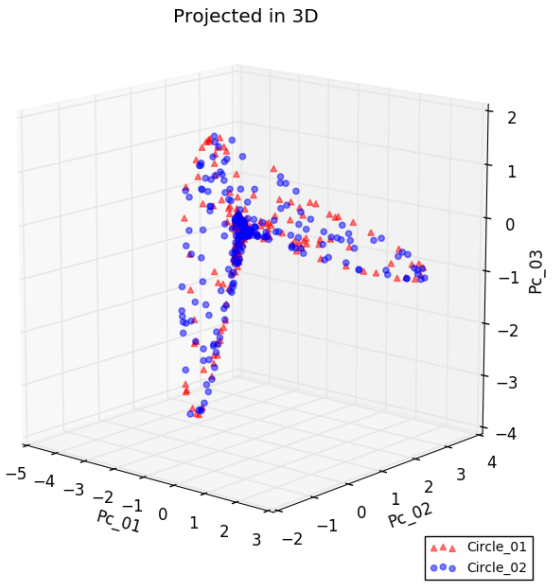



Figure 15: (a) Projection in 3d $\gamma = 5$

For $\gamma = 20, 40, 60$ we have respectively:

**Kernel PCA with Gaussian Kernel**

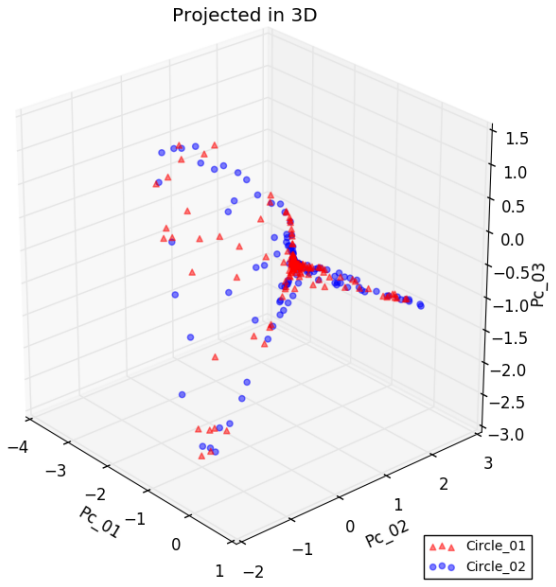



Figure 16: Projection in 3d $\gamma = 20$

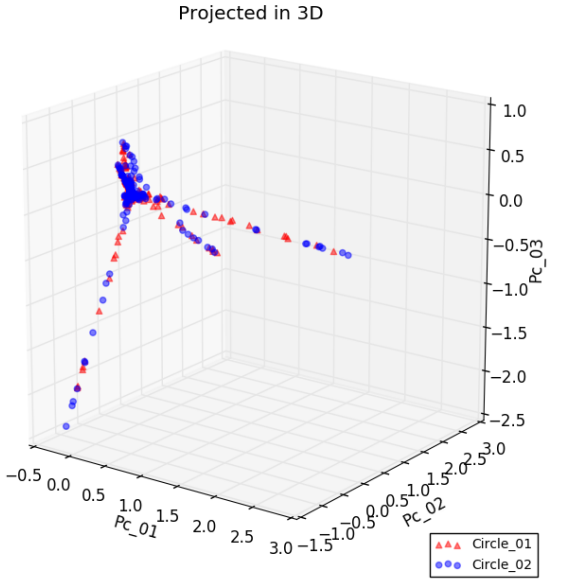



Figure 17: Projection in 3d $\gamma = 40$

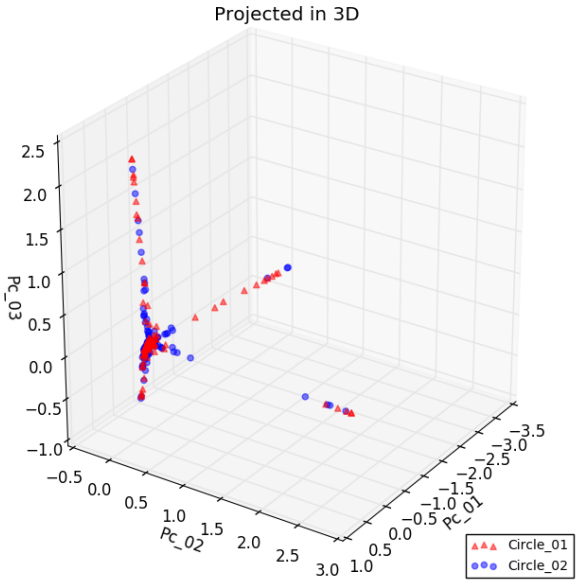



Figure 18: Projection in 3d $\gamma = 60$

Increasing the *noise* in the samples data, increases the complexity between them ,so it is difficult to make them separate. It is interesting, yet, to see that for different values of $\gamma$ the *projection* tends to be orthogonal with respect to each principal component.

### 3.1.2 Part (ii): Plot of Eigenvalue spectrum

In this part, we generate a dataset $X$ which is randomly Gaussian distributed with mean a $10 \times 1$ vector with 1 in every position and 10 samples and make a plot of the eigenvalues in descending order. Then, we keep only 5 samples and inspect the eigenvalues. Below is a combined plot for both of the eigenvalues spectrum: While reducing dimensionality $10 \rightarrow 5$ *columns* , the number of *observations* is larger than the number of *samples*, in other words $D < N$. So, what we expect to see is that, in the second spectrum, $D - N$ eigenvalues tend $\rightarrow 0$ while all the others are *raised*, as a result of the *noise* in Covariance Matrix.

The reader is encouraged to read more about the *noise* in covariance matrices and the determination of the eigenvalue spectrum when the number of observations is limited in the [5].

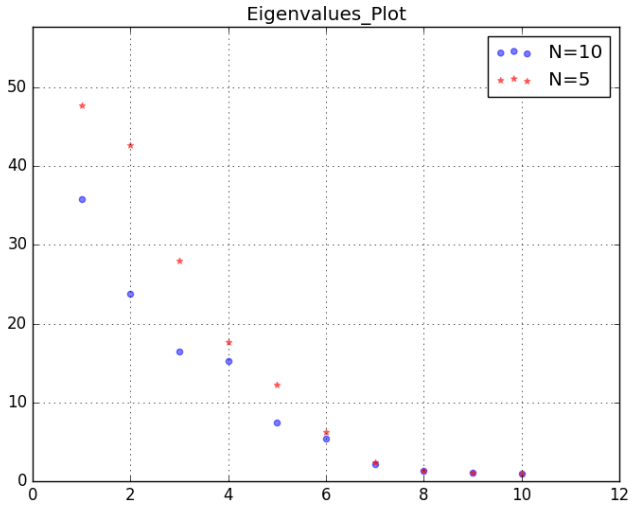



Figure 19: Eigenvalues Spectrum

**Plot of the eigenvalue spectrum for the** 10 **samples set (circle shapes) and for the** 5 **samples set (star shapes).**

.

## 3.2 Practical Problem

The aim of this exercise is to perform Principal Component Analysis in a real data set. 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 implemented *PCA Algorithm* is good approach for this data set especially in comparison to PCA analysis with the partiular $built-in$ module in *Python*.

The number of the observations, as mentinoed above, is 51, while the number of dimensions 45281. We perform Principal Component Analysis, using the *EM Algorithm*, for $\sigma^2 = 0$ and $\sigma^2 > 0$. As we mentioned in theory, In both cases, $\sigma^2 \to 0$ *or not* , the $E$ and $M$ steps of the algorithm are repeated until a suitable *convergence criterion* is satisfied. Here, we make use of **Root Mean Squared Error (RMSE)** for the convergence of the $W$ Matrix.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Basically, it is the square root of the mean of the square of all of the error. The use of $RMSE$ is very common and it makes an excellent general purpose error metric for numerical predictions.Compared to the similar Mean Absolute Error, RMSE amplifies and severely punishes large errors.
For the convergence of $\sigma^2$ we make use of

$$\|\sigma^2_{new} - \sigma^2_{old}\|$$

**EM Algorithm with $\sigma^2 = 0$**

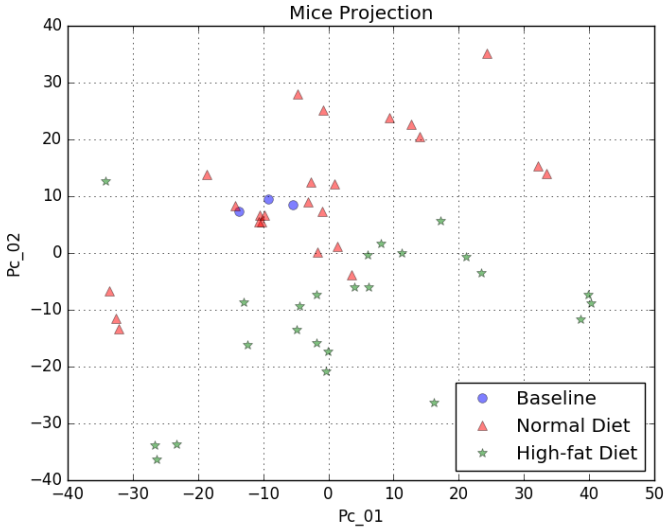



Figure 20: Projection in 2d, and number of iterations: 10

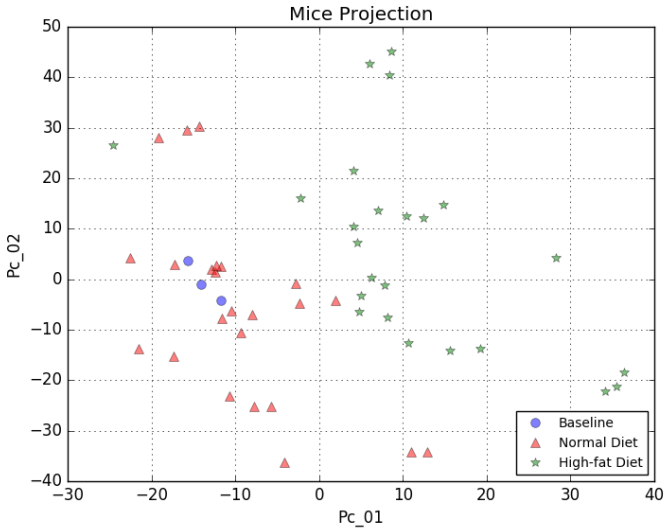



Figure 21: Projection in 2d, and number of iterations: 10

**EM Algorithm with $\sigma^2 = 0$**

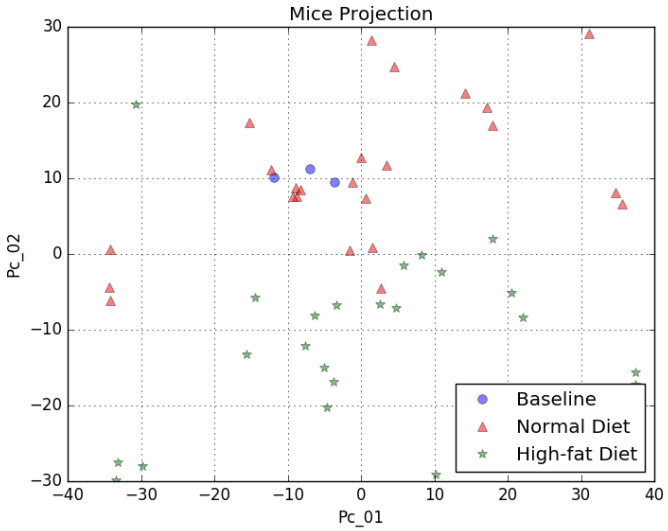



Figure 22: Projection in 2d, and number of iterations: 20

The previous plot is one of the various ones that gained after performing, all displaying the same projection ,only rotated a little bit. But, the key point is that it is very similar to the projection of built in function for PCA algorithm:

**PCA Analysis**

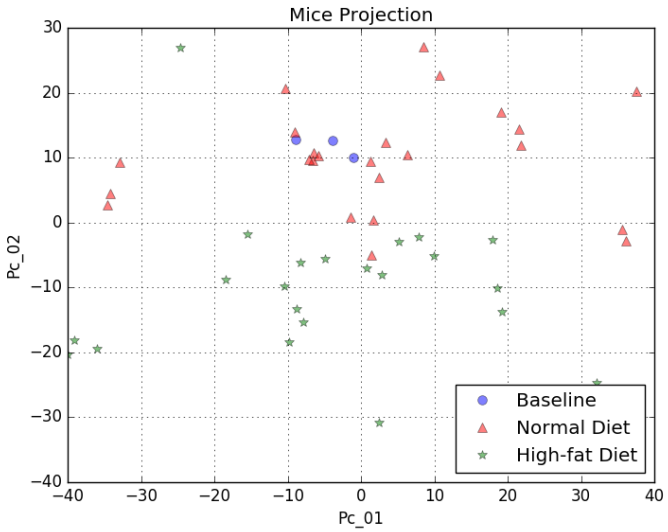



Figure 23: Projection in 2d using PCA built algorithm

**Number of Iterations** refers to the number of calls of EM steps, so as to get a *mean* $W_{new}$ matrix.

**EM Algorithm with $\sigma^2 > 0$**

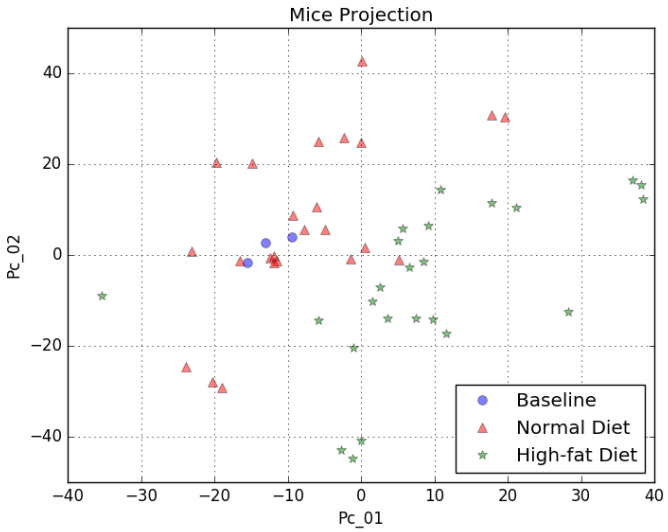



Figure 24: Projection in 2d, and number of iterations: 10

**EM Algorithm with $\sigma^2 > 0$**

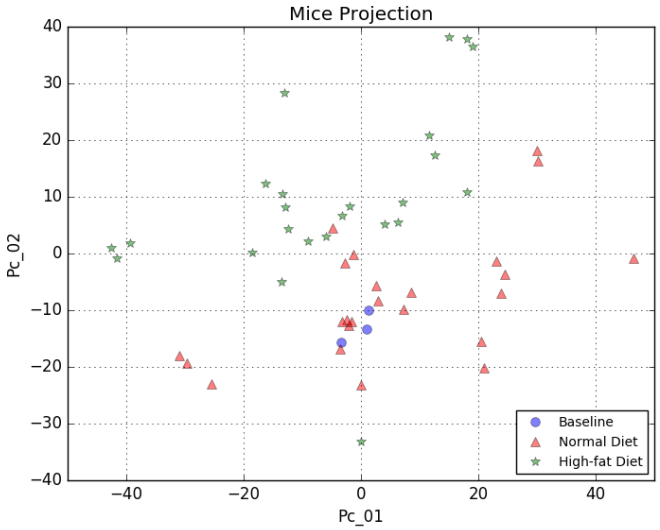



Figure 25: Projection in 2d, and number of iterations: 20

**PCA Analysis**

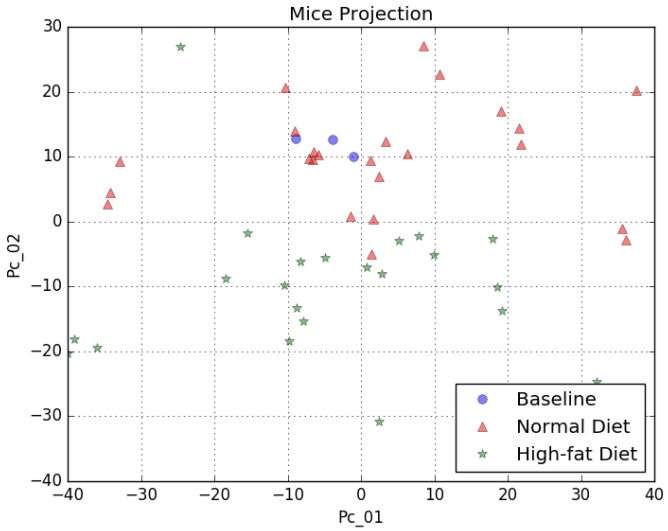



Figure 26: Projection in 2d using PCA built algorithm

In this case as well, we get the same projection close to PCA Algorithm, only the subspace is rotated.

# 4 Python Codes

File: main.py

```
import time
import os
from Principal_Methods 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 selected____****\n")

  Sub_quest = input("Choose Subquestion for the Theoretical part\n Enter A for Subquestion (i) and B for Subquestion (ii): ")

  if Sub_quest == 'A':
    print("****____Theoretical Part: Subquestion (i)____****")

    Method = input("Choose Method: Enter PCA for Principal Component Analysis,\nEM for Probabilistic PCA and \nKERNEL for Kernel Method: ")
    Dimension = int(input("Give Dimensionality of Projection\n Notice that for either PCA or EM should be 1 or 2:  "))
    M = Dimension

    ##  Data given for the theoretical problem . Subquestion (i) with different noises:

    n = input("Enter A for 0.05 noise: ")
    if n == 'A':
      X, y = make_circles(n_samples=1000,noise=0.05, factor=0.3)
    else:
      X, y = make_circles(n_samples=1000,noise=3, factor=0.3)

    Data = X.T

    if Method == 'PCA':
      PCA(Data,y,M)
    elif Method == 'EM':
      PPCA(Data,y,M)
    else:
      KERNEL(Data,y,M)
  else:
    print("****____Theoretical Part: Subquestion (ii)____****\n")

    ##  Data given for the theoretical problem . Subquestion (ii):

    mean = np.array([1,1,1,1,1,1,1,1,1,1])#; print(mean.shape)
    cov  = np.eye((10))#; print(cov.shape)

    Xa = np.random.multivariate_normal(mean,cov,10)
    Xb = Xa[:,0:5]

    Theoretical_II(Xa,Xb)

else:
  y = [0]
  print("****____Practical Problem has been selected____****\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...')

  X = np.loadtxt("Final.txt")     ## Constructing the Data Array
  print(X.shape)

  Method = input("Choose Method: Enter PCA for Principal Component Analysis, EM for Probabilistic PCA and KERNEL for Kernel Method: ")
  Dimension = int(input("Give Dimensionality of Projection:"))
  M = Dimension

  if Method == 'PCA':
    PCA(X,y,M)
  elif Method == 'EM':
    PPCA(X,y,M)
  else:
    KERNEL(X,y,M)
```

```
78
79    ##### PCA with Built in class
80
81    from sklearn.decomposition import PCA as pca
82
83    n_components = Dimension
84    my_pca = pca(n_components)
85
86    Projected_Data = my_pca.fit_transform(X.T).T
87
88    if Dimension == 2:
89      Practical_Plots(Projected_Data)
90    else:
91      D_Plots(Projected_Data)
92
93  print("\n\n*****____End of Process____*****\n\n")
94
95
96  Time = time.process_time()
97  print(Time)
```

File: PrincipalMethods.py

```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)

########### 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)


########### 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
```

```
  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()
```

```
435 def D_Plots(Arr):
436
437   fig = plt.figure(figsize=(8,8))
438   ax  = fig.add_subplot(111, projection='3d')
439   plt.rcParams['legend.fontsize'] = 10
440
441   ax.plot(Arr[0,0:3],Arr[1,0:3],Arr[2,0:3], 'o', markersize=7, color='blue', alpha=0.5,
          label='Baseline')
442   ax.plot(Arr[0,3:27],Arr[1,3:27],Arr[2,3:27], '^', markersize=7, color='red', alpha=0.5,
          label='Normal Diet')
443   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')
444
445   ax.grid(True)
446   ax.set_xlabel("Pc_01")
447   ax.set_ylabel("Pc_02")
448   ax.set_zlabel("Pc_03")
449   plt.title('Projected Mice 3D')
450   ax.legend(numpoints=1,loc='lower right')
451   plt.savefig("Mice 3d Plot.png")
452   plt.show()
453   plt.close()
```

# 5 Suggested Bibliography

## References


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

[2] I. T.Jolliffe *Principal Component Analysis, Second Edition*

[3] Sebastian Raschka *Python Machine Learning*

[4] Lingbo Yu, Robert R. Snapp, Teresa Ruiz and Michael Radermacher *Probabilistic Principal Component Analysis with Expectation Maximization (PPCA-EM) Facilitates Volume Classification and Estimates the Missing Data* https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3353830/

[5] Richard Everson, Stephen Roberts *Inferring the eigenvalues of covariance matrices from limited, noisy data* http://empslocal.ex.ac.uk/people/staff/reverson/uploads/Site/spectrum.pdf

[6] César Souza, *Kernel Functions for Machine Learning Applications* http://crsouza.com/2010/03/17/kernel-functions-for-machine-learning-applications/#kernel_choosing