Technical University of Crete
School of Electrical and Computer Engineer

ΤΗΛ 311 – Statistical Modelling and Pattern Recognition – 2024

Instructor: Vassilis Digalakis | **1st Set Exercises** | Resolve up: 19/05/2024

Report

Section 1: Principal Component Analysis (PCA)

In this exercise we will first use Principal Component Analysis (PCA) on a dataset describing benign and malignant breast tumours. We will then apply PCA to a larger dataset of 5000 facial images.

Part 1:

1. First, we read the file `data/breast_cancer_data.csv` which contains 569 samples, each of which is described by a vector of 30 attributes. The last value in the vector is the classification of the sample into benign (0) and malignant (1).

   From the perspective of code implementation, the following results are achieved:

```
Data=csvread('data/breast_cancer_data.csv');
NSamples = 100; %Get the first NSamples only for better
   visualization
X=Data(1:NSamples,1:end-1); % Get all features
Y=Data(1:NSamples,end);
```

Listing 1: Code for loading the dataset

2. We then select several pairs and visualise the samples in 2D space, introducing another benchmark, the Mahalanobis distance, to confirm the visual results. In a p-dimensional space, the Malanobis distance is defined as:

$$D(x, y) = \sqrt{(x - y)^T \cdot S^{-1}(x - y)} \tag{1}$$

   where x,y are p-dimensional column vectors representing points in space and S^{-1} is the data covariance matrix. We prefer the Mahalanobis distance as opposed to the Euclidean distance because it does not require normalization in our samples; of course, if normalization with mean 0 and variance 1 is applied, it is easily shown that the Mahalanobis distance converts to the Euclidean distance.

From the perspective of code implementation, we have:

```
% Define the feature pairs you want to plot
featurePairs = [3 4; 17 10; 9 12; 1 12; 1 2];  % Each row defines a pair of features to plot
numPairs = size(featurePairs, 1);

% Compute covariance matrix and mean vector
C = cov(X);
mu = mean(X);

for i = 1:numPairs
    % Create a new figure for each pair
    figure;

    % Plot the ith pair of features
    plot(X(Y==0, featurePairs(i, 1)), X(Y==0, featurePairs(i, 2)), 'bo', ...
         X(Y==1, featurePairs(i, 1)), X(Y==1, featurePairs(i, 2)), 'ro');

    % Compute Mahalanobis distance for each observation
    numObs = size(X, 1);  % Number of observations
    D = zeros(numObs, 1);  % Initialize array to store distances
    for j = 1:numObs
        x = X(j, featurePairs(i, :));  % Extract the features for the j-th observation
        D(j) = sqrt((x - mu(featurePairs(i, :))) * inv(C(featurePairs(i, :), featurePairs(i, :))) * (x - mu(featurePairs(i, :)))');
    end

    % Set plot properties
    axis square;
    xlabel(sprintf('Feature %d', featurePairs(i, 1)));
    ylabel(sprintf('Feature %d', featurePairs(i, 2)));
    title(sprintf('Feature %d vs Feature %d (Mahalanobis Distance)', featurePairs(i, 1), featurePairs(i, 2)));

    % Display Mahalanobis distance in the title
    distanceMean = mean(D);
```

```
33      titleText = sprintf('Feature %d vs Feature %d (Mean
            Mahalanobis Distance: %.2f)', featurePairs(i, 1),
            featurePairs(i, 2), distanceMean);
34      title(titleText);
35  end
36
37  pause
```

Listing 2: Code for displaying different samples in 2D space

We also receive the following images:

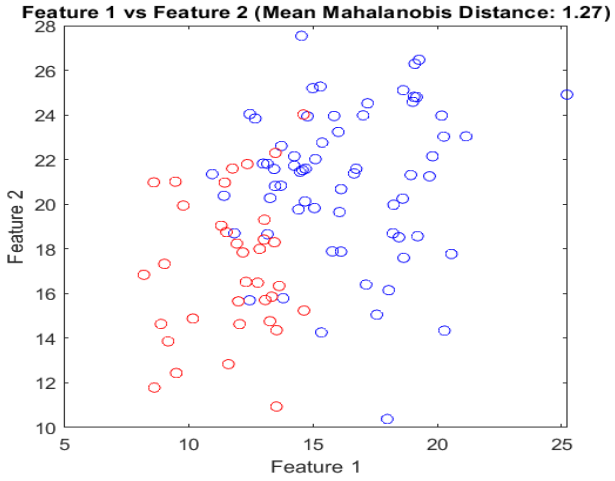

Figure 1: Απεικόνιση Feature 1 vs Feature 2

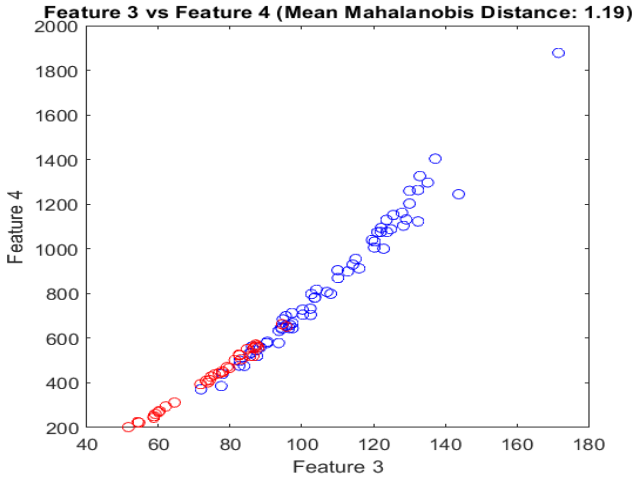

Figure 2: Απεικόνιση Feature 3 vs Feature 4

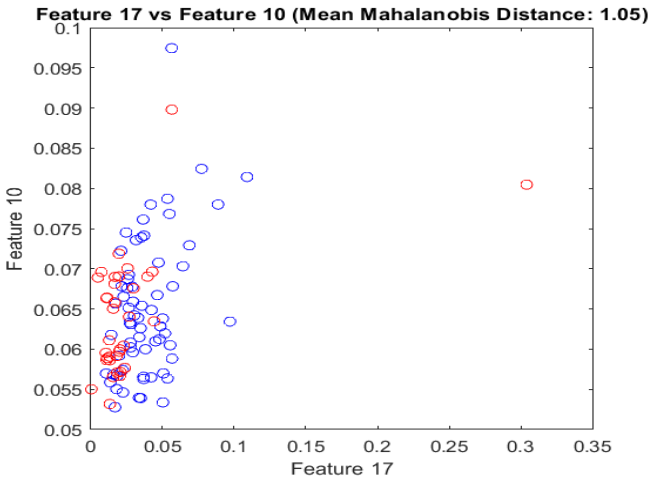

Figure 3: Απεικόνιση Feature 17 vs Feature 10

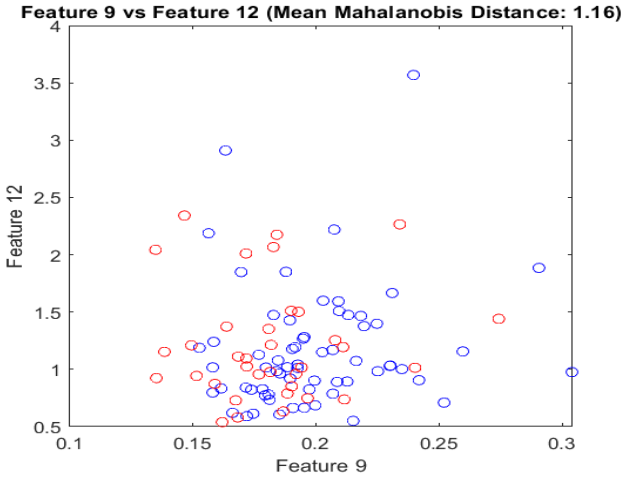

Figure 4: Απεικόνιση Feature 9 vs Feature 12

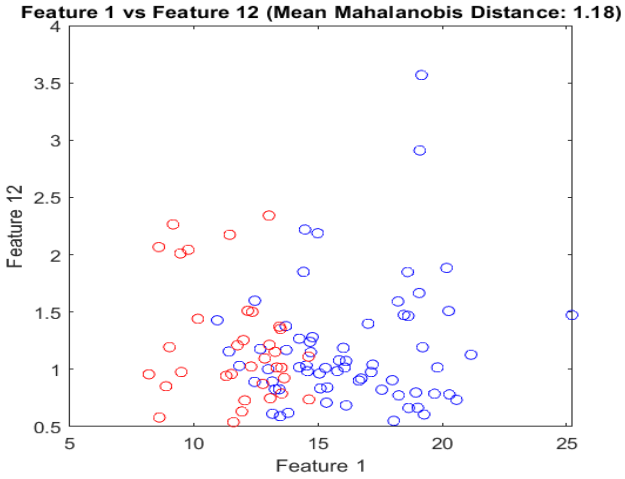

Figure 5: Απεικόνιση Feature 1 vs Feature 12

We continue by normalising our original samples with a mean of zero and a variance of 1. This is implemented using the FeatureNormalise function, whose code is

```
function [X_norm, mu, sigma] = featureNormalize(X)
% Get the number of samples and features
[nSamples, nFeat] = size(X);

% Initialize output variables --- Preallocation
X_norm = zeros(nSamples, nFeat);
mu = zeros(1, nFeat);
sigma = zeros(1, nFeat);

% Normalize each feature
for j = 1:nFeat
    mu(j) = mean(X(:, j));
    sigma(j) = std(X(:, j));
    X_norm(:, j) = (X(:, j) - mu(j)) / sigma(j);
end

end
```

Listing 3: Code for standardization

And we get the following picture:

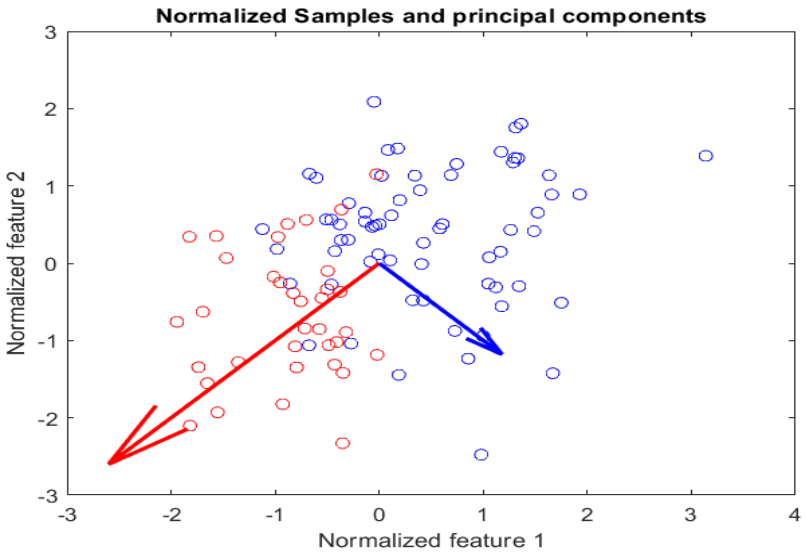

Figure 6: Κανονικοποιημένα δείγματα και Principal Components

In the same figure, the principal components are plotted through the myPCA function as follows:

```
function [ eigenval, eigenvec, order] = myPCA(X)

```

```
% Useful values
[m, n] = size(X);
eigenvec = zeros(m);
eigenval = zeros(n);

% Make sure each feature from the data is zero mean
X_centered = X - mean(X);

% ====================== YOUR CODE HERE ======================

% Compute the covariance matrix
Sigma = (1/(m)) * (X_centered' * X_centered);


% Compute eigenvalues and eigenvectors
[V, D] = eig(Sigma);

% Extract the diagonal of D as a vector
eigenval = diag(D);

% Sort the eigenvalues in descending order and get the order
[eigenval,order]=sort(eigenval,1,'descend'); %Sort them

% Reorder the eigenvectors according to the sorted eigenvalues
eigenvec=V(:,order); %Corresponding eigenvectors
```

Listing 4: Code for creating the principal components

And we get the following picture:

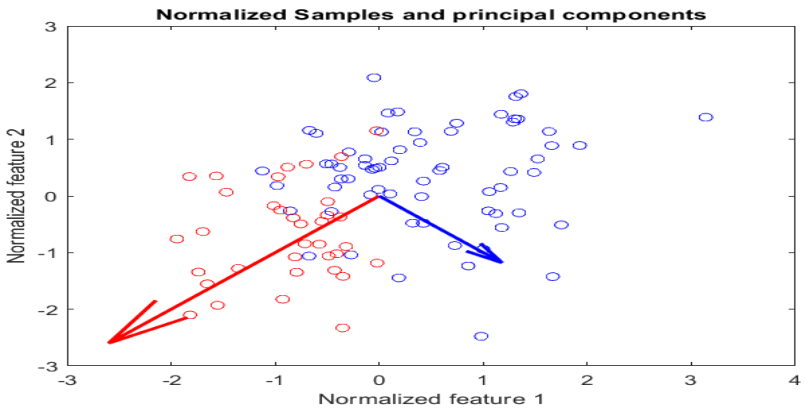

Figure 7: Normalized Samples and Principal Components

Following the implementation of standardisation, it is evident that the samples are centred around

the mean.Additionally, it is observed that the principal components of the PCA are perpendicular to each other.

Furthermore, we calculate the contribution of each component (explained variance) to the total variance. We define as explained variance:

$$Var(PC_i) = \frac{\lambda_i}{\sum_{j=1}^{L} \lambda_j}$$

Σε επίπεδο MATLAB έχουμε:

```
ExplainedVar = eigvals/sum(eigvals);
fprintf(' Explained Variance(1st PC = %f) (2nd PC = %f)\n',
   ExplainedVar(1), ExplainedVar(2));
```

Listing 5: Code for the calculation of explained variance

The numerical data obtained are enumerated as follows:

Explained Variances:
1st Principal Component: 0.687891
2nd Principal Component: 0.312109

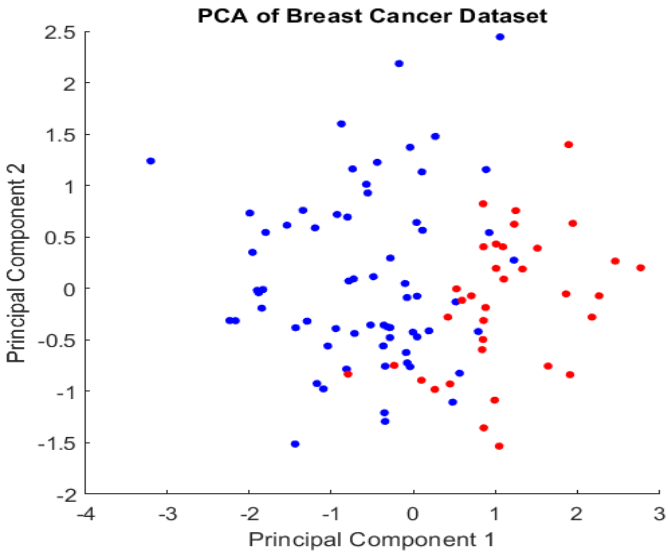

Figure 8: View vectors on the principal components

In order to reduce the dimension of our vectors, we properly populate the projectData function:

```
function Z = projectData(X, U, K)
Z = zeros(size(X, 1), K);

% Instructions: Compute the projection of the data using only
   the top K
%               eigenvectors
```

```
6  %
7
8  Z = X * U(:, 1:K);
9
10
11 end
```

Listing 6: Code for dimensional reduction

The projection of the data into a reduced spatial domain results in the following image:

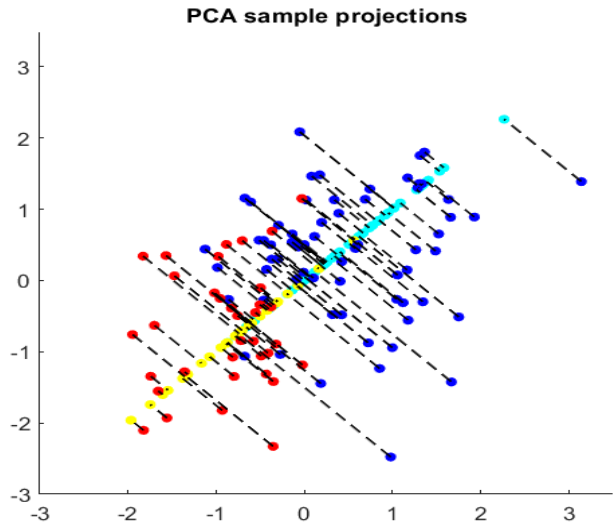

Figure 9: Ανακτημένα δείγματα και προβολή στον χώρο των υψηλών διαστάσεων

If we repeat the above procedure, this time using the entire feature vector (30 features), we obtain the following image:

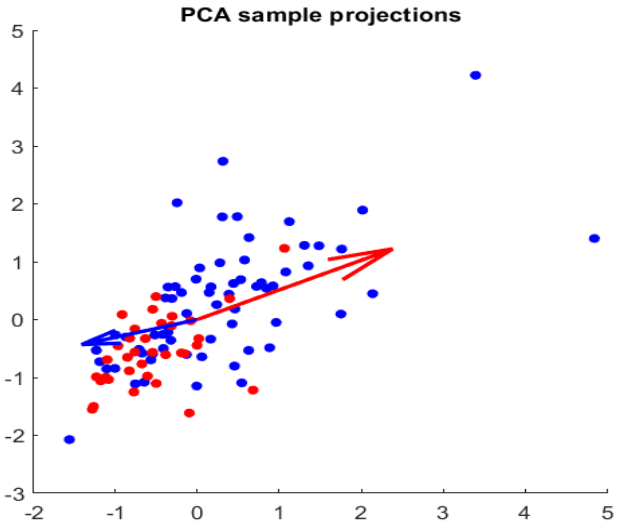

Figure 10: Projection of samples on the main components

In this image, we can see that the principal components do not appear to be equal, however, our analysis in this case has used all 30 features while projecting the samples in 2D space; therefore, it is expected that deformations are present, but theoretically they are vertical.

Part 2:

In this part, we repeat the above procedure using the data from the 5000 face images given. We start by plotting the first 100 faces from our dataset using the display function and obtain the following image:

Figure 11: The first 100 persons face design

Then we apply standardisation and calculate the principal components using the myPCA function and finally we draw a new image with the first 36 principal components and the result is as follows:

Figure 12: Design of faces with the first 36 main components

We observe that the faces show high alteration of their features and the colour tends to fluctuate around dark grey.

Next, we reduce the dimension of our samples using the first 100 principal components and then plot the reduced dimension samples, after projecting them into the original space, of course:

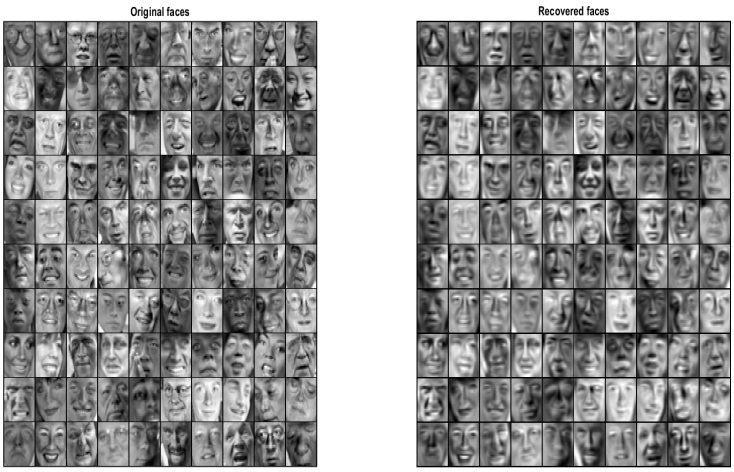

Figure 13: Comparison of our original samples with the reduced dimension samples

Section 2: LDA (Linear Discriminant Analysis) classifier design

In this part we will design an LDA classifier and compute the projection vector. To begin, we express the Fisher criterion as a function of the matrices S_w and S_b :

$$J(w) = \frac{w^T S_B w}{w^T S_w w}$$

We multiply and equate to zero to find the maximum of $J(w)$:

$$\begin{aligned}
\frac{d}{dw}J(w) = \frac{d}{dw}\left(\frac{w^T S_B w}{w^T S_w w}\right) = 0 \\
\Rightarrow (w^T S_w w)\frac{d}{dw}(w^T S_B w) - (w^T S_B w)\frac{d}{dw}(w^T S_w w) = 0 \\
\Rightarrow (w^T S_w w)2S_B w - (w^T S_B w)2S_B w = 0
\end{aligned}$$

Divide by $2w^T S_w w$:

$$\begin{aligned}
\Rightarrow S_B w - J(w)S_w w = 0 \\
\Rightarrow S_B^{-1} S_B w - J(w)w = 0
\end{aligned}$$

The following step is the resolution of the generalized eigenvalue problem.

$$S_w^{-1} S_B w = \lambda w$$

where $\lambda = J(w)$ and we have:

$$w^* = \text{argmax} J(w) = \text{argmax}\left(\frac{w^T S_B w}{w^T S_w w}\right) = S_w^{-1}(\mu_A - \mu_B)$$

Let us now calculate the table S_w :

$$\begin{aligned}
S_w &= P(\omega_A)\Sigma_A + P(\omega_B)\Sigma_B \\
&= 0.4\begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} + 0.6\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \\
&= \begin{bmatrix} 1.2 & 0.4 \\ 0.4 & 0.8 \end{bmatrix} + \begin{bmatrix} 1.2 & 0.6 \\ 0.6 & 1.8 \end{bmatrix} \\
&= \begin{bmatrix} 2.4 & 1 \\ 1 & 2.6 \end{bmatrix}
\end{aligned}$$

Moreover,

$$\mu_A - \mu_B = \begin{bmatrix} 0 \\ 2 \end{bmatrix} - \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$$

The inverse of the matrix ${S_w}^{-1}$ is given by the formula:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

So we have:

$${S_w}^{-1} = \frac{1}{2.4 \cdot 2.6 - 1 \cdot 1}\begin{bmatrix} 2.6 & -1 \\ -1 & 2.4 \end{bmatrix} = \begin{bmatrix} 0.496 & -0.19 \\ -0.19 & 0.458 \end{bmatrix}$$

At last we have that:

$$w = \begin{bmatrix} 0.496 & -0.19 \\ -0.19 & 0.458 \end{bmatrix} \cdot \begin{bmatrix} -2 \\ 2 \end{bmatrix} = \begin{bmatrix} -1.372 \\ 1.296 \end{bmatrix}$$

Part 2:

To calculate the projection of the vectors $x_A = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$ and $x_B = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$ we will use the formula:

$$y = w^T x$$

Therefore, we have:

$$y_1 = w^T x_A = 2.516 \text{ and } y_2 = w^T x_B = -2.82$$

Section 3: Linear Discriminant Analysis(LDA) vs PCA

Part 1:

In this exercise we will use Linear Discriminant Analysis to reduce the dimension of a feature vector and compare the results with the PCA method.

So we start by loading our data and standardising with a mean of 0 and a variance of 1 and we get the following picture:

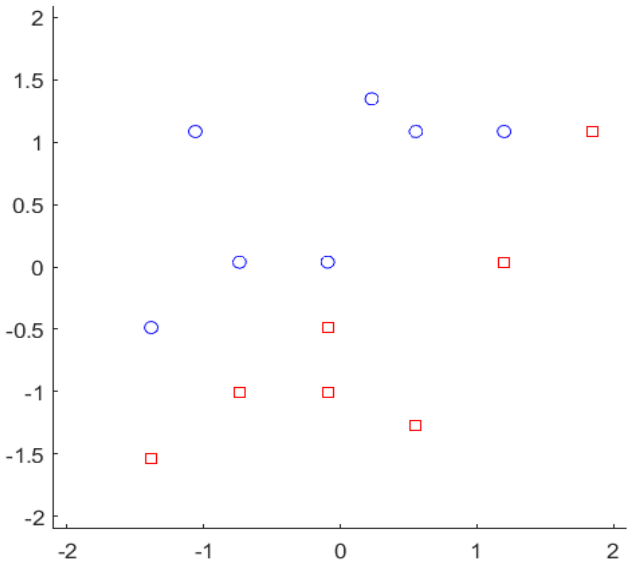

Figure 14

Then we correctly implement the fisherLinearDisciminant function, reducing the dimension in our samples:

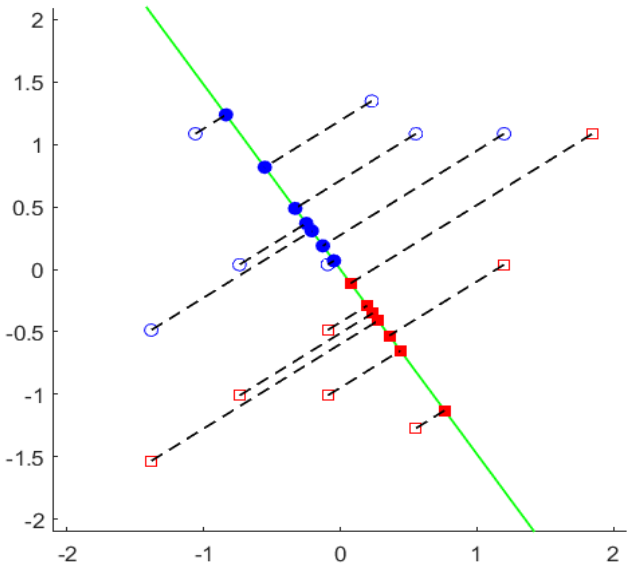

Figure 15: Comparison of our original samples with the reduced dimension samples

We follow a similar procedure with the PCA method and the picture we get is as follows:

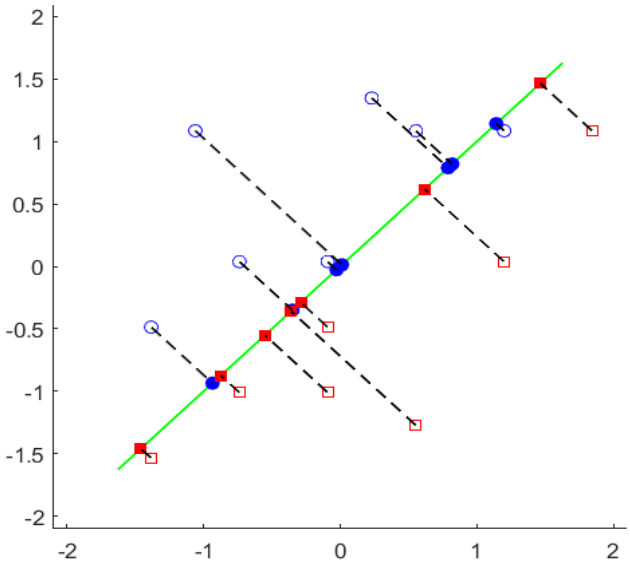

Figure 16: Comparison of our original samples with the reduced size samples

Part 2:

In this part, we apply the LDA algorithm to a very popular dataset in machine learning, the Iris database, which contains 3 different species of the Iris flower family.As part of our analysis, we load our samples and perform a standardisation with mean 0 and variance 1 and plot the first 2 features in a 2D space and obtain the following image:

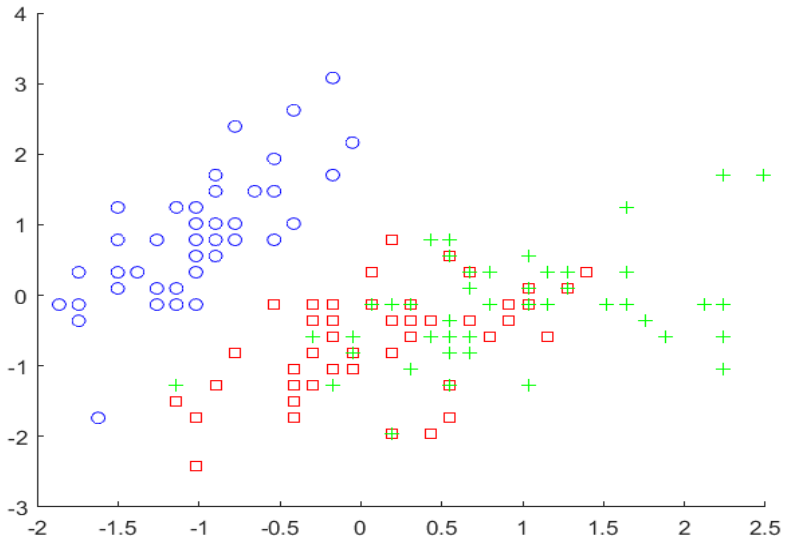

Figure 17: Plot the first 2 features from the Iris set.

Subsequently, the LDA function is implemented, which is the algorithm that has been requested. The code that is utilised is as follows:

```
function A = myLDA(Samples, Labels, NewDim)
% Input:
%   Samples: The Data Samples
%   Labels: The labels that correspond to the Samples
%   NewDim: The New Dimension of the Feature Vector after applying
   LDA

%A=zeros(NumFeatures,NewDim);

[NumSamples, NumFeatures] = size(Samples);
NumLabels = length(Labels);
if(NumSamples ~= NumLabels) then
    fprintf('\nNumber of Samples are not the same with the Number
       of Labels.\n\n');
    exit
end
    Classes = unique(Labels);
    NumClasses = length(Classes);  %The number of classes

% Initialize matrices
Sw = zeros(NumFeatures, NumFeatures); % Within-class scatter matrix
m0 = mean(Samples, 1); % Global mean
Sb = zeros(NumFeatures, NumFeatures); % Between-class scatter
```

```
    matrix
22
23  %For each class i
24  %Find the necessary statistics
25      for i = 1:NumClasses
26          % Class-specific samples and statistics
27          classSamples = Samples(Labels == Classes(i), :);
28          mu(i, :) = mean(classSamples, 1); % %Calculate the Class
                Mean
29          P(i) = size(classSamples, 1) / NumSamples; %Calculate the
                Class Prior Probability
30
31          % Within-class scatter -- Calculate the Within Class
                Scatter Matrix
32          classScatter = classSamples - mu(i, :); % Deviation from
                mean
33          Sw = Sw + (classScatter' * classScatter); % Sum of squares
34
35          % Between-class scatter -- %Calculate the Between Class
                Scatter Matrix
36          meanDiff = (mu(i, :) - m0)';
37          Sb = Sb + P(i) * (meanDiff * meanDiff');
38      end
39
40  %Eigen matrix EigMat=inv(Sw)*Sb
41  EigMat = inv(Sw)*Sb;
42
43  %Perform Eigendecomposition
44  [eigenvectors, eigenvalues] = eig(EigMat);
45
46  % Extract eigenvalues as a vector
47  eigenvalues = diag(eigenvalues);
48
49  % Sort eigenvalues (and vectors) in descending order
50  [sortedEigenvalues, sortOrder] = sort(eigenvalues, 'descend');
51  sortedEigenvectors = eigenvectors(:, sortOrder);
52
53
54  %% You need to return the following variable correctly.
55  % Select the NewDim eigenvectors corresponding to the top NewDim
        eigenvalues
56  % Assuming they are NewDim <= NumClasses - 1
```

```
57 A = sortedEigenvectors(:, 1:NewDim);  % Return the LDA projection
       vectors
```

Listing 7: Κώδικας για την υλοποίηση του αλγόριθμου LDA

In conclusion, the projection vectors calculated with LDA are applied to the original samples with the projectDataLDA function, with the objective of reducing their dimension to 2. The following picture is obtained:

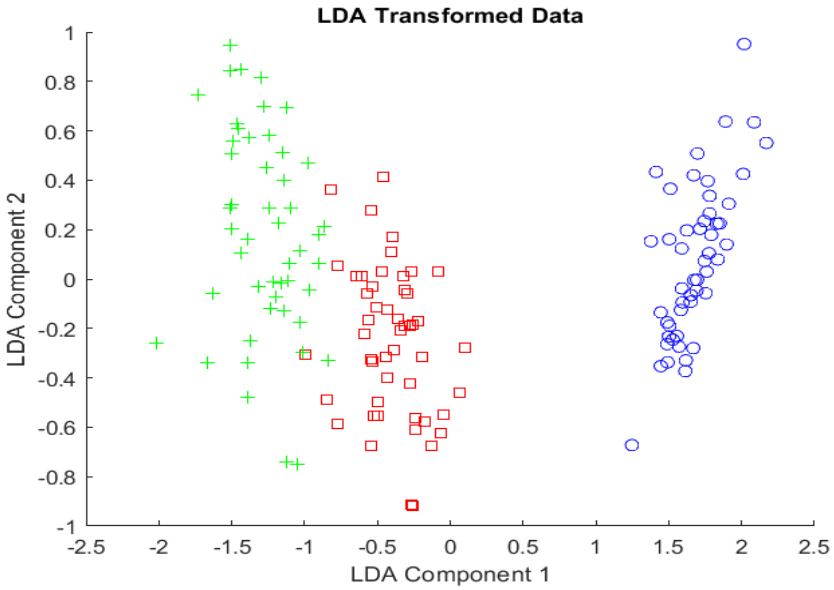

Figure 18: Projecting the samples via the projectDataLDA function.

Section 4: Bayes

For two classes following a multivariate normal distribution with mean values μ_1, μ_2 and covariance tables Σ_1, Σ_2 the decision boundary is given by:

$$P(\omega_1|x) = P(\omega_2|x)$$

However, as a consequence of Bayes' rule, the aforementioned equation is to be transformed as follows:

$$P(x|\omega_1)P(\omega_1) = P(x|\omega_2)P(\omega_2)$$

The calculation of the aforementioned equation yields the following result:

$$\log(P(x|\omega_1)P(\omega_1)) = \log(P(x|\omega_2)P(\omega_2)) \iff$$
$$\log(P(x|\omega_1)) + \log(P(\omega_1)) = \log(P(x|\omega_2)) + \log(P(\omega_2))$$

Of course, for the probability $P(x|\omega_1)$ and $P(x|\omega_2)$ follow a multivariate normal distribution so the PDF is:

$$f_X(X) = \frac{1}{2\pi^{\frac{n}{2}}|\Sigma_X|^{\frac{1}{2}}} \cdot exp\left(-\frac{1}{2}(X-\mu_X)^T\Sigma_X^{-1}(X-\mu_X)\right)$$

Combining the aforementioned relationships, the following relationships can be deduced:

$$
\begin{aligned}
&-\frac{1}{2}(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1)+\log(P(\omega_1))-\frac{1}{2}\log(|\Sigma_1|)\\
&=-\frac{1}{2}(x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)+\log(P(\omega_2))-\frac{1}{2}\log(|\Sigma_2|) \iff\\
&(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1)-(x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)\\
&=\log\frac{|\Sigma_2|}{|\Sigma_1|}+2\log\left(\frac{P(\omega_2)}{P(\omega_1)}\right)
\end{aligned}
$$

The subsequent step will be to express the equation in a more familiar form.
We expand the term: $(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1)$ and we have:

$$(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1)=x^T\Sigma_1^{-1}x-2\mu_1^T\Sigma_1^{-1}x+\mu_1^T\Sigma_1^{-1}\mu_1$$

Similar for the other terms we have:

$$(x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)=x^T\Sigma_2^{-1}x-2\mu_2^T\Sigma_1^{-1}x+\mu_2^T\Sigma_2^{-1}\mu_2$$

Subsequent to the application of the aforementioned equations, the original equation can be expressed in the following manner:

$$
\begin{aligned}
&x^T\Sigma_1^{-1}x-2\mu_1^T\Sigma_1^{-1}x+\mu_1^T\Sigma_1^{-1}\mu_1-(x^T\Sigma_2^{-1}x-2\mu_2^T\Sigma_1^{-1}x+\mu_2^T\Sigma_2^{-1}\mu_2)\\
&=\log\frac{|\Sigma_2|}{|\Sigma_1|}+2\log\left(\frac{P(\omega_2)}{P(\omega_1)}\right) \iff\\
&x^T(\Sigma_1^{-1}-\Sigma_2^{-1})x+2(\mu_2^T\Sigma_2^{-1}-\mu_1^T\Sigma_1^{-1})x+(\mu_1^T\Sigma_1^{-1}\mu_1-\mu_2^T\Sigma_2^{-1}\mu_2)\\
&=\log\frac{|\Sigma_2|}{|\Sigma_1|}+2\log\left(\frac{P(\omega_2)}{P(\omega_1)}\right)
\end{aligned}
$$

Let denote

$$A=\Sigma_1^{-1}-\Sigma_2^{-1} \tag{2}$$

$$B=2(\mu_2^T\Sigma_2^{-1}-\mu_1^T\Sigma_1^{-1}) \tag{3}$$

$$C=\mu_1^T\Sigma_1^{-1}\mu_1-\mu_2^T\Sigma_2^{-1}\mu_2-2\log\left(\frac{P(\omega_2)}{P(\omega_1)}\right)-\log\left(\frac{|\Sigma_2|}{|\Sigma_1|}\right) \tag{4}$$

The decision boundary equation is therefore:

$$x^TAx+B^Tx+C=0 \tag{5}$$

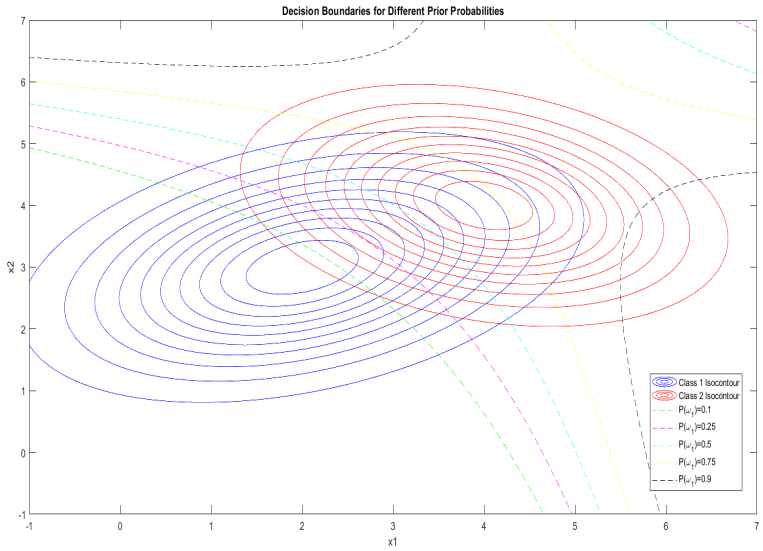

Figure 19: Isopic curves and decision boundaries for different values

As illustrated in Figure, the decision boundaries are characterised by the presence of curves, a consequence of the distinct nature of the covariance matrices.

In the case where the covariance matrices are such that:

$$\Sigma = \Sigma_1 = \Sigma_2 = \begin{bmatrix} 1.2 & 0.4 \\ 0.4 & 1.2 \end{bmatrix}$$

Recalling the previously calculated generalised equation, we have the following result:

$$\begin{aligned} &(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1) - (x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2) \\ &= \log\frac{|\Sigma_2|}{|\Sigma_1|} + 2\log\left(\frac{P(\omega_2)}{P(\omega_1)}\right) \iff \\ &(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) - (x-\mu_2)^T\Sigma^{-1}(x-\mu_2) \\ &= \log\frac{|\Sigma|}{|\Sigma|} + 2\log\left(\frac{P(\omega_2)}{P(\omega_1)}\right) \iff \\ &(x-\mu_1)^T\Sigma^{-1}(x-\mu_1) - (x-\mu_2)^T\Sigma^{-1}(x-\mu_2) \\ &= 2\log\left(\frac{P(\omega_2)}{P(\omega_1)}\right) \end{aligned}$$

Let $y_1 = (x-\mu_1)$ and $y_2 = (x-\mu_2)$ we have that:

$$y_1^T\Sigma^{-1}(y_1) - (y_2)^T\Sigma^{-1}y_2$$

where the last equation is the decision boundary equation.

Section 5: Feature extraction and Bayes Classification.

In this exercise, the objective is to implement a Bayes classifier, which will initially be employed to classify digits 1 and 2 using various features. The first feature for our classification is the aspect ratio, which is implemented as follows in Python:

aspect_ratio

```
def aspect_ratio(image):
  """Calculates the aspect ratio of the bounding box around the foreground
  ↪ pixels."""
  try:
    # Extract image data and reshape it (assuming data is in a column
    ↪ named 'image')
    img = image.values.reshape(28, 28)

    # Find non-zero foreground pixels
    nonzero_pixels = np.nonzero(img)

    # Check if there are any foreground pixels
    if nonzero_pixels[0].size == 0:
      return np.nan  # Return NaN if no foreground pixels found

    # Get minimum and maximum coordinates of foreground pixels
    min_row = np.min(nonzero_pixels[0])
    max_row = np.max(nonzero_pixels[0])
    min_col = np.min(nonzero_pixels[1])
    max_col = np.max(nonzero_pixels[1])

    # Calculate bounding box dimensions
    height = max_row - min_row + 1
    width = max_col - min_col + 1

    # Calculate aspect ratio
    aspect_ratio = width / height if height > 0 else np.nan

    return aspect_ratio
```

We confirm that our calculations are correct by drawing two random samples of our digits inside a parallelogram and then applying min-max normalisation, which we implement in Python as follows:

min_max_scaling

```
def min_max_scaling(X, min_val=-1, max_val=1):
  """Scales features to a range between min_val and max_val."""
  X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
  X_scaled = X_std * (max_val - min_val) + min_val
  return X_scaled
```

We continue to implement the prior probabilities $P(C_1)$ as well as $P(C_2)$ via the train function

which is implemented as follows:

train

```
  def train(self, X, y):
"""
Train the classifier under the assumption of Gaussian distributions:
  calculate priors and Gaussian distribution parameters for each class.

Args:
X (pd.DataFrame): DataFrame with features.
y (pd.Series): Series with target class labels.
"""
self.classes_ = np.unique(y)
for class_label in self.classes_:
  # Filter data by class
  X_class = X[y == class_label]

  # Calculate prior probability for the class
  self.class_priors[class_label] = len(X_class) / len(X)
```

If we further assume that the distribution of the aspect ratio attribute in each class follows a normal distribution with mean $\overline{\mu} = \frac{1}{N}\Sigma_{i=1}^{N} x_i$ and standard deviation $\sigma = \sqrt{\frac{1}{N}\Sigma_{i=1}^{N}(x_i - \overline{\mu})^2}$ the train function is completed as follows:

train

```
# Calculate mean and covariance for the class
# Adding a small value to the covariance for numerical stability
self.class_stats[class_label] = {
  'mean': X_class.mean(),
  'cov': X_class.cov() + 1e-3 * np.eye(X_class.shape[1])
}
```

Then, we calculate the prediction of our classifier and we have:

predict

```
def predict(self, X):
"""
Predict class labels for each test sample in X.

Args:
X (pd.DataFrame): DataFrame with features to predict.
```

```
    Returns:
    np.array: Predicted class labels.
    """
    predictions = []
    for index, sample in X.iterrows():
        log_probs = {}
        for class_label, stats in self.class_stats.items():
            prior = np.log(self.class_priors[class_label])
            mean = stats['mean']
            cov = stats['cov']
            diff = sample.values - mean
            likelihood = multivariate_normal.logpdf(diff, mean=mean,
             ↪  cov=cov)
            log_probs[class_label] = prior + likelihood
        prediction = max(log_probs, key=log_probs.get)
        predictions.append(prediction)
    return np.array(predictions)
```

We estimate the result and the accuracy of our classifier is 0.9587, so even with 1 feature we get quite satisfactory results. Another feature we can introduce is the number of non-black pixels that make up the foreground in each image (foreground_pixels).Implementing the function in Python we get

foreground_pixels

```
def foreground_pixels(image):
    """
    Calculate the pixel density of the image, defined as the
    count of non-zero pixels

    Args:
    image (np.array): A 1D numpy array representing the image.

    Returns:
    int: The pixel density of the image.
    """
    try:
        # Extract image data and reshape it (assuming data is in a column
         ↪  named 'image')
        img = image.values.reshape(28, 28)
```

```
        # Find non-zero foreground pixels
        nonzero_pixels = np.count_nonzero(img)
        if nonzero_pixels == 0:
            print(f"Warning: Couldn't find nonzero pixels on
             ↪ {image.name}")
            return np.nan  # Return NaN if no foreground pixels found
    except (KeyError, ValueError) as e:
        print(f"Error processing image in row  {image.name}: {e}")
        return np.nan  # Return NaN for rows with errors

    return nonzero_pixels
```

In this case, adding another feature increases the accuracy of the classifier, but not dramatically (accuracy = 0.9711).We introduce one last feature, the centroid, which is defined as coordinates in the image plane.We implement the query in Python and we have :

calculate_centroid

```
  def calculate_centroid(image):
"""
Calculate the normalized centroid (center of mass) of the image.

Returns:
tuple: The (x, y) coordinates of the centroid normalized by image
 ↪  dimensions.
"""
# Extract image data and reshape it (assuming data is in a column named
 ↪  'image')
img = image.values.reshape(28, 28)
rows, cols = img.shape

# Calculate total mass (sum of all pixel values)
total_mass = img.sum()

# Calculate x and y center of mass
x_center = (img.sum(axis=0) * np.arange(cols)).sum() / total_mass
y_center = (img.sum(axis=1) * np.arange(rows)).sum() / total_mass

# Create a single scalar as a centroid feature using x+(y * cols) where
 ↪  cols is the width of the image
centroid = x_center + (y_center * cols)
```

```
return centroid
```

We estimate the accuracy of our classifier and get the highest so far, accuracy = 0.9752. However, if we introduce the number 0 in the training symbols, the accuracy of the classifier decreases drastically and we get an accuracy = 0.7156. This is because the third feature does not help much in the classification and also because the symbols 2 and 0 have similar statistical properties.

Section 6: Minumum risk:
From the minimization criterion we know that:

$$l_1 = \lambda_{11}P(\omega_1)P(x|\omega_1) + \lambda_{21}P(\omega_2)P(x|\omega_2) = \lambda_{21}P(\omega_2)P(x|\omega_2)$$

and,

$$l_2 = \lambda_{12}P(\omega_1)P(x|\omega_1) + \lambda_{22}P(\omega_2)P(x|\omega_2) = \lambda_{12}P(\omega_1)P(x|\omega_1)$$

The decision threshold is:

$$l_1 = l_2 \Rightarrow \lambda_{21}P(\omega_2)P(x|\omega_2) = \lambda_{12}P(\omega_1)P(x|\omega_1)$$

Moreover, $P(\omega_1) = P(\omega_2)$ so:

$$\lambda_{21}\frac{x_0}{\sigma_2^2}e^{-\frac{{x_0}^2}{2\sigma_2^2}} = \lambda_{12}\frac{x_0}{\sigma_1^2}e^{-\frac{{x_0}^2}{2\sigma_2^1}}$$

Yet, we know that: $\lambda_{21} = 1, \lambda_{12} = 0.5, \sigma_1 = 1$ and $\sigma_2 = 2$ Consequently:

$$\begin{aligned}
\frac{x_0}{4}e^{-\frac{x_0^2}{8}} &= \frac{x_0}{2}e^{-\frac{x_0^2}{2}} \iff \\
2x_0e^{-\frac{{x_0}^2}{8}} &= 4x_0e^{-\frac{{x_0}^2}{2}} \Rightarrow \\
\ln(e^{-\frac{{x_0}^2}{8}}) &= \ln(2e^{-\frac{{x_0}^2}{2}}) \Rightarrow \\
-\frac{{x_0}^2}{8} &= \ln(2) - \frac{{x_0}^2}{2} \Rightarrow \\
-x_0^2 &= 8\ln(2) - 4x_0^2 \Rightarrow \\
3x_0^2 &= 8\ln(2) \Rightarrow \\
x_0 &= \sqrt{\frac{8\ln 2}{3}}
\end{aligned}$$

If the decision threshold is 1.36, then for $x < 1.36$ it is classified in the first class and for $x > 1.36$ it is classified in the second class.