Laboratory 3

In this laboratory we will implement Principal Component Analysis and Linear Discriminant Analysis

Principal Component Analysis

Principal Component Analysis (PCA) allows reducing the dimensionality of a dataset by projecting the data over the principal components. These can be computed from the eigenvectors of the data covariance matrix corresponding to the largest eigenvalues. The first step to implement PCA therefore requires computing the data covariance matrix

$$\boldsymbol{C} = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}_i - \boldsymbol{\mu})(\boldsymbol{x}_i - \boldsymbol{\mu})^T$$

where $\boldsymbol{\mu}$ is the dataset mean

$$\boldsymbol{\mu} = \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{x}_i$$

A straightforward implementation would be based on `for` loops, e.g., assuming `D` is the data matrix (with samples being columns of `D`):

```
mu = 0
for i in range(D.shape[1]):
    mu = mu + D[:, i:i+1]

mu = mu / float(D.shape[1])

C = 0
for i in range(D.shape[1]):
    C = C + numpy.dot(D[:, i:i+1] - mu, (D[:, i:i+1] - mu).T)

C = C / float(D.shape[1])
```

As we have seen in the previous Laboratory, for loops in Python are slow, and we can obtain better performance if we can express the computations as matrix operations, and / or use library functions, which are usually implemented in C and thus much faster.

We have also seen that `numpy` allows computing the mean of an array through the method `.mean` . The method allows specifying an axis — for 2-D arrays, `axis = 0` allows computing the mean of the rows, whereas `axis = 1` allows computing the mean of the columns of the matrix. We can thus compute the dataset mean as

```
mu = D.mean(1)
```

Remember that, in this case, `mu` is a 1-D array, rather than a column vector as in the previous example.

To *center the data*, i.e. to remove the mean from all points, we exploit `numpy` broadcasting: if we reshape `mu` to be a column vector, we can then remove it from all columns of `D` simply with

```
DC = D - mu.reshape((mu.size, 1))
```

Suggestion: we will often have to reshape 1-D vectors as column or row vectors. Write a function `vcol` and a function `vrow` that implements the reshaping

If $\boldsymbol{D}_C$ is the matrix of centered data, the covariance matrix can be represented as

$$\boldsymbol{C} = \frac{1}{N}\boldsymbol{D}_C\boldsymbol{D}_C^T \tag{1}$$

which can be directly computed through matrix multiplication (@ operator or `numpy.dot` function)

The results should be

$$\boldsymbol{\mu} = \begin{bmatrix} 5.84333333 \\ 3.05733333 \\ 3.758 \\ 1.19933333 \end{bmatrix} \qquad \boldsymbol{C} = \begin{bmatrix} 0.68112222 & -0.04215111 & 1.26582 & 0.51282889 \\ -0.04215111 & 0.18871289 & -0.32745867 & -0.12082844 \\ 1.26582 & -0.32745867 & 3.09550267 & 1.286972 \\ 0.51282889 & -0.12082844 & 1.286972 & 0.57713289 \end{bmatrix}$$

Once we have computed the data covariance matrix, we need to compute its eigenvectors and eigenvalues. For a generic square matrix we can use the library function `numpy.linalg.eig` . Since the covariance matrix is symmetric, we can use the more specific function `numpy.linalg.eigh`

```
s, U = numpy.linalg.eigh(C)
```

which returns the eigenvalues, sorted from smallest to largest, and the corresponding eigenvectors (columns of `U`). Note that `eig` does not, instead, sort the eigenvalues and eigenvectors. Check the documentation of the function — you can check on the `numpy` web page or, if you're using `ipython`, simply executing `numpy.linalg.eigh?` (question mark included). The m leading eigenvectors can be retrieved from `U` (here we also reverse the order of the columns of `U` so that the leading eigenvectors are in the first `m` columns):

```
P = U[:, ::-1][:, 0:m]
```

Since the covariance matrix is semi-definite positive, we can also get the sorted eigenvectors from the Singular Value Decomposition

$$\boldsymbol{C} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$$

In fact, in this case $\boldsymbol{V}^T = \boldsymbol{U}^T$, thus $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ is also an eigen–decomposition of $\boldsymbol{C}$. The SVD can be computed by

```
U, s, Vh  = numpy.linalg.svd(C)
```

In this case, the singular values (which are equal to the eigenvalues) are sorted in descending order, and the columns of `U` are the corresponding eigenvectors

```
P = U[:, 0:m]
```

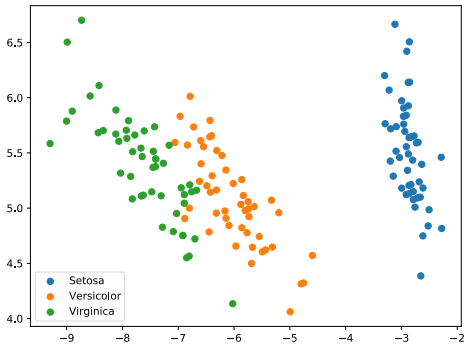
Finally, we can apply the projection to a single point `x` or to a matrix of samples `D` as

```
y = numpy.dot(P.T, x)
```

or

```
DP = numpy.dot(P.T, D)
```

Write the function that computes the PCA projection matrix for any dimensionality `m`. The file `Solution/IRIS_PCA_matrix_m4.npy` contains the matrix of sorted eigenvectors, which you can use to check your solution (the file can be loaded using `numpy.load`). Keep in mind that the solution is not unique: for example, we can change the sign of any eigenvector and obtain an equivalent projection matrix (the directions are the same, but their orientation is different). Applying PCA to map the IRIS dataset to a 2-D space you should obtain the following result (x-axis is the first principal direction, y-axis is the second principal direction)



NOTE: **your image may be flipped over one or both axes**, due to the fact that eigenvectors are defined up to their sign. Flipping the sign of an eigenvector would result in a flipped image, but the solutions are equivalent.

Linear Discriminant Analysis

To compute the LDA transformation matrix $\boldsymbol{W}$ we need to compute the between and within class covariance matrices

$$\boldsymbol{S}_B = \frac{1}{N}\sum_{c=1}^{K} n_c \left(\boldsymbol{\mu}_c - \boldsymbol{\mu}\right)\left(\boldsymbol{\mu}_c - \boldsymbol{\mu}\right)^T$$

$$\boldsymbol{S}_W = \frac{1}{N}\sum_{c=1}^{K}\sum_{i=1}^{n_c} \left(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c\right)\left(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c\right)^T$$

In the IRIS dataset classes are labeled as `0,1,2`. We can select the samples of the i–th class as

```
D[:, L==i]
```

The within class covariance matrix can be computed as a weighted sum the covariance matrices of each class:

$$\boldsymbol{S}_w = \frac{1}{N}\sum_{c=1}^{K} n_c \boldsymbol{S}_{W,c}$$

where

$$\boldsymbol{S}_{W,c} = \frac{1}{n_c}\sum_{i=1}^{n_c} \left(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c\right)\left(\boldsymbol{x}_{c,i} - \boldsymbol{\mu}_c\right)^T \ .$$

The covariance of each class $\boldsymbol{S}_{W,c}$ can be computed as we did for PCA:

- Compute the mean for the class samples
- Remove the mean from the class data
- Compute $\boldsymbol{S}_{W,c}$ as in (1) using the centered data matrix of the class samples, and n_c as the number of samples

The between class covariance matrix can be computed directly from its definition. Write a function that computes the two matrices.

The result should be

$$\boldsymbol{S}_B = \begin{bmatrix} 0.42141422 & -0.13301778 & 1.101656 & 0.47519556 \\ -0.13301778 & 0.07563289 & -0.38159733 & -0.15288444 \\ 1.101656 & -0.38159733 & 2.91401867 & 1.24516 \\ 0.47519556 & -0.15288444 & 1.24516 & 0.53608889 \end{bmatrix}$$

$$\boldsymbol{S}_W = \begin{bmatrix} 0.259708 & 0.09086667 & 0.164164 & 0.03763333 \\ 0.09086667 & 0.11308 & 0.05413867 & 0.032056 \\ 0.164164 & 0.05413867 & 0.181484 & 0.041812 \\ 0.03763333 & 0.032056 & 0.041812 & 0.041044 \end{bmatrix}$$

Once we have computed the matrices, we need to find the solution to the LDA objective. We analyze two methods to compute the LDA directions

Generalized eigenvalue problem

The LDA directions (columns of $\boldsymbol{W}$) can be computed solving the generalized eigenvalue problem

$$\boldsymbol{S}_B \boldsymbol{w} = \lambda \boldsymbol{S}_W \boldsymbol{w}$$

Assuming (as in our case) that $\boldsymbol{S}_W$ is positive definite (all eigenvalues are greater than 0), we can use the `scipy.linalg.eigh` function (you need to `import scipy.linalg`), which solves the generalized eigenvalue problem for hermitian (including real symmetric) matrices (note that we pass both $\boldsymbol{S}_B$ and $\boldsymbol{S}_W$ to the function; also, the `numpy.linalg.eigh` function cannot be used as it does not solve the generalized problem):

```
s, U = scipy.linalg.eigh(SB, SW)
W = U[:, ::-1][:, 0:m]
```

Notice that the columns of `W` are not necessarily orthogonal. If we want, we can find a basis `U` for the subspace spanned by `W` using the singular value decomposition of `W`:

```
UW, _, _ = numpy.linalg.svd(W)
U = UW[:, 0:m]
```

Solving the eigenvalue problem by joint diagonalization of $\boldsymbol{S}_B$ and $\boldsymbol{S}_W$

We have seen that the LDA solution can be implemented as a first transformation that whitens the within class covariance matrix, followed by a projection on the leading eigenvectors of the transformed between class covariance.
The first step consists in estimating matrix $\boldsymbol{P}_1$ such that the within class covariance of the transformed points $\boldsymbol{P}_1\boldsymbol{x}$ is the identity. Applying the transformation $\boldsymbol{P}_1$ the covariance matrix becomes $\boldsymbol{P}_1\boldsymbol{S}_W\boldsymbol{P}_1^T$, thus we can compute a matrix $\boldsymbol{P}_1$ that whitens the within-class covariance as

$$\boldsymbol{P}_1 = \boldsymbol{U}\boldsymbol{\Sigma}^{-\frac{1}{2}}\boldsymbol{U}^T$$

where $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^T$ is the SVD of $\boldsymbol{S}_W$. The SVD can be computed as

```
U, s, _ = numpy.linalg.svd(SW)
```

`s` is a 1-dimensional array containing the diagonal of $\boldsymbol{\Sigma}$. The diagonal of $\boldsymbol{\Sigma}^{-\frac{1}{2}}$ can be computed as `1.0/s**0.5`. We can exploit broadcasting to compute $\boldsymbol{U}\boldsymbol{\Sigma}^{-\frac{1}{2}}$ as `U * vrow(1.0/(s**0.5))`. $\boldsymbol{P}_1$ can then be computed as

```
P1 = numpy.dot(U * vrow(1.0/(s**0.5)), U.T)
```

We can also use `numpy.diag` to build a diagonal matrix from a one-dimensional array:

```
P1 = numpy.dot( numpy.dot(U, numpy.diag(1.0/(s**0.5))), U.T )
```

The transformed between class covariance $\boldsymbol{S}_{BT}$ can be computed as

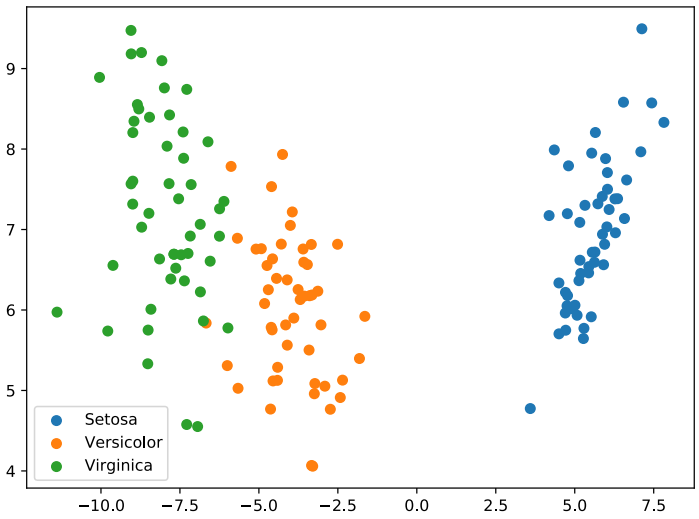
$$\boldsymbol{S}_{BT} = \boldsymbol{P}_1\boldsymbol{S}_B\boldsymbol{P}_1^T$$

We finally need to compute the matrix $\boldsymbol{P}_2$ of eigenvectors of $\boldsymbol{S}_{BT}$ corresponding to its m highest eigenvalues. The transformation from the original space to the LDA subspace can then be expressed as $\boldsymbol{y} = \boldsymbol{P}_2^T\boldsymbol{P}_1\boldsymbol{x}$. Thus, the LDA matrix $\boldsymbol{W}$ is given by $\boldsymbol{W} = \boldsymbol{P}_1^T\boldsymbol{P}_2$, and the LDA transformation is $\boldsymbol{y} = \boldsymbol{W}^T\boldsymbol{x}$. Again, we can observe that the solution is not orthogonal.

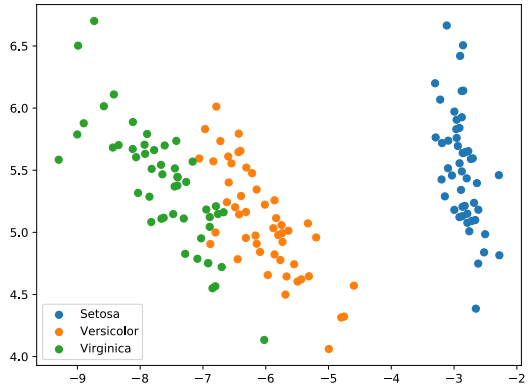
You can compare your solution with the one contained in `Solution/IRIS_LDA_matrix_m2.npy`
Remember that we can compute at most 2 discriminant directions, since we have 3 classes. Since different solutions can be found also for LDA, you should check that your solution describes the same subspace as the provided solution. Let $\boldsymbol{U}$ and $\boldsymbol{V}$ be two solutions, with m columns. A quick way to check their columns span the same subspace is to check that the matrix $[\boldsymbol{U}\boldsymbol{V}]$ has at most m non-zero singular values. You can check the singular values as

```
numpy.linalg.svd(numpy.hstack([U, V]))[1]
```

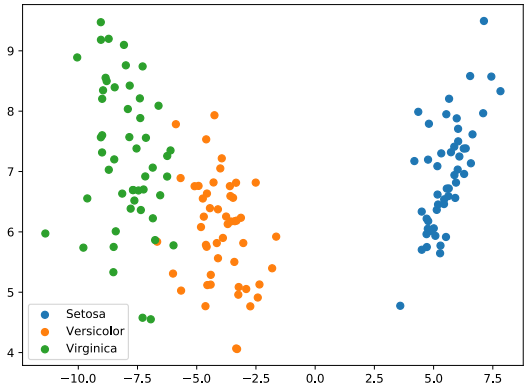
Applying LDA on the IRIS dataset we obtain the following features:



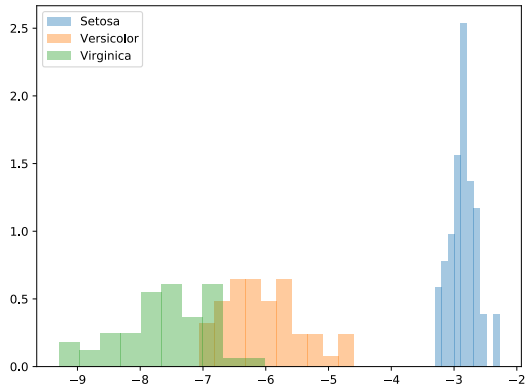
We can compare with PCA:



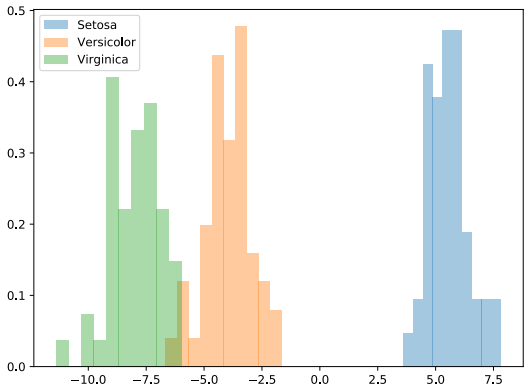
(a) PCA (1st and 2nd direction)



(b) LDA (1st and 2nd direction)



(c) PCA (1st direction)



(d) LDA (1st direction)

We can observe that the first LDA direction (x-axis in the scatter plots) results in lower overlapping of the green and the orange classes, which can thus be better separated along this direction.

PCA and LDA for classification

We now turn our attention to applications of PCA and LDA, focusing on a binary classification task. We consider the problem of separating the Iris versicolor and Iris virginica classes. We can build the new dataset from the Iris data matrix and label array as

```
def load_iris():
    return sklearn.datasets.load_iris()['data'].T, sklearn.datasets.load_iris
        ()['target']

DIris, LIris = load_iris()
D = DIris[:, LIris != 0]
L = LIris[LIris != 0]
```

`D` and `L` will contain the samples and labels of classes 1 and 2 (versicolor and virginica).
In the previous sections we have already employed PCA and LDA to analyze the full dataset. In this section, we want to apply PCA and LDA to the binary classification problem. In particular, we employ LDA to identify the (single, since we have only 2 classes) discriminant direction of the two-class version of the dataset (which, in general, may differ from the discriminant directions of the 3-class problem). We will also employ PCA as a pre-processing for LDA.

We want to be able to assess the quality of our model, and we also want to choose optimal values for hyperparameters such as the PCA subspace dimensionality and the LDA classification threshold, therefore we need to be able to evaluate the goodness of the predictions of our LDA-based classifier. We cannot directly evaluate the performance of the model on the data that was used to learn the model parameters. Indeed, the result may be over-optimistic, since the real data that we will want to classify will be data that has not been seen during training. To reduce the bias, we therefore need to simulate an application scenario, i.e. we want to evaluate the model on data that was not used during training.

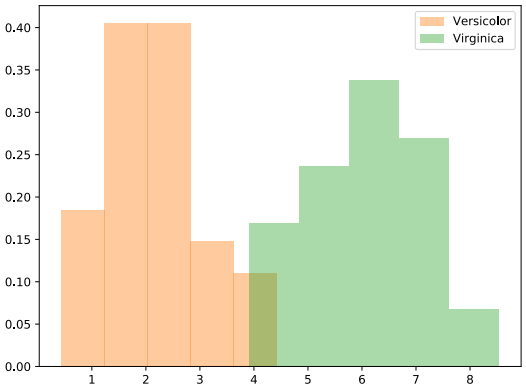
We split the training set in two parts: model training data (that will be used to estimate the model parameters) and validation data (that will be used for evaluation and model selection). Both datasets should contain data from all classes. For the moment we employ a simple random split that assigns $^2/_3$ of the samples to the model training partition and $^1/_3$ of the samples to the validation partition.

You can try implementing yourself the split. The results in the following sections are based on the following code:

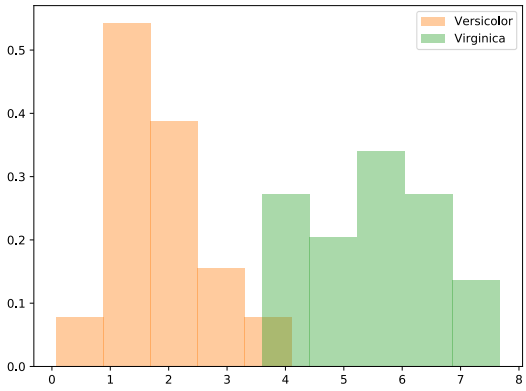
```
def split_db_2to1(D, L, seed=0):

    nTrain = int(D.shape[1]*2.0/3.0)
    numpy.random.seed(seed)
    idx = numpy.random.permutation(D.shape[1])
    idxTrain = idx[0:nTrain]
    idxTest = idx[nTrain:]

    DTR = D[:, idxTrain]
    DVAL = D[:, idxTest]
    LTR = L[idxTrain]
    LVAL = L[idxTest]

    return (DTR, LTR), (DVAL, LVAL)

# DTR and LTR are model training data and labels
# DVAL and LVAL are validation data and labels
(DTR, LTR), (DVAL, LVAL) = split_db_2to1(D, L)
```

where `D` and `L` are the data matrix and label array of the 2-class problem.
To build our LDA classifier we need to train an LDA model over the *model training* part of the dataset `(DTR, LTR)`, i.e., without using the validation data. Train the LDA model and then project the *validation* samples. We can compare the histogram of the projected validation samples to the histogram of the projected model training samples (we use 5 bins only, since each dataset contains only few samples).

We observe that the distribution look similar, our model is not showing overfitting issues (although the low number of samples makes our analysis less reliable).



(a) Model training set (`DTR, LTR`)



(b) Validation set (`DVAL, LVAL`)

NOTE: since the LDA direction is defined up to its sign, in order to perform classification we need to fix the orientation and choose a threshold, so that we can properly label samples above and below the selected threshold. In the following we choose the orientation that results in the Virginica class being on the *right* of the plot, i.e., in the mean of the projected samples of the Virigincal class being larger than the mean of the projected samples of the Versicolor class. If your LDA model returned the opposite orientation, simply change the sign of the LDA matrix, otherwise the code below will **not** provide the correct results. To choose the threshold, for the moment we simply employ the mean of the projected class means, *computed on the model training data* (better ways to choose a good threshold will be analyzed in the next laboratories)

```
threshold = (DTR_lda[0, LTR==1].mean() + DTR_lda[0, LTR==2].mean()) / 2.0 #
    Projected samples have only 1 dimension
```

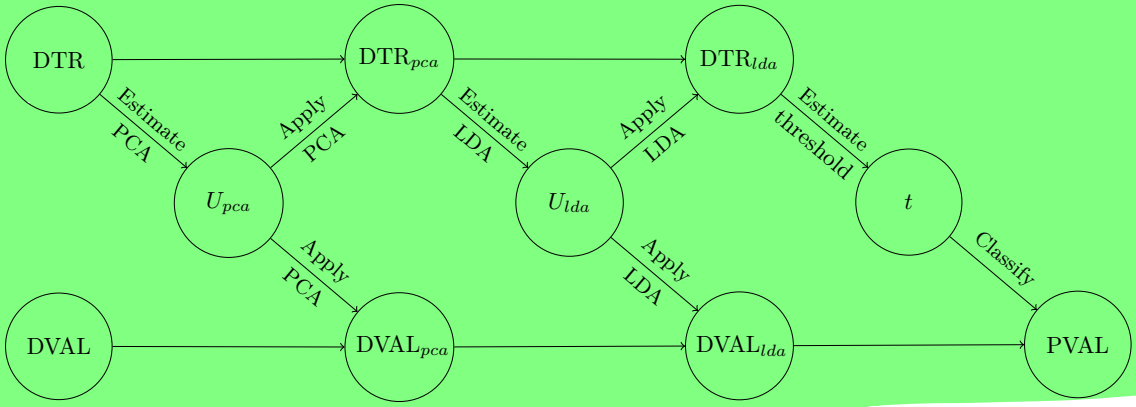
where `DTR_lda` and `DVAL_lda` are the projected samples. To predict the class label, build an array of predicted labels, compare the projected samples to the threshold and appropriately fill the array (elements on the right of the threshold should be labeled as class 2, elements on the left as class 1)

```
PVAL = numpy.zeros(shape=LVAL.shape, dtype=numpy.int32)
PVAL[DVAL_lda[0] >= threshold] = 2
PVAL[DVAL_lda[0] < threshold] = 1
```

Count the number of times `LVAL` and `PVAL` disagree to get the error rate (you should get 2 errors out of 34 validation samples).

Try again using the first PCA direction (principal component) in place of the LDA direction (pay attention again to choose the orientation so that the mean of the projected Virginica samples is larger that the mean of the projected Versicolor samples). What do you observe? Is the first component as effective for classification?

Try applying PCA *before* LDA to reduce the feature size to either 2 or 3 dimensions (reducing to 1 dimension makes LDA irrelevant, whereas reducing to 4 dimensions does not change the LDA subspace). Remember that you must estimate both the PCA projection matrix and the LDA matrix *only* on the model training part of the dataset (`DTR, LTR`). The full pipeline is depicted below:



Project

Apply PCA and LDA to the project data. Start analyzing the effects of PCA on the features. Plot the histogram of the projected features for the 6 PCA directions, starting from the principal (largest variance). What do you observe? What are the effects on the class distributions? Can you spot the different clusters inside each class?

Apply LDA (1 dimensional, since we have just two classes), and compute the histogram of the projected LDA samples. What do you observe? Do the classes overlap? Compared to the histograms of the 6 features you computed in Laboratory 2, is LDA finding a good direction with little class overlap?

Try applying LDA as classifier. Divide the dataset in model training and validation sets (you can reuse the previous function to split the dataset). Apply LDA, select the orientation that results in the projected mean of class True (label 1) being larger than the projected mean of class False (label 0), and select the threshold as in the previous sections, i.e., as the average of the projected class means. Compute the predictions on the validation data, and the corresponding error rate.

Now try changing the value of the threshold. What do you observe? Can you find values that improve the classification accuracy?

Finally, try pre-processing the features with PCA. Apply PCA (estimated on the model training data only), and then classify the validation data with LDA. Analyze the performance as a function of the number of PCA dimensions m. What do you observe? Can you find values of m that improve the accuracy on the validation set? Is PCA beneficial for the task when combined with the LDA classifier?