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 means of 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 implemented through `numpy.dot`

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 more specific the 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

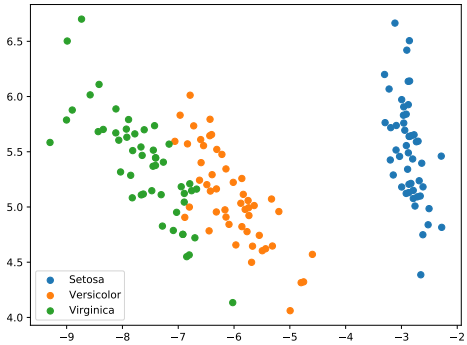
```
y = numpy.dot(P.T, x)
```

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):



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]
```

To compute the within class covariance matrix, we have to sum the covariance matrices of each class. The covariance of each class can be computed as we did for PCA:

- Compute the mean for the class samples
- Remove the mean from the class data
- Compute C as in (1)

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 becomes $\boldsymbol{P}_1\boldsymbol{S}_W\boldsymbol{P}_1^T$, thus we can compute $\boldsymbol{P}_1$ 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 is

```
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

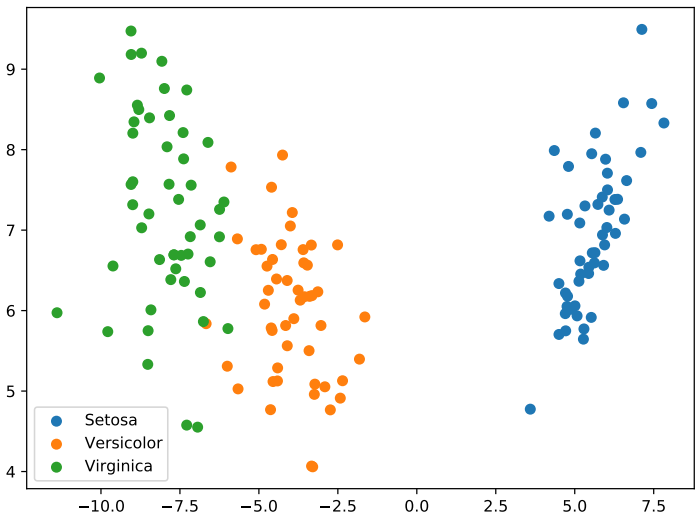
$$\boldsymbol{S}_{BT} = \boldsymbol{P}_1\boldsymbol{S}_B\boldsymbol{P}_1^T$$

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.

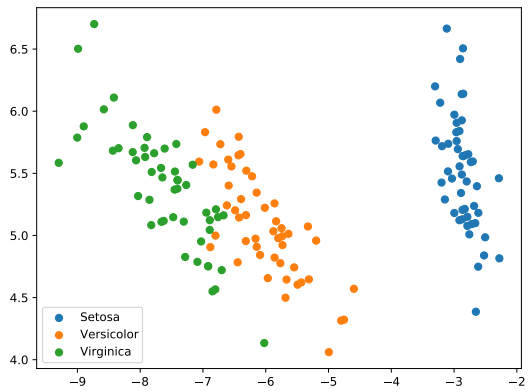
You can compare your solution with the one contained in `Solution/IRIS_LDA_matrix_m2.npy` Remember that we can compute at most 2 discriminant directions, since we have 3 classes. Since different solutions can be found also for LDA, you should check that your solution describes the same subspace as the 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]
```

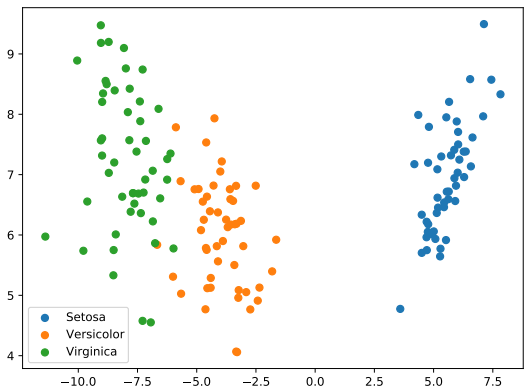
Applying LDA on the IRIS dataset we obtain the following features:



We can compare with PCA:



(a) PCA



(b) LDA

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