TP2 : Prétraitement et visualisation de données

A. Normalisation de données

1- Créez la matrice X :

Importez les librairies numpy (calcul scientifique) et preprocessing (prétraitement de données)

In [2]:
```
import numpy as np
from sklearn import preprocessing
X = np.array([[1, -1, 2], [2, 0, 0], [0, 1, -1]])
```

2- Visualisez X et calculez la moyenne et la variance de X.

In [3]:
```
print('> X :\n',X)
print('> Mean : ',np.mean(X))
print('> Std : ', np.std(X))
```

```
> X :
 [[ 1 -1  2]
 [ 2  0  0]
 [ 0  1 -1]]
> Mean :  0.4444444444444444
> Std :  1.0657403385139377
```

3- Utilisez la fonction scale pour normaliser la matrice X. Que constatez vous ?

In [4]:
```
X_norm = preprocessing.scale(X)
print(X_norm)
```

```
[[ 0.         -1.22474487  1.33630621]
 [ 1.22474487  0.         -0.26726124]
 [-1.22474487  1.22474487 -1.06904497]]
```

Que constatez vous ?

Réponse :

la variance permet de combiner toutes les valeurs à l'intérieur d'un ensemble de données afin d'obtenir la mesure de dispersion. dans la matrice X les données sont plus disperssés par rapport à la matrice X aprés la normalisation.

4- Calculer la moyenne et la variance de la matrice X normalisé. Expliquez le résultat obtenu.

In [6]:
```
print('> La moyenne  : ',np.mean(X_norm))
print('> La variance : ', np.std(X_norm))
```

```
> La moyenne  :  4.9343245538895844e-17
> La variance :  1.0
```

Expliquez le résultat obtenu:

Réponse :

La matrice est mise à l'échelle (centrée-réduite) car la moyenne est proche de zéro et la variance (ou l'écart-type vaut 1)

B. Normalisation MinMax

1- Créez la matrice de données X2 suivante :

1, -1, 2,

2, 0, 0,

0, 1, -1

In [8]:
```
X2 = X
```

2- Visualisez la matrice et calculez la moyenne sur les variables.

In [10]:
```
print('> X :\n',X2)
print('> La moyenne pour chaque variable : ', X2.mean(0))
```

```
> X :
 [[ 1 -1  2]
 [ 2  0  0]
 [ 0  1 -1]]
> La moyenne pour chaque variable :  [1.         0.         0.33333333]
```

3- Normalisez les données dans l'intervalle [0 1]. Visualisez les données normalisées et calculez la moyenne sur les variables. Que constatez-vous ?

In [11]:
```
scaleX2 = preprocessing.MinMaxScaler((0, 1))
X2_norm = scaleX2.fit_transform(X2)
```

Visualisez les données normalisées et calculez la moyenne sur les variables.

In [13]:
```
print('> X2_norm :\n', X2_norm)
print('>La moyenne sur les variables : ', X2_norm.mean(0))
```

```
> X2_norm :
 [[0.5        0.         1.        ]
 [1.         0.5        0.33333333]
 [0.         1.         0.        ]]
>La moyenne sur les variables :  [0.5        0.5        0.44444444]
```

Que constatez-vous?

MinMaxScaler: Les valeurs de la matrice après la normalisation se trouvent entre la valeur moyenne minimum et la valeur moyenne maximum des variables en ne dépassant pas l'intervalle [0,1]

C. Visualisation de données

1- Chargez les données Iris

In [14]:
```
from sklearn import datasets
iris = datasets.load_iris()
data = iris.data
targets = iris.target
```

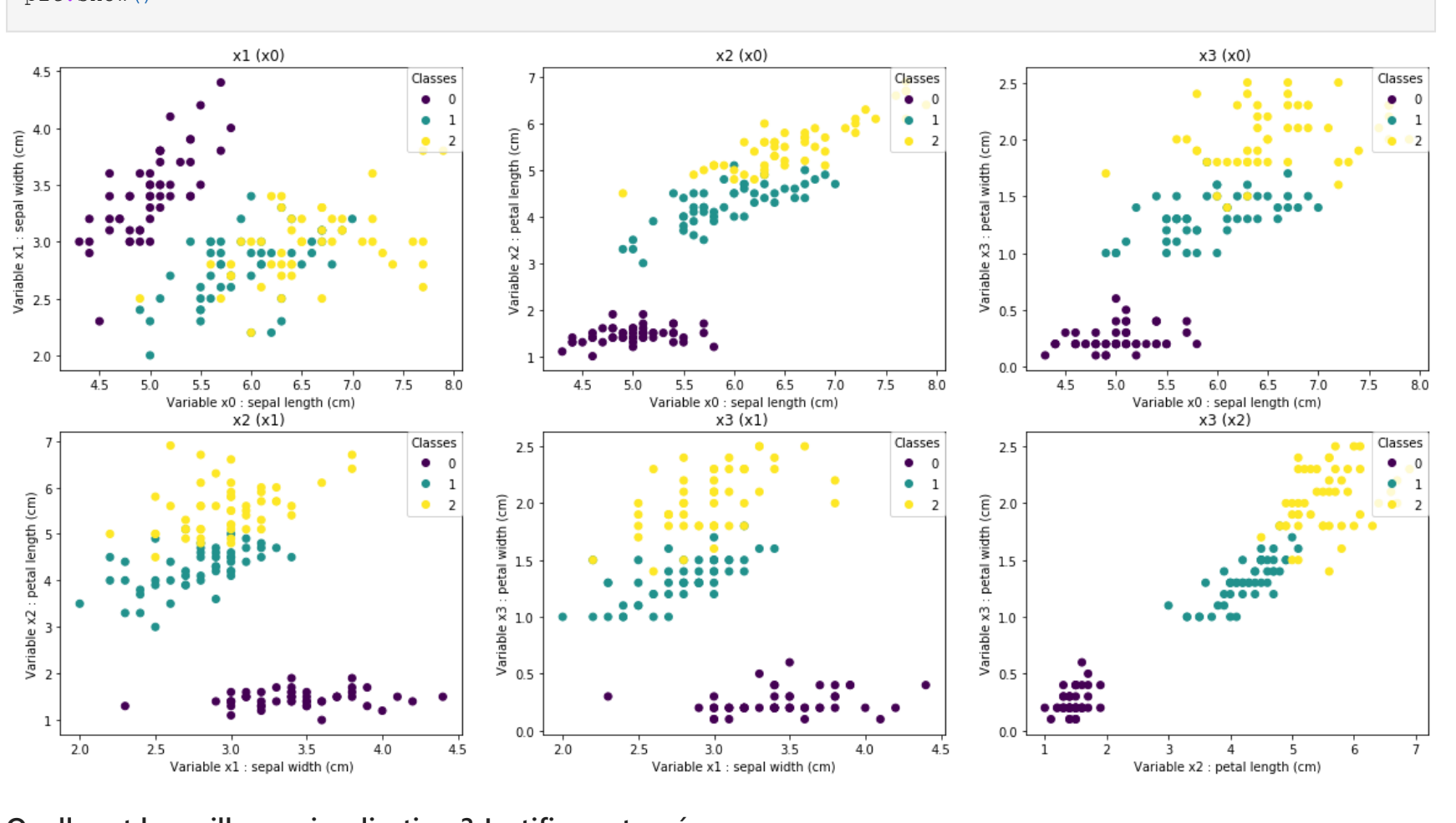
2- Visualisez le nuage de points en 2D avec des couleurs correspondant aux classes en utilisant toutes les combinaisons de variables.

In [15]:
```
data.shape
```

Out[15]:
```
(150, 4)
```

In [22]:
```
import matplotlib.pyplot as plt

plt.subplots(2,3,figsize=(20,10))
cpp = 1
for i in range(4):
    for j in range(i+1, 4):
        plt.subplot(2,3,cpp)
        cpp+=1
        scatter = plt.scatter(data[:,i], data[:,j], c=targets)
        plt.title('x'+str(j)+' (x'+str(i)+') ')
        plt.xlabel('Variable x'+str(i)+' : '+iris.feature_names[i])
        plt.ylabel('Variable x'+str(j)+' : '+iris.feature_names[j])
        plt.legend(*scatter.legend_elements(), loc="upper right", title="Classes", borderaxespad=0.)
plt.show()
```



Quelle est la meilleure visualisation ? Justifiez votre réponse.

La meilleure visualisation est : le plot x3 en fonction de x2, car les classes son bien séparée

D. Réduction de dimensions et visualisation de données

L'Analyse en Composantes Principales (ACP) a comme objectif d'identifier la combinaison d'attributs (composants principaux, ou les directions dans l'espace de caractéristique), qui représentent le plus la variance dans les données.
L'Analyse discriminante linéaire (ADL) tente d'identifier les attributs qui représentent le plus la variance entre les classes. En particulier, l'ADL, contrairement à l'APC, est un procédé supervisé en utilisant les étiquettes de classe connus.

1- Les méthodes PCA et LDA peuvent etre importé à partir des package suivants :

In [23]:
```
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
```

2- Analysez le manuel d'aide pour ces deux fonctions (pca et lda) et appliquez les sur la base Iris. Il faudra utiliser pca.fit(Iris).transform(Iris) et sauvegardez les résultats dans IrisPCA pour la PCA et IrisLDA pour la LDA.

In [24]:
```
help(PCA)
```

```
Help on class PCA in module sklearn.decomposition._pca:

class PCA(sklearn.decomposition._base._BasePCA)
 |  PCA(n_components=None, *, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power='auto', ra
ndom_state=None)
 |
 |  Principal component analysis (PCA).
 |
 |  Linear dimensionality reduction using Singular Value Decomposition of the
 |  data to project it to a lower dimensional space. The input data is centered
 |  but not scaled for each feature before applying the SVD.
 |
 |  It uses the LAPACK implementation of the full SVD or a randomized truncated
 |  SVD by the method of Halko et al. 2009, depending on the shape of the input
 |  data and the number of components to extract.
 |
 |  It can also use the scipy.sparse.linalg ARPACK implementation of the
 |  truncated SVD.
 |
 |  Notice that this class does not support sparse input. See
 |  :class:`TruncatedSVD` for an alternative with sparse data.
 |
 |  Read more in the :ref:`User Guide <PCA>`.
 |
 |  Parameters
 |  ----------
 |  n_components : int, float, None or str
 |      Number of components to keep.
 |      if n_components is not set all components are kept::
 |
 |          n_components == min(n_samples, n_features)
 |
 |      If ``n_components == 'mle'`` and ``svd_solver == 'full'``, Minka's
 |      MLE is used to guess the dimension. Use of ``n_components == 'mle'``
 |      will interpret ``svd_solver == 'auto'`` as ``svd_solver == 'full'``.
 |
 |      If ``0 < n_components < 1`` and ``svd_solver == 'full'``, select the
 |      number of components such that the amount of variance that needs to be
 |      explained is greater than the percentage specified by n_components.
 |
 |      If ``svd_solver == 'arpack'``, the number of components must be
 |      strictly less than the minimum of n_features and n_samples.
 |
 |      Hence, the None case results in::
 |
 |          n_components == min(n_samples, n_features) - 1
 |
 |  copy : bool, default=True
 |      If False, data passed to fit are overwritten and running
 |      fit(X).transform(X) will not yield the expected results,
 |      use fit_transform(X) instead.
 |
 |  whiten : bool, optional (default False)
 |      When True (False by default) the `components_` vectors are multiplied
 |      by the square root of n_samples and then divided by the singular values
 |      to ensure uncorrelated outputs with unit component-wise variances.
 |
 |      Whitening will remove some information from the transformed signal
 |      (the relative variance scales of the components) but can sometime
 |      improve the predictive accuracy of the downstream estimators by
 |      making their data respect some hard-wired assumptions.
 |
 |  svd_solver : str {'auto', 'full', 'arpack', 'randomized'}
 |      If auto :
 |          The solver is selected by a default policy based on `X.shape` and
 |          `n_components`: if the input data is larger than 500x500 and the
 |          number of components to extract is lower than 80% of the smallest
 |          dimension of the data, then the more efficient 'randomized'
 |          method is enabled. Otherwise the exact full SVD is computed and
 |          optionally truncated afterwards.
 |      If full :
 |          run exact full SVD calling the standard LAPACK solver via
 |          `scipy.linalg.svd` and select the components by postprocessing
 |      If arpack :
 |          run SVD truncated to n_components calling ARPACK solver via
 |          `scipy.sparse.linalg.svds`. It requires strictly
 |          0 < n_components < min(X.shape)
 |      If randomized :
 |          run randomized SVD by the method of Halko et al.
 |
 |      .. versionadded:: 0.18.0
 |
 |  tol : float >= 0, optional (default .0)
 |      Tolerance for singular values computed by svd_solver == 'arpack'.
 |
 |      .. versionadded:: 0.18.0
 |
 |  iterated_power : int >= 0, or 'auto', (default 'auto')
 |      Number of iterations for the power method computed by
 |      svd_solver == 'randomized'.
 |
 |      .. versionadded:: 0.18.0
 |
 |  random_state : int, RandomState instance, default=None
 |      Used when ``svd_solver`` == 'arpack' or 'randomized'. Pass an int
 |      for reproducible results across multiple function calls.
 |      See :term:`Glossary <random_state>`.
 |
 |      .. versionadded:: 0.18.0
 |
 |  Attributes
 |  ----------
 |  components_ : array, shape (n_components, n_features)
 |      Principal axes in feature space, representing the directions of
 |      maximum variance in the data. The components are sorted by
 |      ``explained_variance_``.
 |
 |  explained_variance_ : array, shape (n_components,)
 |      The amount of variance explained by each of the selected components.
 |
 |      Equal to n_components largest eigenvalues
 |      of the covariance matrix of X.
 |
 |      .. versionadded:: 0.18
 |
 |  explained_variance_ratio_ : array, shape (n_components,)
 |      Percentage of variance explained by each of the selected components.
 |
 |      If ``n_components`` is not set then all components are stored and the
 |      sum of the ratios is equal to 1.0.
 |
 |  singular_values_ : array, shape (n_components,)
 |      The singular values corresponding to each of the selected components.
 |      The singular values are equal to the 2-norms of the ``n_components``
 |      variables in the lower-dimensional space.
 |
 |      .. versionadded:: 0.19
 |
 |  mean_ : array, shape (n_features,)
 |      Per-feature empirical mean, estimated from the training set.
 |
 |      Equal to `X.mean(axis=0)`.
 |
 |  n_components_ : int
 |      The estimated number of components. When n_components is set
 |      to 'mle' or a number between 0 and 1 (with svd_solver == 'full') this
 |      number is estimated from input data. Otherwise it equals the parameter
 |      n_components, or the lesser value of n_features and n_samples
 |      if n_components is None.
 |
 |  n_features_ : int
 |      Number of features in the training data.
 |
 |  n_samples_ : int
 |      Number of samples in the training data.
 |
 |  noise_variance_ : float
 |      The estimated noise covariance following the Probabilistic PCA model
 |      from Tipping and Bishop 1999. See "Pattern Recognition and
 |      Machine Learning" by C. Bishop, 12.2.1 p. 574 or
 |      http://www.miketipping.com/papers/met-mppca.pdf. It is required to
 |      compute the estimated data covariance and score samples.
 |
 |      Equal to the average of (min(n_features, n_samples) - n_components)
 |      smallest eigenvalues of the covariance matrix of X.
 |
 |  See Also
 |  --------
 |  KernelPCA : Kernel Principal Component Analysis.
 |  SparsePCA : Sparse Principal Component Analysis.
 |  TruncatedSVD : Dimensionality reduction using truncated SVD.
 |  IncrementalPCA : Incremental Principal Component Analysis.
 |
 |  References
 |  ----------
 |  For n_components == 'mle', this class uses the method of *Minka, T. P.
 |  "Automatic choice of dimensionality for PCA". In NIPS, pp. 598-604*
 |
 |  Implements the probabilistic PCA model from:
 |  Tipping, M. E., and Bishop, C. M. (1999). "Probabilistic principal
 |  component analysis". Journal of the Royal Statistical Society:
 |  Series B (Statistical Methodology), 61(3), 611-622.
 |  via the score and score_samples methods.
 |  See http://www.miketipping.com/papers/met-mppca.pdf
 |
 |  For svd_solver == 'arpack', refer to `scipy.sparse.linalg.svds`.
 |
 |  For svd_solver == 'randomized', see:
 |  *Halko, N., Martinsson, P. G., and Tropp, J. A. (2011).
 |  "Finding structure with randomness: Probabilistic algorithms for
 |  constructing approximate matrix decompositions".
 |  SIAM review, 53(2), 217-288.* and also
 |  *Martinsson, P. G., Rokhlin, V., and Tygert, M. (2011).
 |  "A randomized algorithm for the decomposition of matrices".
 |  Applied and Computational Harmonic Analysis, 30(1), 47-68.*
 |
 |  Examples
 |  --------
 |  >>> import numpy as np
 |  >>> from sklearn.decomposition import PCA
 |  >>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
 |  >>> pca = PCA(n_components=2)
 |  >>> pca.fit(X)
 |  PCA(n_components=2)
 |  >>> print(pca.explained_variance_ratio_)
 |  [0.9924... 0.0075...]
 |  >>> print(pca.singular_values_)
 |  [6.30061... 0.54980...]
 |
 |  >>> pca = PCA(n_components=2, svd_solver='full')
 |  >>> pca.fit(X)
 |  PCA(n_components=2, svd_solver='full')
 |  >>> print(pca.explained_variance_ratio_)
 |  [0.9924... 0.00755...]
 |  >>> print(pca.singular_values_)
 |  [6.30061... 0.54980...]
 |
 |  >>> pca = PCA(n_components=1, svd_solver='arpack')
 |  >>> pca.fit(X)
 |  PCA(n_components=1, svd_solver='arpack')
 |  >>> print(pca.explained_variance_ratio_)
 |  [0.99244...]
 |  >>> print(pca.singular_values_)
 |  [6.30061...]
 |
 |  Method resolution order:
 |      PCA
 |      sklearn.decomposition._base._BasePCA
 |      sklearn.base.TransformerMixin
 |      sklearn.base.BaseEstimator
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, n_components=None, *, copy=True, whiten=False, svd_solver='auto', tol=0.0, iterated_power
='auto', random_state=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y=None)
 |      Fit the model with X.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_features)
 |          Training data, where n_samples is the number of samples
 |          and n_features is the number of features.
 |
 |      y : None
 |          Ignored variable.
 |
 |      Returns
 |      -------
 |      self : object
 |          Returns the instance itself.
 |
 |  fit_transform(self, X, y=None)
 |      Fit the model with X and apply the dimensionality reduction on X.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_features)
 |          Training data, where n_samples is the number of samples
 |          and n_features is the number of features.
 |
 |      y : None
 |          Ignored variable.
 |
 |      Returns
 |      -------
 |      X_new : array-like, shape (n_samples, n_components)
 |          Transformed values.
 |
 |      Notes
 |      -----
 |      This method returns a Fortran-ordered array. To convert it to a
 |      C-ordered array, use 'np.ascontiguousarray'.
 |
 |  score(self, X, y=None)
 |      Return the average log-likelihood of all samples.
 |
 |      See. "Pattern Recognition and Machine Learning"
 |      by C. Bishop, 12.2.1 p. 574
 |      or http://www.miketipping.com/papers/met-mppca.pdf
 |
 |      Parameters
 |      ----------
 |      X : array, shape(n_samples, n_features)
 |          The data.
 |
 |      y : None
 |          Ignored variable.
 |
 |      Returns
 |      -------
 |      ll : float
 |          Average log-likelihood of the samples under the current model.
 |
 |  score_samples(self, X)
 |      Return the log-likelihood of each sample.
 |
 |      See. "Pattern Recognition and Machine Learning"
 |      by C. Bishop, 12.2.1 p. 574
 |      or http://www.miketipping.com/papers/met-mppca.pdf
 |
 |      Parameters
 |      ----------
 |      X : array, shape(n_samples, n_features)
 |          The data.
 |
 |      Returns
 |      -------
 |      ll : array, shape (n_samples,)
 |          Log-likelihood of each sample under the current model.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.decomposition._base._BasePCA:
 |
 |  get_covariance(self)
 |      Compute data covariance with the generative model.
 |
 |      ``cov = components_.T * S**2 * components_ + sigma2 * eye(n_features)``
 |      where S**2 contains the explained variances, and sigma2 contains the
 |      noise variances.
 |
 |      Returns
 |      -------
 |      cov : array, shape=(n_features, n_features)
 |          Estimated covariance of data.
 |
 |  get_precision(self)
 |      Compute data precision matrix with the generative model.
 |
 |      Equals the inverse of the covariance but computed with
 |      the matrix inversion lemma for efficiency.
 |
 |      Returns
 |      -------
 |      precision : array, shape=(n_features, n_features)
 |          Estimated precision of data.
 |
 |  inverse_transform(self, X)
 |      Transform data back to its original space.
 |
 |      In other words, return an input X_original whose transform would be X.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_components)
 |          New data, where n_samples is the number of samples
 |          and n_components is the number of components.
 |
 |      Returns
 |      -------
 |      X_original array-like, shape (n_samples, n_features)
 |
 |      Notes
 |      -----
 |      If whitening is enabled, inverse_transform will compute the
 |      exact inverse operation, which includes reversing whitening.
 |
 |  transform(self, X)
 |      Apply dimensionality reduction to X.
 |
 |      X is projected on the first principal components previously extracted
 |      from a training set.
 |
 |      Parameters
 |      ----------
 |      X : array-like, shape (n_samples, n_features)
 |          New data, where n_samples is the number of samples
 |          and n_features is the number of features.
 |
 |      Returns
 |      -------
 |      X_new : array-like, shape (n_samples, n_components)
 |
 |      Examples
 |      --------
 |
 |      >>> import numpy as np
 |      >>> from sklearn.decomposition import IncrementalPCA
 |      >>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
 |      >>> ipca = IncrementalPCA(n_components=2, batch_size=3)
 |      >>> ipca.fit(X)
 |      IncrementalPCA(batch_size=3, n_components=2)
 |      >>> ipca.transform(X) # doctest: +SKIP
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.TransformerMixin:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
```

In [26]:
```
pca = PCA(n_components=2)
IrisPCA = pca.fit_transform(iris.data)
print(IrisPCA[:5])
```

```
[[-2.68412563  0.31939725]
 [-2.71414169 -0.17700123]
 [-2.88899057 -0.14494943]
 [-2.74534286 -0.31829898]
 [-2.72871654  0.32675451]]
```

In [27]:
```
help(LDA)
```

```
Help on class LinearDiscriminantAnalysis in module sklearn.discriminant_analysis:

class LinearDiscriminantAnalysis(sklearn.base.BaseEstimator, sklearn.linear_model._base.LinearClassifierMixi
n, sklearn.base.TransformerMixin)
 |  LinearDiscriminantAnalysis(*, solver='svd', shrinkage=None, priors=None, n_components=None, store_covari
ance=False, tol=0.0001)
 |
 |  Linear Discriminant Analysis
 |
 |  A classifier with a linear decision boundary, generated by fitting class
 |  conditional densities to the data and using Bayes' rule.
 |
 |  The model fits a Gaussian density to each class, assuming that all classes
 |  share the same covariance matrix.
 |
 |  The fitted model can also be used to reduce the dimensionality of the input
 |  by projecting it to the most discriminative directions, using the
 |  `transform` method.
 |
 |  .. versionadded:: 0.17
 |     *LinearDiscriminantAnalysis*.
 |
 |  Read more in the :ref:`User Guide <lda_qda>`.
 |
 |  Parameters
 |  ----------
 |  solver : {'svd', 'lsqr', 'eigen'}, default='svd'
 |      Solver to use, possible values:
 |        - 'svd': Singular value decomposition (default).
 |          Does not compute the covariance matrix, therefore this solver is
 |          recommended for data with a large number of features.
 |        - 'lsqr': Least squares solution, can be combined with shrinkage.
 |        - 'eigen': Eigenvalue decomposition, can be combined with shrinkage.
 |
 |  shrinkage : 'auto' or float, default=None
 |      Shrinkage parameter, possible values:
 |        - None: no shrinkage (default).
 |        - 'auto': automatic shrinkage using the Ledoit-Wolf lemma.
 |        - float between 0 and 1: fixed shrinkage parameter.
 |
 |      Note that shrinkage works only with 'lsqr' and 'eigen' solvers.
 |
 |  priors : array-like of shape (n_classes,), default=None
 |      The class prior probabilities. By default, the class proportions are
 |      inferred from the training data.
 |
 |  n_components : int, default=None
 |      Number of components (<= min(n_classes - 1, n_features)) for
 |      dimensionality reduction. If None, will be set to
 |      min(n_classes - 1, n_features). This parameter only affects the
 |      `transform` method.
 |
 |  store_covariance : bool, default=False
 |      If True, explicitely compute the weighted within-class covariance
 |      matrix when solver is 'svd'. The matrix is always computed
 |      and stored for the other solvers.
 |
 |      .. versionadded:: 0.17
 |
 |  tol : float, default=1.0e-4
 |      Absolute threshold for a singular value of X to be considered
 |      significant, used to estimate the rank of X. Dimensions whose
 |      singular values are non-significant are discarded. Only used if
 |      solver is 'svd'.
 |
 |      .. versionadded:: 0.17
 |
 |  Attributes
 |  ----------
 |  coef_ : ndarray of shape (n_features,) or (n_classes, n_features)
 |      Weight vector(s).
 |
 |  intercept_ : ndarray of shape (n_classes,)
 |      Intercept term.
 |
 |  covariance_ : array-like of shape (n_features, n_features)
 |      Weighted within-class covariance matrix. It corresponds to
 |      `sum_k prior_k * C_k` where `C_k` is the covariance matrix of the
 |      samples in class `k`. The `C_k` are estimated using the (potentially
 |      shrunk) biased estimator of covariance. If solver is 'svd', only
 |      exists when `store_covariance` is True.
 |
 |  explained_variance_ratio_ : ndarray of shape (n_components,)
 |      Percentage of variance explained by each of the selected components.
 |      If ``n_components`` is not set then all components are stored and the
 |      sum of explained variances is equal to 1.0. Only available when eigen
 |      or svd solver is used.
 |
 |  means_ : array-like of shape (n_classes, n_features)
 |      Class-wise means.
 |
 |  priors_ : array-like of shape (n_classes,)
 |      Class priors (sum to 1).
 |
 |  scalings_ : array-like of shape (rank, n_classes - 1)
 |      Scaling of the features in the space spanned by the class centroids.
 |      Only available for 'svd' and 'eigen' solvers.
 |
 |  xbar_ : array-like of shape (n_features,)
 |      Overall mean. Only present if solver is 'svd'.
 |
 |  classes_ : array-like of shape (n_classes,)
 |      Unique class labels.
 |
 |  See also
 |  --------
 |  sklearn.discriminant_analysis.QuadraticDiscriminantAnalysis: Quadratic
 |      Discriminant Analysis
 |
 |  Examples
 |  --------
 |  >>> import numpy as np
 |  >>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
 |  >>> X = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
 |  >>> y = np.array([1, 1, 1, 2, 2, 2])
 |  >>> clf = LinearDiscriminantAnalysis()
 |  >>> clf.fit(X, y)
 |  LinearDiscriminantAnalysis()
 |  >>> print(clf.predict([[-0.8, -1]]))
 |  [1]
 |
 |  Method resolution order:
 |      LinearDiscriminantAnalysis
 |      sklearn.base.BaseEstimator
 |      sklearn.linear_model._base.LinearClassifierMixin
 |      sklearn.base.ClassifierMixin
 |      sklearn.base.TransformerMixin
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *, solver='svd', shrinkage=None, priors=None, n_components=None, store_covariance=False,
tol=0.0001)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  decision_function(self, X)
 |      Apply decision function to an array of samples.
 |
 |      The decision function is equal (up to a constant factor) to the
 |      log-posterior of the model, i.e. `log p(y = k | x)`. In a binary
 |      classification setting this instead corresponds to the difference
 |      `log p(y = 1 | x) - log p(y = 0 | x)`. See :ref:`lda_qda_math`.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Array of samples (test vectors).
 |
 |      Returns
 |      -------
 |      C : ndarray of shape (n_samples,) or (n_samples, n_classes)
 |          Decision function values related to each class, per sample.
 |          In the two-class case, the shape is (n_samples,), giving the
 |          log likelihood ratio of the positive class.
 |
 |  fit(self, X, y)
 |      Fit LinearDiscriminantAnalysis model according to the given
 |         training data and parameters.
 |
 |         .. versionchanged:: 0.19
 |            *store_covariance* has been moved to main constructor.
 |
 |         .. versionchanged:: 0.19
 |            *tol* has been moved to main constructor.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Training data.
 |
 |      y : array-like of shape (n_samples,)
 |          Target values.
 |
 |  predict_log_proba(self, X)
 |      Estimate log probability.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input data.
 |
 |      Returns
 |      -------
 |      C : ndarray of shape (n_samples, n_classes)
 |          Estimated log probabilities.
 |
 |  predict_proba(self, X)
 |      Estimate probability.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input data.
 |
 |      Returns
 |      -------
 |      C : ndarray of shape (n_samples, n_classes)
 |          Estimated probabilities.
 |
 |  transform(self, X)
 |      Project data to maximize class separation.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Input data.
 |
 |      Returns
 |      -------
 |      X_new : ndarray of shape (n_samples, n_components)
 |          Transformed data.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : mapping of string to any
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as pipelines). The latter have parameters of the form
 |      ``<component>__<parameter>`` so that it's possible to update each
 |      component of a nested object.
 |
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |
 |      Returns
 |      -------
 |      self : object
 |          Estimator instance.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from sklearn.base.BaseEstimator:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.linear_model._base.LinearClassifierMixin:
 |
 |  predict(self, X)
 |      Predict class labels for samples in X.
 |
 |      Parameters
 |      ----------
 |      X : array_like or sparse matrix, shape (n_samples, n_features)
 |          Samples.
 |
 |      Returns
 |      -------
 |      C : array, shape [n_samples]
 |          Predicted class label per sample.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.ClassifierMixin:
 |
 |  score(self, X, y, sample_weight=None)
 |      Return the mean accuracy on the given test data and labels.
 |
 |      In multi-label classification, this is the subset accuracy
```

```
 |      which is a harsh metric since you require for each sample that
 |      each label set be correctly predicted.
 |      
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |      
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True labels for X.
 |      
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |      
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of self.predict(X) wrt. y.
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.TransformerMixin:
 |  
 |  fit_transform(self, X, y=None, **fit_params)
 |      Fit to data, then transform it.
 |      
 |      Fits transformer to X and y with optional parameters fit_params
 |      and returns a transformed version of X.
 |      
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix, dataframe} of shape                 (n_samples, n_features)
 |      
 |      y : ndarray of shape (n_samples,), default=None
 |          Target values.
 |      
 |      **fit_params : dict
 |          Additional fit parameters.
 |      
 |      Returns
 |      -------
 |      X_new : ndarray array of shape (n_samples, n_features_new)
 |          Transformed array.
```

In [28]:

```
clf = LDA(n_components=2)
IrisLDA = clf.fit(iris.data,iris.target).transform(iris.data)
print(IrisLDA[:5])
```

```
[[-8.06179978  0.30042062]
 [-7.12868772 -0.78666043]
 [-7.48982797 -0.26538449]
 [-6.81320057 -0.67063107]
 [-8.13230933  0.51446253]]
```

3- Visualisez les nuages de points avec les nouvelles axes obtenus : une image pour l'ACP et une autre pour l'ADL et utiliser la classe de Iris comme couleurs de points. Quelle différence constatez--vous entre les deux visualisations? Expliquer votre raisonnement.

In [29]:

```
plt.figure()
plt.subplots(2,3,figsize=(16,5))
plt.subplot(1,2,1)
scatter = plt.scatter(IrisPCA[:,0], IrisPCA[:,1], c=targets)
plt.title('PCA')
plt.xlabel('Variable x1')
plt.ylabel('Variable x2')
plt.legend(*scatter.legend_elements(), loc="upper right", title="Classes", borderaxespad=0.)

plt.subplot(1,2,2)
scatter = plt.scatter(IrisLDA[:,0], IrisLDA[:,1], c=targets)
plt.title('LDA')
plt.xlabel('Variable x1')
plt.ylabel('Variable x2')
plt.legend(*scatter.legend_elements(), loc="upper right", title="Classes", borderaxespad=0.)

plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```


Quelle différence constatez-vous entre les deux visualisations? Expliquer votre raisonnement.

Le LDA et le PCA sont des techniques de transformation linéaire: le LDA est un supervisé tandis que le PCA non supervisé - le PCA ignore les étiquettes de classe. Nous pouvons imaginer l'ACP comme une technique qui trouve les directions de la variance maximale. (L'ACP a tendance à entraîner de meilleurs résultats de classification dans une tâche de reconnaissance d'image si le nombre d'échantillons pour une classe donnée était relativement faible.) Contrairement à PCA, LDA tente de trouver un sous-espace de fonctionnalités qui maximise la séparabilité des classes. LDA fait des hypothèses sur les classes normalement distribuées et les covariances de classe égales.

Links

- E-mail : zakaria.abbou199434@gmail.com
- GitHub : github.com/ZakariaAABBOU
- Linkedin : linkedin.com/in/zakaria-aabbou/

In [ ]: