mlpy Documentation

Release 3.5.0

Davide Albanese

March 12, 2012

CONTENTS

Release 3.5

Date March 12, 2012

Homepage http://mlpy.sourceforge.net

Machine Learning PYthon (mlpy) is a high-performance Python library for predictive modeling.

This reference manual details functions, modules, and objects included in mlpy.

CHAPTER

ONE

INSTALL

1.1 Download

Download latest version for your OS from http://sourceforge.net/projects/mlpy/files/

1.2 Installing on *nix from source

On GNU/Linux, OSX and FreeBSD you need the following requirements:

- GCC
- Python >= 2.6 or 3.X
- NumPy >= 1.3.0 (with header files)
- SciPy >= 0.7.0
- GSL >= 1.11 (with header files)

From a terminal run:

```
$ python setup.py install
```

If you don't have root access, installing mlpy in a directory by specifying the `--prefix` argument. Then you need to set `PYTHONPATH`:

```
$ python setup.py install --prefix=/path/to/modules
$ export PYTHONPATH=$PYTHONPATH:/path/to/modules/lib/python{version}/site-packages
```

If the GSL header files or shared library are in non-standard locations on your system, use the `--include-dirs` and `--rpath` options to `build_ext`:

```
$ python setup.py build_ext --include-dirs=/path/to/header --rpath=/path/to/lib
$ python setup.py install
```

1.3 Installing on Windows Xp/Vista/7 from binary installer

Requirements:

- Python 2.6, 2.7, 3.1, 3.2 Windows installer (x86)
- NumPy >= 1.3.0 win32 installer

- SciPy >= 0.8.0 win32 installer

The GSL library is pre-compiled (by Visual Studio Express 2008) and included in mlpy.

Download and run the mlpy Windows installer (.exe).

CHAPTER TWO

INTRODUCTION

2.1 Conventions

- *x* is a matrix $n \times p$ which represents a set of n samples in $\Re^p$.
- *y* is a vector n which represents the target values (integers in classification problems, floats in regression problems).

CHAPTER

THREE

TUTORIAL

If you are new in Python and NumPy see: http://docs.python.org/tutorial/ http://www.scipy.org/Tentative_NumPy_Tutorial and http://matplotlib.sourceforge.net/.

A learning problem usually considers a set of p-dimensional samples (observations) of data and tries to predict properties of unknown data.

3.1 Tutorial 1 - Iris Dataset

The well known Iris dataset represents 3 kinds of Iris flowers with 150 observations and 4 attributes: sepal length, sepal width, petal length and petal width.

A dimensionality reduction and learning tasks can be performed by the mlpy library with just a few number of commands.

Download `Iris dataset`

Load the modules:

```
>>> import numpy as np
>>> import mlpy
>>> import matplotlib.pyplot as plt # required for plotting
```

Load the Iris dataset:

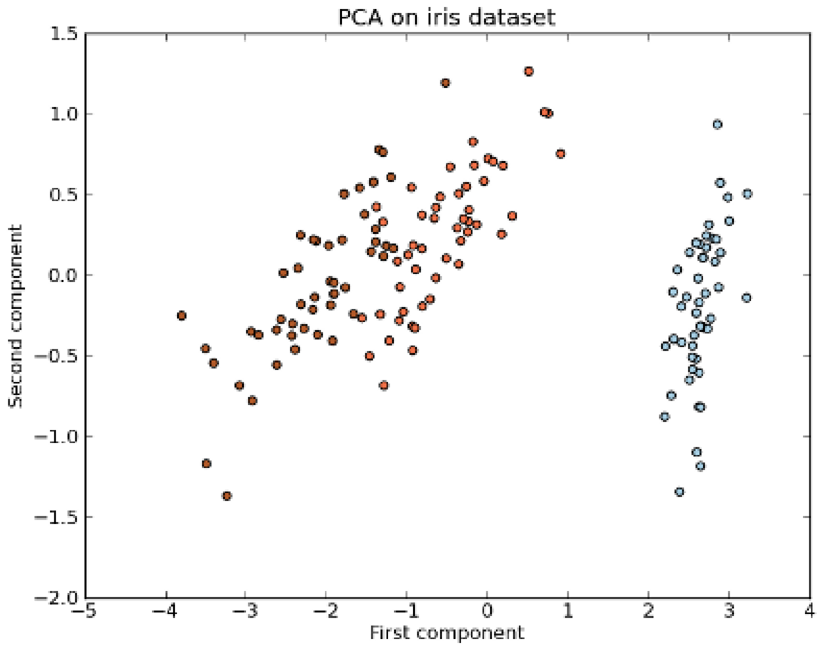
```
>>> iris = np.loadtxt('iris.csv', delimiter=',')
>>> x, y = iris[:, :4], iris[:, 4].astype(np.int) # x: (observations x attributes) matrix, y: classe
>>> x.shape
(150, 4)
>>> y.shape
(150, )
```

Dimensionality reduction by Principal Component Analysis (PCA)

```
>>> pca = mlpy.PCA() # new PCA instance
>>> pca.learn(x) # learn from data
>>> z = pca.transform(x, k=2) # embed x into the k=2 dimensional subspace
>>> z.shape
(150, 2)
```

Plot the principal components:

```
>>> plt.set_cmap(plt.cm.Paired)
>>> fig1 = plt.figure(1)
>>> title = plt.title("PCA on iris dataset")
>>> plot = plt.scatter(z[:, 0], z[:, 1], c=y)
>>> labx = plt.xlabel("First component")
>>> laby = plt.ylabel("Second component")
>>> plt.show()
```



Learning by Kernel Support Vector Machines (SVMs) on principal components:

```
>>> linear_svm = mlpy.LibSvm(kernel_type='linear') # new linear SVM instance
>>> linear_svm.learn(z, y) # learn from principal components
```

For plotting purposes, we build the grid where we will compute the predictions (*zgrid*):

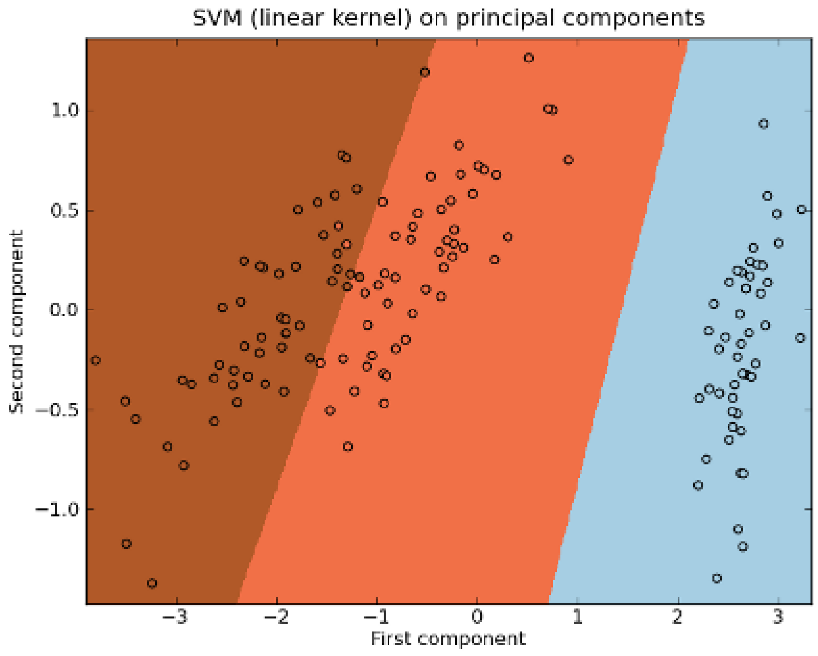
```
>>> xmin, xmax = z[:,0].min()-0.1, z[:,0].max()+0.1
>>> ymin, ymax = z[:,1].min()-0.1, z[:,1].max()+0.1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.01), np.arange(ymin, ymax, 0.01))
>>> zgrid = np.c_[xx.ravel(), yy.ravel()]
```

Now we perform the predictions on the grid. The *pred()* method returns the prediction for each point in zgrid:

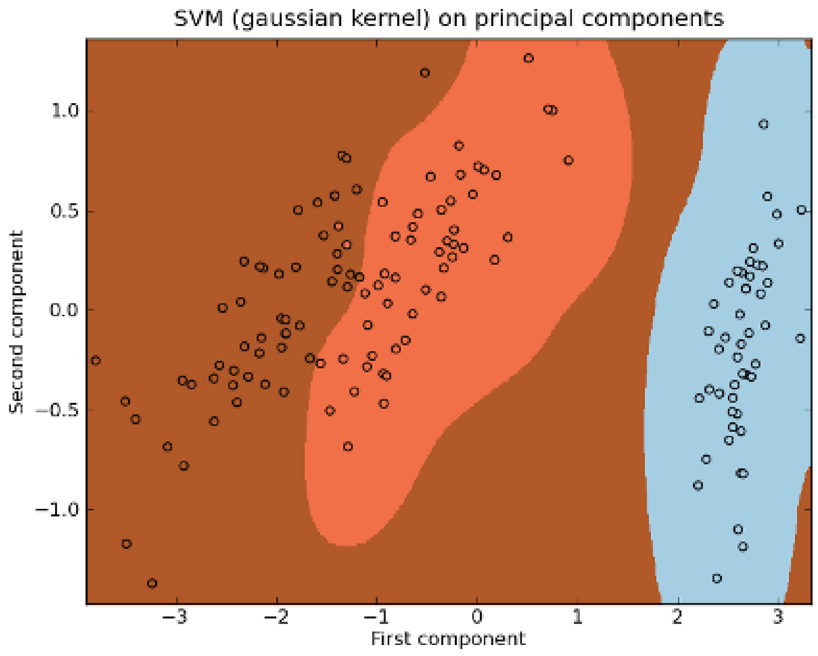
```
>>> yp = linear_svm.pred(zgrid)
```

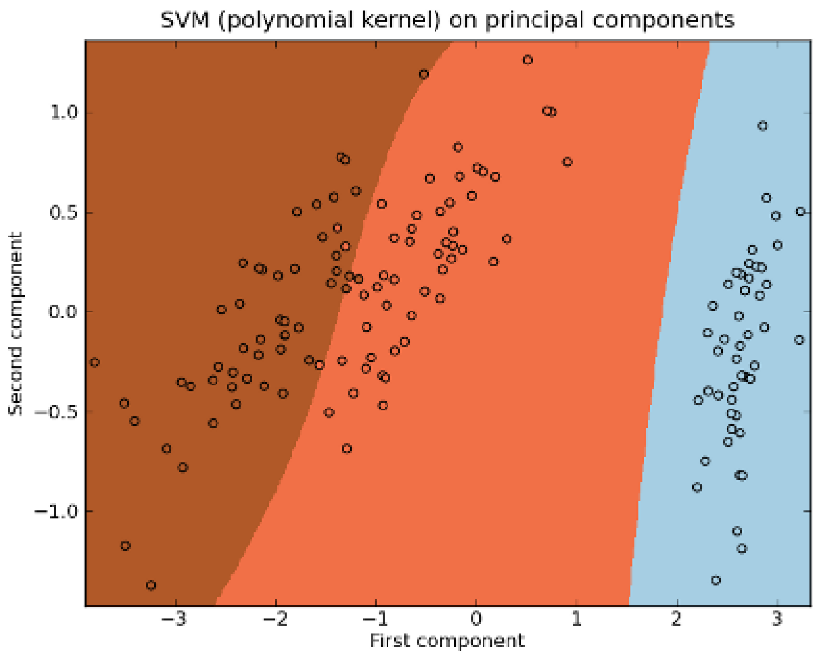
Plot the predictions:

```
>>> plt.set_cmap(plt.cm.Paired)
>>> fig2 = plt.figure(2)
>>> title = plt.title("SVM (linear kernel) on principal components")
>>> plot1 = plt.pcolormesh(xx, yy, yp.reshape(xx.shape))
>>> plot2 = plt.scatter(z[:, 0], z[:, 1], c=y)
>>> labx = plt.xlabel("First component")
>>> laby = plt.ylabel("Second component")
>>> limx = plt.xlim(xmin, xmax)
>>> limy = plt.ylim(ymin, ymax)
>>> plt.show()
```



We can try to use different kernels to obtain:




SVM (polynomial kernel) on principal components
1.0
0.5
0.0
−0.5
−1.0
Second component
−3
−2
−1
0
1
2
3
First component

CHAPTER

FOUR

LINEAR METHODS FOR REGRESSION

4.1 Ordinary Least Squares

`mlpy.`**`ols_base`**(*x*, *y*, *tol*)

Ordinary (Linear) Least Squares.

Solves the equation X beta = y by computing a vector beta that minimize ||y - X beta||^2 where ||.|| is the L^2 norm This function uses numpy.linalg.lstsq().

X must be centered by columns.

Parameters

x [2d array_like object] training data (samples x features)

y [1d array_like object integer (two classes)] target values

tol [float] Cut-off ratio for small singular values of x. Singular values are set to zero if they are smaller than *tol* times the largest singular value of x. If *tol* < 0, machine precision is used instead.

Returns

beta, rank = 1d numpy array, float beta, rank of matrix *x*.

class `mlpy.`**`OLS`**(*tol=-1*)

Ordinary (Linear) Least Squares Regression (OLS).

Initialization.

Parameters

tol [float] Cut-off ratio for small singular values of x. Singular values are set to zero if they are smaller than *tol* times the largest singular value of x. If *tol* < 0, machine precision is used instead.

`beta`()

Return b1, ..., bp.

`beta0`()

Return b0.

`learn`(*x*, *y*)

Learning method.

Parameters

x [2d array_like object] training data (samples x features)

y [1d array_like object integer (two classes)] target values

pred(*t*)
Compute the predicted response.

Parameters

t [1d or 2d array_like object] test data

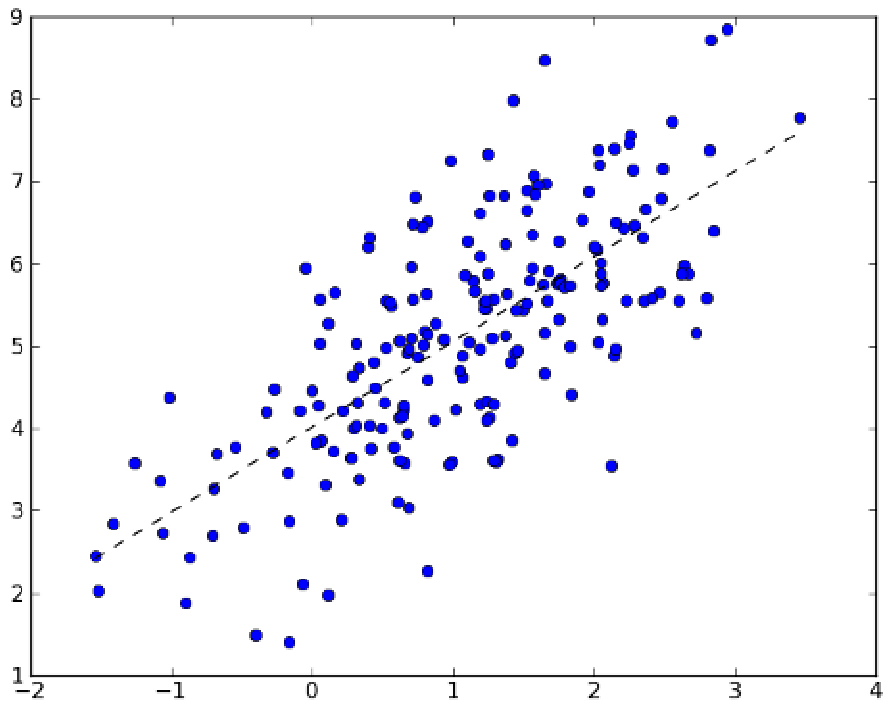
Returns

p [integer or 1d numpy darray] predicted response

rank()
Rank of matrix *x*.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean, cov, n = [1, 5], [[1,1],[1,2]], 200
>>> d = np.random.multivariate_normal(mean, cov, n)
>>> x, y = d[:, 0].reshape(-1, 1), d[:, 1]
>>> x.shape
(200, 1)
>>> ols = mlpy.OLS()
>>> ols.learn(x, y)
>>> xx = np.arange(np.min(x), np.max(x), 0.01).reshape(-1, 1)
>>> yy = ols.pred(xx)
>>> fig = plt.figure(1) # plot
>>> plot = plt.plot(x, y, 'o', xx, yy, '--k')
>>> plt.show()
```



4.2 Ridge Regression

See [Hoerl70]. Ridge regression is also known as regularized least squares. It avoids overfitting by controlling the size of the model vector β, measured by its ℓ^2-norm.

`mlpy.`**`ridge_base`**(*x*, *y*, *lmb*)

Solves the equation X beta = y by computing a vector beta that minimize ||y - X beta||^2 + ||lambda beta||^2 where ||.|| is the L^2 norm (X is a NxP matrix). When if N >= P the function solves the normal equation (primal solution), when N < P the function solves the dual solution.

X must be centered by columns.

Parameters

x [2d array_like object] training data (N x P)

y [1d array_like object (N)] target values

lmb [float (> 0.0)] lambda, regularization parameter

Returns

beta [1d numpy array] beta

class `mlpy.`**`Ridge`**(*lmb=1.0*)

Ridge Regression.

Solves the equation X beta = y by computing a vector beta that minimize ||y - X beta||^2 + ||lambda beta||^2 where ||.|| is the L^2 norm (X is a NxP matrix). When if N >= P the function solves the normal equation (primal solution), when N < P the function solves the dual solution.

Initialization.

Parameters

lmb [float (>= 0.0)] regularization parameter

beta ()

Return b1, ..., bp.

beta0 ()

Return b0.

learn (*x*, *y*)

Compute the regression coefficients.

Parameters:

x [2d array_like object] training data (N, P)

y [1d array_like object (N)] target values

pred (*t*)

Compute the predicted response.

Parameters

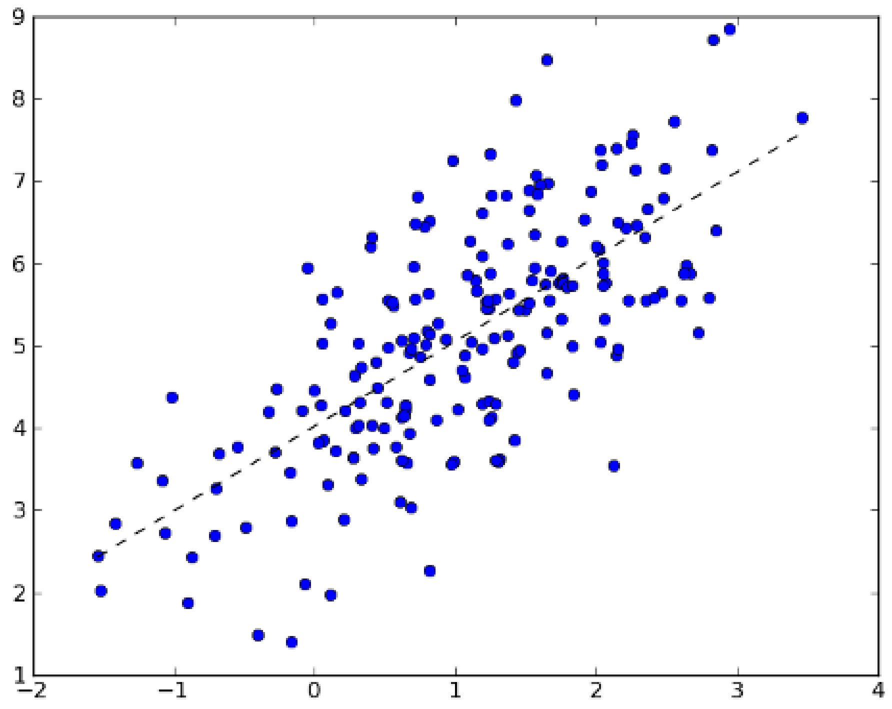
t [1d or 2d array_like object ([M,] P)] test data

Returns

p [integer or 1d numpy darray] predicted response

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean, cov, n = [1, 5], [[1,1],[1,2]], 200
>>> d = np.random.multivariate_normal(mean, cov, n)
>>> x, y = d[:, 0].reshape(-1, 1), d[:, 1]
>>> x.shape
(200, 1)
>>> ridge = mlpy.Ridge()
>>> ridge.learn(x, y)
>>> xx = np.arange(np.min(x), np.max(x), 0.01).reshape(-1, 1)
>>> yy = ridge.pred(xx)
>>> fig = plt.figure(1) # plot
>>> plot = plt.plot(x, y, 'o', xx, yy, '--k')
>>> plt.show()
```



4.3 Partial Least Squares

class `mlpy.PLS`(*iters*)

Multivariate primal Partial Least Squares (PLS) algorithm as described in [Taylor04].

Initialization.

Parameters

iters [int (>= 1)] number of iterations. iters should be <= min(N-1, P)

`beta`()

Returns the regression coefficients.

beta is a (P) vector in the univariate case and a (P, M) matrix in the multivariate case, where M is the number of target outputs.

`beta0`()

Returns offset(s).

beta is a float in the univariate case, and a (M) vector in the multivariate case, where M is the number of target outputs.

`learn`(*x*, *y*)

Compute the regression coefficients.

Parameters:

x [2d array_like object] training data (N, P)

y [1d array_like object (N [,M])] target values

pred(*t*)
Compute the predicted response(s).

Parameters

t [1d or 2d array_like object ([M,] P)] test data

Returns

p [integer or 1d numpy darray] predicted response(s)

4.4 Last Angle Regression (LARS)

mlpy.**lars_base**(*x*, *y*, *maxsteps=None*)
Least Angle Regression.

x should be centered and normalized by columns, and *y* should be centered.

Parameters

x [2d array_like object (N x P)] matrix of regressors

y [1d array_like object (N)] response

maxsteps [int (> 0) or None] maximum number of steps. If *maxsteps* is None, the maximum number of steps is min(N-1, P), where N is the number of variables and P is the number of features.

Returns

active, est, steps [1d numpy array, 2d numpy array, int] active features, all LARS estimates, number of steps performed

class mlpy.**LARS**(*maxsteps=None*)
Least Angle Regression.

Initialization.

Parameters

maxsteps [int (> 0) or None] maximum number of steps.

active()
Returns the active features.

beta()
Return b_1, ..., b_p.

beta0()
Return b_0.

est()
Returns all LARS estimates.

learn(*x*, *y*)
Compute the regression coefficients.

Parameters

x [2d array_like object (N x P)] matrix of regressors

y [1d array_like object (N)] response

pred (*t*)
Compute the predicted response.

Parameters

t [1d or 2d array_like object ([M,] P)] test data

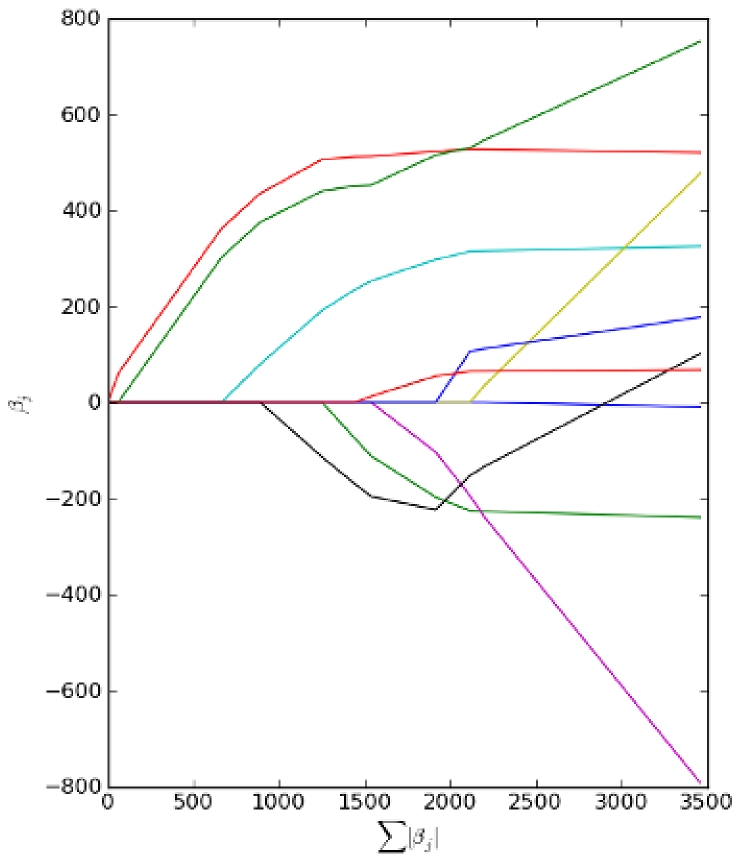
Returns

p [float or 1d numpy array] predicted response

steps ()
Return the number of steps performed.

This example replicates the Figure 3 in [Efron04]. The diabetes data can be downloaded from http://www.stanford.edu/~hastie/Papers/LARS/diabetes.data

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> diabetes = np.loadtxt("diabetes.data", skiprows=1)
>>> x = diabetes[:, :-1]
>>> y = diabetes[:, -1]
>>> x -= np.mean(x, axis=0) # center x
>>> x /= np.sqrt(np.sum((x)**2, axis=0)) # normalize x
>>> y -= np.mean(y) # center y
>>> lars = mlpy.LARS()
>>> lars.learn(x, y)
>>> lars.steps() # number of steps performed
10
>>> lars.beta()
array([ -10.0098663 , -239.81564367,  519.84592005,  324.3846455 ,
       -792.17563855,  476.73902101,  101.04326794,  177.06323767,
        751.27369956,   67.62669218])
>>> lars.beta0()
4.7406304540474682e-14
>>> est = lars.est() # returns all LARS estimates
>>> beta_sum = np.sum(np.abs(est), axis=1)
>>> fig = plt.figure(1)
>>> plot1 = plt.plot(beta_sum, est)
>>> xl = plt.xlabel(r'$\sum{|\beta_j|}$')
>>> yl = plt.ylabel(r'$\beta_j$')
>>> plt.show()
```



4.5 Elastic Net

Documentation and implementation is taken from http://web.mit.edu/lrosasco/www/contents/code/ENcode.html

Computes the coefficient vector which solves the elastic-net regularization problem

$$\min\{\|X\beta - Y\|^2 + \lambda(\|\beta\|_2^2 + \epsilon\|\beta\|_1\}$$

Elastic Net Regularization is an algorithm for learning and variable selection. It is based on a regularized least square procedure with a penalty which is the sum of an L1 penalty (like Lasso) and an L2 penalty (like ridge regression). The first term enforces the sparsity of the solution, whereas the second term ensures democracy among groups of correlated variables. The second term has also a smoothing effect that stabilizes the obtained solution.

mlpy.**elasticnet_base**(*x*, *y*, *lmb*, *eps*, *supp=True*, *tol=0.01*)

Elastic Net Regularization via Iterative Soft Thresholding.

x should be centered and normalized by columns, and *y* should be centered.

Computes the coefficient vector which solves the elastic-net regularization problem min {|| X beta - Y ||^2 + lambda(|beta|^2_2 + eps |beta|_1}. The solution beta is computed via iterative soft-thresholding, with damping factor 1/(1+eps*lambda), thresholding factor eps*lambda, null initialization vector and step 1 / (eig_max(XX^T)*1.1).

Parameters

x [2d array_like object (N x P)] matrix of regressors

y [1d array_like object (N)] response

lmb [float] regularization parameter controlling overfitting. *lmb* can be tuned via cross validation.

eps [float] correlation parameter preserving correlation among variables against sparsity. The solutions obtained for different values of the correlation parameter have the same prediction properties but different feature representation.

supp [bool] if True, the algorithm stops when the support of beta reached convergence. If False, the algorithm stops when the coefficients reached convergence, that is when the beta_{l}(i) - beta_{l+1}(i) > tol * beta_{l}(i) for all i.

tol [double] tolerance for convergence

Returns

beta, iters [1d numpy array, int] beta, number of iterations performed

class `mlpy.`**`ElasticNet`**(*lmb*, *eps*, *supp=True*, *tol=0.01*)

Elastic Net Regularization via Iterative Soft Thresholding.

Computes the coefficient vector which solves the elastic-net regularization problem min {|| X beta - Y ||^2 + lambda(|beta|^2_2 + eps |beta|_1}. The solution beta is computed via iterative soft-thresholding, with damping factor 1/(1+eps*lambda), thresholding factor eps*lambda, null initialization vector and step 1 / (eig_max(XX^T)*1.1).

Initialization.

Parameters

lmb [float] regularization parameter controlling overfitting. *lmb* can be tuned via cross validation.

eps [float] correlation parameter preserving correlation among variables against sparsity. The solutions obtained for different values of the correlation parameter have the same prediction properties but different feature representation.

supp [bool] if True, the algorithm stops when the support of beta reached convergence. If False, the algorithm stops when the coefficients reached convergence, that is when the beta_{l}(i) - beta_{l+1}(i) > tol * beta_{l}(i) for all i.

tol [double] tolerance for convergence

`beta`()

Return b_1, ..., b_p.

`beta0`()

Return b_0.

`iters`()

Return the number of iterations performed.

`learn`(*x*, *y*)

Compute the regression coefficients.

Parameters

x [2d array_like object (N x P)] matrix of regressors

y [1d array_like object (N)] response

pred(*t*)

Compute the predicted response.

Parameters

t [1d or 2d array_like object ([M,] P)] test data

Returns

p [float or 1d numpy array] predicted response

CHAPTER

FIVE

LINEAR METHODS FOR CLASSIFICATION

5.1 Linear Discriminant Analysis Classifier (LDAC)

See [Hastie09], page 106.

class `mlpy.`**`LDAC`**

Linear Discriminant Analysis Classifier.

Initialization.

`bias``()`

Returns the bias. For multiclass classification this method returns a 1d numpy array where b[i] contains the coefficients of label i. For binary classification an float (b_1 - b_0) is returned.

`labels``()`

Outputs the name of labels.

`learn``(`*x*, *y*`)`

Learning method.

Parameters

x [2d array_like object] training data (N, P)

y [1d array_like object integer] target values (N)

`pred``(`*t*`)`

Does classification on test vector(s) *t*.

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

p [integer or 1d numpy array] predicted class(es)

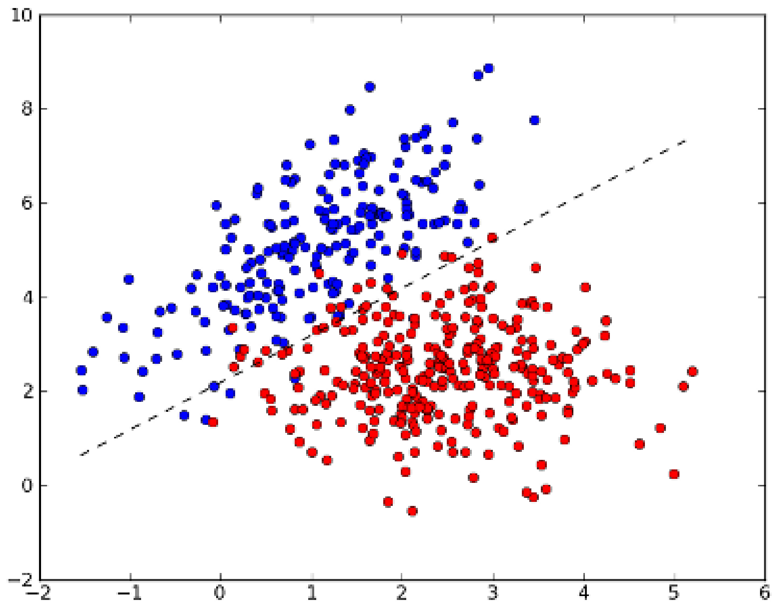
`w``()`

Returns the coefficients. For multiclass classification this method returns a 2d numpy array where w[i] contains the coefficients of label i. For binary classification an 1d numpy array (w_1 - w_0) is returned.

5.1.1 Examples

Binary classification:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class -1
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = -np.ones(n2, dtype=np.int)
>>> x = np.concatenate((x1, x2), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2))
>>> ldac = mlpy.LDAC()
>>> ldac.learn(x, y)
>>> w = ldac.w()
>>> w
array([ 2.5948979  -2.58553746])
>>> b = ldac.bias()
>>> b
5.63727441841
>>> xx = np.arange(np.min(x[:,0]), np.max(x[:,0]), 0.01)
>>> yy = - (w[0] * xx + b) / w[1] # separator line
>>> fig = plt.figure(1) # plot
>>> plot1 = plt.plot(x1[:, 0], x1[:, 1], 'ob', x2[:, 0], x2[:, 1], 'or')
>>> plot2 = plt.plot(xx, yy, '--k')
>>> plt.show()
```

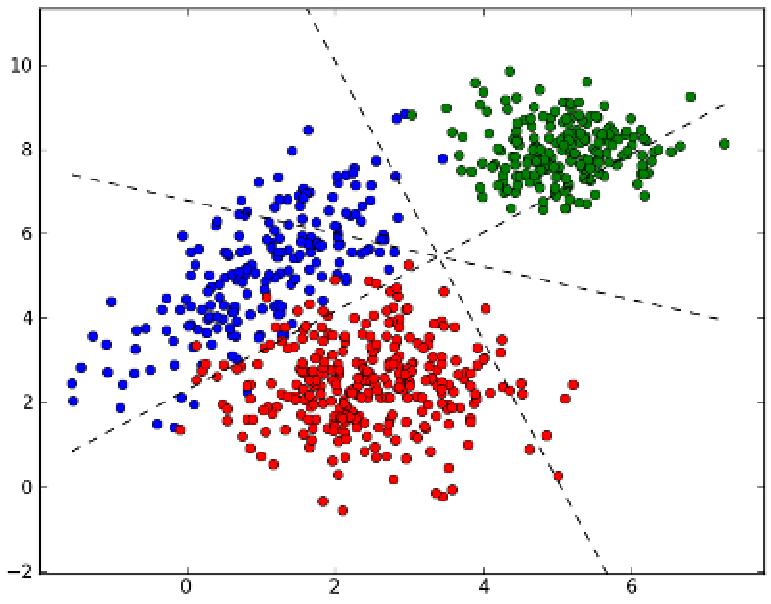


```
>>> test = [[0, 2], [4, 2]] # test points
>>> ldac.pred(test)
array([-1, -1])
```

```
>>> ldac.labels()
array([-1,  1])
```

Multiclass classification:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 25], [[1,1],[1,2]], 200  # 200 samples of class 0
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.zeros(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 22.5], [[1,0],[0,1]], 300 # 300 samples of class 1
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = np.ones(n2, dtype=np.int)
>>> mean3, cov3, n3 = [5, 28], [[0.5,0],[0,0.5]], 200 # 200 samples of class 2
>>> x3 = np.random.multivariate_normal(mean3, cov3, n3)
>>> y3 = 2 * np.ones(n3, dtype=np.int)
>>> x = np.concatenate((x1, x2, x3), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2, y3))
>>> ldac = mlpy.LDAC()
>>> ldac.learn(x, y)
>>> w = ldac.w()
>>> w # w[i]: coefficients label ldac.labels()[i]
array([[-0.30949939  4.53041257]
       [ 2.52002288  1.50501818]
       [ 4.2499381   5.90569921]])
>>> b = ldac.bias()
>>> b # b[i]: bias for label ldac.labels()[i]
array([-12.65129158  -5.7628039  -35.63605709])
>>> xx = np.arange(np.min(x[:,0]), np.max(x[:,0]), 0.01)
>>> yy1 = (xx* (w[1][0]-w[0][0]) + b[1] - b[0]) / (w[0][1]-w[1][1])
>>> yy2 = (xx* (w[2][0]-w[0][0]) + b[2] - b[0]) / (w[0][1]-w[2][1])
>>> yy3 = (xx* (w[2][0]-w[1][0]) + b[2] - b[1]) / (w[1][1]-w[2][1])
>>> fig = plt.figure(1) # plot
>>> plot1 = plt.plot(x1[:, 0], x1[:, 1], 'ob', x2[:, 0], x2[:, 1], 'or', x3[:, 0], x3[:, 1], 'og')
>>> plot2 = plt.plot(xx, yy1, '--k')
>>> plot3 = plt.plot(xx, yy2, '--k')
>>> plot4 = plt.plot(xx, yy3, '--k')
>>> plt.show()
```



```
>>> test = [[6,7], [4, 2]] # test points
>>> ldac.pred(test)
array([2, 1])
>>> ldac.labels()
array([0, 1, 2])
```

5.2 Basic Perceptron

class `mlpy.`**`Perceptron`**(*alpha=0.10000000000000001*, *thr=0.0*, *maxiters=1000*)

Perceptron binary classifier.

The algorithm stops when the iteration error is less or equal than *thr*, or a predetermined number of iterations (*maxiters*) have been completed.

Parameters

alpha [float, in range (0.0, 1]] learning rate

thr [float, in range [0.0, 1.0]] iteration error (e.g. thr=0.13 for error=13%)

maxiters [integer (>0)] maximum number of iterations

`bias`()

Returns the bias.

`err`()

Returns the iteration error

`iters`()

Returns the number of iterations

`labels`()

Outputs the name of labels.

learn (*x*, *y*)
Learning method.

Parameters

x [2d array_like object] training data (N, P)

y [1d array_like object integer (only two classes)] target values (N)

pred (*t*)
Prediction method.

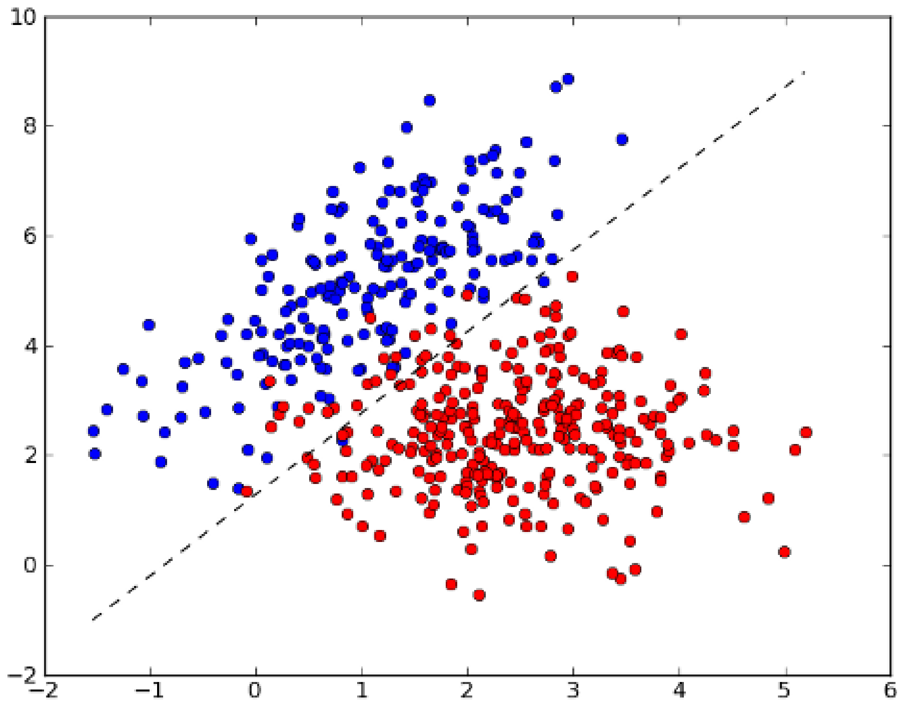
Parameters

t [1d or 2d array_like object] testing data ([M,], P)

w ()
Returns the coefficients.

5.2.1 Examples

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class -1
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = -np.ones(n2, dtype=np.int)
>>> x = np.concatenate((x1, x2), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2))
>>> p = mlpy.Perceptron(alpha=0.1, thr=0.05, maxiters=100) # basic perceptron
>>> p.learn(x, y)
>>> w = p.w()
>>> w
array([-69.00185254,  46.49202132])
>>> b = p.bias()
>>> b
-59.600000000000001
>>> p.err()
0.050000000000000003
>>> p.iters()
46
>>> xx = np.arange(np.min(x[:,0]), np.max(x[:,0]), 0.01)
>>> yy = - (w[0] * xx + b) / w[1] # separator line
>>> fig = plt.figure(1) # plot
>>> plot1 = plt.plot(x1[:, 0], x1[:, 1], 'ob', x2[:, 0], x2[:, 1], 'or')
>>> plot2 = plt.plot(xx, yy, '--k')
>>> plt.show()
```



```
>>> test = [[0, 2], [4, 2]] # test points
>>> p.pred(test)
array([ 1, -1])
>>> p.labels()
array([-1,  1])
```

5.3 Elastic Net Classifier

See [Hastie09], Chapter 18, page 661.

class `mlpy.`**`ElasticNetC`**(*lmb*, *eps*, *supp=True*, *tol=0.01*)

Elastic Net Regularization via Iterative Soft Thresholding for classification.

See the ElasticNet class documentation.

Initialization.

Parameters

lmb [float] regularization parameter controlling overfitting. *lmb* can be tuned via cross validation.

eps [float] correlation parameter preserving correlation among variables against sparsity. The solutions obtained for different values of the correlation parameter have the same prediction properties but different feature representation.

supp [bool] if True, the algorithm stops when the support of beta reached convergence. If False, the algorithm stops when the coefficients reached convergence, that is when the beta_{l}(i) - beta_{l+1}(i) > tol * beta_{l}(i) for all i.

tol [double] tolerance for convergence

bias()
Returns the bias.

labels()
Outputs the name of labels.

learn(*x*, *y*)
Compute the classification coefficients.

Parameters

x [2d array_like object (N x P)] matrix

y [1d array_like object integer (N)] class labels

pred(*t*)
Compute the predicted labels.

Parameters

t [1d or 2d array_like object ([M,] P)] test data

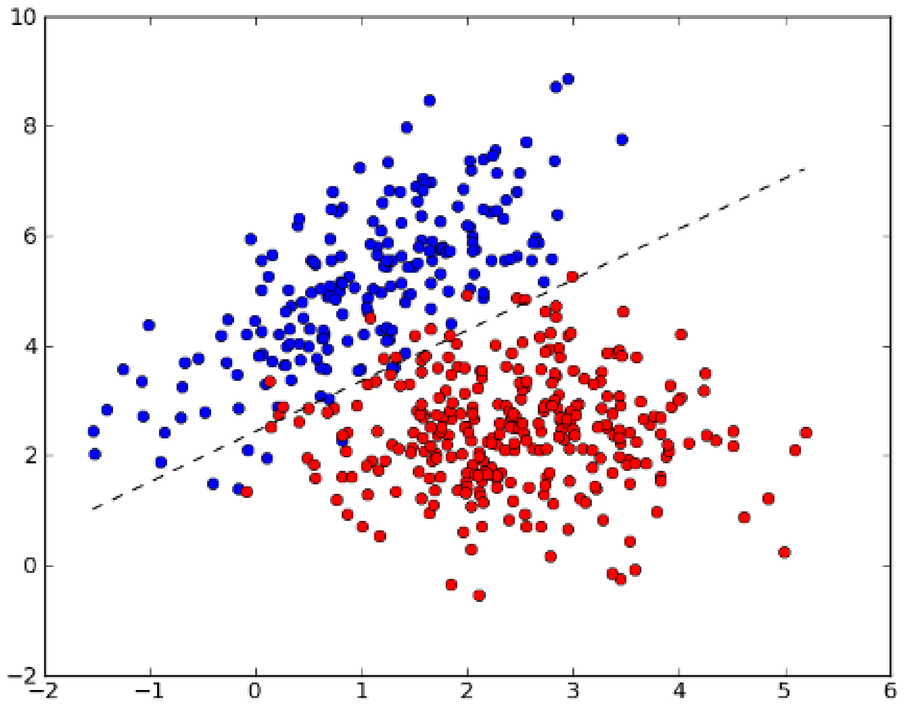
Returns

p [integer or 1d numpy array] predicted labels

w()
Returns the coefficients.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class -1
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = -np.ones(n2, dtype=np.int)
>>> x = np.concatenate((x1, x2), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2))
>>> en = mlpy.ElasticNetC(lmb=0.01, eps=0.001)
>>> en.learn(x, y)
>>> w = en.w()
>>> w
array([-0.27733363,  0.30115026])
>>> b = en.bias()
>>> b
-0.73445916200332606
>>> en.iters()
1000
>>> xx = np.arange(np.min(x[:,0]), np.max(x[:,0]), 0.01)
>>> yy = - (w[0] * xx + b) / w[1] # separator line
>>> fig = plt.figure(1) # plot
>>> plot1 = plt.plot(x1[:, 0], x1[:, 1], 'ob', x2[:, 0], x2[:, 1], 'or')
>>> plot2 = plt.plot(xx, yy, '--k')
>>> plt.show()
```



```
>>> test = [[1, 4], [2, 2]] # test points
>>> en.pred(test)
array([ 1., -1.])
```

5.4 Logistic Regression

See *Large Linear Classification from [LIBLINEAR]*

5.5 Support Vector Classification

See *Large Linear Classification from [LIBLINEAR]*

5.6 Diagonal Linear Discriminant Analysis (DLDA)

See [Hastie09], page 651.

class `mlpy.`**`DLDA`**`(`*delta*`)`

Diagonal Linear Discriminant Analysis classifier. The algorithm uses the procedure called Nearest Shrunken Centroids (NSC).

Initialization.

Parameters

delta [float] regularization parameter

dprime ()
Return the dprime d'_kj (C, P), where C is the number of classes.

labels ()
Outputs the name of labels.

learn (*x*, *y*)
Learning method.

Parameters

x [2d array_like object] training data (N, P)

y [1d array_like object integer] target values (N)

pred (*t*)
Does classification on test vector(s) t.

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

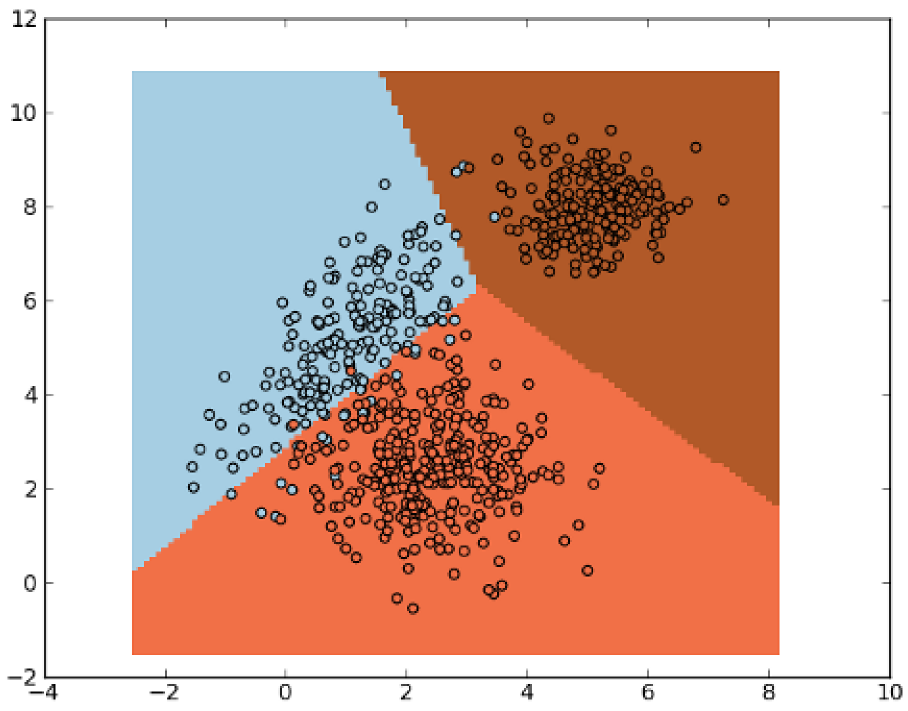
p [int or 1d numpy array] the predicted class(es) for t is returned.

prob (*t*)
For each sample returns C (number of classes) probability estimates.

sel ()
Returns the most important features (the features that have a nonzero dprime for at least one of the classes).

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 0
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.zeros(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class 1
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = np.ones(n2, dtype=np.int)
>>> mean3, cov3, n3 = [5, 8], [[0.5,0],[0,0.5]], 200 # 200 samples of class 2
>>> x3 = np.random.multivariate_normal(mean3, cov3, n3)
>>> y3 = 2 * np.ones(n3, dtype=np.int)
>>> x = np.concatenate((x1, x2, x3), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2, y3))
>>> da = mlpy.DLDA(delta=0.1)
>>> da.learn(x, y)
>>> xmin, xmax = x[:,0].min()-1, x[:,0].max()+1
>>> ymin, ymax = x[:,1].min()-1, x[:,1].max()+1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.1), np.arange(ymin, ymax, 0.1))
>>> xnew = np.c_[xx.ravel(), yy.ravel()]
>>> ynew = da.pred(xnew).reshape(xx.shape)
>>> fig = plt.figure(1)
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> plot1 = plt.pcolormesh(xx, yy, ynew)
>>> plot2 = plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```



5.7 Golub Classifier

class `mlpy.`**`Golub`**

Golub binary classifier described in [Golub99].

Decision function is D(x) = w (x-mu), where w is defined as w_i = (mu_i(+) - mu_i(-)) / (std_i(+) + std_i(-)) and mu id defined as (mu(+) + mu(-)) / 2.

Initialization.

`labels`()

Outputs the name of labels.

`learn`(*x*, *y*)

Learning method.

Parameters

x [2d array_like object] training data (N, P)

y [1d array_like object integer (only two classes)] target values (N)

`pred`(*t*)

Prediction method.

Parameters

t [1d or 2d array_like object] testing data ([M,], P)

`w`()

Returns the coefficients.

CHAPTER

SIX

KERNELS

6.1 Kernel Functions

A kernel is a function κ that for all $\mathbf{t}, \mathbf{x} \in X$ satisfies $\kappa(\mathbf{t}, \mathbf{x}) = \langle\Phi(\mathbf{t}), \Phi(\mathbf{x})\rangle$, where Φ is a mapping from X to an (inner product) feature space F, $\Phi : \mathbf{t} \longmapsto \Phi(\mathbf{t}) \in F$.

The following functions take two array-like objects `t` (M, P) and `x` (N, P) and compute the (M, N) matrix $\mathbf{K^t}$ with entries

$$\mathbf{K^t}_{ij} = \kappa(\mathbf{t}_i, \mathbf{x}_j).$$

6.2 Kernel Classes

class `mlpy.`**`Kernel`**
Base class for kernels.

class `mlpy.`**`KernelLinear`**
Linear kernel, t_i' x_j.

class `mlpy.`**`KernelPolynomial`**(*gamma=1.0*, *b=1.0*, *d=2.0*)
Polynomial kernel, (gamma t_i' x_j + b)^d.

class `mlpy.`**`KernelGaussian`**(*sigma=1.0*)
Gaussian kernel, exp(-||t_i - x_j||^2 / 2 * sigma^2).

class `mlpy.`**`KernelExponential`**(*sigma=1.0*)
Exponential kernel, exp(-||t_i - x_j|| / 2 * sigma^2).

class `mlpy.`**`KernelSigmoid`**(*gamma=1.0*, *b=1.0*)
Sigmoid kernel, tanh(gamma t_i' x_j + b).

6.3 Functions

`mlpy.`**`kernel_linear`**(*t*, *x*)
Linear kernel, t_i' x_j.

`mlpy.`**`kernel_polynomial`**(*t*, *x*, *gamma=1.0*, *b=1.0*, *d=2.0*)
Polynomial kernel, (gamma t_i' x_j + b)^d.

mlpy.**kernel_gaussian**(*t*, *x*, *sigma=1.0*)
Gaussian kernel, exp(-||t_i - x_j||^2 / 2 * sigma^2).

mlpy.**kernel_exponential**(*t*, *x*, *sigma=1.0*)
Exponential kernel, exp(-||t_i - x_j|| / 2 * sigma^2).

mlpy.**kernel_sigmoid**(*t*, *x*, *gamma=1.0*, *b=1.0*)
Sigmoid kernel, tanh(gamma t_i' x_j + b).

Example:

```
>>> import mlpy
>>> x = [[5, 1, 3, 1], [7, 1, 11, 4], [0, 4, 2, 9]] # three training points
>>> K = mlpy.kernel_gaussian(x, x, sigma=10) # compute the kernel matrix K_ij = k(x_i, x_j)
>>> K
array([[ 1.        ,  0.68045064,  0.60957091],
       [ 0.68045064,  1.        ,  0.44043165],
       [ 0.60957091,  0.44043165,  1.        ]])
>>> t = [[8, 1, 5, 1], [7, 1, 11, 4]] # two test points
>>> Kt = mlpy.kernel_gaussian(t, x, sigma=10) # compute the test kernel matrix Kt_ij = <Phi(t_i), Ph
>>> Kt
array([[ 0.93706746,  0.7945336 ,  0.48190899],
       [ 0.68045064,  1.        ,  0.44043165]])
```

6.4 Centering in Feature Space

The centered kernel matrix $\tilde{\mathbf{K}}^{\mathbf{t}}$ is computed by:

$$\tilde{\mathbf{K}}^{\mathbf{t}}_{ij} = \left\langle \Phi(\mathbf{t}_i) - \frac{1}{N}\sum_{m=1}^{N}\Phi(\mathbf{x}_m), \Phi(\mathbf{x}_j) - \frac{1}{N}\sum_{n=1}^{N}\Phi(\mathbf{x}_n) \right\rangle.$$

We can express $\tilde{\mathbf{K}}^{\mathbf{t}}$ in terms of $\mathbf{K}^{\mathbf{t}}$ and $\mathbf{K}$:

$$\tilde{\mathbf{K}}^{\mathbf{t}}_{ij} = \mathbf{K}^{\mathbf{t}} - \mathbf{1}_N^T\mathbf{K} - \mathbf{K}^{\mathbf{t}}\mathbf{1}_N + \mathbf{1}_N^T\mathbf{K}\mathbf{1}_N$$

where $\mathbf{1}_N$ is the $N \times M$ matrix with all entries equal to $1/N$ and $\mathbf{K}$ is $\mathbf{K}_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$.

mlpy.**kernel_center**(*Kt*, *K*)
Centers the testing kernel matrix Kt respect the training kernel matrix K. If Kt = K (kernel_center(K, K), where K = k(x_i, x_j)), the function centers the kernel matrix K.

Parameters

Kt [2d array_like object (M, N)] test kernel matrix Kt_ij = k(t_i, x_j). If Kt = K the function centers the kernel matrix K

K [2d array_like object (N, N)] training kernel matrix K_ij = k(x_i, x_j)

Returns

Ktcentered [2d numpy array (M, N)] centered version of Kt

Example:

```
>>> Kcentered = mlpy.kernel_center(K, K) # center K
>>> Kcentered
array([[ 0.19119746, -0.07197215, -0.11922531],
       [-0.07197215,  0.30395696, -0.23198481],
       [-0.11922531, -0.23198481,  0.35121011]])
>>> Ktcentered = mlpy.kernel_center(Kt, K) # center the test kernel matrix Kt respect to K
>>> Ktcentered
array([[ 0.15376875,  0.06761464, -0.22138339],
       [-0.07197215,  0.30395696, -0.23198481]])
```

6.5 Make a Custom Kernel

TODO

CHAPTER

SEVEN

NON LINEAR METHODS FOR REGRESSION

7.1 Kernel Ridge Regression

class `mlpy.`**`KernelRidge`**(*lmb=1.0*, *kernel=None*)

Kernel Ridge Regression (dual).

Initialization.

Parameters

lmb [float (>= 0.0)] regularization parameter

kernel [None or mlpy.Kernel object.] if kernel is None, K and Kt in .learn() and in .pred() methods must be precomputed kernel matricies, else K and Kt must be training (resp. test) data in input space.

`alpha`()

Return alpha.

`b`()

Return b.

`learn`(*K*, *y*)

Compute the regression coefficients.

Parameters:

K: 2d array_like object precomputed training kernel matrix (if kernel=None); training data in input space (if kernel is a Kernel object)

y [1d array_like object (N)] target values

`pred`(*Kt*)

Compute the predicted response.

Parameters

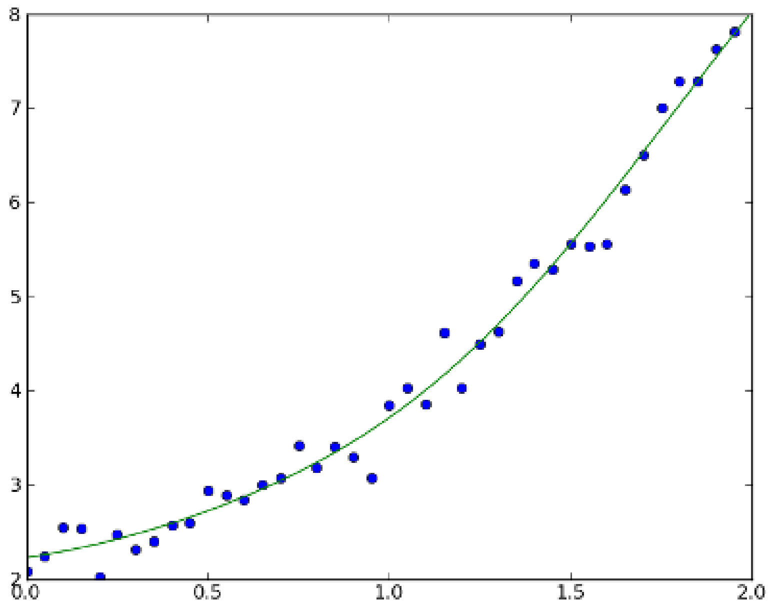
Kt [1d or 2d array_like object] precomputed test kernel matrix. (if kernel=None); test data in input space (if kernel is a Kernel object).

Returns

p [integer or 1d numpy darray] predicted response

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> x = np.arange(0, 2, 0.05).reshape(-1, 1) # training points
>>> y = np.ravel(np.exp(x)) + np.random.normal(1, 0.2, x.shape[0]) # target values
>>> xt = np.arange(0, 2, 0.01).reshape(-1, 1) # testing points
>>> K = mlpy.kernel_gaussian(x, x, sigma=1) # training kernel matrix
>>> Kt = mlpy.kernel_gaussian(xt, x, sigma=1) # testing kernel matrix
>>> krr = KernelRidge(lmb=0.01)
>>> krr.learn(K, y)
>>> yt = krr.pred(Kt)
>>> fig = plt.figure(1)
>>> plot1 = plt.plot(x[:, 0], y, 'o')
>>> plot2 = plt.plot(xt[:, 0], yt)
>>> plt.show()
```



7.2 Support Vector Regression

See *Support Vector Machines (SVMs)*

CHAPTER

EIGHT

NON LINEAR METHODS FOR CLASSIFICATION

8.1 Parzen-based classifier

class `mlpy.`**`Parzen`**(*kernel=None*)

Parzen based classifier (binary).

Initialization.

Parameters

kernel [None or mlpy.Kernel object.] if kernel is None, K and Kt in .learn() and in .pred() methods must be precomputed kernel matricies, else K and Kt must be training (resp. test) data in input space.

`alpha`()

Return alpha.

`b`()

Return b.

`labels`()

Outputs the name of labels.

`learn`(*K*, *y*)

Compute alpha and b.

Parameters:

K: 2d array_like object precomputed training kernel matrix (if kernel=None); training data in input space (if kernel is a Kernel object)

y [1d array_like object (N)] target values

`pred`(*Kt*)

Compute the predicted class.

Parameters

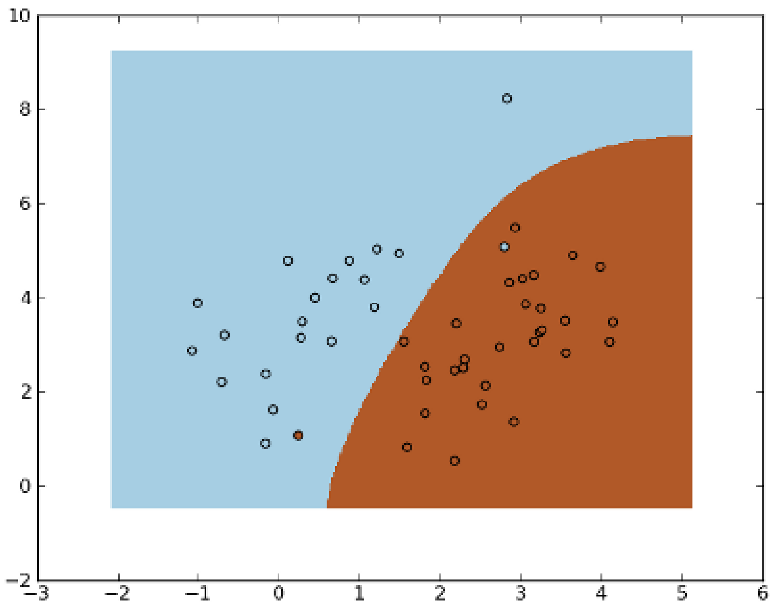
Kt [1d or 2d array_like object] precomputed test kernel matrix. (if kernel=None); test data in input space (if kernel is a Kernel object).

Returns

p [integer or 1d numpy array] predicted class

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 4.5], [[1,1],[1,2]], 20  # 20 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,1],[1,2]], 30 # 30 samples of class 2
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = 2 * np.ones(n2, dtype=np.int)
>>> x = np.concatenate((x1, x2), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2))
>>> K = mlpy.kernel_gaussian(x, x, sigma=2) # kernel matrix
>>> parzen = mlpy.Parzen()
>>> parzen.learn(K, y)
>>> xmin, xmax = x[:,0].min()-1, x[:,0].max()+1
>>> ymin, ymax = x[:,1].min()-1, x[:,1].max()+1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.02), np.arange(ymin, ymax, 0.02))
>>> xt = np.c_[xx.ravel(), yy.ravel()] # test points
>>> Kt = mlpy.kernel_gaussian(xt, x, sigma=2) # test kernel matrix
>>> yt = parzen.pred(Kt).reshape(xx.shape)
>>> fig = plt.figure(1)
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> plot1 = plt.pcolormesh(xx, yy, yt)
>>> plot2 = plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```



8.2 Support Vector Classification

See *Support Vector Machines (SVMs)*

8.3 Kernel Fisher Discriminant Classifier

class `mlpy.`**`KFDAC`**(*lmb=0.001*, *kernel=None*)

Kernel Fisher Discriminant Analysis Classifier (binary classifier).

The bias term (b) is computed as in [Gavin03].

Initialization.

Parameters

lmb [float (>= 0.0)] regularization parameter

kernel [None or mlpy.Kernel object.] if kernel is None, K and Kt in .learn() and in .transform() methods must be precomputed kernel matricies, else K and Kt must be training (resp. test) data in input space.

`alpha`()

Return alpha.

`b`()

Return b.

`labels`()

Outputs the name of labels.

`learn`(*K*, *y*)

Learning method.

Parameters

K: 2d array_like object precomputed training kernel matrix (if kernel=None); training data in input space (if kernel is a Kernel object)

y [1d array_like object integer (N)] class labels (only two classes)

`pred`(*Kt*)

Compute the predicted response.

Parameters

Kt [1d or 2d array_like object] precomputed test kernel matrix. (if kernel=None); test data in input space (if kernel is a Kernel object).

Returns

p [integer or 1d numpy array] the predicted class(es)

8.4 k-Nearest-Neighbor

class `mlpy.`**`KNN`**(*k*)

k-Nearest Neighbor (euclidean distance)

Parameters

k [int] number of nearest neighbors

`KNN.`**`learn`**(*x*, *y*)

Learn method.

Parameters

x [2d array_like object (N,P)] training data

y [1d array_like integer] class labels

`KNN.pred(t)`
Predict KNN model on a test point(s).

Parameters

t [1d or 2d array_like object ([M,] P)] test point(s)

Returns

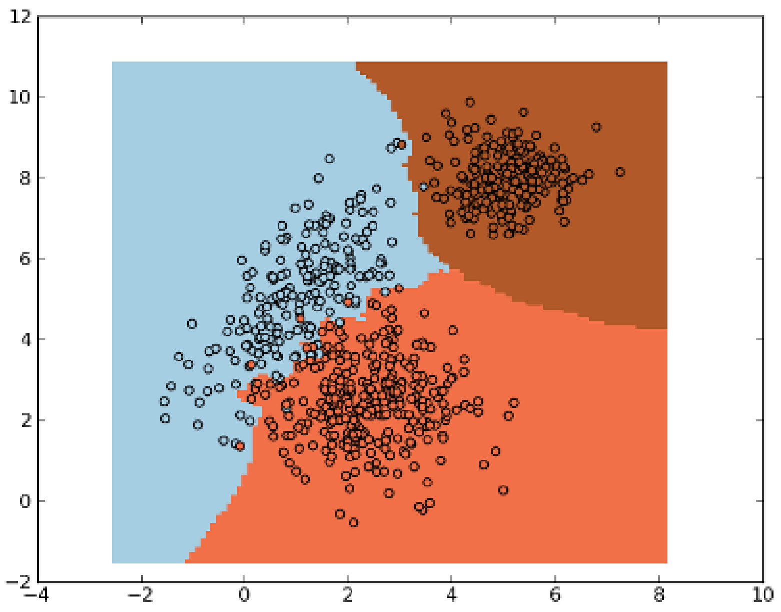
p [int or 1d numpy array] the predicted value(s). Retuns the smallest label minus one (KNN.labels()[0]-1) when the classification is not unique.

`KNN.nclasses()`
Returns the number of classes.

`KNN.labels()`
Outputs the name of labels.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class 2
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = 2 * np.ones(n2, dtype=np.int)
>>> mean3, cov3, n3 = [5, 8], [[0.5,0],[0,0.5]], 200 # 200 samples of class 3
>>> x3 = np.random.multivariate_normal(mean3, cov3, n3)
>>> y3 = 3 * np.ones(n3, dtype=np.int)
>>> x = np.concatenate((x1, x2, x3), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2, y3))
>>> knn = mlpy.KNN(k=3)
>>> knn.learn(x, y)
>>> xmin, xmax = x[:,0].min()-1, x[:,0].max()+1
>>> ymin, ymax = x[:,1].min()-1, x[:,1].max()+1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.1), np.arange(ymin, ymax, 0.1))
>>> xnew = np.c_[xx.ravel(), yy.ravel()]
>>> ynew = knn.pred(xnew).reshape(xx.shape)
>>> ynew[ynew == 0] = 1 # set the samples with no unique classification to 1
>>> fig = plt.figure(1)
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> plot1 = plt.pcolormesh(xx, yy, ynew)
>>> plot2 = plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```



8.5 Classification Tree

class `mlpy.`**`ClassTree`**(*stumps=0*, *minsize=1*)

Classification Tree (gini index)

Parameters

stumps [bool] True: compute single split or False: standard tree

minsize [int (>=0)] minimum number of cases required to split a leaf

`ClassTree.`**`learn`**(*x*, *y*)

Learn method.

Parameters

x [2d array_like object (N x P)] training data

y [1d array_like integer] class labels

`ClassTree.`**`pred`**(*t*)

Predict Tree model on a test point(s).

Parameters

t [1d or 2d array_like object ([M,] P)] test point(s)

Returns

p [int or 1d numpy array] the predicted value(s). Retuns the smallest label minus one (ClassTree.labels()[0]-1) when the classification is not unique.

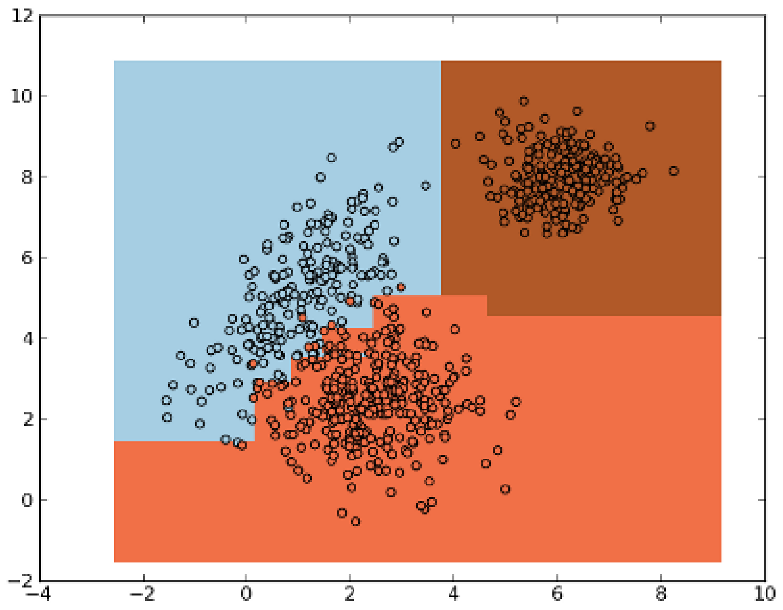
`ClassTree.`**`nclasses`**()

Returns the number of classes.

ClassTree.**labels**()
: Outputs the name of labels.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class 2
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = 2 * np.ones(n2, dtype=np.int)
>>> mean3, cov3, n3 = [6, 8], [[0.5,0],[0,0.5]], 200 # 200 samples of class 3
>>> x3 = np.random.multivariate_normal(mean3, cov3, n3)
>>> y3 = 3 * np.ones(n3, dtype=np.int)
>>> x = np.concatenate((x1, x2, x3), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2, y3))
>>> tree = mlpy.ClassTree(minsize=10)
>>> tree.learn(x, y)
>>> xmin, xmax = x[:,0].min()-1, x[:,0].max()+1
>>> ymin, ymax = x[:,1].min()-1, x[:,1].max()+1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.1), np.arange(ymin, ymax, 0.1))
>>> xnew = np.c_[xx.ravel(), yy.ravel()]
>>> ynew = tree.pred(xnew).reshape(xx.shape)
>>> ynew[ynew == 0] = 1 # set the samples with no unique classification to 1
>>> fig = plt.figure(1)
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> plot1 = plt.pcolormesh(xx, yy, ynew)
>>> plot2 = plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```



8.6 Maximum Likelihood Classifier

class `mlpy.`**`MaximumLikelihoodC`**

Maximum Likelihood Classifier

`MaximumLikelihoodC.`**`learn`**`(x, y)`

Learn method.

Parameters

x [2d array_like object (N,P)] training data

y [1d array_like integer] class labels

`MaximumLikelihoodC.`**`pred`**`(t)`

Predict Maximum Likelihood model on a test point(s).

Parameters

t [1d or 2d array_like object ([M,] P)] test point(s)

Returns

p [int or 1d numpy array] the predicted value(s). Retuns the smallest label minus one (MaximumLikelihoodC.labels()[0]-1) when the classification is not unique.

`MaximumLikelihoodC.`**`nclasses`**`()`

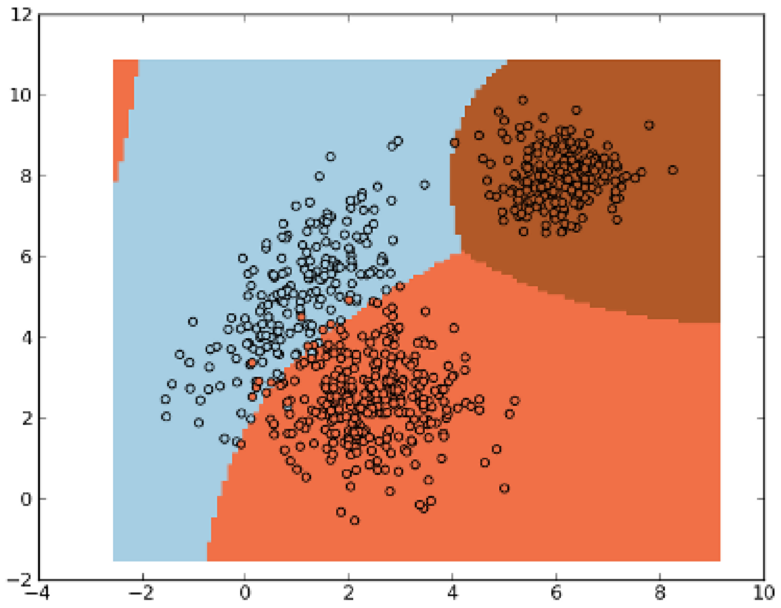
Returns the number of classes.

`MaximumLikelihoodC.`**`labels`**`()`

Outputs the name of labels.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class 2
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = 2 * np.ones(n2, dtype=np.int)
>>> mean3, cov3, n3 = [6, 8], [[0.5,0],[0,0.5]], 200 # 200 samples of class 3
>>> x3 = np.random.multivariate_normal(mean3, cov3, n3)
>>> y3 = 3 * np.ones(n3, dtype=np.int)
>>> x = np.concatenate((x1, x2, x3), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2, y3))
>>> ml = mlpy.MaximumLikelihoodC()
>>> ml.learn(x, y)
>>> xmin, xmax = x[:,0].min()-1, x[:,0].max()+1
>>> ymin, ymax = x[:,1].min()-1, x[:,1].max()+1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.1), np.arange(ymin, ymax, 0.1))
>>> xnew = np.c_[xx.ravel(), yy.ravel()]
>>> ynew = ml.pred(xnew).reshape(xx.shape)
>>> ynew[ynew == 0] = 1 # set the samples with no unique classification to 1
>>> fig = plt.figure(1)
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> plot1 = plt.pcolormesh(xx, yy, ynew)
>>> plot2 = plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```


12
10
8
6
4
2
0
−2
−4
−2
0
2
4
6
8
10

CHAPTER

NINE

SUPPORT VECTOR MACHINES (SVMS)

9.1 Support Vector Machines from [LIBSVM]

class `mlpy.`**`LibSvm`**(*svm_type='c_svc', kernel_type='linear', degree=3, gamma=0.001, coef0=0, C=1, nu=0.5, eps=0.001, p=0.1, cache_size=100, shrinking=True, probability=False, weight={}*)

LibSvm.

Parameters

svm_type [string] SVM type, can be one of: 'c_svc', 'nu_svc', 'one_class', 'epsilon_svr', 'nu_svr'

kernel_type [string] kernel type, can be one of: 'linear' (uT*v), 'poly' ((gamma*uT*v + coef0)^degree), 'rbf' (exp(-gamma*|u-v|^2)), 'sigmoid' (tanh(gamma*uT*v + coef0))

degree [int (for 'poly' kernel_type)] degree in kernel

gamma [float (for 'poly', 'rbf', 'sigmoid' kernel_type)] gamma in kernel (e.g. 1 / number of features)

coef0 [float (for 'poly', 'sigmoid' kernel_type)] coef0 in kernel

C [float (for 'c_svc', 'epsilon_svr', 'nu_svr')] cost of constraints violation

nu [float (for 'nu_svc', 'one_class', 'nu_svr')] nu parameter

eps [float] stopping criterion, usually 0.00001 in nu-SVC, 0.001 in others

p [float (for 'epsilon_svr')] p is the epsilon in epsilon-insensitive loss function of epsilon-SVM regression

cache_size [float [MB]] size of the kernel cache, specified in megabytes

shrinking [bool] use the shrinking heuristics

probability [bool] predict probability estimates

weight [dict] changes the penalty for some classes (if the weight for a class is not changed, it is set to 1). For example, to change penalty for classes 1 and 2 to 0.5 and 0.8 respectively set weight={1:0.5, 2:0.8}

`LibSvm.`**`learn`**(*x, y*)

Constructs the model. For classification, y is an integer indicating the class label (multi-class is supported). For regression, y is the target value which can be any real number. For one-class SVM, it's not used so can be any number.

Parameters

x [2d array_like object] training data (N, P)

y [1d array_like object] target values (N)

`LibSvm.`**`pred`**`(`*t*`)`

Does classification or regression on test vector(s) t.

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

p [for a classification model, the predicted class(es) for t is] returned. For a regression model, the function value(s) of t calculated using the model is returned. For an one-class model, +1 or -1 is returned.

`LibSvm.`**`pred_probability`**`(`*t*`)`

Returns C (number of classes) probability estimates. For a 'c_svc' and 'nu_svc' classification models with probability information, this method computes 'number of classes' probability estimates.

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

probability estimates [1d (C) or 2d numpy array (M,C)] probability estimates for each observation.

`LibSvm.`**`pred_values`**`(`*t*`)`

Returns D decision values. For a classification model with C classes, this method returns D=C*(C-1)/2 decision values for each test sample. The order is label[0] vs. label[1], ..., label[0] vs. label[C-1], label[1] vs. label[2], ..., label[C-2] vs. label[C-1], where label can be obtained from the method labels().

For a one-class model, this method returns D=1 decision value for each test sample.

For a regression model, this method returns the predicted value as in pred()

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

decision values [1d (D) or 2d numpy array (M,D)] decision values for each observation.

`LibSvm.`**`labels`**`()`

For a classification model, this method outputs the name of labels. For regression and one-class models, this method returns None.

`LibSvm.`**`nclasses`**`()`

Get the number of classes. = 2 in regression and in one class SVM

`LibSvm.`**`nsv`**`()`

Get the total number of support vectors.

`LibSvm.`**`label_nsv`**`()`

Return a dictionary containing the number of support vectors for each class (for classification).

static `LibSvm.`**`load_model`**`(`*filename*`)`

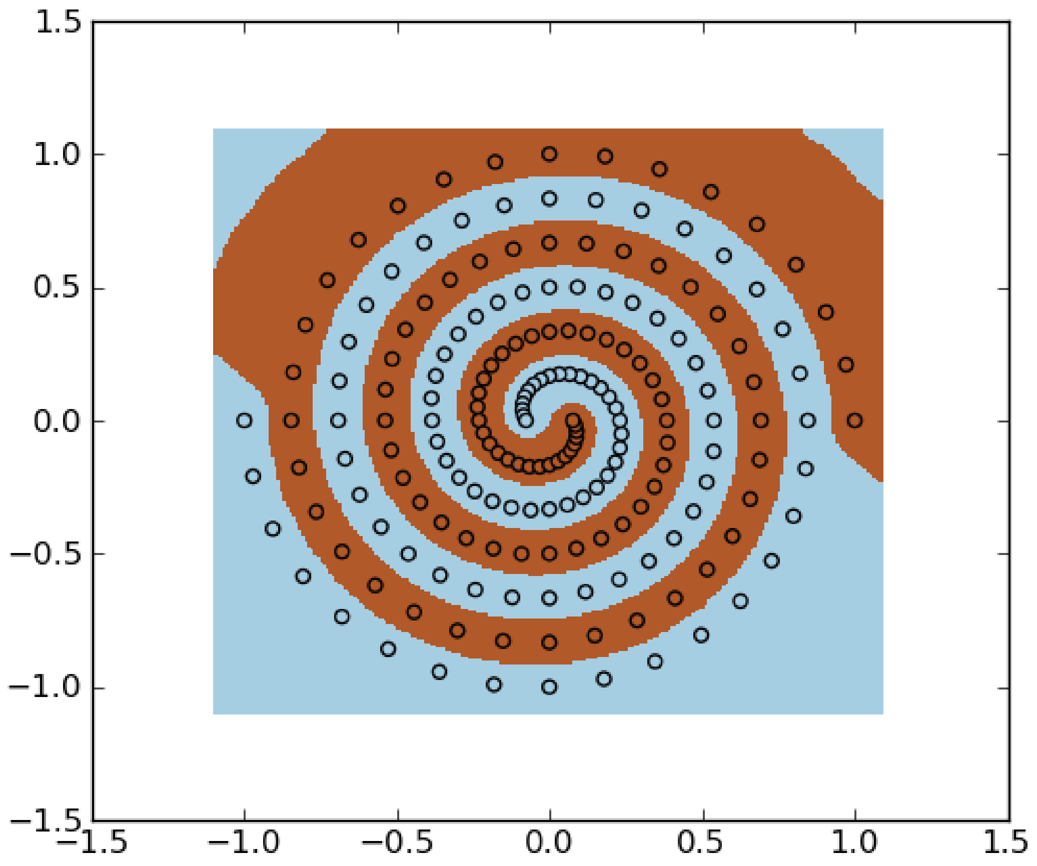
Loads model from file. Returns a LibSvm object with the learn() method disabled.

`LibSvm.`**`save_model`**`(`*filename*`)`

Saves model to a file.

Example on `spiral dataset`:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> f = np.loadtxt("spiral.data")
>>> x, y = f[:, :2], f[:, 2]
>>> svm = mlpy.LibSvm(svm_type='c_svc', kernel_type='rbf', gamma=100)
>>> svm.learn(x, y)
>>> xmin, xmax = x[:,0].min()-0.1, x[:,0].max()+0.1
>>> ymin, ymax = x[:,1].min()-0.1, x[:,1].max()+0.1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.01), np.arange(ymin, ymax, 0.01))
>>> xnew = np.c_[xx.ravel(), yy.ravel()]
>>> ynew = svm.pred(xnew).reshape(xx.shape)
>>> fig = plt.figure(1)
>>> plt.set_cmap(plt.cm.Paired)
>>> plt.pcolormesh(xx, yy, ynew)
>>> plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```



9.2 Kernel Adatron

class `mlpy.KernelAdatron`(*C=1000*, *maxsteps=1000*, *eps=0.01*)
Kernel Adatron algorithm without-bias-term (binary classifier).

The algoritm handles a version of the 1-norm soft margin support vector machine. If C is very high the algoritm handles a version of the hard margin SVM.

Use positive definite kernels (such as Gaussian and Polynomial kernels)

Parameters

C [float] upper bound on the value of alpha

maxsteps [integer (> 0)] maximum number of steps

eps [float (>=0)] the algoritm stops when abs(1 - margin) < eps

`KernelAdatron.learn`(*K*, *y*)
Learn.

Parameters:

K: 2d array_like object (N, N) precomputed kernel matrix

y [1d array_like object (N)] target values

`KernelAdatron.pred`(*Kt*)
Compute the predicted class.

Parameters

Kt [1d or 2d array_like object ([M], N)] test kernel matrix. Precomputed inner products (in feature space) between M testing and N training points.

Returns

p [integer or 1d numpy array] predicted class

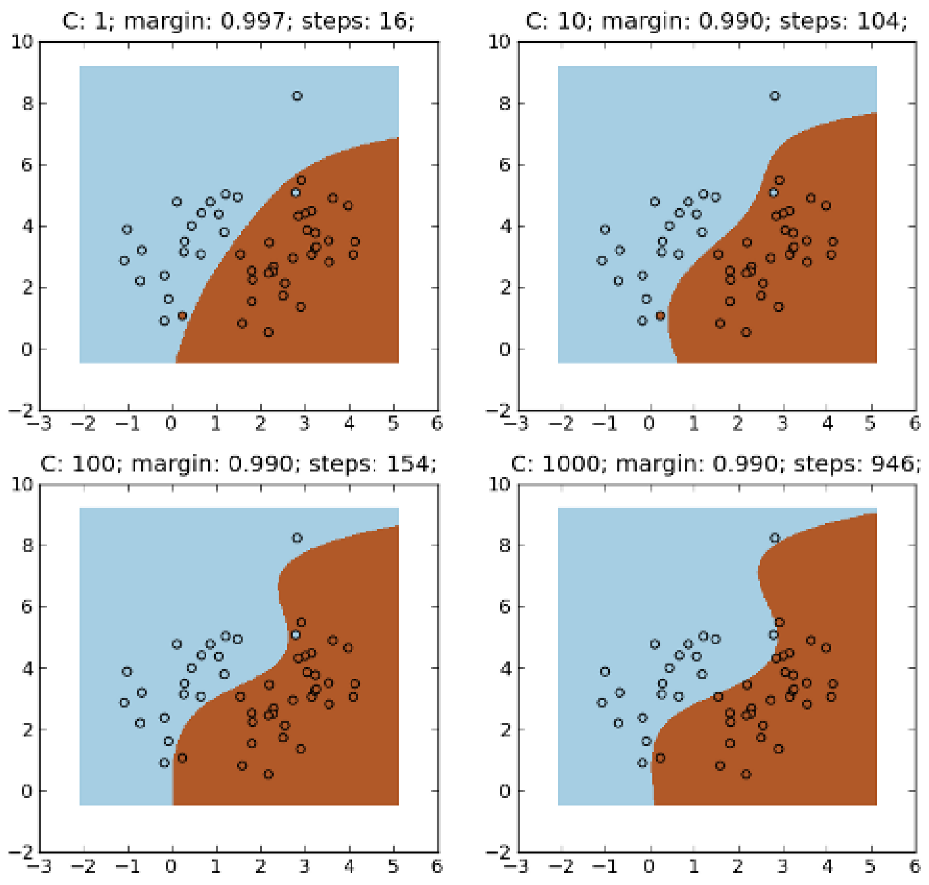
`KernelAdatron.margin`()
Return the margin.

`KernelAdatron.steps`()
Return the number of steps performed.

`KernelAdatron.alpha`()
Return alpha

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 4.5], [[1,1],[1,2]], 20  # 20 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,1],[1,2]], 30 # 30 samples of class 2
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = 2 * np.ones(n2, dtype=np.int)
>>> x = np.concatenate((x1, x2), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2))
>>> K = mlpy.kernel_gaussian(x, x, sigma=2) # kernel matrix
>>> xmin, xmax = x[:,0].min()-1, x[:,0].max()+1
>>> ymin, ymax = x[:,1].min()-1, x[:,1].max()+1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.02), np.arange(ymin, ymax, 0.02))
>>> xt = np.c_[xx.ravel(), yy.ravel()] # test points
>>> Kt = mlpy.kernel_gaussian(xt, x, sigma=2) # test kernel matrix
>>> fig = plt.figure(1)
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> for i, c in enumerate([1, 10, 100, 1000]):
...     ka = mlpy.KernelAdatron(C=c)
...     ax = plt.subplot(2, 2, i+1)
...     ka.learn(K, y)
...     ytest = ka.pred(Kt).reshape(xx.shape)
...     title = ax.set_title('C: %s; margin: %.3f; steps: %s;' % (c, ka.margin(), ka.steps()))
...     plot1 = plt.pcolormesh(xx, yy, ytest)
...     plot2 = plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```



CHAPTER

TEN

LARGE LINEAR CLASSIFICATION FROM [LIBLINEAR]

Solvers:

- **l2r_lr**: L2-regularized logistic regression (primal)
- **l2r_l2loss_svc_dual**: L2-regularized L2-loss support vector classification (dual)
- **l2r_l2loss_svc**: L2-regularized L2-loss support vector classification (primal)
- **l2r_l1loss_svc_dual**: L2-regularized L1-loss support vector classification (dual)
- **mcsvm_cs**: multi-class support vector classification by Crammer and Singer
- **l1r_l2loss_svc**: L1-regularized L2-loss support vector classification
- **l1r_lr**: L1-regularized logistic regression
- **l2r_lr_dual**: L2-regularized logistic regression (dual)

class `mlpy.`**`LibLinear`**(*solver_type='l2r_lr'*, *C=1*, *eps=0.01*, *weight={}*)

LibLinear is a simple class for solving large-scale regularized linear classification. It currently supports L2-regularized logistic regression/L2-loss support vector classification/L1-loss support vector classification, and L1-regularized L2-loss support vector classification/ logistic regression.

Parameters

solver_type [string] solver, can be one of 'l2r_lr', 'l2r_l2loss_svc_dual', 'l2r_l2loss_svc', 'l2r_l1loss_svc_dual', 'mcsvm_cs', 'l1r_l2loss_svc', 'l1r_lr', 'l2r_lr_dual'

C [float] cost of constraints violation

eps [float] stopping criterion

weight [dict] changes the penalty for some classes (if the weight for a class is not changed, it is set to 1). For example, to change penalty for classes 1 and 2 to 0.5 and 0.8 respectively set weight={1:0.5, 2:0.8}

`LibLinear.`**`learn`**(*x*, *y*)

Learning method.

Parameters

x [2d array_like object] training data (N, P)

y [1d array_like object] target values (N)

`LibLinear.`**`pred`**(*t*)

Does classification on test vector(s) t.

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

p [int or 1d numpy array] the predicted class(es) for t is returned.

`LibLinear.`**`pred_values`**`(`*t*`)`

Returns D decision values. D is 1 if there are two classes except multi-class svm by Crammer and Singer ('mcsvm_cs'), and is the number of classes otherwise. The pred() method returns the class with the highest decision value.

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

decision values [1d (D) or 2d numpy array (M, D)] decision values for each observation.

`LibLinear.`**`pred_probability`**`(`*t*`)`

Returns C (number of classes) probability estimates. The simple probability model of logistic regression is used.

Parameters

t [1d (one sample) or 2d array_like object] test data ([M,] P)

Returns

probability estimates [1d (C) or 2d numpy array (M, C)] probability estimates for each observation.

`LibLinear.`**`w`**`()`

Returns the coefficients. For 'mcsvm_cs' solver and for multiclass classification returns a 2d numpy array where w[i] contains the coefficients of label i. For binary classification an 1d numpy array is returned.

`LibLinear.`**`bias`**`()`

Returns the bias term(s). For 'mcsvm_cs' solver and for multiclass classification returns a 1d numpy array where b[i] contains the bias of label i (.labels()[i]). For binary classification a float is returned.

`LibLinear.`**`nfeature`**`()`

Returns the number of attributes.

`LibLinear.`**`nclasses`**`()`

Returns the number of classes.

`LibLinear.`**`labels`**`()`

Outputs the name of labels.

static `LibLinear.`**`load_model`**`(`*filename*`)`

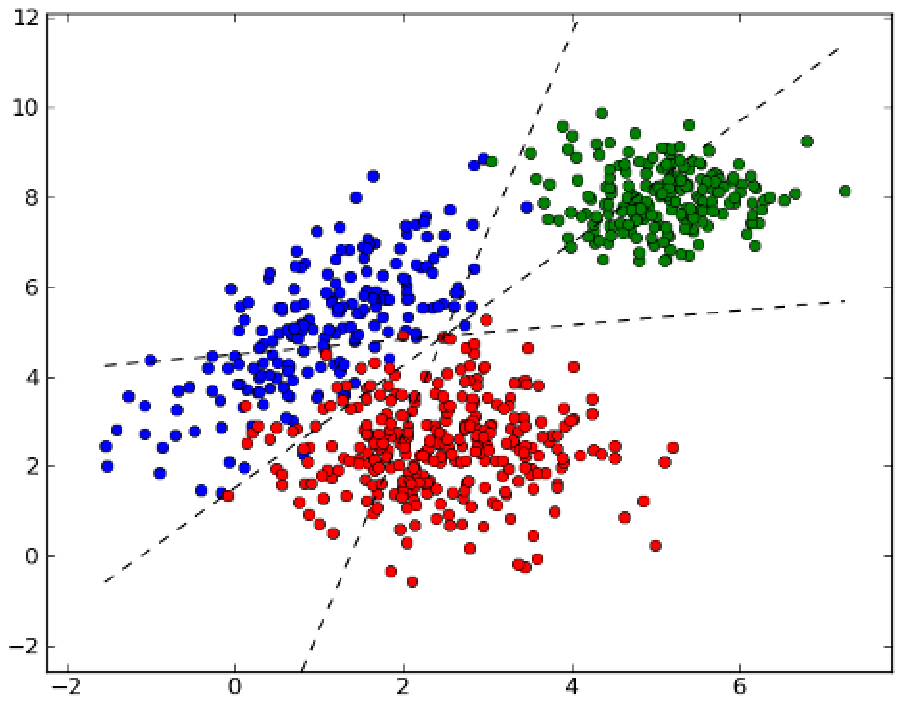
Loads model from file. Returns a LibLinear object with the learn() method disabled.

`LibLinear.`**`save_model`**`(`*filename*`)`

Saves a model to a file.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200  # 200 samples of class 0
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.zeros(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 samples of class 1
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = np.ones(n2, dtype=np.int)
>>> mean3, cov3, n3 = [5, 8], [[0.5,0],[0,0.5]], 200 # 200 samples of class 2
>>> x3 = np.random.multivariate_normal(mean3, cov3, n3)
>>> y3 = 2 * np.ones(n3, dtype=np.int)
>>> x = np.concatenate((x1, x2, x3), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2, y3))
>>> svm = mlpy.LibLinear(solver_type='l2r_l2loss_svc_dual', C=0.01)
>>> svm.learn(x, y)
>>> w = svm.w() # w[i]: coefficients for label svm.labels()[i]
>>> w
array([[-0.73225278,  0.33309388],
       [ 0.32295557, -0.44097029],
       [ 0.23192595,  0.11536679]])
>>> b = svm.bias() # b[i]: bias for label svm.labels()[i]
>>> b
array([-0.21631629,  0.96014472, -1.53933202])
>>> xx = np.arange(np.min(x[:,0]), np.max(x[:,0]), 0.01)
>>> yy1 = (xx* (w[1][0]-w[0][0]) + b[1] - b[0]) / (w[0][1]-w[1][1])
>>> yy2 = (xx* (w[2][0]-w[0][0]) + b[2] - b[0]) / (w[0][1]-w[2][1])
>>> yy3 = (xx* (w[2][0]-w[1][0]) + b[2] - b[1]) / (w[1][1]-w[2][1])
>>> fig = plt.figure(1) # plot
>>> plot1 = plt.plot(x1[:, 0], x1[:, 1], 'ob', x2[:, 0], x2[:, 1], 'or', x3[:, 0], x3[:, 1], 'og')
>>> plot2 = plt.plot(xx, yy1, '--k')
>>> plot3 = plt.plot(xx, yy2, '--k')
>>> plot4 = plt.plot(xx, yy3, '--k')
>>> plt.show()
```



```
>>> test = [[6,7], [4, 2]] # test points
>>> print svm.pred(test)
array([2, 1])
```

CHAPTER

ELEVEN

CLUSTER ANALYSIS

11.1 Hierarchical Clustering

Hierarchical Clustering algorithm derived from the R package 'amap' [Amap].

The condensed distance matrix *y* can be computed by `pdist()` function in **scipy** (<http://docs.scipy.org/doc/scipy/reference/spatial.distance.html>)

class `mlpy.`**`HCluster`**(*method='complete'*)

Hierarchical Cluster.

Initialization.

Parameters

method [string ('ward', 'single', 'complete', 'average', 'mcquitty', 'median', 'centroid')] the agglomeration method to be used

`cut`(*t*)

Cuts the tree into several groups by specifying the cut height.

Parameters

t [float] the threshold to apply when forming flat clusters

Returns

clust [1d numpy array] group memberships. Groups are in 0, ..., N-1.

`linkage`(*y*)

Performs hierarchical clustering on the condensed distance matrix y.

Parameters

y [1d array_like object] condensed distance matrix y. y must be a C(n, 2) sized vector where n is the number of original observations paired in the distance matrix.

11.2 Memory-saving Hierarchical Clustering

Memory-saving Hierarchical Clustering derived from the R and Python package 'fastcluster' [fastcluster].

class `mlpy.`**`MFastHCluster`**(*method='single'*)

Memory-saving Hierarchical Cluster (only euclidean distance).

This method needs O(NP) memory for clustering of N point in R^P.

Initialization.

Parameters

method [string ('single', 'centroid', 'median', 'ward')] the agglomeration method to be used

`Z()`

Returns the hierarchical clustering encoded as a linkage matrix. See *scipy.cluster.hierarchy.linkage*.

`cut(t)`

Cuts the tree into several groups by specifying the cut height.

Parameters

t [float] the threshold to apply when forming flat clusters

Returns

clust [1d numpy array] group memberships. Groups are in 0, ..., N-1.

`linkage(x)`

Performs hierarchical clustering.

Parameters

x [2d array_like object (N, P)] vector data, N observations in R^P

11.3 k-means

`mlpy.kmeans(x, k, plus=False, seed=0)`

k-means clustering.

Parameters

x [2d array_like object (N, P)] data

k [int (1<k<N)] number of clusters

plus [bool] k-means++ algorithm for initialization

seed [int] random seed for initialization

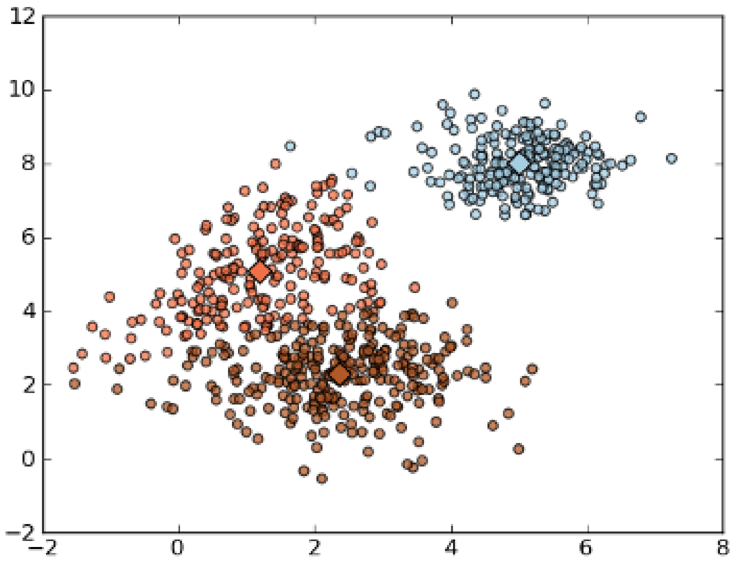
Returns

clusters, means, steps: 1d array, 2d array, int cluster membership in 0,...,K-1, means (K,P), number of steps

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 5], [[1,1],[1,2]], 200 # 200 points, mean=(1,5)
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,0],[0,1]], 300 # 300 points, mean=(2.5,2.5)
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> mean3, cov3, n3 = [5, 8], [[0.5,0],[0,0.5]], 200 # 200 points, mean=(5,8)
>>> x3 = np.random.multivariate_normal(mean3, cov3, n3)
>>> x = np.concatenate((x1, x2, x3), axis=0) # concatenate the samples
>>> cls, means, steps = mlpy.kmeans(x, k=3, plus=True)
>>> steps
13
```

```
>>> fig = plt.figure(1)
>>> plot1 = plt.scatter(x[:,0], x[:,1], c=cls, alpha=0.75)
>>> plot2 = plt.scatter(means[:,0], means[:,1], c=np.unique(cls), s=128, marker='d') # plot the mean
>>> plt.show()
```



CHAPTER

TWELVE

ALGORITHMS FOR FEATURE WEIGHTING

12.1 Iterative RELIEF

class `mlpy.`**`IRelief`**`(`*T=1000*, *sigma=1.0*, *theta=0.001*`)`

Iterative RELIEF for feature weighting.

Parameters

T [integer (> 0)] max loops

sigma [float (> 0.0)] kernel width

theta [float (> 0.0)] convergence parameter

`learn``(`*x*, *y*`)`

Compute the feature weights.

Parameters

x [2d array_like object] training data (N, P)

y [1d array_like object integer (only two classes)] target values (N)

Raises SigmaError

`loops``()`

Returns the number of loops.

`weights``()`

Returns the feature weights.

CHAPTER

THIRTEEN

FEATURE SELECTION

13.1 Recursive Feature Elimination

mlpy.**rfe_w2**(*x*, *y*, *p*, *classifier*)
RFE algorithm, where the ranking criteria is w^2, described in [Guyon02]. *classifier* must be an linear classifier with learn() and w() methods.

Parameters

x: 2d array_like object (N,P) training data

y [1d array_like object integer (N)] class labels (only two classes)

p [float [0.0, 1.0]] percentage of features (upper rounded) to remove at each iteration (p=0 one variable)

classifier [object with learn() and w() methods] object

Returns

ranking [1d numpy array int] feature ranking. ranking[i] contains the feature index ranked in i-th position.

mlpy.**rfe_kfda**(*x*, *y*, *p*, *lmb*, *kernel*)
KFDA-RFE algorithm based on the Rayleigh coefficient proposed in [Louw06]. The algorithm works with only two classes.

Parameters

x: 2d array_like object (N,P) training data

y [1d array_like object integer (N)] class labels (only two classes)

p [float [0.0, 1.0]] percentage of features (upper rounded) to remove at each iteration (p=0 one variable)

lmb [float (>= 0.0)] regularization parameter

kernel [mlpy.Kernel object.] kernel.

Returns

ranking [1d numpy array int] feature ranking. ranking[i] contains the feature index ranked in i-th position.

CHAPTER

FOURTEEN

DIMENSIONALITY REDUCTION

14.1 Linear Discriminant Analysis (LDA)

class `mlpy.LDA`(*method='cov'*)

Linear Discriminant Analysis.

Initialization.

Parameters

method [str] 'cov' or 'fast'

`coeff`()

Returns the tranformation matrix (P,C-1), where C is the number of classes. Each column contains coefficients for one transformation vector.

`learn`(*x*, *y*)

Computes the transformation matrix. *x* is a matrix (N,P) and *y* is a vector containing the class labels. Each column of *x* represents a variable, while the rows contain observations.

`transform`(*t*)

Embed *t* (M,P) into the C-1 dimensional space. Returns a (M,C-1) matrix.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 4.5], [[1,1],[1,2]], 20  # 20 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,1],[1,2]], 30 # 30 samples of class 2
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = 2 * np.ones(n2, dtype=np.int)
>>> x = np.concatenate((x1, x2), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2))
>>> lda = mlpy.LDA()
>>> lda.learn(x, y) # compute the tranformation matrix
>>> z = lda.transform(x) # embedded x into the C-1 = 1 dimensional space
```

14.2 Spectral Regression Discriminant Analysis (SRDA)

class `mlpy.`**`SRDA`**(*alpha=0.001*)

Spectral Regression Discriminant Analysis.

Initialization.

Parameters

alpha [float (>=0)] regularization parameter

`coeff`()

Returns the tranformation matrix (P,C-1), where C is the number of classes. Each column contains coefficients for one transformation vector.

`learn`(*x*, *y*)

Computes the transformation matrix. *x* is a matrix (N,P) and *y* is a vector containing the class labels. Each column of *x* represents a variable, while the rows contain observations.

`transform`(*t*)

Embed t (M,P) into the C-1 dimensional space. Returns a (M,C-1) matrix.

14.3 Kernel Fisher Discriminant Analysis (KFDA)

class `mlpy.`**`KFDA`**(*lmb=0.001*, *kernel=None*)

Kernel Fisher Discriminant Analysis.

Initialization.

Parameters

lmb [float (>= 0.0)] regularization parameter

kernel [None or mlpy.Kernel object.] if kernel is None, K and Kt in .learn() and in .transform() methods must be precomputed kernel matricies, else K and Kt must be training (resp. test) data in input space.

`coeff`()

Returns the tranformation vector (N,1).

`learn`(*K*, *y*)

Computes the transformation vector.

Parameters

K: 2d array_like object precomputed training kernel matrix (if kernel=None); training data in input space (if kernel is a Kernel object)

y [1d array_like object integer (N)] class labels (only two classes)

`transform`(*Kt*)

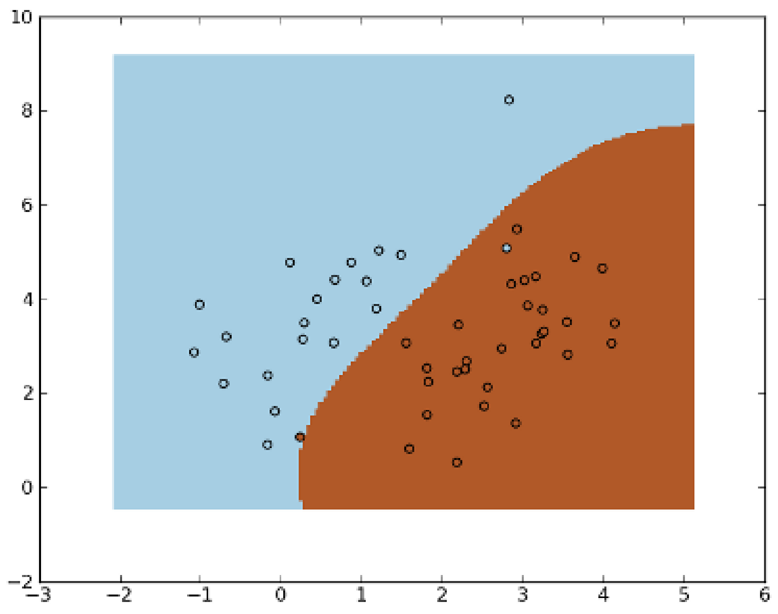
Embed Kt into the 1d kernel fisher space.

Parameters

Kt [1d or 2d array_like object] precomputed test kernel matrix. (if kernel=None); test data in input space (if kernel is a Kernel object).

Example - KNN in kernel fisher space:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean1, cov1, n1 = [1, 4.5], [[1,1],[1,2]], 20  # 20 samples of class 1
>>> x1 = np.random.multivariate_normal(mean1, cov1, n1)
>>> y1 = np.ones(n1, dtype=np.int)
>>> mean2, cov2, n2 = [2.5, 2.5], [[1,1],[1,2]], 30 # 30 samples of class 2
>>> x2 = np.random.multivariate_normal(mean2, cov2, n2)
>>> y2 = 2 * np.ones(n2, dtype=np.int)
>>> x = np.concatenate((x1, x2), axis=0) # concatenate the samples
>>> y = np.concatenate((y1, y2))
>>> K = mlpy.kernel_gaussian(x, x, sigma=3) # compute the kernel matrix
>>> kfda = mlpy.KFDA(lmb=0.01)
>>> kfda.learn(K, y) # compute the tranformation vector
>>> z = kfda.transform(K) # embedded x into the kernel fisher space
>>> knn = mlpy.KNN(k=5)
>>> knn.learn(z, y) # learn KNN in the kernel fisher space
>>> xmin, xmax = x[:,0].min()-1, x[:,0].max()+1
>>> ymin, ymax = x[:,1].min()-1, x[:,1].max()+1
>>> xx, yy = np.meshgrid(np.arange(xmin, xmax, 0.05), np.arange(ymin, ymax, 0.05))
>>> xt = np.c_[xx.ravel(), yy.ravel()]
>>> Kt = mlpy.kernel_gaussian(xt, x, sigma=3) # compute the kernel matrix Kt
>>> zt = kfda.transform(Kt) # embedded xt into the kernel fisher space
>>> yt = KNN.pred(zt).reshape(xx.shape) # perform the KNN prediction in the kernel fisher space
>>> fig = plt.figure(1)
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> plot1 = plt.pcolormesh(xx, yy, yt)
>>> plot2 = plt.scatter(x[:,0], x[:,1], c=y)
>>> plt.show()
```



14.4 Principal Component Analysis (PCA)

class `mlpy.PCA`(*method='svd'*, *whiten=False*)

Principal Component Analysis.

Initialization.

Parameters

method [str] method, 'svd' or 'cov'

whiten [bool] whitening. The eigenvectors will be scaled by eigenvalues**-(1/2)

`coeff`()

Returns the tranformation matrix (P,L), where L=min(N,P), sorted by decreasing eigenvalue. Each column contains coefficients for one principal component.

`coeff_inv`()

Returns the inverse of tranformation matrix (L,P), where L=min(N,P), sorted by decreasing eigenvalue.

`evals`()

Returns sorted eigenvalues (L), where L=min(N,P).

`learn`(*x*)

Compute the principal component coefficients. *x* is a matrix (N,P). Each column of *x* represents a variable, while the rows contain observations.

`transform`(*t*, *k=None*)

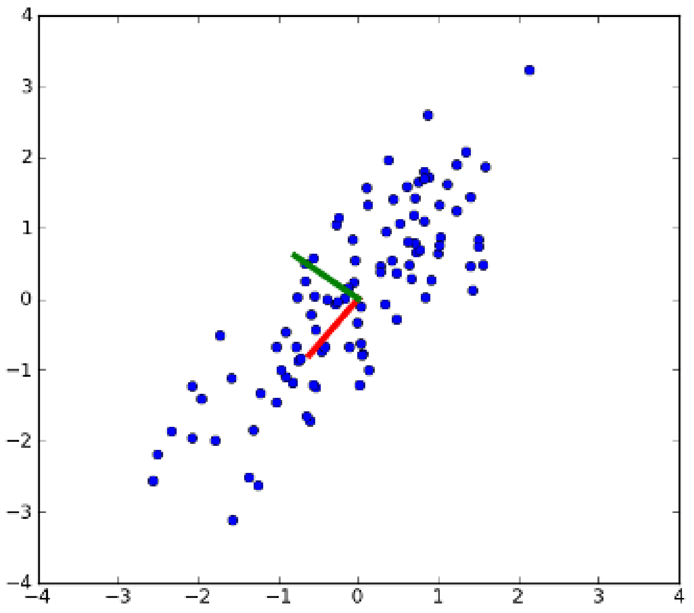
Embed *t* (M,P) into the k dimensional subspace. Returns a (M,K) matrix. If *k* =None will be set to min(N,P)

`transform_inv`(*z*)

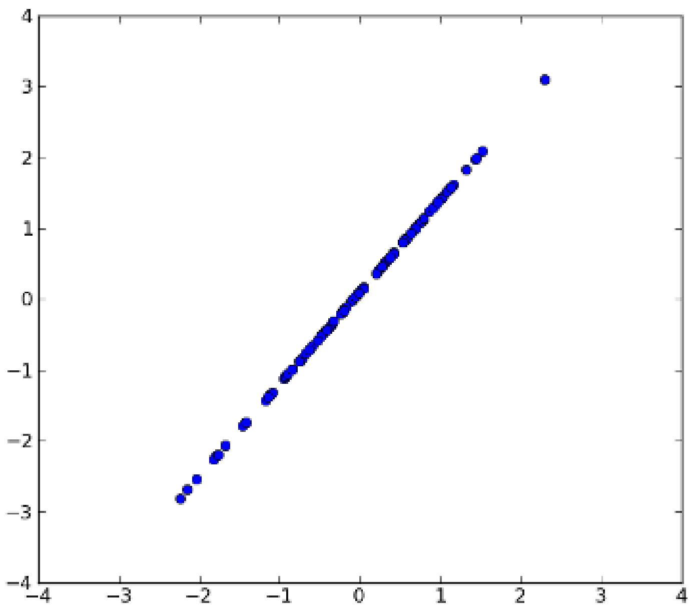
Transform data back to its original space, where *z* is a (M,K) matrix. Returns a (M,P) matrix.

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> mean, cov, n = [0, 0], [[1,1],[1,1.5]], 100
>>> x = np.random.multivariate_normal(mean, cov, n)
>>> pca.learn(x)
>>> coeff = pca.coeff()
>>> fig = plt.figure(1) # plot
>>> plot1 = plt.plot(x[:, 0], x[:, 1], 'o')
>>> plot2 = plt.plot([0,coeff[0, 0]], [0, coeff[1, 0]], linewidth=4, color='r') # first PC
>>> plot3 = plt.plot([0,coeff[0, 1]], [0, coeff[1, 1]], linewidth=4, color='g') # second PC
>>> xx = plt.xlim(-4, 4)
>>> yy = plt.ylim(-4, 4)
>>> plt.show()
```



```
>>> z = pca.transform(x, k=1) # transform x using the first PC
>>> xnew = pca.transform_inv(z) # transform data back to its original space
>>> fig2 = plt.figure(2) # plot
>>> plot1 = plt.plot(xnew[:, 0], xnew[:, 1], 'o')
>>> xx = plt.xlim(-4, 4)
>>> yy = plt.ylim(-4, 4)
>>> plt.show()
```



14.5 Fast Principal Component Analysis (PCAFast)

Fast PCA implementation described in [Sharma07].

class `mlpy.PCAFast(k=2, eps=0.01)`

Fast Principal Component Analysis.

Initialization.

Parameters

k [integer] the number of principal axes or eigenvectors required

eps [float (> 0)] tolerance error

`coeff()`

Returns the tranformation matrix (P,K) sorted by decreasing eigenvalue. Each column contains coefficients for one principal component.

`coeff_inv()`

Returns the inverse of tranformation matrix (K,P), sorted by decreasing eigenvalue.

`learn(x)`

Compute the firsts *k* principal component coefficients. *x* is a matrix (N,P). Each column of *x* represents a variable, while the rows contain observations.

`transform(t)`

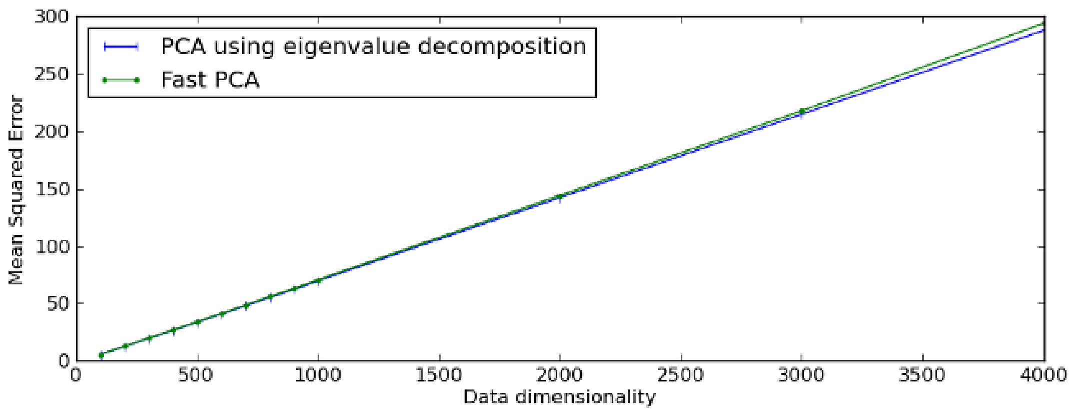
Embed t (M,P) into the *k* dimensional subspace. Returns a (M,K) matrix.

`transform_inv(z)`

Transform data back to its original space, where *z* is a (M,K) matrix. Returns a (M,P) matrix.

Example reproducing Figure 1 of [Sharma07]:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> h = 10 # dimension reduced to h=10
>>> n = 100 # number of samples
>>> d = np.array([100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000]) # number of
>>> mse_eig, mse_fast = np.zeros(len(d)), np.zeros(len(d))
>>> pca = mlpy.PCA(method='cov') # pca (eigenvalue decomposition)
>>> pca_fast=  mlpy.PCAFast(k=h) # fast pca
>>> for i in range(d.shape[0]):
...     x = np.random.rand(n, d[i])
...     pca.learn(x) # pca (eigenvalue decomposition)
...     y_eig = pca.transform(x, k=h) # reduced dimensional feature vectors
...     xhat_eig = pca.transform_inv(y_eig) # reconstructed vector
...     pca_fast.learn(x) # pca (eigenvalue decomposition)
...     y_fast = pca_fast.transform(x) # reduced dimensional feature vectors
...     xhat_fast = pca_fast.transform_inv(y_fast) # reconstructed vector
...     for j in range(n):
...         mse_eig[i] += np.sum((x[j] - xhat_eig[j])**2)
...         mse_fast[i] += np.sum((x[j] - xhat_fast[j])**2)
...     mse_eig[i] /= n
...     mse_fast[i] /= n
...
>>> fig = plt.figure(1)
>>> plot1 = plt.plot(d, mse_eig, '|-b', label="PCA using eigenvalue decomposition")
>>> plot2 = plt.plot(d, mse_fast, '.-g', label="Fast PCA")
>>> leg = plt.legend(loc = 'best')
>>> xl = plt.xlabel("Data dimensionality")
>>> yl = plt.ylabel("Mean Squared Error")
>>> plt.show()
```



14.6 Kernel Principal Component Analysis (KPCA)

class `mlpy.KPCA`(*kernel=None*)

Kernel Principal Component Analysis.

Initialization.

Parameters

kernel [None or mlpy.Kernel object.] if kernel is None, K and Kt in .learn() and in .transform() methods must be precomputed kernel matricies, else K and Kt must be training (resp. test) data in input space.

`coeff`()

Returns the tranformation matrix (N,N) sorted by decreasing eigenvalue.

`evals`()

Returns sorted eigenvalues (N).

`learn`(*K*)

Compute the kernel principal component coefficients.

Parameters

K: 2d array_like object precomputed training kernel matrix (if kernel=None); training data in input space (if kernel is a Kernel object)

`transform`(*Kt*, *k=None*)

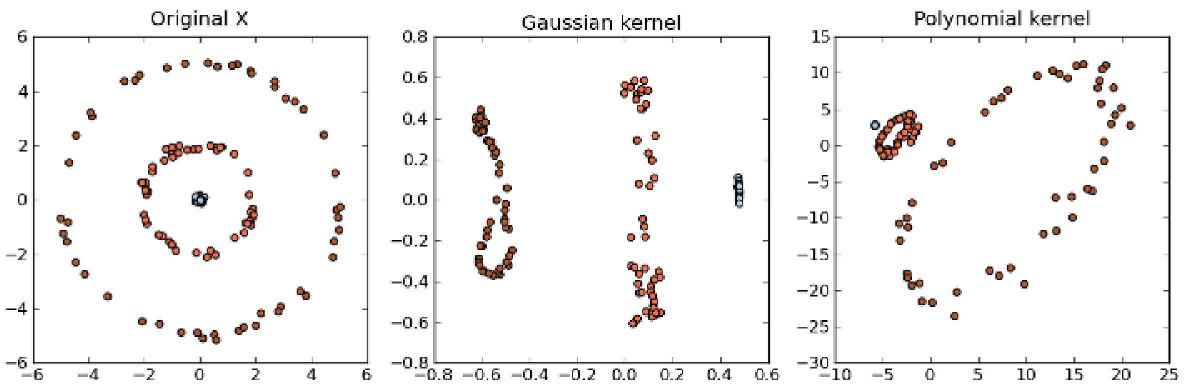
Embed Kt into the *k* dimensional subspace.

Parameters

Kt [1d or 2d array_like object] precomputed test kernel matrix. (if kernel=None); test data in input space (if kernel is a Kernel object).

Example:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> np.random.seed(0)
>>> np.random.seed(0)
>>> x = np.zeros((150, 2))
>>> y = np.empty(150, dtype=np.int)
>>> theta = np.random.normal(0, np.pi, 50)
>>> r = np.random.normal(0, 0.1, 50)
>>> x[0:50, 0] = r * np.cos(theta)
>>> x[0:50, 1] = r * np.sin(theta)
>>> y[0:50] = 0
>>> theta = np.random.normal(0, np.pi, 50)
>>> r = np.random.normal(2, 0.1, 50)
>>> x[50:100, 0] = r * np.cos(theta)
>>> x[50:100, 1] = r * np.sin(theta)
>>> y[50:100] = 1
>>> theta = np.random.normal(0, np.pi, 50)
>>> r = np.random.normal(5, 0.1, 50)
>>> x[100:150, 0] = r * np.cos(theta)
>>> x[100:150, 1] = r * np.sin(theta)
>>> y[100:150] = 2
>>> cmap = plt.set_cmap(plt.cm.Paired)
>>> gK = mlpy.kernel_gaussian(x, x, sigma=2) # gaussian kernel matrix
>>> pK = mlpy.kernel_polynomial(x, x, gamma=1.0, b=1.0, d=2.0) # polynomial kernel matrix
>>> gaussian_pca = mlpy.KPCA()
>>> polynomial_pca = mlpy.KPCA()
>>> gaussian_pca.learn(gK)
>>> polynomial_pca.learn(pK)
>>> gz = gaussian_pca.transform(gK, k=2)
>>> pz = polynomial_pca.transform(pK, k=2)
>>> fig = plt.figure(1)
>>> ax1 = plt.subplot(131)
>>> plot1 = plt.scatter(x[:, 0], x[:, 1], c=y)
>>> title1 = ax1.set_title('Original X')
>>> ax2 = plt.subplot(132)
>>> plot2 = plt.scatter(gz[:, 0], gz[:, 1], c=y)
>>> title2 = ax2.set_title('Gaussian kernel')
>>> ax3 = plt.subplot(133)
>>> plot3 = plt.scatter(pz[:, 0], pz[:, 1], c=y)
>>> title3 = ax3.set_title('Polynomial kernel')
>>> plt.show()
```



CHAPTER

FIFTEEN

CROSS VALIDATION

15.1 Leave-one-out and k-fold

`mlpy.cv_kfold`(*n*, *k*, *strat=None*, *seed=0*)

Returns train and test indexes for k-fold cross-validation.

Parameters

n [int (n > 1)] number of indexes

k [int (k > 1)] number of iterations (folds). The case $k = n$ is known as leave-one-out cross-validation.

strat [None or 1d array_like integer (of length *n*)] labels for stratification. If *strat* is not None returns 'stratified' k-fold CV indexes, where each subsample has roughly the same label proportions of *strat*.

seed [int] random seed

Returns

idx: list of tuples list of *k* tuples containing the train and test indexes

Example:

```
>>> import mlpy
>>> idx = mlpy.cv_kfold(n=12, k=3)
>>> for tr, ts in idx: tr, ts
...
(array([2, 8, 1, 7, 9, 3, 0, 5]), array([ 6, 11,  4, 10]))
(array([ 6, 11,  4, 10,  9,  3,  0,  5]), array([2, 8, 1, 7]))
(array([ 6, 11,  4, 10,  2,  8,  1,  7]), array([9, 3, 0, 5]))
>>> strat = [0,0,0,0,0,0,0,0,1,1,1,1]
>>> idx = mlpy.cv_kfold(12, k=4, strat=strat)
>>> for tr, ts in idx: tr, ts
...
(array([ 1,  7,  3,  0,  5,  4,  8, 10,  9]), array([ 6,  2, 11]))
(array([ 6,  2,  3,  0,  5,  4, 11, 10,  9]), array([1, 7, 8]))
(array([ 6,  2,  1,  7,  5,  4, 11,  8,  9]), array([ 3,  0, 10]))
(array([ 6,  2,  1,  7,  3,  0, 11,  8, 10]), array([5, 4, 9]))
```

15.2 Random Subsampling (*aka MonteCarlo*)

mlpy.**cv_random**(*n*, *k*, *p*, *strat=None*, *seed=0*)

Returns train and test indexes for random subsampling cross-validation. The proportion of the train/test indexes is not dependent on the number of iterations *k*.

Parameters

n [int (n > 1)] number of indexes

k [int (k > 0)] number of iterations (folds)

p [float (0 <= p <= 100)] percentage of indexes in test

strat [None or 1d array_like integer (of length *n*)] labels for stratification. If *strat* is not None returns 'stratified' random subsampling CV indexes, where each subsample has roughly the same label proportions of *strat*.

seed [int] random seed

Returns

idx: list of tuples list of *k* tuples containing the train and test indexes

Example:

```
>>> import mlpy
>>> ap = mlpy.cv_random(n=12, k=4, p=30)
>>> for tr, ts in ap: tr, ts
...
(array([ 6, 11,  4, 10,  2,  8,  1,  7,  9]), array([3, 0, 5]))
(array([ 5,  2,  3,  4,  9,  0, 11,  7,  6]), array([ 1, 10,  8]))
(array([ 6,  1, 10,  2,  7,  5, 11,  0,  3]), array([4, 9, 8]))
(array([2, 4, 8, 9, 5, 6, 1, 0, 7]), array([10, 11,  3]))
```

15.3 All Combinations

mlpy.**cv_all**(*n*, *p*)

Returns train and test indexes for all-combinations cross-validation.

Parameters

n [int (n > 1)] number of indexes

p [float (0 <= p <= 100)] percentage of indexes in test

Returns

idx [list of tuples] list of tuples containing the train and test indexes

Example

```
>>> import mlpy
>>> idx = mlpy.cv_all(n=4, p=50)
>>> for tr, ts in idx: tr, ts
...
(array([2, 3]), array([0, 1]))
(array([1, 3]), array([0, 2]))
(array([1, 2]), array([0, 3]))
(array([0, 3]), array([1, 2]))
(array([0, 2]), array([1, 3]))
```

```
(array([0, 1]), array([2, 3]))
>>> idx = mlpy.cv_all(a, 10) # ValueError: p must be >= 25.000
```

CHAPTER

SIXTEEN

METRICS

Compute metrics for assessing the performance of classification/regression models.

16.1 Classification

mlpy.**error**(*t*, *p*)

Error for binary and multiclass classification problems.

Parameters

t [1d array_like object integer] target values

p [1d array_like object integer] predicted values

Returns error : float, in range [0.0, 1.0]

mlpy.**accuracy**(*t*, *p*)

Accuracy for binary and multiclass classification problems.

Parameters

t [1d array_like object integer] target values

p [1d array_like object integer] predicted values

Returns accuracy : float, in range [0.0, 1.0]

Examples:

```
>>> import mlpy
>>> t = [3,2,3,3,3,1,1,1]
>>> p = [3,2,1,3,3,2,1,1]
>>> mlpy.error(t, p)
0.25
>>> mlpy.accuracy(t, p)
0.75
```

16.1.1 Binary Classification Only

The Confusion Matrix:

Total Samples (ts)	Actual Positives (ap)	Actual Negatives (an)
Predicted Positives (pp)	True Positives (tp)	False Positives (fp)
Predicted Negatives (pn)	False Negatives (fn)	True Negatives (tn)

`mlpy.`**`error_p`**(*t*, *p*)

Compute the positive error as:

error_p = fn / ap

Only binary classification problems with t[i] = -1/+1 are allowed.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object integer (-1/+1)] predicted values

Returns errorp : float, in range [0.0, 1.0]

`mlpy.`**`error_n`**(*t*, *p*)

Compute the negative error as:

error_n = fp / an

Only binary classification problems with t[i] = -1/+1 are allowed.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object integer (-1/+1)] predicted values

Returns errorp : float, in range [0.0, 1.0]

`mlpy.`**`sensitivity`**(*t*, *p*)

Sensitivity, computed as:

sensitivity = tp / ap

Only binary classification problems with t[i] = -1/+1 are allowed.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object integer (-1/+1)] predicted values

Returns sensitivity : float, in range [0.0, 1.0]

`mlpy.`**`specificity`**(*t*, *p*)

Specificity, computed as:

specificity = tn / an

Only binary classification problems with t[i] = -1/+1 are allowed.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object integer (-1/+1)] predicted values

Returns sensitivity : float, in range [0.0, 1.0]

`mlpy.`**`ppv`**(*t*, *p*)

Positive Predictive Value (PPV) computed as:

ppv = tp / pp

Only binary classification problems with t[i] = -1/+1 are allowed.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object integer (-1/+1)] predicted values

Returns PPV : float, in range [0.0, 1.0]

`mlpy.`**`npv`**(*t*, *p*)

Negative Predictive Value (NPV), computed as:

npv = tn / pn

Only binary classification problems with t[i] = -1/+1 are allowed.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object integer (-1/+1)] predicted values

Returns NPV : float, in range [0.0, 1.0]

`mlpy.`**`mcc`**(*t*, *p*)

Matthews Correlation Coefficient (MCC), computed as:

MCC = ((tp*tn)-(fp*fn)) / sqrt((tp+fn)*(tp+fp)*(tn+fn)*(tn+fp))

Only binary classification problems with t[i] = -1/+1 are allowed.

Returns a value between -1 and +1. A MCC of +1 represents a perfect prediction, 0 an average random prediction and -1 an inverse prediction. If any of the four sums in the denominator is zero, the denominator is set to one; this results in a Matthews Correlation Coefficient of zero, which can be shown to be the correct limiting value.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object integer (-1/+1)] predicted values

Returns MCC : float, in range [-1.0, 1.0]

`mlpy.`**`auc_wmw`**(*t*, *p*)

Compute the AUC by using the Wilcoxon-Mann-Whitney statistic. Only binary classification problems with t[i] = -1/+1 are allowed.

Parameters

t [1d array_like object integer (-1/+1)] target values

p [1d array_like object (negative/positive values)] predicted values

Returns AUC : float, in range [0.0, 1.0]

Examples:

```
>>> import mlpy
>>> t = [1, 1, 1,-1, 1,-1,-1,-1]
>>> p = [1,-1, 1, 1, 1,-1, 1,-1]
>>> mlpy.error_p(t, p)
0.25
>>> mlpy.error_n(t, p)
0.5
>>> mlpy.sensitivity(t, p)
0.75
>>> mlpy.specificity(t, p)
0.5
>>> mlpy.ppv(t, p)
0.59999999999999998
>>> mlpy.npv(t, p)
```

```
0.66666666666666663
>>> mlpy.mcc(t, p)
0.2581988897471611
>>> p = [2.3,-0.4, 1.6, 0.6, 3.2,-4.9, 1.3,-0.3]
>>> mlpy.auc_wmw(t, p)
0.8125
>>> p = [2.3,0.4, 1.6, -0.6, 3.2,-4.9, -1.3,-0.3]
>>> mlpy.auc_wmw(t, p)
1.0
```

16.2 Regression

mlpy.**mse**(*t*, *p*)

Mean Squared Error (MSE).

Parameters

t [1d array_like object] target values

p [1d array_like object] predicted values

Returns MSE : float

mlpy.**r2**(*t*, *p*)

Coefficient of determination (R^2) computed as 1 - (sserr/sstot), where *sserr* is the sum of squares of residuals and *sstot* is the total sum of squares.

Parameters

t [1d array_like object] target values

p [1d array_like object] predicted values

Returns R^2 : float

mlpy.**r2_corr**(*t*, *p*)

Coefficient of determination (R^2) computed as square of the correlation coefficient.

Parameters

t [1d array_like object] target values

p [1d array_like object] predicted values

Returns R^2 : float

Example:

```
>>> import mlpy
>>> t = [2.4,0.4,1.2,-0.2,3.3,-4.9,-1.1,-0.1]
>>> p = [2.3,0.4,1.6,-0.6,3.2,-4.9,-1.3,-0.3]
>>> mlpy.mse(t, p)
0.052499999999999998
```

CHAPTER

SEVENTEEN

A SET OF STATISTICAL FUNCTIONS

`mlpy.`**`bootstrap_ci`**(*x*, *B=1000*, *alpha=0.050000000000000003*, *seed=0*)

Computes the (1-alpha) Bootstrap confidence interval from empirical bootstrap distribution of sample mean.

The lower and upper confidence bounds are the (B*alpha/2)-th and B * (1-alpha/2)-th ordered means, respectively. For B = 1000 and alpha = 0.05 these are the 25th and 975th ordered means.

`mlpy.`**`quantile`**(*x*, *f*)

Returns a quantile value of *x*.

The quantile is determined by the *f*, a fraction between 0 and 1. For example, to compute the value of the 75th percentile *f* should have the value 0.75.

CANBERRA DISTANCES AND STABILITY INDICATOR OF RANKED LISTS

18.1 Canberra distance

mlpy.**canberra**(*x*, *y*)

Returns the Canberra distance between two P-vectors x and y: sum_i(abs(x_i - y_i) / (abs(x_i) + abs(y_i))).

18.2 Canberra Distance with Location Parameter

See [Jurman08].

mlpy.**canberra_location**(*x*, *y*, *k=None*)

Returns the Canberra distance between two position lists, *x* and *y*. A position list of length P contains the position (from 0 to P-1) of P elements. k is the location parameter, if k=None will be set to P.

The function computes:

$$\sum_i \frac{|\min\{x_i+1,k+1\}-\min\{y_i+1,k+1\}|}{\min\{x_i+1,k+1\}+\min\{y_i+1,k+1\}}$$

mlpy.**canberra_location_expected**(*p*, *k=None*)

Returns the expected value of the Canberra location distance, where *p* is the number of elements and *k* is the number of positions to consider.

18.3 Canberra Stability Indicator

See [Jurman08].

mlpy.**canberra_stability**(*x*, *k=None*)

Returns the Canberra stability indicator between N position lists, where *x* is an (N, P) matrix. A position list of length P contains the position (from 0 to P-1) of P elements. k is the location parameter, if k=None will be set to P. The lower the indicator value, the higher the stability of the lists.

The stability is computed by the mean distance of all the (N(N-1))/2 non trivial values of the distance matrix (computed by canberra_location()) scaled by the expected (average) value of the Canberra metric.

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([[2,4,1,3,0], [3,4,1,2,0], [2,4,3,0,1]])  # 3 position lists
>>> mlpy.canberra_stability(x, 3) # stability indicator
0.74862979571499755
```

CHAPTER

NINETEEN

BORDA COUNT

`mlpy.borda_count(x, k=None)`

Given N ranked ids lists of length P compute the number of extractions on top-k positions and the mean position for each id. Sort the element ids with decreasing number of extractions, and element ids with equal number of extractions will be sorted with increasing mean positions.

Parameters

x [2d array_like object integer (N, P)] ranked ids lists. For each list ids must be unique in [0, P-1].

k [None or integer] compute borda on top-k position (None -> k = P)

Returns

borda [1d numpy array objects] sorted-ids, number of extractions, mean positions

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = [[2,4,1,3,0], # first ranked list
...      [3,4,1,2,0], # second ranked list
...      [2,4,3,0,1], # third ranked list
...      [0,1,4,2,3]] # fourth ranked list
>>> mlpy.borda_count(x=x, k=3)
(array([4, 1, 2, 3, 0]), array([4, 3, 2, 2, 1]), array([ 1.25      ,  1.66666667,  0.        ,
```

- Id 4 is in the first position with 4 extractions and mean position 1.25.
- Id 1 is in the first position with 3 extractions and mean position 1.67.
- ...

CHAPTER

TWENTY

FIND PEAKS

`mlpy.findpeaks_dist()`

Find peaks. With *mindist* parameter the algorithm ignore small peaks that occur in the neighborhood of a larger peak.

Parameters

x [1d array_like object] input data

mindist [integer (>=2)] minimum peak distance (minimum separation between peaks)

Returns

idx [1d numpy array int] peaks indexes

Example:

```
>>> import mlpy
>>> x = [6,2,2,1,3,4,1,3,1,1,1,6,2,2,7,1]
>>> mlpy.findpeaks_dist(x, mindist=3)
array([ 0,  5, 11, 14])
```

`mlpy.findpeaks_win()`

Find peaks with a sliding window of width *span*.

Parameters

x [1d array_like object] input data

span [odd integer (>=3)] span

Returns

idx [1d numpy array int] peaks indexes

Example:

```
>>> import mlpy
>>> x = [6,2,2,1,3,4,1,3,1,1,1,6,2,2,7,1]
>>> mlpy.findpeaks_win(x, span=3)
array([ 0,  5,  7, 11, 14])
```

CHAPTER

TWENTYONE

DYNAMIC TIME WARPING (DTW)

21.1 Standard DTW

mlpy.**dtw_std**(*x*, *y*, *dist_only=True*)

Standard DTW as described in [Muller07], using the Euclidean distance (absolute value of the difference) or squared Euclidean distance (as in [Keogh01]) as local cost measure.

Parameters

x [1d array_like object (N)] first sequence

y [1d array_like object (M)] second sequence

dist_only [bool] compute only the distance

squared [bool] squared Euclidean distance

Returns

dist [float] unnormalized minimum-distance warp path between sequences

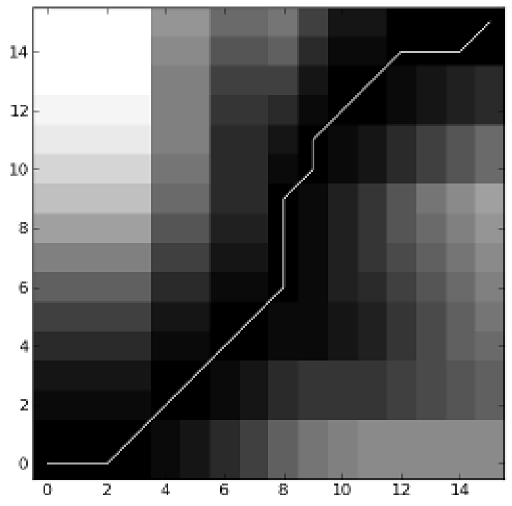
cost [2d numpy array (N,M) [if dist_only=False]] accumulated cost matrix

path [tuple of two 1d numpy array (path_x, path_y) [if dist_only=False]] warp path

Example

Reproducing the Fig. 2 example in [Salvador04].

```
>>> import mlpy
>>> import matplotlib.pyplot as plt
>>> import matplotlib.cm as cm
>>> x = [0,0,0,0,1,1,2,2,3,2,1,1,0,0,0,0]
>>> y = [0,0,1,1,2,2,3,3,3,3,2,2,1,1,0,0]
>>> dist, cost, path = mlpy.dtw_std(x, y, dist_only=False)
>>> dist
0.0
>>> fig = plt.figure(1)
>>> ax = fig.add_subplot(111)
>>> plot1 = plt.imshow(cost.T, origin='lower', cmap=cm.gray, interpolation='nearest')
>>> plot2 = plt.plot(path[0], path[1], 'w')
>>> xlim = ax.set_xlim((-0.5, cost.shape[0]-0.5))
>>> ylim = ax.set_ylim((-0.5, cost.shape[1]-0.5))
>>> plt.show()
```



21.2 Subsequence DTW

`mlpy.dtw_subsequence(x, y)`

Subsequence DTW as described in [Muller07], assuming that the length of *y* is much larger than the length of *x* and using the Manhattan distance (absolute value of the difference) as local cost measure.

Returns the subsequence of *y* that are close to *x* with respect to the minimum DTW distance.

Parameters

x [1d array_like object (N)] first sequence

y [1d array_like object (M)] second sequence

Returns

dist [float] unnormalized minimum-distance warp path between x and the subsequence of y

cost [2d numpy array (N,M) [if dist_only=False]] complete accumulated cost matrix

path [tuple of two 1d numpy array (path_x, path_y)] warp path

CHAPTER

TWENTYTWO

LONGEST COMMON SUBSEQUENCE (LCS)

22.1 Standard LCS

mlpy.**lcs_std**(*x*, *y*)

Standard Longest Common Subsequence (LCS) algorithm as described in [Cormen01].

The elements of sequences must be coded as integers.

Parameters

x [1d integer array_like object (N)] first sequence

y [1d integer array_like object (M)] second sequence

Returns

length [integer] length of the LCS of x and y

path [tuple of two 1d numpy array (path_x, path_y)] path of the LCS

Example

Reproducing the example in figure 15.6 of [Cormen01], where sequence X = (A, B, C, B, D, A, B) and Y = (B, D, C, A, B, A).

```
>>> import mlpy
>>> x = [0,1,2,1,3,0,1] # (A, B, C, B, D, A, B)
>>> y = [1,3,2,0,1,0] # (B, D, C, A, B, A)
>>> length, path = mlpy.lcs_std(x, y)
>>> length
4
>>> path
(array([1, 2, 3, 5]), array([0, 2, 4, 5]))
```

22.2 LCS for real series

mlpy.**lcs_real**(*x*, *y*, *eps*, *delta*)

Longest Common Subsequence (LCS) for series composed by real numbers as described in [Vlachos02].

Parameters

x [1d integer array_like object (N)] first sequence

y [1d integer array_like object (M)] second sequence

eps [float (>=0)] matching threshold

delta [int (>=0)] controls how far in time we can go in order to match a given point from one series to a point in another series

Returns

length [integer] length of the LCS of x and y

path [tuple of two 1d numpy array (path_x, path_y)] path of the LCS

CHAPTER

TWENTYTHREE

MLPY.WAVELET - WAVELET TRANSFORM

23.1 Padding

`mlpy.wavelet.pad(x, method='reflection')`

Pad to bring the total length N up to the next-higher power of two.

Parameters

x [1d array_like object] data

method [string ('reflection', 'periodic', 'zeros')] method

Returns

xp, orig [1d numpy array, 1d numpy array bool] padded version of *x* and a boolean array with value True where xp contains the original data

23.2 Discrete Wavelet Transform

Discrete Wavelet Transform based on the GSL DWT [Gsldwt].

For the forward transform, the output is the discrete wavelet transform $f_i \rightarrow w_{j,k}$ in a packed triangular storage layout, where j is the index of the level $j = 0 \dots J-1$ and k is the index of the coefficient within each level, $k = 0 \dots (2^j)-1$. The total number of levels is $J = \log_2(n)$. The output data has the following form,

$$(s_{-1,0}, d_{0,0}, d_{1,0}, d_{1,1}, d_{2,0}, \dots, d_{j,k}, ..., d_{J-1,2^{J-1}-1})$$

where the first element is the smoothing coefficient $s_{-1,0}$, followed by the detail coefficients $d_{j,k}$ for each level j. The backward transform inverts these coefficients to obtain the original data.

Note: from GSL online manual (http://www.gnu.org/software/gsl/manual/)

`mlpy.wavelet.dwt(x, wf, k, centered=False)`

Discrete Wavelet Tranform

Parameters

x [1d array_like object (the length is restricted to powers of two)] data

wf [string ('d': daubechies, 'h': haar, 'b': bspline)] wavelet family

k [integer] member of the wavelet family

- daubechies : k = 4, 6, ..., 20 with k even
- haar : the only valid choice of k is k = 2
- bspline : k = 103, 105, 202, 204, 206, 208, 301, 303, 305 307, 309

centered [bool] align the coefficients of the various sub-bands on edges. Thus the resulting visualization of the coefficients of the wavelet transform in the phase plane is easier to understand.

Returns

X [1d numpy array] discrete wavelet transformed data

Example

```
>>> import numpy as np
>>> import mlpy.wavelet as wave
>>> x = np.array([1,2,3,4,3,2,1,0])
>>> wave.dwt(x=x, wf='d', k=6)
array([ 5.65685425,  3.41458985,  0.29185347, -0.29185347, -0.28310081,
       -0.07045258,  0.28310081,  0.07045258])
```

`mlpy.wavelet.idwt` (*X*, *wf*, *k*, *centered=False*)

Inverse Discrete Wavelet Tranform

Parameters

X [1d array_like object] discrete wavelet transformed data

wf [string ('d': daubechies, 'h': haar, 'b': bspline)] wavelet type

k [integer] member of the wavelet family

- daubechies : k = 4, 6, ..., 20 with k even
- haar : the only valid choice of k is k = 2
- bspline : k = 103, 105, 202, 204, 206, 208, 301, 303, 305 307, 309

centered [bool] if the coefficients are aligned

Returns

x [1d numpy array] data

Example:

```
>>> import numpy as np
>>> import mlpy.wavelet as wave
>>> X = np.array([ 5.65685425,  3.41458985,  0.29185347, -0.29185347, -0.28310081,
...               -0.07045258,  0.28310081,  0.07045258])
>>> wave.idwt(X=X, wf='d', k=6)
array([  1.00000000e+00,   2.00000000e+00,   3.00000000e+00,
         4.00000000e+00,   3.00000000e+00,   2.00000000e+00,
         1.00000000e+00,  -3.53954610e-09])
```

23.3 Undecimated Wavelet Transform

Undecimated Wavelet Transform (also known as stationary wavelet transform, redundant wavelet transform, translation invariant wavelet transform, shift invariant wavelet transform or Maximal overlap wavelet transform) based on the "wavelets" R package.

`mlpy.wavelet.uwt` (*x*, *wf*, *k*, *levels=0*)

Undecimated Wavelet Tranform

Parameters

x [1d array_like object (the length is restricted to powers of two)] data

wf [string ('d': daubechies, 'h': haar, 'b': bspline)] wavelet family

k [int] member of the wavelet family

- daubechies: k = 4, 6, ..., 20 with k even
- haar: the only valid choice of k is k = 2
- bspline: k = 103, 105, 202, 204, 206, 208, 301, 303, 305 307, 309

levels [int] level of the decomposition (J). If levels = 0 this is the value J such that the length of X is at least as great as the length of the level J wavelet filter, but less than the length of the level J+1 wavelet filter. Thus, j <= log_2((n-1)/(l-1)+1), where n is the length of x

Returns

X [2d numpy array (2J * len(x))] misaligned scaling and wavelet coefficients:

```
[[wavelet coefficients W_1]
 [wavelet coefficients W_2]
              :
 [wavelet coefficients W_J]
 [scaling coefficients V_1]
 [scaling coefficients V_2]
             :
 [scaling coefficients V_J]]
```

`mlpy.wavelet.iuwt` (*X*, *wf*, *k*)

Inverse Undecimated Wavelet Tranform

Parameters

X [2d array_like object (the length is restricted to powers of two)] misaligned scaling and wavelet coefficients

wf [string ('d': daubechies, 'h': haar, 'b': bspline)] wavelet family

k [int] member of the wavelet family

- daubechies: k = 4, 6, ..., 20 with k even
- haar: the only valid choice of k is k = 2
- bspline: k = 103, 105, 202, 204, 206, 208, 301, 303, 305 307, 309

Returns

x [1d numpy array] data

`mlpy.wavelet.uwt_align_h2` (*X*, *inverse=False*)

UWT h2 coefficients aligment.

If inverse = True performs the misalignment for a correct reconstruction.

`mlpy.wavelet.uwt_align_d4` (*X*, *inverse=False*)

UWT d4 coefficients aligment.

If inverse = True performs the misalignment for a correct reconstruction.

23.4 Continuous Wavelet Transform

Continuous Wavelet Transform based on [Torrence98].

`mlpy.wavelet.`**`cwt`**(*x*, *dt*, *scales*, *wf='dog'*, *p=2*)

Continuous Wavelet Tranform.

Parameters

x [1d array_like object] data

dt [float] time step

scales [1d array_like object] scales

wf [string ('morlet', 'paul', 'dog')] wavelet function

p [float] wavelet function parameter ('omega0' for morlet, 'm' for paul and dog)

Returns

X [2d numpy array] transformed data

`mlpy.wavelet.`**`icwt`**(*X*, *dt*, *scales*, *wf='dog'*, *p=2*)

Inverse Continuous Wavelet Tranform. The reconstruction factor is not applied.

Parameters

X [2d array_like object] transformed data

dt [float] time step

scales [1d array_like object] scales

wf [string ('morlet', 'paul', 'dog')] wavelet function

p [float] wavelet function parameter

Returns

x [1d numpy array] data

`mlpy.wavelet.`**`autoscales`**(*N*, *dt*, *dj*, *wf*, *p*)

Compute scales as fractional power of two.

Parameters

N [integer] number of data samples

dt [float] time step

dj [float] scale resolution (smaller values of dj give finer resolution)

wf [string] wavelet function ('morlet', 'paul', 'dog')

p [float] omega0 ('morlet') or order ('paul', 'dog')

Returns

scales [1d numpy array] scales

`mlpy.wavelet.`**`fourier_from_scales`**(*scales*, *wf*, *p*)

Compute the equivalent fourier period from scales.

Parameters

scales [list or 1d numpy array] scales

wf [string ('morlet', 'paul', 'dog')] wavelet function

p [float] wavelet function parameter ('omega0' for morlet, 'm' for paul and dog)

Returns fourier wavelengths

`mlpy.wavelet.`**`scales_from_fourier`**(*f*, *wf*, *p*)

Compute scales from fourier period.

Parameters

f [list or 1d numpy array] fourier wavelengths

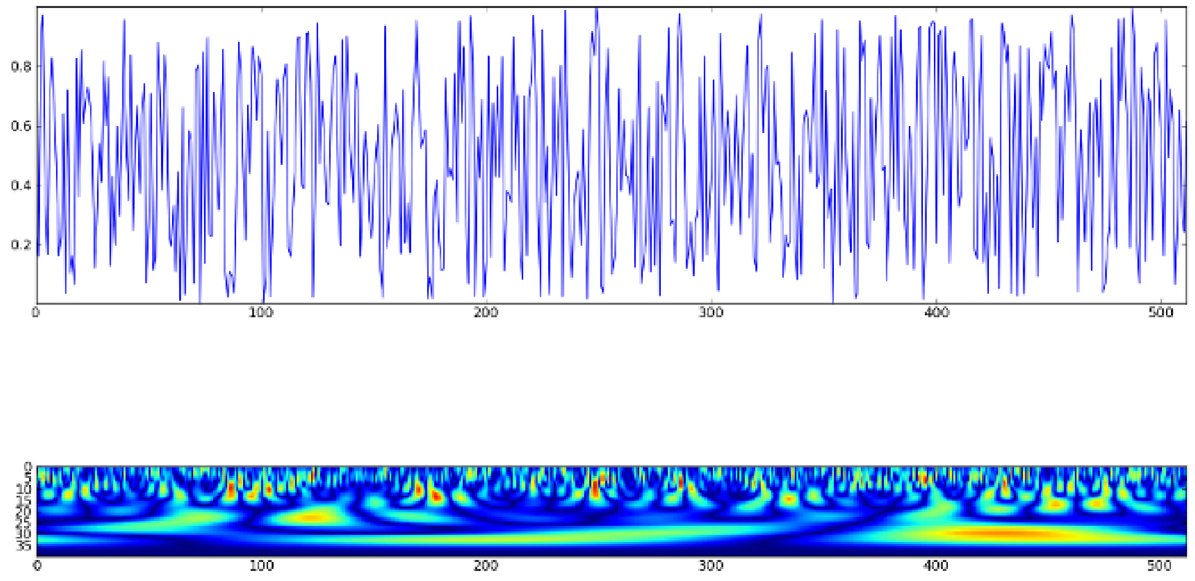
wf [string ('morlet', 'paul', 'dog')] wavelet function

p [float] wavelet function parameter ('omega0' for morlet, 'm' for paul and dog)

Returns scales

Example (requires matplotlib)

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy.wavelet as wave
>>> x = np.random.sample(512)
>>> scales = wave.autoscales(N=x.shape[0], dt=1, dj=0.25, wf='dog', p=2)
>>> X = wave.cwt(x=x, dt=1, scales=scales, wf='dog', p=2)
>>> fig = plt.figure(1)
>>> ax1 = plt.subplot(2,1,1)
>>> p1 = ax1.plot(x)
>>> ax1.autoscale_view(tight=True)
>>> ax2 = plt.subplot(2,1,2)
>>> p2 = ax2.imshow(np.abs(X), interpolation='nearest')
>>> plt.show()
```



CHAPTER

TWENTYFOUR

SHORT GUIDE TO CENTERING AND SCALING

Centering:

1d array	`>>> x - np.mean(x)`
2d array along rows	`>>> x - np.mean(x, axis=1).reshape(-1, 1)`
2d array along cols	`>>> x - np.mean(x, axis=0)`

Unit length scaling (normalization). Elements are scaled to have and unit length ($\sum_{i=1}^{n} x_i^2 = 1$):

1d array	`>>> x / np.sqrt(np.sum((x)**2))`
2d array along rows	`>>> x / np.sqrt(np.sum((x)**2, axis=1)).reshape(-1`
2d array along cols	`>>> x / np.sqrt(np.sum((x)**2, axis=0))`

Standardization. Elements are scaled to have unit standard deviation. The standard deviation is computed using $n-1$ instead of n (Bessel's correction).

1d array	`>>> x / np.std(x, ddof=1) # ddof=1: Bessel's corre`
2d array along rows	`>>> x / np.std(x, axis=1, ddof=1).reshape(-1, 1)`
2d array along cols	`>>> x / np.std(x, axis=0, ddof=1)`

CHAPTER

TWENTYFIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[Taylor04] J Shawe-Taylor and N Cristianini. Kernel Methods for Pattern Analysis.

[DeMol08] C De Mol, E De Vito and L Rosasco. Elastic Net Regularization in Learning Theory,CBCL paper #273/ CSAILTechnical Report #TR-2008-046, Massachusetts Institute of Technology, Cambridge, MA, July 24, 2008. arXiv:0807.3423 (to appear in the Journal of Complexity).

[Efron04] Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani. Least Angle Regression. Annals of Statistics, 2004, volume 32, pages 407-499.

[Hoerl70] A E Hoerl and R W Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. Technometrics. Vol. 12, No. 1, 1970, pp. 55–67.

[Golub99] T R Golub et al. Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring. Science, 1999.

[Hastie09] T Hastie, R Tibshirani, J Friedman. The Elements of Statistical Learning. Second Edition.

[Scolkopf98] Bernhard Scholkopf, Alexander Smola, and Klaus-Robert Muller. Nonlinear component analysis as a kernel eigenvalue problem. Neural Computation, 10(5):1299–1319, July 1998.

[Gavin03] Gavin C. et al. Efficient Cross-Validation of Kernel Fisher Discriminant Classifers. ESANN'2003 proceedings - European Symposium on Artificial Neural Networks, 2003.

[LIBSVM] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

[Cristianini] N Cristianini and J Shawe-Taylor. An introduction to support vector machines. Cambridge University Press.

[Vapnik95] V Vapnik. The Nature of Statistical Learning Theory. Springer-Verlag, 1995.

[Friess] Friess, Cristianini, Campbell. The Kernel-Adatron Algorithm: a Fast and Simple Learning Procedure for Support Vector Machines.

[Kecman03] Kecman, Vogt, Huang. On the Equality of Kernel AdaTron and Sequential Minimal Optimization in Classification and Regression Tasks and Alike Algorithms for Kernel Machines. ESANN'2003 proceedings - European Symposium on Artificial Neural Networks, ISBN 2-930307-03-X, pp. 215-222.

[LIBLINEAR] Machine Learning Group at National Taiwan University. http://www.csie.ntu.edu.tw/~cjlin/liblinear/

[Amap] amap: Another Multidimensional Analysis Package, http://cran.r-project.org/web/packages/amap/index.html

[fastcluster] Fast hierarchical clustering routines for R and Python, http://cran.r-project.org/web/packages/fastcluster/index.html

[Sun07] Yijun Sun. Iterative RELIEF for Feature Weighting: Algorithms, Theories, and Applications. IEEE Trans. Pattern Anal. Mach. Intell. 29(6): 1035-1051, 2007.

[Guyon02] I Guyon, J Weston, S Barnhill and V Vapnik. Gene Selection for Cancer Classification using Support Vector Machines. Machine Learning, 2002.

[Louw06] N Louw and S J Steel. Variable selection in kernel Fisher discriminant analysis by means of recursive feature elimination. Journal Computational Statistics & Data Analysis, 2006.

[Cai08] D Cai, X He, J Han. SRDA: An Efficient Algorithm for Large-Scale Discriminant Analysis. Knowledge and Data Engineering, IEEE Transactions on Volume 20, Issue 1, Jan. 2008 Page(s):1 - 12.

[Sharma07] A Sharma, K K Paliwal. Fast principal component analysis using fixed-point algorithm. Pattern Recognition Letters 28 (2007) 1151–1155.

[Mika99] S Mika et al. Fisher Discriminant Analysis with Kernels. Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop.

[Scholkopf96] B Scholkopf, A Smola, KR Muller. Nonlinear Component Analysis as a Kernel EigenValue Problem

[Jurman08] G Jurman, S Riccadonna, R Visintainer and C Furlanello. Algebraic stability indicators for ranked lists in molecular profiling. Bioinformatics Vol. 24 no. 2 2008, pages 258–264.

[Muller07] M Muller. Information Retrieval for Music and Motion. Springer, 2007.

[Keogh01] E J Keogh, M J Pazzani. Derivative Dynamic Time Warping. In First SIAM International Conference on Data Mining, 2001.

[Salvador04] S Salvador and P Chan. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. 3rd Wkshp. on Mining Temporal and Sequential Data, ACM KDD ‘04, 2004.

[Cormen01] H Cormen et al.. Introduction to Algorithms, Second Edition. The MIT Press, 2001.

[Vlachos02] M Vlachos et al.. Discovering Similar Multidimensional Trajectories. In Proceedings of the 18th international conference on data engineering, 2002

[Torrence98] C Torrence and G P Compo. Practical Guide to Wavelet Analysis

[Gsldwt] Gnu Scientific Library, http://www.gnu.org/software/gsl/

PYTHON MODULE INDEX

INDEX