mlpy Documentation

Release 2.2.0

Davide Albanese, Giuseppe Jurman, Roberto Visintainer

August 06, 2017

CONTENTS

Homepage: https://mlpy.fbk.eu *Module author: mlpy Developers <albanese@fbk.eu>*

Section author: Davide Albanese <albanese@fbk.eu>

`mlpy` is a high-performance Python package for predictive modeling. It makes extensive use of NumPy (http://scipy.org) to provide fast N-dimensional array manipulation and easy integration of C code. `mlpy` provides high level procedures that support, with few lines of code, the design of rich Data Analysis Protocols (DAPs) for preprocessing, clustering, predictive classification and feature selection. Methods are available for feature weighting and ranking, data resampling, error evaluation and experiment landscaping. The package includes tools to measure stability in sets of ranked feature lists.

`mlpy` is a project of the MPBA Research Unit at FBK, the Bruno Kessler Foundation in Trento, Italy (http://mpba.fbk.eu).

CHAPTER

ONE

TUTORIAL

1.1 A Simple Example

In this example the performance of SVM classifier is evaluated in a stratified k-fold resampling schema.

First, import NumPy and mlpy modules:

```
>>> import numpy as np
>>> import mlpy
```

Then, load a data file (*data.dat*) containing 30 samples described by 100 features (x) and labels (y):

```
>>> x, y = mlpy.data_fromfile('data.dat') # import data file
>>> x.shape
(30, 100)
```

Initialize SVM classifier, specifying kernel type (*linear*) and regularization parameter (C):

```
>>> classifier = mlpy.Svm(kernel = 'linear', C = 1.0)  # initialize the svm classifier
```

Define a stratified 10-fold resampling schema, where *idx* contains the sample indexes (list of train/test pairs):

```
>>> idx = mlpy.kfoldS(cl = y, sets = 10)
```

Actually build train and test data. Train the model on *xtr* and test it on *xts*. The performance is evaluated computing the average prediction error:

```
>>> pred_err = 0.0
>>> for idxtr, idxts in idx:
...     xtr, xts = x[idxtr], x[idxts]       # build training data
...     ytr, yts = y[idxtr], y[idxts]       # build test data
...     ret = classifier.compute(xtr, ytr)  # compute the model
...     pred = classifier.predict(xts)      # test the model on test data
...     pred_err += mlpy.err(yts, pred)          # compute the prediction error
>>> av_pred_err = pred_err / len(idx)       # compute the average prediction error
>>> av_pred_err
0.17499999999999999
```

CHAPTER

TWO

WAVELET TRANSFORM

2.1 Extend data

This function should be used in `dwt()` and `uwt()` to extend the length of data to power of two. `cwt()` use it as internal function.

`mlpy.`**`extend`**(*x*, *method='reflection'*, *length='powerof2'*)

Extend the 1D numpy array x beyond its original length.

Parameters

x [1d ndarray] data

method [string ('reflection', 'periodic', 'zeros')] indicates which extension method to use

length [string ('powerof2', 'double')] indicates how to determinate the length of the extended data

Returns

xext [1d ndarray] extended version of x

Example:

```
>>> import numpy as np
>>> import mlpy
>>> a = np.array([1,2,3,4,5])
>>> mlpy.extend(a, method='periodic', length='powerof2')
array([1, 2, 3, 4, 5, 1, 2, 3])
```

New in version 2.0.6.

2.2 Discrete Wavelet Transform

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

`mlpy.`**`dwt`**(*x*, *wf*, *k*)

Discrete Wavelet Tranform

Parameters

x [1d ndarray float (the length is restricted to powers of two)] 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

Returns

X [1d ndarray float] discrete wavelet transformed data

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([1,2,3,4,3,2,1,0])
>>> mlpy.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.**idwt**(*X*, *wf*, *k*)

Inverse Discrete Wavelet Tranform

Parameters

X [1d ndarray float] 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

Returns

x [1d ndarray float] data

Example:

```
>>> import numpy as np
>>> import mlpy
>>> X = np.array([ 5.65685425,  3.41458985,  0.29185347, -0.29185347, -0.28310081,
...                -0.07045258,  0.28310081,  0.07045258])
>>> mlpy.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])
```

2.3 Undecimated Wavelet Transform

Undecimated Wavelet Transform based on the "wavelets" R package.

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

Undecimated Wavelet Tranform

Parameters

x [1d ndarray float (the length is restricted to powers of two)] 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

levels [integer] 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 ndarray float (2J x len(x))] undecimated wavelet transformed data

Data:

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

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([1,2,3,4,3,2,1,0])
>>> mlpy.uwt(x=x, wf='d', k=6, levels=0)
array([[ 0.0498175 ,  0.22046721,  0.2001825 , -0.47046721, -0.0498175 ,
        -0.22046721, -0.2001825 ,  0.47046721],
       [ 0.28786838,  0.8994525 ,  2.16140162,  3.23241633,  3.71213162,
         3.1005475 ,  1.83859838,  0.76758367]])
```

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

Inverse Undecimated Wavelet Tranform

Parameters

X [2d ndarray float] undecimated 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

Returns

x [1d ndarray float] data

Example:

```
>>> import numpy as np
>>> import mlpy
>>> X = np.array([[ 0.0498175 ,  0.22046721,  0.2001825 , -0.47046721, -0.0498175,
...                -0.22046721, -0.2001825 ,  0.47046721],
...               [ 0.28786838,  0.8994525 ,  2.16140162,  3.23241633,  3.
↪71213162,
...                 3.1005475 ,  1.83859838,  0.76758367]])
>>> mlpy.iuwt(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,   2.29246158e-09])
```

New in version 2.0.2.

2.4 Continuous Wavelet Transform

Continuous Wavelet Transform based on *[Torrence98]*.

`mlpy.cwt`(*x*, *dt*, *dj*, *wf='dog'*, *p=2*, *extmethod='none'*, *extlength='powerof2'*)

Continuous Wavelet Tranform.

Parameters

x [1d ndarray float] data

dt [float] time step

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

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

p [float] wavelet function parameter

extmethod [string ('none', 'reflection', 'periodic', 'zeros')] indicates which extension method to use

extlength [string ('powerof2', 'double')] indicates how to determinate the length of the extended data

Returns

(X, scales) [(2d ndarray complex, 1d ndarray float)] transformed data, scales

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([1,2,3,4,3,2,1,0])
>>> mlpy.cwt(x=x, dt=1, dj=2, wf='dog', p=2)
(array([[ -4.66713159e-02 -6.66133815e-16j,
          -3.05311332e-16 +2.77555756e-16j,
           4.66713159e-02 +1.38777878e-16j,
           6.94959463e-01 -8.60422844e-16j,
           4.66713159e-02 +6.66133815e-16j,
           3.05311332e-16 -2.77555756e-16j,
          -4.66713159e-02 -1.38777878e-16j,
          -6.94959463e-01 +8.60422844e-16j],
        [ -2.66685280e+00 +2.44249065e-15j,
          -1.77635684e-15 -4.44089210e-16j,
           2.66685280e+00 -3.10862447e-15j,
```

```
          3.77202823e+00 -8.88178420e-16j,
          2.66685280e+00 -2.44249065e-15j,
          1.77635684e-15 +4.44089210e-16j,
         -2.66685280e+00 +3.10862447e-15j,
         -3.77202823e+00 +8.88178420e-16j]]), array([ 0.50329212,  2.01316848]))
```

mlpy.**icwt**(*X*, *dt*, *dj*, *wf='dog'*, *p=2*, *recf=True*)

Inverse Continuous Wavelet Tranform.

Parameters

X [2d ndarray complex] transformed data

dt [float] time step

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

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

p [float] wavelet function parameter

- morlet : 2, 4, 6
- paul : 2, 4, 6
- dog : 2, 6, 10

recf [bool] use the reconstruction factor ($C_\delta \Psi_0(0)$)

Returns

x [1d ndarray float] data

Example:

```
>>> import numpy as np
>>> import mlpy
>>> X = np.array([[ -4.66713159e-02 -6.66133815e-16j,
...                 -3.05311332e-16 +2.77555756e-16j,
...                  4.66713159e-02 +1.38777878e-16j,
...                  6.94959463e-01 -8.60422844e-16j,
...                  4.66713159e-02 +6.66133815e-16j,
...                  3.05311332e-16 -2.77555756e-16j,
...                 -4.66713159e-02 -1.38777878e-16j,
...                 -6.94959463e-01 +8.60422844e-16j],
...               [ -2.66685280e+00 +2.44249065e-15j,
...                 -1.77635684e-15 -4.44089210e-16j,
...                  2.66685280e+00 -3.10862447e-15j,
...                  3.77202823e+00 -8.88178420e-16j,
...                  2.66685280e+00 -2.44249065e-15j,
...                  1.77635684e-15 +4.44089210e-16j,
...                 -2.66685280e+00 +3.10862447e-15j,
...                 -3.77202823e+00 +8.88178420e-16j]])
>>> mlpy.icwt(X=X, dt=1, dj=2, wf='dog', p=2)
array([ -1.24078928e+00,  -1.07301771e-15,   1.24078928e+00,
         2.32044753e+00,   1.24078928e+00,   1.07301771e-15,
        -1.24078928e+00,  -2.32044753e+00])
```

2.4.1 Other functions

See *[Torrence98]*.

mlpy.**angularfreq**(*N*, *dt*)
Compute angular frequencies.

Input

- *N* - [integer] number of data samples
- *dt* - [float] time step

Output

- *angular frequencies* - [1D numpy array float]

mlpy.**scales**(*N*, *dj*, *dt*, *s0*)
Compute scales.

Input

- *N* - [integer] number of data samples
- *dj* - [float] scale resolution
- *dt* - [float] time step

Output

- *scales* - [1D numpy array float]

mlpy.**compute_s0**(*dt*, *p*, *wf*)
Compute s0.

Input

- *dt* - [float] time step
- *p* - [float] omega0 ('morlet') or order ('paul', 'dog')
- *wf* - [string] wavelet function ('morlet', 'paul', 'dog')

Output

- *s0* - [float]

CHAPTER

THREE

IMPUTING

3.1 Purify

mlpy.**purify**(*x*, *th0=0.1*, *th1=0.1*)

Return the matrix x without rows and cols containing respectively more than th0 * x.shape[1] and th1 * x.shape[0] NaNs.

Returns

(xout, v0, v1) [(2d ndarray, 1d ndarray int, 1d ndarray int)] v0 are the valid index at dimension 0 and v1 are the valid index at dimension 1

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([[1,      4,      4     ],
...               [2,      9,      np.NaN],
...               [2,      5,      8     ],
...               [8,      np.NaN, np.NaN],
...               [np.NaN, 4,      4     ]])
>>> y = np.array([1, -1, 1, -1, -1])
>>> x, v0, v1 = mlpy.purify(x, 0.4, 0.4)
>>> x
array([[  1.,   4.,   4.],
       [  2.,   9.,  NaN],
       [  2.,   5.,   8.],
       [ NaN,   4.,   4.]])
>>> v0
array([0, 1, 2, 4])
>>> v1
array([0, 1, 2])
```

New in version 2.0.4.

3.2 KNN imputing

mlpy.**knn_imputing**(*x*, *k*, *dist='e'*, *method='mean'*, *y=None*, *ldep=False*)

Knn imputing

Parameters

x [2d ndarray float (samples x feats)] data to impute

k [integer] number of nearest neighbor

dist [string ('se' = SQUARED EUCLIDEAN, 'e' = EUCLIDEAN)] adopted distance

method [string ('mean', 'median')] method to compute the missing values

y [1d ndarray] labels

ldep [bool] label depended (if y != None)

Returns

xout [2d ndarray float (samples x feats)] data imputed

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([[1,      4,      4     ],
...               [2,      9,      np.NaN],
...               [2,      5,      8     ],
...               [8,      np.NaN, np.NaN],
...               [np.NaN, 4,      4     ]])
>>> y = np.array([1, -1, 1, -1, -1])
>>> x, v0, v1 = mlpy.purify(x, 0.4, 0.4)
>>> x
array([[  1.,   4.,   4.],
       [  2.,   9.,  NaN],
       [  2.,   5.,   8.],
       [ NaN,   4.,   4.]])
>>> v0
array([0, 1, 2, 4])
>>> v1
array([0, 1, 2])
>>> y = y[v0]
>>> x = mlpy.knn_imputing(x, 2, dist='e', method='median')
>>> x
array([[ 1. ,  4. ,  4. ],
       [ 2. ,  9. ,  6. ],
       [ 2. ,  5. ,  8. ],
       [ 1.5,  4. ,  4. ]])
```

New in version 2.0.4.

CHAPTER

FOUR

DISTANCE COMPUTATIONS

4.1 Dynamic Time Warping

Features:

- Naive and Derivative *[Keogh01]* DTW
- Symmetric, Asymmetric, Quasi-Symmetric implementation with Slope Constraint Condition P=0 *[Sakoe78]*
- Sakoe-Chiba window condition *[Sakoe78]* option
- Linear space-complexity implementation option

class `mlpy.Dtw`(*derivative=False*, *startbc=True*, *steppattern='symmetric0'*, *wincond='nowindow'*, *r=0.0*, *onlydist=True*)

Dynamic Time Warping.

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([1,1,2,2,3,3,4,4,4,4,3,3,2,2,1,1])
>>> y = np.array([1,1,1,1,1,1,1,1,1,1,2,2,3,3,4,3,2,2,1,2,3,4])
>>> mydtw = mlpy.Dtw(onlydist=False)
>>> mydtw.compute(x, y)
0.36842105263157893
>>> mydtw.px
array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  2,  3,  4,  5,  6,  7,  8,
        9, 10, 11, 12, 12, 12, 13, 14, 15], dtype=int32)
>>> mydtw.py
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 14, 14,
       14, 15, 15, 16, 17, 18, 19, 20, 21], dtype=int32)
```

Parameters

derivative [bool] derivative DTW (DDTW)

startbc [bool] forces x=0 and y=0 boundary condition

steppattern [string ('symmetric', 'asymmetric', 'quasisymmetric')] step pattern

wincond [string ('nowindow', 'sakoechiba')] window condition

r [float] sakoe-chiba window length

onlydist [bool] linear space-complexity implementation. Only the current and previous columns are kept in memory.

New in version 2.0.7.

compute (*x*, *y*)

Parameters

x [1d ndarray or list] first time series

y [1d ndarray or list] second time series

Returns

d [float] normalized distance

Attributes

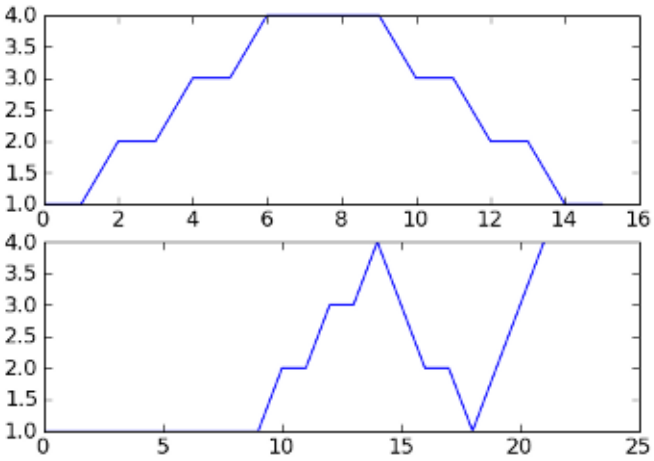
Dtw.px [1d ndarray int32] optimal warping path (for x time series) (if onlydist=False)

Dtw.py [1d ndarray int32] optimal warping path (for y time series) (if onlydist=False)

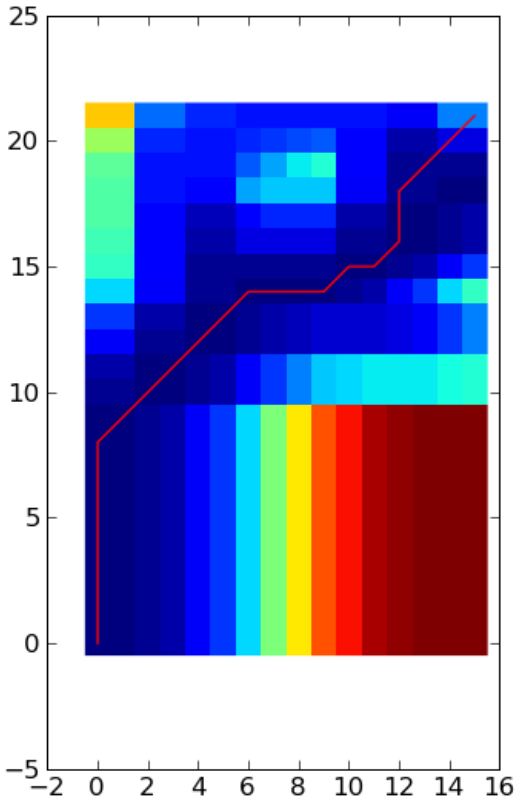
Dtw.cost [2dndarray float] cost matrix (if onlydist=False)

Extended example (requires matplotlib module):

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import mlpy
>>> x = np.array([1,1,2,2,3,3,4,4,4,4,3,3,2,2,1,1])
>>> y = np.array([1,1,1,1,1,1,1,1,1,2,2,3,3,4,3,2,2,1,2,3,4])
>>> plt.figure(1)
>>> plt.subplot(211)
>>> plt.plot(x)
>>> plt.subplot(212)
>>> plt.plot(y)
>>> plt.show()
```




```
>>> mydtw = mlpy.Dtw()
>>> d = mydtw.compute(x, y)
>>> plt.figure(2)
>>> plt.imshow(mydtw.cost.T, interpolation='nearest', origin='lower')
>>> plt.plot(mydtw.px, mydtw.py, 'r')
>>> plt.show()
```



4.2 Minkowski Distance

class `mlpy.`**`Minkowski`**(*p*)

Computes the Minkowski distance between two vectors `x` and `y`.

$$||x-y||_p = (\sum |x_i - y_i|^p)^{1/p}.$$

Initialize Minkowski class.

Parameters

p [float] The norm of the difference $||x-y||_p$

New in version 2.0.8.

`compute`(*x*, *y*)

Compute Minkowski distance

Parameters

x [ndarray] An 1-dimensional vector.

y [ndarray] An 1-dimensional vector.

Returns

d [float] The Minkowski distance between vectors `x` and `y`

CHAPTER

FIVE

CLUSTERING

5.1 Hierarchical Clustering

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

class mlpy.**HCluster**(*method='euclidean'*, *link='complete'*)

Hierarchical Cluster.

Initialize Hierarchical Cluster.

Parameters

method [string ('euclidean')] the distance measure to be used

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

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([[ 1. ,  1.5],
...               [ 1.1,  1.8],
...               [ 2. ,  2.8],
...               [ 3.2,  3.1],
...               [ 3.4,  3.2]])
>>> hc = mlpy.HCluster()
>>> hc.compute(x)
>>> hc.ia
array([-4, -1, -3,  2])
>>> hc.ib
array([-5, -2,  1,  3])
>>> hc.heights
array([ 0.2236068 ,  0.31622776,  1.4560219 ,  2.94108844])
>>> hc.cut(0.5)
array([0, 0, 1, 2, 2])
```

compute(*x*)

Compute Hierarchical Cluster.

Parameters

x [ndarray] An 2-dimensional vector (sample x features).

Returns

self.ia [ndarray (1-dimensional vector)] merge

self.ib [ndarray (1-dimensional vector)] merge

self.heights [ndarray (1-dimensional vector)] a set of n-1 non-decreasing real values. The clustering height: that is, the value of the criterion associated with the clustering method for the particular agglomeration.

Element i of merge describes the merging of clusters at step i of the clustering. If an element j is negative, then observation -j was merged at this stage. If j is positive then the merge was with the cluster formed at the (earlier) stage j of the algorithm. Thus negative entries in merge indicate agglomerations of singletons, and positive entries indicate agglomerations of non-singletons.

cut (*ht*)

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

Parameters

ht [float] height where the tree should be cut

Returns

cl [ndarray (1-dimensional vector)] group memberships. Groups are in 0, ..., N-1

5.2 k-means

class mlpy.**Kmeans** (*k*, *init='std'*, *seed=0*)

k-means algorithm.

Initialization.

Parameters

k [int (>1)] number of clusters

init [string ('std', 'plus')]

initialization algorithm

- 'std' : randomly selected
- 'plus' : k-means++ algorithm

seed [int (>=0)] random seed

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([[ 1. ,  1.5],
...               [ 1.1,  1.8],
...               [ 2. ,  2.8],
...               [ 3.2,  3.1],
...               [ 3.4,  3.2]])
>>> kmeans = mlpy.Kmeans(k=3, init="plus", seed=0)
>>> kmeans.compute(x)
array([1, 1, 2, 0, 0], dtype=int32)
>>> kmeans.means
array([[ 3.3 ,  3.15],
[ 1.05,  1.65],
[ 2.  ,  2.8 ]])
>>> kmeans.steps
2
```

New in version 2.2.0.

compute(*x*)
Compute Kmeans.

Parameters

x [ndarray] an 2-dimensional vector (number of points x dimensions)

Returns

cls [ndarray (1-dimensional vector)] cluster membership. Clusters are in 0, ..., k-1

Attributes

Kmeans.means [2d ndarray float (k x dim)] means

Kmeans.steps [int] number of steps

5.3 k-medoids

class mlpy.**Kmedoids**(*k*, *dist*, *maxloops=100*, *rs=0*)
k-medoids algorithm.

Initialize Kmedoids.

Parameters

k [int] Number of clusters/medoids

dist [class] class with a .compute(x, y) method which returns a distance

maxloops [int] maximum number of loops

rs [int] random seed

Example:

```
>>> import numpy as np
>>> import mlpy
>>> x = np.array([[ 1. ,  1.5],
...               [ 1.1,  1.8],
...               [ 2. ,  2.8],
...               [ 3.2,  3.1],
...               [ 3.4,  3.2]])
>>> dtw = mlpy.Dtw(onlydist=True)
>>> km = mlpy.Kmedoids(k=3, dist=dtw)
>>> km.compute(x)
(array([4, 0, 2]), array([3, 1]), array([0, 1]), 0.072499999999999981)
```

Samples 4, 0, 2 are medoids and represent cluster 0, 1, 2 respectively.

- cluster 0: samples 4 (medoid) and 3
- cluster 1: samples 0 (medoid) and 1
- cluster 2: sample 2 (medoid)

New in version 2.0.8.

compute(*x*)
Compute Kmedoids.

Parameters

x [ndarray] An 2-dimensional vector (sample x features).

Returns

m [ndarray (1-dimensional vector)] medoids indexes

n [ndarray (1-dimensional vector)] non-medoids indexes

cl [ndarray 1-dimensional vector)] cluster membership for non-medoids. Groups are in 0, ..., k-1

co [double] total cost of configuration

CHAPTER

SIX

KERNELS

Methods:

.matrix(*x*)
Return the kernel matrix $K_{ij} = k(x_i, x_j)$.

.vector(*a*, *x*)
Return the kernel vector $K_i = k(x_i, a)$.

6.1 Linear Kernel

class mlpy.**KernelLinear**
Linear Kernel

$$K(x, x') = x \cdot x'$$

6.2 Gaussian Kernel

class mlpy.**KernelGaussian**(*sigma*)
Gaussian Kernel

$$K(x, x') = e^{-\frac{\|x-x'\|}{2\sigma^2}}$$

6.3 Polynomial Kernel

class mlpy.**KernelPolynomial**(*d*)
Polynomial Kernel

$$K(x, x') = (x \cdot x' + 1)^d$$

CHAPTER

SEVEN

SUPERVISED CLASSIFICATION

Every classifier must be initialized with a specific set of parameters. Two distinct methods are deployed for the *training* (`compute()`) and the *testing* (`predict()`) phases. Whenever possible, the real valued prediction is stored in the *realpred* variable.

7.1 Support Vector Machines (SVMs)

class `mlpy.`**`Svm`**(*kernel='linear'*, *kp=0.1*, *C=1.0*, *tol=0.001*, *eps=0.001*, *maxloops=1000*, *cost=0.0*, *alpha_tversky=1.0*, *beta_tversky=1.0*, *opt_offset=True*)

Support Vector Machines (SVM).

Example

```
>>> import numpy as np
>>> import mlpy
>>> xtr = np.array([[1.0, 2.0, 3.0, 1.0],  # first sample
...                 [1.0, 2.0, 3.0, 2.0],  # second sample
...                 [1.0, 2.0, 3.0, 1.0]]) # third sample
>>> ytr = np.array([1, -1, 1])             # classes
>>> mysvm = mlpy.Svm()                     # initialize Svm class
>>> mysvm.compute(xtr, ytr)                # compute SVM
1
>>> mysvm.predict(xtr)                     # predict SVM model on training data
array([ 1, -1,  1])
>>> xts = np.array([4.0, 5.0, 6.0, 7.0])   # test point
>>> mysvm.predict(xts)                     # predict SVM model on test point
-1
>>> mysvm.realpred                         # real-valued prediction
-5.5
>>> mysvm.weights(xtr, ytr)                # compute weights on training data
array([ 0.,  0.,  0.,  1.])
```

Initialize the Svm class

Parameters

kernel [string ['linear', 'gaussian', 'polynomial', 'tr', 'tversky']] kernel

kp [float] kernel parameter (two sigma squared) for gaussian and polynomial kernel

C [float] regularization parameter

tol [float] tolerance for testing KKT conditions

eps [float] convergence parameter

maxloops [integer] maximum number of optimization loops

cost [float [-1.0, ..., 1.0]] for cost-sensitive classification

alpha_tversky [float] positive multiplicative parameter for the norm of the first vector

beta_tversky [float] positive multiplicative parameter for the norm of the second vector

opt_offset [bool] compute the optimal offset

compute(*x*, *y*)
Compute SVM model

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

conv [integer] svm convergence (0: false, 1: true)

predict(*p*)
Predict svm model on a test point(s)

Parameters

p [1d or 2d ndarray float (samples x feats)] test point(s)training dataInput

Returns

cl [integer or 1d ndarray integer] class(es) predicted

Attributes

Svm.realpred [float or 1d ndarray float] real valued prediction

weights(*x*, *y*)
Return feature weights

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

fw [1d ndarray float] feature weights

Note: For *tr* kernel (Terminated Ramp Kernel) see *[Merler06]*.

7.2 K Nearest Neighbor (KNN)

class mlpy.**Knn**(*k*, *dist='se'*)
k-Nearest Neighbor (KNN).

Example:

```
>>> import numpy as np
>>> import mlpy
>>> xtr = np.array([[1.0, 2.0, 3.1, 1.0],  # first sample
...                 [1.0, 2.0, 3.0, 2.0],  # second sample
...                 [1.0, 2.0, 3.1, 1.0]]) # third sample
>>> ytr = np.array([1, -1, 1])             # classes
>>> myknn = mlpy.Knn(k = 1)                # initialize knn class
>>> myknn.compute(xtr, ytr)               # compute knn
1
>>> myknn.predict(xtr)                    # predict knn model on training data
array([ 1, -1,  1])
>>> xts = np.array([4.0, 5.0, 6.0, 7.0]) # test point
>>> myknn.predict(xts)                    # predict knn model on test point
-1
>>> myknn.realpred                        # real-valued prediction
0.0
```

Initialize the Knn class.

Parameters

k [int (odd >= 1)] number of NN

dist [string ('se' = SQUARED EUCLIDEAN, 'e' = EUCLIDEAN)] adopted distance

compute (*x*, *y*)

Store x and y data.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1 for binary classification)] : 1d ndarray integer (1, ..., nclasses for multiclass classificatio) classes

Returns 1

Raises

ValueError if not (1 <= k <= #samples)

ValueError if there aren'e at least 2 classes

ValueError if, in case of 2-classes problems, the lables are not 1 and -1

ValueError if, in case of n-classes problems, the lables are not int from 1 to n

predict (*p*)

Predict knn model on a test point(s).

Parameters

p [1d or 2d ndarray float (sample(s) x feats)] test sample(s)

Returns the predicted value(s) on success: integer or 1d numpy array integer (-1 or 1) for binary classification integer or 1d numpy array integer (1, ..., nclasses) for multiclass classification 0 on succes with non unique classification -2 otherwise

Raises

StandardError if no Knn method computed

7.3 Fisher Discriminant Analysis (FDA)

Described in *[Mika01]*.

class mlpy.**Fda**(*C=1*)
Fisher Discriminant Analysis.

Example:

```
>>> import numpy as np
>>> import mlpy
>>> xtr = np.array([[1.0, 2.0, 3.1, 1.0],  # first sample
...                 [1.0, 2.0, 3.0, 2.0],  # second sample
...                 [1.0, 2.0, 3.1, 1.0]]) # third sample
>>> ytr = np.array([1, -1, 1])             # classes
>>> myfda = mlpy.Fda()                    # initialize fda class
>>> myfda.compute(xtr, ytr)               # compute fda
1
>>> myfda.predict(xtr)                    # predict fda model on training data
array([ 1, -1,  1])
>>> xts = np.array([4.0, 5.0, 6.0, 7.0]) # test point
>>> myfda.predict(xts)                    # predict fda model on test point
-1
>>> myfda.realpred                        # real-valued prediction
-42.51475717037367
>>> myfda.weights(xtr, ytr)               # compute weights on training data
array([  9.60629896,   9.77148463,   9.82027615,  11.58765243])
```

Initialize Fda class.

Parameters

C [float] regularization parameter

compute(*x*, *y*)
Compute fda model.

Parameters

x [2d numpy array float (sample x feature)] training data

y [1d numpy array integer (two classes, 1 or -1)] classes

Returns 1

predict(*p*)
Predict fda model on test point(s).

Parameters

p [1d or 2d ndarray float (sample(s) x feats)] test sample(s)

Returns

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

Attributes

self.realpred [float or 1d numpy array float] real valued prediction

weights(*x*, *y*)
Return feature weights.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

fw [1d ndarray float] feature weights

7.4 Spectral Regression Discriminant Analysis (SRDA)

Described in *[Cai08]*.

class `mlpy.Srda`(*alpha=1.0*)

Spectral Regression Discriminant Analysis (SRDA).

Example:

```
>>> import numpy as np
>>> import mlpy
>>> xtr = np.array([[1.0, 2.0, 3.1, 1.0],  # first sample
...                 [1.0, 2.0, 3.0, 2.0],  # second sample
...                 [1.0, 2.0, 3.1, 1.0]]) # third sample
>>> ytr = np.array([1, -1, 1])             # classes
>>> mysrda = mlpy.Srda()                  # initialize srda class
>>> mysrda.compute(xtr, ytr)              # compute srda
1
>>> mysrda.predict(xtr)                   # predict srda model on training data
array([ 1, -1,  1])
>>> xts = np.array([4.0, 5.0, 6.0, 7.0]) # test point
>>> mysrda.predict(xts)                   # predict srda model on test point
-1
>>> mysrda.realpred                       # real-valued prediction
-6.8283034257748758
>>> mysrda.weights(xtr, ytr)              # compute weights on training data
array([ 0.10766721,  0.21533442,  0.51386623,  1.69331158])
```

Initialize the Srda class.

Parameters

alpha [float(>=0.0)] regularization parameter

compute(*x*, *y*)

Compute Srda model. Initialize array of alphas a.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns 1

Raises

LinAlgError if x is singular matrix in __PenRegrModel

predict(*p*)

Predict Srda model on test point(s).

Parameters

p [1d or 2d ndarray float (sample(s) x feats)] test sample(s)

Returns

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

Attributes

self.realpred [float or 1d numpy array float] real valued prediction

weights (*x*, *y*)
Return feature weights.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

fw [1d ndarray float] feature weights

7.5 Penalized Discriminant Analysis (PDA)

Described in *[Ghosh03]*.

class mlpy.**Pda** (*Nreg=3*)
Penalized Discriminant Analysis (PDA).

Example:

```
>>> import numpy as np
>>> import mlpy
>>> xtr = np.array([[1.0, 2.0, 3.1, 1.0],  # first sample
...                 [1.0, 2.0, 3.0, 2.0],  # second sample
...                 [1.0, 2.0, 3.1, 1.0]]) # third sample
>>> ytr = np.array([1, -1, 1])             # classes
>>> mypda = mlpy.Pda()                    # initialize pda class
>>> mypda.compute(xtr, ytr)               # compute pda
1
>>> mypda.predict(xtr)                    # predict pda model on training data
array([ 1, -1,  1])
>>> xts = np.array([4.0, 5.0, 6.0, 7.0]) # test point
>>> mypda.predict(xts)                    # predict pda model on test point
-1
>>> mypda.realpred                        # real-valued prediction
-7.6106885609535624
>>> mypda.weights(xtr, ytr)               # compute weights on training data
array([  4.0468174 ,   8.0936348 ,  18.79228266,  58.42466988])
```

Initialize Pda class.

Parameters

Nreg [int] number of regressions

compute (*x*, *y*)
Compute Pda model.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns 1

Raises

LinAlgError if x is singular matrix in __PenRegrModel

predict (*p*)

Predict Pda model on test point(s).

Parameters

p [1d or 2d ndarray float (sample(s) x feats)] test sample(s)

Returns

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

Attributes

self.realpred [float or 1d numpy array float] real valued prediction

weights (*x*, *y*)

Compute feature weights.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

fw [1d ndarray float] feature weights

7.6 Diagonal Linear Discriminant Analysis (DLDA)

class mlpy.**Dlda** (*nf=0*, *tol=10*, *overview=False*, *bal=False*)

Diagonal Linear Discriminant Analysis.

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> xtr = array([[1.1, 2.4, 3.1, 1.0],  # first sample
...              [1.2, 2.3, 3.0, 2.0],  # second sample
...              [1.3, 2.2, 3.5, 1.0],  # third sample
...              [1.4, 2.1, 3.2, 2.0]]) # fourth sample
>>> ytr = array([1, -1, 1, -1])         # classes
>>> mydlda = Dlda(nf = 2)                    # initialize dlda class
>>> mydlda.compute(xtr, ytr)        # compute dlda
1
>>> mydlda.predict(xtr)             # predict dlda model on training data
array([ 1, -1,  1, -1])
>>> xts = array([4.0, 5.0, 6.0, 7.0])   # test point
>>> mydlda.predict(xts)                 # predict dlda model on test point
-1
>>> mydlda.realpred                     # real-valued prediction
```

```
-21.999999999999954
>>> mydlda.weights(xtr, ytr)        # compute weights on training data
array([  2.13162821e-14,   0.00000000e+00,   0.00000000e+00,   4.00000000e+00])
```

Initialize Dlda class.

Parameters

nf [int (1 <= nf >= #features)] the number of the best features that you want to use in the model. If nf = 0 the system stops at a number of features corresponding to a peak of accuracy

tol [int] in case of nf = 0 it's the number of steps of classification to be calculated after the peak to avoid a local maximum

overview [bool] set True to print informations about the accuracy of the classifier at every step of the compute

bal [bool] set True if it's reasonable to consider the unbalancement of the test set similar to the one of the training set

compute (*x*, *y*, *mf=0*)

Compute Dlda model.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

mf [int] number of classification steps to be calculated more on a model already computed

Returns 1

Raises

LinAlgError if x is singular matrix

predict (*p*)

Predict Dlda model on test point(s).

Parameters

p [1d or 2d ndarray float (sample(s) x feats)] test sample(s)

Returns

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

Attributes

self.realpred [float or 1d numpy array float] real valued prediction

weights (*x*, *y*)

Return feature weights.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

fw [1d ndarray float] feature weights, they are going to be > 0 for the features chosen for the classification and = 0 for all the others

CHAPTER

EIGHT

REGRESSION

8.1 Ordinary Least Squares and Ridge Regression

class mlpy.**RidgeRegression**(*alpha=0.0*)

Ridge Regression and Ordinary Least Squares (OLS).

Initialization.

Parameters

alpha [float (>= 0.0)] regularization (0.0: OLS)

New in version 2.2.0.

beta()

Return b_1, ..., b_p.

beta0()

Return b_0.

learn(*x*, *y*)

Compute the regression coefficients.

Parameters

x [numpy 2d array (n x p)] matrix of regressors

y [numpy 1d array (n)] response

pred(*x*)

Compute the predicted response.

Parameters

x [numpy 2d array (nxp)] matrix of regressors

Returns

yp [1d ndarray] predicted response

selected()

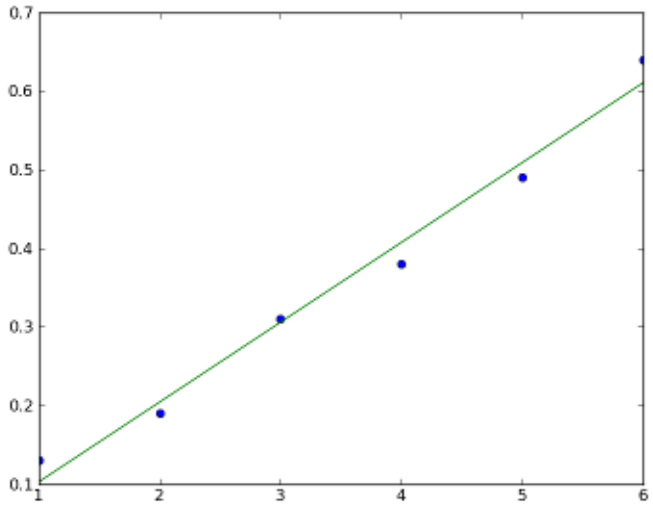
Returns the regressors ranking.

Note: The predicted response is computed as:

$$\hat{y} = \beta_0 + X\boldsymbol{\beta}$$

Example (requires matplotlib module):

```
>>> import numpy as np
>>> import mlpy
>>> import matplotlib.pyplot as plt
>>> x = np.array([[1], [2], [3], [4], [5], [6]]) # p = 1
>>> y = np.array([0.13, 0.19, 0.31, 0.38, 0.49, 0.64])
>>> rr = mlpy.RidgeRegression(alpha=0.0) # OLS
>>> rr.learn(x, y)
>>> y_hat = rr.pred(x)
>>> plt.figure(1)
>>> plt.plot(x[:, 0], y, 'o') # show y
>>> plt.plot(x[:, 0], y_hat) # show y_hat
>>> plt.show()
```




```
>>> rr.beta0()
0.004666666666667078
>>> rr.beta()
array([ 0.10057143])
```

8.2 Kernel Ridge Regression

class mlpy.**KernelRidgeRegression**(*kernel*, *alpha*)

Ridge Regression and Ordinary Least Squares (OLS).

Initialization.

Parameters alpha : float (> 0.0)

New in version 2.2.0.

learn(*x*, *y*)

Compute the regression coefficients.

Parameters

x [numpy 2d array (n x p)] matrix of regressors

y [numpy 1d array (n)] response

pred(*x*)
Compute the predicted response.

Parameters

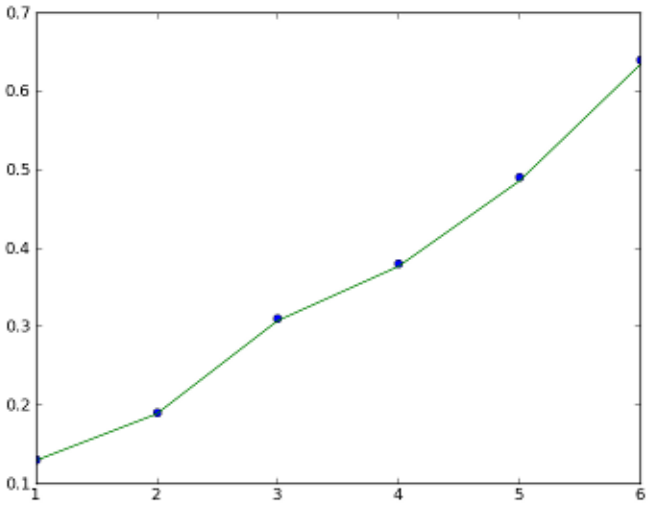
x [numpy 2d array (n x p)] matrix of regressors

Returns

yp [1d ndarray] predicted response

Example (requires matplotlib module):

```
>>> import numpy as np
>>> import mlpy
>>> import matplotlib.pyplot as plt
>>> x = np.array([[1], [2], [3], [4], [5], [6]]) # p = 1
>>> y = np.array([0.13, 0.19, 0.31, 0.38, 0.49, 0.64])
>>> kernel = mlpy.KernelGaussian(sigma=0.01)
>>> krr = mlpy.KernelRidgeRegression(kernel=kernel, alpha=0.01)
>>> krr.learn(x,y)
>>> y_hat = krr.pred(x)
>>> plt.figure(1)
>>> plt.plot(x[:, 0], y, 'o') # show y
>>> plt.plot(x[:, 0], y_hat) # show y_hat
>>> plt.show()
```



8.3 Least Angle Regression (LAR)

Least Angle Regression is described in *[Efron04]*.

Covariates should be standardized to have mean 0 and unit length, and the response should have mean 0:

$$\sum_{i=1}^{n} x_{ij} = 0, \qquad \sum_{i=1}^{n} x_{ij}^2 = 1, \qquad \sum_{i=1}^{n} y_i = 0 \qquad \text{for } j = 1, 2, \ldots, p.$$

class mlpy.**Lar**(*m=None*)
LAR.

Initialization.

Parameters

m [int (> 0)] max number of steps (= number of features selected). If m=None -> m=x.shape[1] in .learn(x, y)

New in version 2.2.0.

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

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

Parameters

x [numpy 2d array (nxp)] matrix of regressors

y [numpy 1d array (n)] response

pred(*x*)
Compute the predicted response.

Parameters

x [numpy 2d array (nxp)] matrix of regressors

Returns

yp [1d ndarray] predicted response

selected()
Returns the regressors ranking.

steps()
Return the number of steps really performed.

8.4 LASSO (LARS implementation)

It implements simple modifications of the LARS algorithm that produces Lasso estimates. See *[Efron04]* and *[Tibshirani96]*.

Covariates should be standardized to have mean 0 and unit length, and the response should have mean 0:

$$\sum_{i=1}^{n} x_{ij} = 0, \qquad \sum_{i=1}^{n} x_{ij}^2 = 1, \qquad \sum_{i=1}^{n} y_i = 0 \qquad \text{for } j = 1, 2, \ldots, p.$$

class mlpy.**Lasso**(*m*)
LASSO computed with LARS algoritm.

Initialization.

Parameters

m [int (> 0)] max number of steps.

New in version 2.2.0.

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

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

Parameters

x [numpy 2d array (nxp)] matrix of regressors

y [numpy 1d array (n)] response

pred(*x*)
Compute the predicted response.

Parameters

x [numpy 2d array (nxp)] matrix of regressors

Returns

yp [1d ndarray] predicted response

selected()
Returns the regressors ranking.

steps()
Return the number of steps really performed.

8.5 Gradient Descent

class mlpy.**GradientDescent**(*kernel*, *t*, *stepsize*)
Gradient Descent Method

Initialization.

Parameters

kernel: kernel object kernel

t [int (> 0)] number of iterations

stepsize: float step size

New in version 2.2.0.

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

Parameters

x [numpy 2d array (n x p)] matrix of regressors

y [numpy 1d array (n)] response

pred(*x*)
Compute the predicted response.

Parameters

x [numpy 2d array (n x p)] matrix of regressors

Returns

yp [1d ndarray] predicted response

CHAPTER

NINE

FEATURE WEIGHTING

Algorithms for assessing the quality of features.

9.1 Classifier-derived methods

See classification.

9.2 Iterative RELIEF (I-RELIEF)

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

Iterative RELIEF for Feature Weighting.

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> x = array([[1.1, 2.1, 3.1, -1.0],  # first sample
...            [1.2, 2.2, 3.2, 1.0],   # second sample
...            [1.3, 2.3, 3.3, -1.0]]) # third sample
>>> y = array([1, 2, 1])               # classes
>>> myir = Irelief()                   # initialize irelief class
>>> myir.weights(x, y)                 # compute feature weights
array([ 0.,  0.,  0.,  1.])
```

Initialize the Irelief class.

Input

- *T* - [integer] (>0) max loops
- *sigma* - [float] (>0.0) kernel width
- *theta* - [float] (>0.0) convergence parameter

weights(*x*, *y*)

Return feature weights.

Input

- *x* - [2D numpy array float] (sample x feature) training data
- *y* - [1D numpy array integer] (two classes) classes

Output

- *fw* - [1D numpy array float] feature weights

exception mlpy.**SigmaError**
Sigma Error

Sigma parameter is too small.

9.3 Feature Weighting/Selection Yijun Sun08

A feature weighting/selection algorithm described in *[Sun08]*.

class mlpy.**FSSun**(*T=1000*, *sigma=1.0*, *theta=0.001*, *lmbd=1.0*, *eps=0.001*, *alpha0=1.0*, *c=0.01*, *rho=0.5*, *debug=False*)
Sun Algorithm for feature weighting/selection

Initialize the FSSun class

Parameters

T [int (> 0)] max loops

sigma [float (> 0.0)] kernel width

theta [float (> 0.0)] convergence parameter

lmbd [float] regularization parameter

eps [float (0 < eps << 1)] termination tolerance for steepest descent method

alpha0 [float (> 0.0)] initial step length (usually 1.0) for line search

c [float (0 < c < 1/2)] costant for line search

rho [flaot (0 < rho < 1)] alpha coefficient for line search

New in version 2.1.0.

weights(*x*, *y*)
Compute the feature weights

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

fw [1d ndarray float] feature weights

Attributes

FSSun.loops [int] number of loops

Raises

ValueError if classes are not -1 or 1

SigmaError if sigma parameter is too small

9.4 Discrete Wavelet Transform based (DWT)

class mlpy.**Dwt**(*specdiff='rpv'*)

Discrete Wavelet Transform (DWT).

Example:

```
>>> import numpy as np
>>> import mlpy
>>> xtr = np.array([[1.0, 2.0, 3.1, 1.0],  # first sample
...                 [1.0, 2.0, 3.0, 2.0],  # second sample
...                 [1.0, 2.0, 3.1, 1.0]]) # third sample
>>> ytr = np.array([1, -1, 1])             # classes
>>> mydwt = mlpy.Dwt()                    # initialize dwt class
>>> mydwt.weights(xtr, ytr)               # compute weights on training data
array([ -2.22044605e-14,  -2.22044605e-14,   6.34755463e+00,  -3.00000000e+02])
```

Initialize the Dwt class.

Input

- *specdiff* - [string] spectral difference method ('rpv', 'arpv', 'crpv')

weights(*x*, *y*)

Return ABSOLUTE feature weights.

Parameters

x [2d ndarray float (samples x feats)] training data

y [1d ndarray integer (-1 or 1)] classes

Returns

fw [1d ndarray float] feature weights

CHAPTER

TEN

FEATURE RANKING (WRAPPER METHODS)

The feature weights are used for selecting and ranking purposes inside one of the implemented schemes:

- *Recursive Feature Elimination family* *[Guyon02]*: RFE, ERFE *[Furlanello03]*, BISRFE, SQRTRFE
- *Recursive Forward Selection* family *[Louw06]*: RFS
- *One-step*

class `mlpy.`**`Ranking`**(*method='rfe'*, *lastsinglesteps=0*)

Ranking class based on Recursive Feature Elimination (RFE) and Recursive Forward Selection (RFS) methods.

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> x = array([[1.1, 2.1, 3.1, -1.0],  # first sample
...            [1.2, 2.2, 3.2, 1.0],   # second sample
...            [1.3, 2.3, 3.3, -1.0]]) # third sample
>>> y = array([1, -1, 1])              # classes
>>> myrank = Ranking()                 # initialize ranking class
>>> mysvm = Svm()                      # initialize svm class
>>> myrank.compute(x, y, mysvm)        # compute feature ranking
array([3, 1, 2, 0])
```

Initialize Ranking class.

Input

- *method* - [string] method ('onestep', 'rfe', 'bisrfe', 'sqrtrfe', 'erfe', 'rfs')
- *lastsinglesteps* - [integer] last single steps with 'rfe'

`compute`(*x*, *y*, *w*, *debug=False*)

Compute the feature ranking.

Input

- *x* - [2D numpy array float] (sample x feature) training data
- *y* - [1D numpy array integer] (1 or -1) classes
- *w* - object (e.g. classifier) with weights() method
- *debug* - [bool] show remaining number of feature at each step (True or False)

Output

- *feature ranking* - [1D numpy array integer] ranked feature indexes

CHAPTER

ELEVEN

RESAMPLING METHODS

11.1 k-fold

mlpy.**kfold**(*nsamples*, *sets*, *rseed=0*, *indexes=None*)
K-fold Resampling Method.

Input

- *nsamples* - [integer] number of samples
- *sets* - [integer] number of subsets (= number of tr/ts pairs)
- *rseed* - [integer] random seed
- *indexes* - [list integer] source indexes (None for [0, nsamples-1])

Output

- *idx* - list of *sets* tuples: ([training indexes], [test indexes])

mlpy.**kfoldS**(*cl*, *sets*, *rseed=0*, *indexes=None*)
Stratified K-fold Resampling Method.

Input

- *cl* - [list (1 or -1)] class label
- *sets* - [integer] number of subsets (= number of tr/ts pairs)
- *rseed* - [integer] random seed
- *indexes* - [list integer] source indexes (None for [0, nsamples-1])

Output

- *idx* - list of *sets* tuples: ([training indexes], [test indexes])

11.2 Monte Carlo

mlpy.**montecarlo**(*nsamples*, *pairs*, *sets*, *rseed=0*, *indexes=None*)
Monte Carlo Resampling Method.

Input

- *nsamples* - [integer] number of samples
- *pairs* - [integer] number of tr/ts pairs
- *sets* - [integer] 1/(fraction of data in test sets)

- *rseed* - [integer] random seed
- *indexes* - [list integer] source indexes (None for [0, nsamples-1])

Output

- *idx* - list of *pairs* tuples: ([training indexes], [test indexes])

mlpy.**montecarloS**(*cl*, *pairs*, *sets*, *rseed=0*, *indexes=None*)
Stratified Monte Carlo Resampling Method.

Input

- *cl* - [list (1 or -1)] class label
- *pairs* - [integer] number of tr/ts pairs
- *sets* - [integer] 1/(fraction of data in test sets)
- *rseed* - [integer] random seed
- *indexes* - [list integer] source indexes (None for [0, nsamples-1])

Output

- *idx* - list of *pairs* tuples: ([training indexes], [test indexes])

11.3 Leave-one-out

mlpy.**leaveoneout**(*nsamples*, *indexes=None*)
Leave-one-out Resampling Method.

Input

- *nsamples* - [integer] number of samples
- *indexes* - [list integer] source indexes (None for [0, nsamples-1])

Output

- *idx* - list of *nsamples* tuples: ([training indexes], [test indexes])

11.4 All Combinations

mlpy.**allcombinations**(*cl*, *sets*, *indexes=None*)
All Combinations Resampling Method.

Input

- *cl* - [list (1 or -1)] class label
- *sets* - [integer] number of subset
- *indexes* - [list integer] source indexes (None for [0, nsamples-1])

Output

- *idx* - list of tuples: ([training indexes], [test indexes])

11.5 Manual Resampling

mlpy.**manresampling**(*cl*, *pairs*, *trp*, *trn*, *tsp*, *tsn*, *rseed=0*)

Manual Resampling.

Input

- *cl* - [list (1 or -1)] class label
- *pairs* - [integer] number of tr/ts pairs
- *trp* - [integer] number of positive samples in training
- *trn* - [integer] number of negative samples in training
- *tsp* - [integer] number of positive samples in test
- *tsn* - [integer] number of negative samples in test

Output

- *idx* - list of *pairs* tuples: ([training indexes], [test indexes])

11.6 Resampling File

mlpy.**resamplingfile**(*nsamples*, *file*, *sep='\t'*)

Resampling file from file.

Returns a list of tuples: ([training indexes],[test indexes])

Read a file in the form:

```
[test indexes 'sep'-separated for the first  replicate]
[test indexes 'sep'-separated for the second replicate]
                        .
                        .
                        .
[test indexes 'sep'-separated for the last   replicate]
```

where indexes must be integers in [0, nsamples-1].

Input

- *file* - [string] test indexes file
- *nsamples* - [integer] number of samples

Output

- *idx* - list of tuples: ([training indexes],[test indexes])

CHAPTER

TWELVE

METRIC FUNCTIONS

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

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.`**`err`**(*y*, *p*)

Compute the Error.

error = (fp + fn) / ts

Input

- *y* - classes (two classes) [1D numpy array integer]
- *p* - prediction (two classes) [1D numpy array integer]

Output

- error

`mlpy.`**`errp`**(*y*, *p*)

Compute the Error for positive samples.

errp = fp / ap

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- error for positive samples

`mlpy.`**`errn`**(*y*, *p*)

Compute the Error for negative samples.

errn = fn / an

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- error for negative samples

mlpy.**acc**(*y*, *p*)
Compute the Accuracy.

accuracy = (tp + tn) / ts

Input

- *y* - classes (two classes) [1D numpy array integer]
- *p* - prediction (two classes) [1D numpy array integer]

Output

- accuracy

mlpy.**sens**(*y*, *p*)
Compute the Sensitivity.

sensitivity = tp / ap

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- sensitivity

mlpy.**spec**(*y*, *p*)
Compute the Specificity.

specificity = tn / an

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- specificity

mlpy.**single_auc**(*y*, *p*)
Compute the single AUC.

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- singleAUC

mlpy.**wmw_auc**(*y*, *r*)
Compute the AUC by using the Wilcoxon-Mann-Whitney formula.

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *r* - real-valued prediction [1D numpy array float]

Output

- wmwAUC

mlpy.**ppv**(*y*, *p*)
Compute the Positive Predictive Value (PPV).

PPV = tp / pp

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- PPV

mlpy.**npv**(*y*, *p*)
Compute the Negative Predictive Value (NPV).

NPV = tn / pn

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- NPV

mlpy.**mcc**(*y*, *p*)
Compute the Matthews Correlation Coefficient (MCC).

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

Input

- *y* - classes (two classes +1 and -1) [1D numpy array integer]
- *p* - prediction (two classes +1 and -1) [1D numpy array integer]

Output

- MCC

mlpy.**mse**(*y*, *p*)
Mean Squared Error

mlpy.**r2**(*y*, *p*)
Coefficient of determination (R^2)

R^2 is computed as square of the correlation coefficient.

CHAPTER

THIRTEEN

FEATURE LIST ANALYSIS

13.1 Canberra Indicator

Canberra stability indicator on top-k positions *[Jurman08]*

mlpy.**canberra**(*lists*, *k*, *dist=False*, *modules=None*)

Compute mean Canberra distance indicator on top-k sublists.

Input

- *lists* - [2D numpy array integer] position lists Positions must be in [0, #elems-1]
- *k* - [integer] top-k sublists
- *modules* - [list] modules (list of group indicies)
- *dist* - [bool] return partial distances (True or False)

Output

- *cd* - [float] canberra distance
- *i1* - [1D numpy array integer] index 1 (if dist == True)
- *i2* - [1D numpy array integer] index 2 (if dist == True)
- *pd* - [1D numpy array float] partial distances for index1 and index2 (if dist == True)

```
>>> from numpy import *
>>> from mlpy import *
>>> lists = array([[2,4,1,3,0],  # first positions list
...                [3,4,1,2,0],  # second positions list
...                [2,4,3,0,1],  # third positions list
...                [0,1,4,2,3]]) # fourth positions list
>>> canberra(lists, 3)
1.0861983059292479
```

mlpy.**canberraq**(*lists*, *complete=True*, *normalize=False*, *dist=False*)

Compute mean Canberra distance indicator on generic lists.

Input

- *lists* - [2D numpy array integer] position lists Positions must be in [-1, #elems-1], where -1 indicates features not present in the list
- *complete* - [bool] complete
- *normalize* - [bool] normalize
- *dist* - [bool] return partial distances (True or False)

Output

- *cd* - [float] canberra distance
- *i1* - [1D numpy array integer] index 1 (if dist == True)
- *i2* - [1D numpy array integer] index 2 (if dist == True)
- *pd* - [1D numpy array float] partial distances for index1 and index2 (if dist == True)

```
>>> from numpy import *
>>> from mlpy import *
>>> lists = array([[2,-1,1,-1,0],  # first positions list
...                [3,4,1,2,0],    # second positions list
...                [2,-1,3,0,1],   # third positions list
...                [0,1,4,2,3]])   # fourth positions list
>>> canberraq(lists)
1.0628570368721744
```

mlpy.**normalizer**(*lists*)

Compute the average length of the partial lists (nm) and the corresponding normalizing factor (nf) given by 1 - a / b where a is the exact value computed on the average length and b is the exact value computed on the whole set of features.

Inputs

- *lists* - [2D numpy array integer] position lists Positions must be in [-1, #elems-1], where -1 indicates features not present in the list

Output

- *(nm, nf)* - (float, float)

13.2 Borda Count, Extraction Indicator, Mean Position Indicator

Borda Count *[Borda1781]*

mlpy.**borda**(*lists*, *k*, *modules=None*)

Compute the number of extractions on top-k sublists and the mean position on lists for each element. Sort the element ids with decreasing number of extractions, AND element ids with equal number of extractions should be sorted with increasing mean positions.

Input

- *lists* - [2D numpy array integer] ranked feature-id lists. Feature-id must be in [0, #elems-1].
- *k* - [integer] on top-k sublists
- *modules* - [list] modules (list of group indicies)

Output

- *borda* - (feature-id, number of extractions, mean positions)

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> lists = array([[2,4,1,3,0],  # first ranked feature-id list
...                [3,4,1,2,0],  # second ranked feature-id list
...                [2,4,3,0,1],  # third ranked feature-id list
...                [0,1,4,2,3]]) # fourth ranked feature-id list
```

```
>>> borda(lists, 3)
(array([4, 1, 2, 3, 0]), array([4, 3, 2, 2, 1]), array([ 1.25      ,  1.66666667,␣
↪ 0.        ,  1.        ,  0.        ]))
```

- Element 4 is in the first position with 4 extractions and mean position 1.25.
- Element 1 is in the first position with 3 extractions and mean position 1.67.
- Element 2 is in the first position with 2 extractions and mean position 0.00.
- Element 3 is in the first position with 2 extractions and mean position 1.00.
- Element 0 is in the first position with 1 extractions and mean position 0.00.

mlpy.**borda_weighted**(*lists*, *w*, *decimals=2*)

Compute the mean position on lists for each element. Sort the element ids with increasing mean weighted positions.

Input

- *lists* - [2D numpy array integer] ranked feature-id lists. Feature-id must be in [0, #elems-1].
- *w* - [1D numpy array float] weights
- *decimals* - [integer >=0] decimals

Output

- *borda* - (feature-id, mean positions)

CHAPTER

FOURTEEN

DATA MANAGEMENT

14.1 Importing and exporting data

mlpy.**data_fromfile**(*file*, *ytype=<type 'int'>*)

Read data file in the form:

```
x11 [TAB] x12 [TAB] ... x1n [TAB] y1
x21 [TAB] x22 [TAB] ... x2n [TAB] y2
 .         .         .    .         .
 .         .          .   .         .
 .         .           .  .         .
xm1 [TAB] xm2 [TAB] ... xmn [TAB] ym
```

where xij are float and yi are of type 'ytype' (numpy.int or numpy.float).

Input

- *file* - data file name
- *ytype* - numpy datatype for labels (numpy.int or numpy.float)

Output

- *x* - data [2D numpy array float]
- *y* - classes [1D numpy array int or float]

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> x, y = data_fromfile('data_example.dat')
>>> x
array([[ 1.1,  2. ,  5.3,  3.1],
...    [ 3.7,  1.4,  2.3,  4.5],
...    [ 1.4,  5.4,  3.1,  1.4]])
>>> y
array([ 1, -1,  1])
```

mlpy.**data_fromfile_wl**(*file*)

Read data file in the form:

```
x11 [TAB] x12 [TAB] ... x1n [TAB]
x21 [TAB] x22 [TAB] ... x2n [TAB]
 .         .         .    .
 .         .          .   .
```

```
 .          .          .   .
xm1 [TAB] xm2 [TAB] ... xmn [TAB]
```

where xij are float.

Input

- *file* - data file name

Output

- *x* - data [2D numpy array float]

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> x, y = data_fromfile('data_example.dat')
>>> x
array([[ 1.1,  2. ,  5.3,  3.1],
...    [ 3.7,  1.4,  2.3,  4.5],
...    [ 1.4,  5.4,  3.1,  1.4]])
```

mlpy.**data_tofile**(*file*, *x*, *y*, *sep='\t'*)
Write data file in the form:

```
x11 [sep] x12 [sep] ... x1n [sep] y1
x21 [sep] x22 [sep] ... x2n [sep] y2
 .          .         .    .         .
 .          .          .   .         .
 .          .           .  .         .
xm1 [sep] xm2 [sep] ... xmn [sep] ym
```

where xij are float and yi are integer.

Input

- *file* - data file name
- *x* - data [2D numpy array float]
- *y* - classes [1D numpy array integer]
- *sep* - separator

mlpy.**data_tofile_wl**(*file*, *x*, *sep='\t'*)
Write data file in the form:

```
x11 [sep] x12 [sep] ... x1n [sep]
x21 [sep] x22 [sep] ... x2n [sep]
 .          .         .    .
 .          .          .   .
 .          .           .  .
xm1 [sep] xm2 [sep] ... xmn [sep]
```

where xij are float.

Input

- *file* - data file name
- *x* - data [2D numpy array float]

- *sep* - separator

14.2 Normalization

mlpy.**data_normalize**(*x*)

Normalize numpy array (2D) x.

Input

- *x* - data [2D numpy array float]

Output

- normalized data

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> x = array([[ 1.1,  2. ,  5.3,  3.1],
...            [ 3.7,  1.4,  2.3,  4.5],
...            [ 1.4,  5.4,  3.1,  1.4]])
>>> data_normalize(x)
array([[-0.9797065 , -0.48295391,  1.33847226,  0.12418815],
...    [ 0.52197912, -1.13395464, -0.48598056,  1.09795608],
...    [-0.75217354,  1.35919078,  0.1451563 , -0.75217354]])
```

Warning: Deprecated in version 2.3

mlpy.**data_standardize**(*x*, *p=None*)

Standardize numpy array (2D) x and optionally standardize p using mean and std of x.

Input

- *x* - data [2D numpy array float]
- *p* - optional data [2D numpy array float]

Output

- standardized data

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> x = array([[ 1.1,  2. ,  5.3,  3.1],
...            [ 3.7,  1.4,  2.3,  4.5],
...            [ 1.4,  5.4,  3.1,  1.4]])
>>> data_standardize(x)
array([[-0.67958381, -0.43266792,  1.1157668 ,  0.06441566],
...    [ 1.1482623 , -0.71081158, -0.81536804,  0.96623494],
...    [-0.46867849,  1.1434795 , -0.30039875, -1.0306506 ]])
```

Warning: Deprecated in version 2.3. Use mlpy.standardize and mlpy.standardize_from instead

mlpy.**standardize**(*x*)
Standardize x.

x is standardized to have mean 0 and unit length by columns. Return standardized x, the mean and the standard deviation.

mlpy.**center**(*y*)
Center y to have mean 0.

Return centered y.

mlpy.**standardize_from**(*x*, *mean*, *std*)
Standardize x using external mean and standard deviation.

Return standardized x.

mlpy.**center_from**(*y*, *mean*)
Center y using external mean.

Return centered y.

CHAPTER

FIFTEEN

MISCELLANEOUS

15.1 Confidence Interval

mlpy.**percentile_ci_median**(*x*, *nboot=1000*, *alpha=0.025*, *rseed=0*)

Percentile confidence interval for the median of a sample x and unknown distribution.

Input

- *x* - [1D numpy array] sample
- *nboot* - [integer] (>1) number of resamples
- *alpha* - [float] confidence level is 100*(1-2*alpha) (0.0<alpha<1.0)
- *rseed* - [integer] random seed

Output

- *ci* - (cimin, cimax) confidence interval

Example:

```
>>> from numpy import *
>>> from mlpy import *
>>> x = array([1,2,4,3,2,2,1,1,2,3,4,3,2])
>>> percentile_ci_median(x, nboot = 100)
(1.8461538461538463, 2.8461538461538463)
```

15.2 Peaks Detection

mlpy.**span_pd**(*x*, *span*)

span peaks detection.

Input

- *x* - [1D numpy array float] data
- *span* - [odd int] span

Output

- *idx* - [1D numpy array integer] peaks indexes

New in version 2.0.7.

15.3 Functions from GSL

mlpy.**gamma**(*x*)
Gamma Function.

Input

- *x* - [float] data

Output

- *gx* - [float] gamma(x)

mlpy.**fact**(*x*)
Factorial x!. The factorial is related to the gamma function by x! = gamma(x+1)

Input

- *x* - [int] data

Output

- *fx* - [float] factorial x!

mlpy.**quantile**(*x*, *f*)
Quantile value of sorted data. The elements of the array must be in ascending numerical order. The quantile is determined by the f, a fraction between 0 and 1. The quantile is found by interpolation, using the formula: quantile = (1 - delta) x_i + delta x_{i+1} where i is floor((n - 1)f) and delta is (n-1)f - i.

Input

- *x* - [1D numpy array float] sorted data
- *f* - [float] fraction between 0 and 1

Output

- *q* - [float] quantile

mlpy.**cdf_gaussian_P**(*x*, *sigma*)
Cumulative Distribution Functions (CDF) P(x) for the Gaussian distribution.

Input

- *x* - [float] data
- *sigma* - [float] standard deviation

Output

- *p* - [float]

New in version 2.0.2.

15.4 Other

mlpy.**away**(*a*, *b*, *d*)
Given numpy 1D array *a* and numpy 1D array *b* compute *c* = { bi : | bi - aj | > d for each i, j}

Input

- *a* - [1D numpy array float]
- *b* - [1D numpy array float]

- *d* - [double]

Output

- *c* - [1D numpy array float]

New in version 2.0.3.

mlpy.**is_power**(*n*, *b*)
Return True if 'n' is power of 'b', False otherwise.

New in version 2.0.6.

mlpy.**next_power**(*n*, *b*)
Returns the smallest integer, greater than or equal to 'n' which can be obtained as power of 'b'.

New in version 2.0.6.

CHAPTER

SIXTEEN

TOOLS

16.1 Landscaping and Parameter Tuning

mlpy includes executable scripts to be used off-the-shelf for landscaping and parameter tuning tasks. The classification and optionally feature ranking operations are organized in a sampling procedure (k-fold or Monte Carlo cross validation).

- **svm-landscape**: landscaping and regularization parameter (*C*) tuning
- **fda-landscape**: landscaping and regularization parameter (*C*) tuning
- **srda-landscape**: landscaping and alpha parameter (*alpha*) tuning
- **pda-landscape**: landscaping and number of regressions parameter (*Nreg*) tuning
- **dlda-landscape**
- **nn-landscape**: landscaping

Error (*mlpy.err()*), Matthews Correlation Coefficient (*mlpy.mcc()*) and optionally Canberra Distance (*mlpy.canberra()*) are retrieved at each parameter step.

mlpy includes executable scripts to be used exclusively for parameter tuning tasks:

- **irelief-sigma**: kernel width parameter (*sigma*) tuning

In order to print help message:

```
$ command --help
```

16.2 Other Tools

borda

Compute Borda Count, Extraction Indicator, Mean Position Indicator from a text file containing feature lists.

canberra

Compute mean Canberra distance indicator on top-k sublists from a text file containing feature lists and one containing the top-k positions.

In order to print help message:

```
$ command --help
```

16.2.1 The Feature Lists File

The feature lists file is a plain text TAB-separated file where each row is a feature ranking (a feature list).

Example:

```
feat6 [TAB] feat2 [TAB] ... [TAB] feat1
feat4 [TAB] feat1 [TAB] ... [TAB] feat7
feat4 [TAB] feat9 [TAB] ... [TAB] feat3
feat2 [TAB] feat3 [TAB] ... [TAB] feat9
feat8 [TAB] feat4 [TAB] ... [TAB] feat2
```

genindex

BIBLIOGRAPHY

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

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

[Senin08] Pavel Senin. Dynamic Time Warping Algorithm Review

[Keogh01] Eamonn J. Keogh and Michael J. Pazzani. Derivative Dynamic Time Warping. First SIAM International Conference on Data Mining (SDM 2001), 2001.

[Sakoe78] Hiroaki Sakoe and Seibi Chiba. Dynamic Programming Algorithm Optimization for Spoken Word Recognition. IEEE Transactions on Acoustics, Speech, and Signal Processing. Volume 26, 1978.

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

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

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

[Merler06] S Merler and G Jurman. Terminated Ramp - Support Vector Machine: a nonparametric data dependent kernel. Neural Network, 19:1597-1611, 2006.

[Nasr09] 18. Nasr, S. Swamidass, and P. Baldi. Large scale study of multiplemolecule queries. Journal of Cheminformatics, vol. 1, no. 1, p. 7, 2009.

[Mika01] S Mika and A Smola and B Scholkopf. An improved training algorithm for kernel fisher discriminants. Proceedings AISTATS 2001, 2001.

[Cristianini02] N Cristianini, J Shawe-Taylor and A Elisseeff. On Kernel-Target Alignment. Advances in Neural Information Processing Systems, Volume 14, 2002.

[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.

[Ghosh03] D Ghosh. Penalized discriminant methods for the classification of tumors from gene expression data. Biometrics on Volume 59, Dec. 2003 Page(s):992 - 1000(9).

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

[Tibshirani96] Robert Tibshirani. Regression shrinkage and selection via the lasso. J. Royal. Statist. Soc B., 1996, volume 58, number 1, pages 267-288.

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

[Sun08] Yijun Sun, S. Todorovic, and S. Goodison. A Feature Selection Algorithm Capable of Handling Extremely Large Data Dimensionality. In Proc. 8th SIAM International Conference on Data Mining (SDM08), pp. 530-540, April 2008.

[Subramani06] P Subramani, R Sahu and S Verma. Feature selection using Haar wavelet power spectrum. In BMC Bioinformatics 2006, 7:432.

[Guyon02] Isabelle Guyon, Jason Weston, Stephen Barnhill, Vladimir Vapnik. Gene Selection for Cancer Classification using Support Vector Machines, Machine Learning, v.46 n.1-3, p.389-422, 2002.

[Furlanello03] C Furlanello, M Serafini, S Merler, and G Jurman. Advances in Neural Network Research: IJCNN 2003, chapter An accelerated procedure for recursive feature ranking on microarray data. Elsevier, 2003.

[Louw06] N Louw and S J Steel. Variable selection in kernel Fisher discriminant analysis by means of recursive feature elimination. Computational Statistics & Data Analysis, Volume 51 Issue 3 Pages 2043-2055, 2006.

[Jurman08] G Jurman, S Merler, A Barla, S Paoli, A Galea, and C Furlanello. Algebraic stability indicators for ranked lists in molecular profiling. Bioinformatics, 24(2):258-264, 2008.

[Borda1781] J C Borda. Mémoire sur les élections au scrutin. Histoire de l'Académie Royale des Sciences, 1781.

PYTHON MODULE INDEX

m

INDEX