MLBase Documentation

Release 0.4.0

Dahua Lin

July 20, 2014

Contents

MLBase.jl is a Julia package that provides useful tools for machine learning applications. It can be considered as a *Swiss knife* for you when you are writing machine learning codes.

Dependencies:

- Reexport: to support name reexport
- StatsBase: all names in StatsBase are reexported
- ArrayViews: `view` is reexported
- Iterators: to support grid search

Contents:

CHAPTER 1

Data Preprocessing Utilities

The package provide a variety of functions for data preprocessing.

1.1 Data Repetition

repeach(*a*, *n*)
Repeat each element in vector `a` for `n` times. Here `n` can be either a scalar or a vector with the same length as `a`.

```
using MLBase

repeach(1:3, 2) # --> [1, 1, 2, 2, 3, 3]
repeach(1:3, [3,2,1]) # --> [1, 1, 1, 2, 2, 3]
```

repeachcol(*a*, *n*)
Repeat each column in matrix `a` for `n` times. Here `n` can be either a scalar or a vector with `length(n) == size(a,2)`.

repeachrow(*a*, *n*)
Repeat each row in matrix `a` for `n` times. Here `n` can be either a scalar or a vector with `length(n) == size(a,1)`.

1.2 Label Processing

In machine learning, we often need to first attach each class with an integer label. This package provides a type `LabelMap` that captures the association between discrete values (*e.g* a finite set of strings) and integer labels.

Together with `LabelMap`, the package also provides a function `labelmap` to construct the map from a sequence of discrete values, and a function `labelencode` to map discrete values to integer labels.

```
julia> lm = labelmap(["a", "a", "b", "b", "c"])
LabelMap (with 3 labels):
[1] a
[2] b
[3] c

julia> labelencode(lm, "b")
2

julia> labelencode(lm, ["a", "c", "b"])
3-element Array{Int64,1}:
```

```
 1
 3
 2
```

Note that `labelencode` can be applied to either single value or an array.

The package also provides a function `groupindices` to group indices based on associated labels.

```
julia> groupindices(3, [1, 1, 1, 2, 2, 3, 2])
3-element Array{Array{Int64,1},1}:
 [1,2,3]
 [4,5,7]
 [6]

 # using lm as constructed above
julia> groupindices(lm, ["a", "a", "c", "b", "b"])
3-element Array{Array{Int64,1},1}:
 [1,2]
 [4,5]
 [3]
```

CHAPTER 2

Classification

A classification procedure, no matter how sophisticated it is, generally consists of two steps: (1) assign a score/distance to each class, and (2) choose the class that yields the highest score/lowest distance.

This package provides a function `classify` and its friends to accomplish the second step, that is, to predict labels based on scores.

`classify` (*x*[, *ord*])

Classify based on scores given in `x` and the order of scores specified in `ord`.

Generally, `ord` can be any instance of type `Ordering`. However, it usually enough to use either `Forward` or `Reverse`:

- `ord = Forward`: higher value indicates better match (*e.g.*, similarity)
- `ord = Reverse`: lower value indicates better match (*e.g.*, distances)

When `ord` is omitted, it is defaulted to `Forward`.

When `x` is a vector, it produces an integer label. When `x` is a matrix, it produces a vector of integers, each for a column of `x`.

```
classify([0.2, 0.5, 0.3])  # --> 2
classify([0.2, 0.5, 0.3], Forward)  # --> 2
classify([0.2, 0.5, 0.3], Reverse)  # --> 1

classify([0.2 0.5 0.3; 0.7 0.6 0.2]') # --> [2, 1]
classify([0.2 0.5 0.3; 0.7 0.6 0.2]', Forward) # --> [2, 1]
classify([0.2 0.5 0.3; 0.7 0.6 0.2]', Reverse) # --> [1, 3]
```

`classify!(r, x[, ord])`

Write predicted labels to `r`.

`classify_withscore` (*x*[, *ord*])

Return a pair as `(label, score)`, where `score` is the input score corresponding to the predicted label.

`classify_withscores` (*x*[, *ord*])

This function applies to a matrix `x` comprised of multiple samples (each being a column). It returns a pair `(labels, scores)`.

`classify_withscores!(r, s, x[, ord])`

Write predicted labels to `r` and corresponding scores to `s`.

CHAPTER 3

Performance Evaluation

This package provides tools to assess the performance of a machine learning algorithm.

3.1 Classification Performance

correctrate(*gt*, *pred*)
Compute correct rate of predictions given by `pred` w.r.t. the ground truths given in `gt`.

errorrate(*gt*, *pred*)
Compute error rate of predictions given by `pred` w.r.t. the ground truths given in `gt`.

confusmat(*k*, *gt*, *pred*)
Compute the confusion matrix of the predictions given by `pred` w.r.t. the ground truths given in `gt`. Here, `k` is the number of classes.

It returns an integer matrix `R` of size `(k, k)`, such that `R(i, j) == countnz((gt .== i) & (pred .== j))`.

Examples:

```
julia> gt = [1, 1, 1, 2, 2, 2, 3, 3];

julia> pred = [1, 1, 2, 2, 2, 3, 3, 3];

julia> C = confusmat(3, gt, pred)   # compute confusion matrix
3x3 Array{Int64,2}:
 2  1  0
 0  2  1
 0  0  2

julia> C ./ sum(C, 2)   # normalize per class
3x3 Array{Float64,2}:
 0.666667  0.333333  0.0
 0.0       0.666667  0.333333
 0.0       0.0       1.0

julia> trace(C) / length(gt)  # compute correct rate from confusion matrix
0.75

julia> correctrate(gt, pred)
0.75
```

3.2 Hit rate (for retrieval tasks)

hitrate(*gt*, *ranklist*, *k*)
Compute the hitrate of rank `k` for a ranked list of predictions given by `ranklist` w.r.t. the ground truths given in `gt`.

Particularly, if `gt[i]` is contained in `ranklist[1:k, i]`, then the prediction for the `i`-th sample is said to be *hit within rank ''k''*. The hitrate of rank `k` is the fraction of predictions that hit within rank `k`.

hitrates(*gt*, *ranklist*, *ks*)
Compute hit-rates of multiple ranks (as given by a vector `ks`). It returns a vector of hitrates `r`, where `r[i]` corresponding to the rank `ks[i]`.

Note that computing hit-rates for multiple ranks jointly is more efficient than computing them separately.

3.3 Receiver Operating Characteristics (ROC)

Receiver Operating Characteristics (ROC) is often used to measure the performance of a detector, thresholded classifier, or a verification algorithm.

3.3.1 The ROC Type

This package uses an immutable type `ROCNums` defined below to capture the ROC of an experiment:

```
immutable ROCNums{T<:Real}
    p::T    # positive in ground-truth
    n::T    # negative in ground-truth
    tp::T   # correct positive prediction
    tn::T   # correct negative prediction
    fp::T   # (incorrect) positive prediction when ground-truth is negative
    fn::T   # (incorrect) negative prediction when ground-truth is positive
end
```

One can compute a variety of performance measurements from an instance of `ROCNums` (say `r`):

true_positive(*r*)
the number of true positives (`r.tp`)

true_negative(*r*)
the number of true negatives (`r.tn`)

false_positive(*r*)
the number of false positives (`r.fp`)

false_negative(*r*)
the number of false negatives (`r.fn`)

true_postive_rate(*r*)
the fraction of positive samples correctly predicted as positive, defined as `r.tp / r.p`

true_negative_rate(*r*)
the fraction of negative samples correctly predicted as negative, defined as `r.tn / r.n`

false_positive_rate(*r*)
the fraction of negative samples incorrectly predicted as positive, defined as `r.fp / r.n`

false_negative_rate(*r*)
the fraction of positive samples incorrectly predicted as negative, defined as `r.fn / r.p`

recall(*r*)
Equivalent to `true_positive_rate(r)`.

precision(*r*)
the fraction of positive predictions that are correct, defined as `r.tp / (r.tp + r.fp)`.

f1score(*r*)
the harmonic mean of `recall(r)` and `precision(r)`.

3.3.2 Computing ROC Curves

The package provides a function `roc` to compute an instance of `ROCNums` or a sequence of such instances from predictions.

roc(*gt*, *pred*)
Compute an ROC instance based on ground-truths given in `gt` and predictions given in `pred`.

roc(*gt*, *scores*, *thres*[, *ord*])
Compute an ROC instance or an ROC curve (a vector of `ROC` instances), based on given scores and a threshold `thres`.

Prediction will be made as follows:

- When `ord = Forward`: predicts `1` when `scores[i] >= thres` otherwise 0.
- When `ord = Reverse`: predicts `1` when `scores[i] <= thres` otherwise 0.

When `ord` is omitted, it is defaulted to `Forward`.

Returns:

- When `thres` is a single number, it produces a single `ROCNums` instance;
- When `thres` is a vector, it produces a vector of `ROCNums` instances.

Note: Jointly evaluating an ROC curve for multiple thresholds is generally much faster than evaluating for them individually.

roc(*gt*, *(preds*, *scores)*, *thres*[, *ord*])
Compute an ROC instance or an ROC curve (a vector of `ROC` instances) for multi-class classification, based on given predictions, scores and a threshold `thres`.

Prediction is made as follows:

- When `ord = Forward`: predicts `preds[i]` when `scores[i] >= thres` otherwise 0.
- When `ord = Reverse`: predicts `preds[i]` when `scores[i] <= thres` otherwise 0.

When `ord` is omitted, it is defaulted to `Forward`.

Returns:

- When `thres` is a single number, it produces a single `ROCNums` instance.
- When `thres` is a vector, it produces an ROC curve (a vector of `ROCNums` instances).

Note: Jointly evaluating an ROC curve for multiple thresholds is generally much faster than evaluating for them individually.

roc(*gt*, *scores*, *n*[, *ord*])
Compute an ROC curve (a vector of `ROC` instances), with respect to `n` evenly spaced thresholds from `minimum(scores)` and `maximum(scores)`. (See above for details)

roc(*gt*, *(preds*, *scores)*, *n*[, *ord*])
Compute an ROC curve (a vector of `ROC` instances) for multi-class classification, with respect to `n` evenly spaced thresholds from `minimum(scores)` and `maximum(scores)`. (See above for details)

roc(*gt, scores, ord]*)
Equivalent to `roc(gt, scores, 100, ord)`.

roc(*gt, (preds, scores), ord]*)
Equivalent to `roc(gt, (preds, scores), 100, ord)`.

roc(*gt*, *scores*)
Equivalent to `roc(gt, scores, 100, Forward)`.

roc(*gt*, *(preds*, *scores)*)
Equivalent to `roc(gt, (preds, scores), 100, Forward)`.

CHAPTER 4

Cross Validation

This package implements several cross validation schemes: `Kfold`, `LOOCV`, and `RandomSub`. Each scheme is an iterable object, of which each element is a vector of indices (indices of samples selected for training).

4.1 Cross Validation Schemes

`Kfold`(*n*, *k*)

`k`-fold cross validation over a set of `n` samples, which are randomly partitioned into `k` disjoint validation sets of nearly the same sizes. This generates `k` training subsets of length about `n*(1-1/k)`.

```
julia> collect(Kfold(10, 3))
3-element Array{Any,1}:
 [1,3,4,6,7,8,10]
 [2,5,7,8,9,10]
 [1,2,3,4,5,6,9]
```

`StratifiedKfold`(*strata*, *k*)

Like `Kfold`, but indexes in each strata (defined by unique values of an iterator *strata*) are distributed approximately equally across the `k` folds. Each strata should have at least `k` members.

```
julia> collect(StratifiedKfold([:a, :a, :a, :b, :b, :c, :c, :a, :b, :c], 3))
3-element Array{Any,1}:
 [1,2,4,6,8,9,10]
 [3,4,5,7,8,10]
 [1,2,3,5,6,7,9]
```

`LOOCV`(*n*)

Leave-one-out cross validation over a set of `n` samples.

```
julia> collect(LOOCV(4))
4-element Array{Any,1}:
 [2,3,4]
 [1,3,4]
 [1,2,4]
 [1,2,3]
```

`RandomSub`(*n*, *sn*, *k*)

Repetitively random subsampling. Particularly, this generates `k` subsets of length `sn` from a data set with `n` samples.

```
julia> collect(RandomSub(10, 5, 3))
3-element Array{Any,1}:
```

```
 [1,2,5,8,9]
 [2,5,7,8,10]
 [1,3,5,6,7]
```

StratifiedRandomSum(*strata*, *sn*, *k*)
Like `RandomSub`, but indexes in each strata (defined by unique values of an iterator *strata*) are distributed approximately equally across the `k` subsets. `sn` should be greater than the number of strata, so that each stratum can be represented in each subset.

```
julia> collect(StratifiedRandomSub([:a, :a, :a, :b, :b, :c, :c, :a, :b, :c], 7, 5))
5-element Array{Any,1}:
 [1,2,3,4,6,7,9]
 [1,3,4,6,8,9,10]
 [1,3,5,7,8,9,10]
 [1,2,4,7,8,9,10]
 [1,2,3,4,5,6,10]
```

4.2 Cross Validation Function

The package also provides a function `cross_validate` as below to run a cross validation procedure.

cross_validate(*estfun*, *evalfun*, *n*, *gen*)
Run a cross validation procedure.

Parameters

- **estfun** – The estimation function, which takes a vector of training indices as input and returns a learned model, as:

  ```
  model = estfun(train_inds)
  ```

- **evalfun** – The evaluation function, which takes a model and a vector of testing indices as input and returns a score that indicates the goodness of the model, as

  ```
  score = evalfun(model, test_inds)
  ```

- **n** – The total number of samples.
- **gen** – An iterable object that provides training indices, *e.g.*, one of the cross validation schemes listed above.

Returns a vector of scores obtained in the multiple runs.

Example:

```
# A simple example to demonstrate the use of cross validation
#
# Here, we consider a simple model: using a mean vector to represent
# a set of samples. The goodness of the model is assessed in terms
# of the RMSE (root-mean-square-error) evaluated on the testing set
#

using MLBase

# functions
compute_center(X::Matrix{Float64}) = vec(mean(X, 2))
```

```
compute_rmse(c::Vector{Float64}, X::Matrix{Float64}) =
    sqrt(mean(sum(abs2(X .- c),1)))

# data
const n = 200
const data = [2., 3.] .+ randn(2, n)

# cross validation
scores = cross_validate(
    inds -> compute_center(data[:, inds]),        # training function
    (c, inds) -> compute_rmse(c, data[:, inds]),  # evaluation function
    n,              # total number of samples
    Kfold(n, 5))    # cross validation plan: 5-fold

# get the mean and std of the scores
(m, s) = mean_and_std(scores)
```

Please refer to `examples/crossval.jl` for the entire script.

CHAPTER 5

Model Tuning

Many machine learning algorithms and models come with design parameters that need to be set in advance. A widely adopted pratice is to search the parameters (usually through brute-force loops) that yields the best performance over a validation set. The package provides functions to facilitate this.

gridtune (*estfun*, *evalfun*, *params...; ...*)

Search the best setting of parameters over a Cartesian grid (*i.e.* all combinations of parameters).

Parameters

- **estfun** – The model estimation function that takes design parameters as input and produces the model.
- **evalfun** – The function that evaluates the model, producing a score value.
- **params** – A series of parameters, given in the form of `(param_name, param_values)`.

Returns a 3-tuple, as `(best_model, best_cfg, best_score)`. Here, `best_cfg` is a tuple comprised of the parameters in the best setting (the one that yields the best score).

Keyword arguments:

- `ord`: It may take either of `Forward` or `Reverse`:
  - `ord=Forward`: higher score value indicates better model (default)
  - `ord=Reverse`: lower score value indicates better model.
- `verbose`: boolean, whether to show progress information. (default = `false`).

Note: For some learning algorithms, there may be some constraint of the parameters (*e.g* one parameter must be smaller than another, etc). If a certain combination of parameters is not valid, the `estfun` may return nothing, in which case, the function would ignore those particular settings.

Example:

```
using MLBase
using MultivariateStats

## prepare data

n_tr = 20  # number of training samples
n_te = 10  # number of testing samples
d = 5      # dimension of observations

theta = randn(d)
X_tr = randn(n_tr, d)
```

```
y_tr = X_tr * theta + 0.1 * randn(n_tr)
X_te = randn(n_te, d)
y_te = X_te * theta + 0.1 * randn(n_te)

## tune the model

function estfun(regcoef, bias)
    s = ridge(X_tr, y_tr, regcoef; bias=bias)
    return bias ? (s[1:end-1], s[end]) : (s, 0.0)
end

evalfun(m) = msd(X_te * m[1] + m[2], y_te)

r = gridtune(estfun, evalfun,
             ("regcoef", [1.0e-3, 1.0e-2, 1.0e-1, 1.0]),
             ("bias", (true, false));
             ord=Reverse,    # smaller msd value indicates better model
             verbose=true)   # show progress information

best_model, best_cfg, best_score = r

## print results

a, b = best_model
println("Best model:")
println("  a = $(a')"),
println("  b = $b")
println("Best config: regcoef = $(best_cfg[1]), bias = $(best_cfg[2])")
println("Best score: $(best_score)")
```