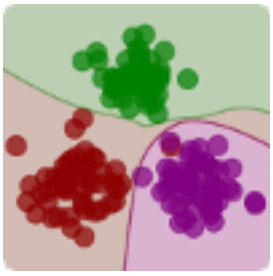

MLDataUtils.jl Documentation

Release v0.1

Christof Stocker, Tom Breloff, and others

November 27, 2016

Contents

This package represents a community effort to provide common functionality to generate, load, split, and process Machine Learning datasets in Julia. As such, it is a part of the JuliaML ecosystem. In contrast to other data-centered packages, MLDataUtils focuses specifically on functionality utilized in a Machine Learning context.

If this is the first time you consider using MLDataUtils, make sure to check out the "Getting Started" section; specifically "How to ...?".

CHAPTER 1

Installation

To install MLDataUtils.jl, start up Julia and type the following code-snipped into the REPL. It makes use of the native Julia package manger.

```
Pkg.add("MLDataUtils")
```

Additionally, for example if you encounter any sudden issues, or in the case you would like to contribute to the package, you can manually choose to be on the latest (untagged) version.

```
Pkg.checkout("MLDataUtils")
```

CHAPTER 2

Getting Started

MLDataUtils is the result of a collaborative effort to design an efficient but also convenient implementation for many of the commonly used data-related subsetting and pre-processing patterns.

Aside from providing common functionality, this library also defines a set of common interfaces and functions, that can (and should) be extended to work with custom user-defined data structures.

2.1 Hello World

This package is registered in the Julia package ecosystem. Once installed the package can be imported just as any other Julia package.

```
using MLDataUtils
```

Let us take a look at a hello world example (with little explanation) to get a feeling for how to use this package in a typical ML scenario. It is a common requirement in machine learning related experiments to partition the dataset of interest in one way or the other.

```
# X is a matrix of floats
# Y is a vector of strings
X, Y = load_iris()

# The iris dataset is ordered according to their labels,
# which means that we should shuffle the dataset before
# partitioning it into training and testset.
Xs, Ys = shuffleobs((X, Y))
# Notice how we use tuples to group data.

# We leave out 15 % of the data for testing
(cv_X, cv_Y), (test_X, test_Y) = splitobs((Xs, Ys); at = 0.85)

# Next we partition the data using a 10-fold scheme.
# Notice how we do not need to splat train into X and Y
for (train, (val_X, val_Y)) in kfolds((cv_X, cv_Y); k = 10)

    # Iterate over the data using mini-batches of 5 observations each
    for (batch_X, batch_Y) in eachbatch(train, size = 5)
        # ... train supervised model on minibatches here
    end
end
```

In the above code snipped, the inner loop for `eachbatch()` is the only place where data other than indices is actually being copied. That is because `cv_X`, `test_X`, `val_X`, etc. are all array views of type `SubArray` (the same applies to all the y's of course). In contrast to this, `batch_X` and `batch_y` will be of type `Array`. Naturally array views only work for arrays, but we provide a generalization of such for any type of datastorage.

Furthermore both, `batch_X` and `batch_y`, will be the same instance each iteration with only their values changed. In other words, they both are a preallocated buffers that will be reused each iteration and filled with the data for the current batch.

Naturally one is not required to work with buffers like this, as stateful iterators can have undesired sideeffects when used without care. For example `collect(eachbatch(X))` would result in an array that has the exact same batch in each position. Oftentimes though, reusing buffers is preferable. This package provides different alternatives for different use-cases.

2.2 How to ... ?

Chances are you ended up here with a very specific use-case in mind. This section outlines a number of different but common scenarios and explains how this package can be utilized to solve them.

- TODO: Split Train test (Val)
- TODO: KFold Cross-validation
- TODO: Labeled Data with inbalanced classes
- TODO: DataFrame
- TODO: GPU Arrays
- TODO: Custom Data Storage Type (ISIC)
- TODO: Custom Data Iterator (stream)

2.3 Getting Help

To get help on specific functionality you can either look up the information here, or if you prefer you can make use of Julia's native doc-system. The following example shows how to get additional information on `DataSubset` within Julia's REPL:

```
?DataSubset
```

If you find yourself stuck or have other questions concerning the package you can find us at gitter or the *Machine Learning* domain on discourse.julialang.org

- Julia ML on Gitter
- Machine Learning on Julialang

If you encounter a bug or would like to participate in the further development of this package come find us on Github.

- JuliaML/MLDataUtils.jl

While the sole focus of the whole package is on data-related functionality, we can further divide the provided types and functions into a number of quite different sub-categories.

CHAPTER 3

Data Access Pattern

The core of the package, and indeed the part that thus far received the most attention, are the data access pattern. These include data-partitioning, -subsampling, and -iteration. The main design principle behind the access pattern is based on the assumption that the data a user is working with is likely of some very user-specific custom type. That said, there was also a lot of attention put into first class support for those types that are most commonly employed to represent the data of interest, such as `Array`.

3.1 Data Subsetting

It is a common requirement in machine learning related experiments to partition the dataset of interest in one way or the other. This section outlines the functionality that this package provides for the typical use-cases.

3.1.1 Design Decisions

One of the interesting strong points of the Julia language is its rich and developer friendly type system. As such we made it a key priority to make as little assumptions as possible about the data at hand.

3.1.2 The DataSubset Type

This package represents subsets of data as a custom type called `DataSubset`; unless a custom subset type is provided, but more on that later. The main purpose for the existence of `DataSubset` is two-fold:

1. To **delay the evaluation** of a subsetting operation until an actual batch of data is needed.
2. To **accumulate subsettings** when different data access pattern are used in combination with each other (which they usually are). (i.e.: train/test splitting -> K-fold CV -> Minibatch-stream)

This design aspect is particularly useful if the data is not located in memory, but on the harddrive or some remote location. In such a scenario one wants to load only the required data only when it is actually needed.

3.1.3 Splitting into Train and Test

Some separation strategies, such as dividing the dataset into a training- and a testset, is often performed offline or predefined by a third party. That said, it is useful to efficiently and conveniently be able to split a given dataset into differently sized subsets.

One such function that this package provides is called `splitobs()`. Note that this function does not shuffle the content, but instead performs a static split at the relative position specified in `at`.

TODO: example splitobs

For the use-cases in which one wants to instead do a completely random partitioning to create a training- and a testset, this package provides a function called *shuffleobs*. Returns a lazy "subset" of data (using all observations), with only the order of the indices permuted. Aside from the indices themseves, this is non-copy operation. Using `shuffleobs()` in combination with `splitobs()` thus results in a random assignment of data-points to the data-partitions.

TODO: example shuffleobs

3.1.4 K-Folds for Cross-validation

Yet another use-case for data partitioning is model selection; that is to determine what hyper-parameter values to use for a given problem. A particularly popular method for that is *k-fold cross-validation*, in which the dataset gets partitioned into k folds. Each model is fit k times, while each time a different fold is left out during training, and is instead used as a validation set. The performance of the k instances of the model is then averaged over all folds and reported as the performance for the particular set of hyper-parameters.

This package offers a general abstraction to perform k-fold partitioning on data sets of arbitrary type. In other words, the purpose of the type `KFolds` is to provide an abstraction to randomly partition some dataset into k disjoint folds. `KFolds` is best utilized as an iterator. If used as such, the dataset will be split into different training and test portions in k different and unqiue ways, each time using a different fold as the validation/testset.

The following code snippets showcase how the function `kfolds()` could be utilized:

TODO: example KFolds

Note: The sizes of the folds may differ by up to 1 observation depending on if the total number of observations is dividable by k.

3.1.5 Observation Dimension

3.2 Data Iterators

Other partition-needs arise from the fact that the interesting datasets are increasing in size as the scientific community continues to improve the state-of-the-art. However, bigger datasets also offer additional challenges in terms of computing resources. Luckily, there are popular techniques in place to deal with such constraints in a surprisingly effective manner. For example, there are a lot of empirical results that demonstrate the efficiency of optimization techniques that continuously update on small subsets of the data. As such, it has become a de facto standard to iterate over a given dataset in minibatches, or even just one observation at a time.

In the case that the size of the dataset is not dividable by the specified (or inferred) size, the remaining observations will be ignored.

The functions `obsview()` or `batchview()` will not shuffle the data, thus the observations within each batch/partition will in general be adjacent to each other. However, one can choose to process the batches in random order by using `shuffleobs()`

3.2.1 RandomBatches

The purpose of `RandomBatches` is to provide a generic `DataIterator` specification for labeled and unlabeled randomly sampled mini-batches that can be used as an iterator. In contrast to `BatchView`, `RandomBatches`

generates completely random mini-batches, in which the containing observations are generally not adjacent to each other in the original dataset.

The fact that the observations within each mini-batch are uniformly sampled has an important consequences. Because observations are independently sampled, it is likely that some observation(s) occur multiple times within the same mini-batch. This may or may not be an issue, depending on the use-case. In the presence of online data-augmentation strategies, this fact should usually not have any noticible impact.

The following code snippets showcase how `RandomBatches` could be utilized:

3.3 Support for User Types

TODO: Only LearnBase dependency needed.

TODO: different level of information available (nobs vs only first etc)

3.3.1 Custom Data Container

For `DataSubset` (and all the data splitting functions for that matter) to work on some custom data-container-type, the desired type `MyType` must implement the following interface:

`LearnBase.`**`getobs`**(*data*, *idx*[, *obsdim*])

Parameters

- **`data`** (*`MyType`*) – The data of your custom user type. It should represent your dataset of interest and somehow know how to access observations of a specific index.
- **`idx`** – The index or indices of the observation(s) in `data` that the subset should represent. Can be of type `Int` or some subtype `AbstractVector{Int}`.
- **`obsdim`** (*`ObsDimension`*) – Support optional. If it makes sense for the type of *data*, *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a typestable manner as a positional argument.

  If support is provided, `obsdim` can take on any of the following values. Their meaning is completely up to the user.

`ObsDim.First()`	`ObsDim.Last()`	`ObsDim.Constant(N)`

Returns Should return the observation(s) indexed by `idx`. In what form is completely up to the user and can be specific to whatever task you have in mind! In other words there is **no** contract that the type of the return value has to fullfill.

`LearnBase.`**`nobs`**(*data*[, *obsdim*])

Parameters

- **`data`** (*`MyType`*) – The data of your custom user type. It should represent your dataset of interest and somehow know how many observations it contains.
- **`obsdim`** (*`ObsDimension`*) – Support optional. If it makes sense for the type of *data*, *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a typestable manner as a positional argument.

  If support is provided, `obsdim` can take on any of the following values. Their meaning is completely up to the user.

`ObsDim.First()`	`ObsDim.Last()`	`ObsDim.Constant(N)`

Returns Should return the number of observations in *data*

The following methods can also be provided and are optional:

`LearnBase.`**`getobs`**(*data*)

By default this function will be the identity function for any type of *data* that does not prove a custom method for it. If that is not the behaviour that you want for your type, you need to provide this method yourself.

Parameters **`data`** (*`MyType`*) – The data of your custom user type. It should represent your dataset of interest and somehow know how to return the full dataset.

Returns Should return all observations in `data`. In what form is completely up to the user and can be specific to whatever task you have in mind! In other words there is **no** contract that the type of the return value has to fullfill.

`LearnBase.`**`getobs!`**(*buffer*, *data*[, *idx*][, *obsdim*])

Inplace version of `getobs()`. If this method is provided for the type of `data`, then `eachobs()` and `eachbatch()` (among others) can preallocate a buffer that is then reused every iteration.

param buffer The preallocated storage to copy the given indices of data into. *Note:* The type and structure should be equivalent to the return value of `getobs()`, since this is how `buffer` is preallocated by default.

Parameters

- **`data`** (*`MyType`*) – The data of your custom user type. It should represent your dataset of interest and somehow know how to access observations of a specific index, and how to store those observation(s) into `buffer`.
- **`idx`** – The index or indices of the observation(s) in `data` that the subset should represent. Can be of type `Int` or some subtype `AbstractVector{Int}`.
- **`obsdim`** (*`ObsDimension`*) – Support optional. If it makes sense for the type of *data*, *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a typestable manner as a positional argument.

  If support is provided, `obsdim` can take on any of the following values. Their meaning is completely up to the user.

`ObsDim.First()`	`ObsDim.Last()`	`ObsDim.Constant(N)`

DataFrames.jl

3.3.2 Custom Data Subset

`LearnBase.`**`datasubset`**(*data*, *idx*[, *obsdim*])

If your custom type has its own kind of subset type, you can return it here. An example for such a case are *SubArray* for representing a subset of some *AbstractArray*. Note: If your type has no use for *obsdim* then dispatch on *::ObsDim.Undefined* in the signature.

3.3.3 Custom Data Iterator

CHAPTER 4

Data Processing

This package contains a number of simple pre-processing strategies that are often applied for ML purposes, such as feature centering and rescaling.

4.1 Feature Normalization

Note: This section will likely be subject to larger changes and/or redesigns. For example none of these function are of yet adapted to work with `ObsDimension`

This package contains a simple model called `FeatureNormalizer`, that can be used to normalize training and test data with the parameters computed from the training data.

```
x = collect(-5:.1:5)
X = [x x.^2 x.^3]'

# Derives the model from the given data
cs = fit(FeatureNormalizer, X)

# Normalizes the given data using the derived parameters
X_norm = predict(cs, X)
```

The underlying functions can also be used directly

4.1.1 Centering

`center!`$(X[, \mu])$

Centers each row of `X` around the corresponding entry in the vector μ. In other words performs feature-wise centering.

Parameters

- **`X`** (*`Array`*) – Feature matrix that should be centered in-place.
- μ (*`Vector`*) – Vector of means. If not specified then it defaults to `mean(X, 2)`.

Returns

Returns the parameters μ itself.

```
μ = center!(X, μ)
```

4.1.2 Rescaling

rescale! (*X*[, *μ*][, *σ*])

Centers each row of `X` around the corresponding entry in the vector μ and then rescaled using the corresponding entry in the vector σ.

Parameters

- **X** (*Array*) – Feature matrix that should be centered and rescaled in-place.
- μ (*Vector*) – Vector of means. If not specified then it defaults to `mean(X, 2)`.
- σ (*Vector*) – Vector of standard deviations. If not specified then it defaults to `std(X, 2)`.

Returns

Returns the parameters μ and σ itself.

```
μ, σ = rescale!(X, μ, σ)
```

4.1.3 Basis Expansion

expand_poly (*x*[, *degree*])

Performs a polynomial basis expansion of the given *degree* for the vector *x*.

Parameters

- **x** (*Vector*) – Feature vector that should be expanded.
- **degree** (*Int*) – The number of polynomes that should be augmented into the resulting matrix `X`

Returns

Result of the expansion. A matrix of size *(degree, length(x))*. Note that all the features of `X` are centered and rescaled.

```
X = expand_poly(x; degree = 5)
```

CHAPTER 5

Data Generators

When studying learning algorithm or other ML related functionality, it is usually of high interest to empirically test the behaviour of the system under specific conditions. Generators can provide the means to fabricate artificial data sets that observe certain attributes, which can help to deepen the understanding of the system under investigation.

5.1 Data Generators

Note: This section may be subject of larger changes and/or redesigns. For example it is planned to absorb joshday/DataGenerator.jl

5.1.1 Noisy Function

noisy_function (*fun*, *x; noise*, *f_rand*) → Tuple

Generates a noisy response `y` for the given function `fun` by adding `noise .* f_randn(length(x))` to the result of `fun(x)`.

Parameters

- **fun** (*Function*) – The function for which one wants to generate some noisy response variables. Can be any univariate function accepting a `Float64`.
- **x** (*Vector*) – The feature vector of numbers that should be used as input for `fun(x)`. This variable will also be returned by the function for consistency with other generators.
- **noise** (*Float64*) – The scaling factor for the noise. This number will be multiplied to the output of `f_rand`.
- **f_rand** (*Function*) – The function creating the random numbers to be added as noise to the result of `fun`.

Returns

A tuple of two vectors. The first vector `x` denotes the independent variable (feature) and the second vector `y` represents a noisy estimate of the given function `fun`, which is "simulated" by adding some rescaled random numbers to its output.

```
x, y = noisy_function(fun, x; noise = 0.01, f_rand = randn)
```

5.1.2 Noisy Sin

noisy_sin(*n*, *start*, *stop; noise*, *f_rand*)

Generates `n` noisy equally spaced samples of a sinus from `start` to `stop` by adding `noise .* f_randn(length(x))` to the result of `fun(x)`.

Parameters

- **n** (*Int*) – Number of observations to generate.
- **start** (*Int*) – The lowest value used as input for `sin`
- **stop** (*Int*) – The largest value used as input for `sin`
- **noise** (*Float64*) – The scaling factor for the noise. This number will be multiplied to the output of `f_rand`.
- **f_rand** (*Function*) – The function creating the random numbers to be added as noise to the result of `sin`.

Returns

A tuple of two vectors. The first vector `x` denotes the independent variable (feature) and the second vector `y` represents a noisy estimate of `sin`, which is "simulated" by adding some rescaled random numbers to its output.

```
x, y = noisy_sin(n, start, stop; noise = 0.3, f_rand = randn)
```

5.1.3 Noisy Polynome

noisy_poly(*coef*, *x; noise*, *f_rand*)

Generates a noisy response for a polynomial of degree `length(coef)` using the vector `x` as input and adding `noise .* f_randn(length(x))` to the result.

Parameters

- **coef** (*Vector*) – Contains the coefficients for the terms of the polynome. The first element denotes the coefficient for the term with the highest degree, while the last element denotes the intercept.
- **x** (*Vector*) – The feature vector of numbers that should be used as the data for the polynome. This variable will also be returned by the function for consistency with other generators.
- **noise** (*Float64*) – The scaling factor for the noise. This number will be multiplied to the output of `f_rand`.
- **f_rand** (*Function*) – The function creating the random numbers to be added as noise to the result of the polynome.

Returns

A tuple of two vectors. The first vector `x` denotes the independent variable (feature) and the second vector `y` represents a noisy estimate of the given polynome, which is "simulated" by adding some rescaled random numbers to its output.

```
x, y = noisy_poly(coef, x; noise = 0.01, f_rand = randn)
```

CHAPTER 6

Example Datasets

We provide a small number of toy datasets. These are mainly intended for didactic and testing purposes.

6.1 Example Datasets

The package contains a few static datasets that are intended to serve as toy examples.

Note: This section may be subject of larger changes. It is possible that in the future the datasets will instead be provided by JuliaML/MLDatasets.jl instead.

6.1.1 Fisher's Iris data set

The Iris data set has become one of the most recognizable machine learning example datasets. It was originally published by Ronald Fisher *[FISHER1936]* and contains the 4 different kind of measurements (that we call features) for 150 observations of a plant called **Iris**. The interesting property of the dataset is that it includes these measurements for 3 different species of Iris (50 observations each) and is thus a dataset that is commonly used to showcase classification or clustering algorithms.

load_iris([*n*]) → Tuple

Loads the first `n` observations from the Iris flower data set introduced by Ronald Fisher (1936).

Parameters **n** (*Int*) – default `150`. Specifies how many of the total 150 observations should be returned (in their native order).

Returns

A tuple of three arrays as the following code snipped shows. The 4 by `n` matrix `X` contains the numeric measurements, in which each individual column denotes an observation. The vector `y` contains the class labels as strings. The optional vector `names` contains the names of the features (i.e. rows of `X`)

```
X, y, names = load_iris(n)
```

Check out the wikipedia entry for more information about the dataset.

6.1.2 Noisy Line Example

This refers to a static pre-defined toy dataset. In order to generate a noisy line using some parameters take a look at *noisy_function()*.

load_line() → Tuple
Loads an artificial example dataset for a noisy line. It is particularly useful to explain under- and overfitting.

Returns

The vector `x` contains 11 equally spaced points between 0 and 1. The vector `y` contains `x ./ 2 + 1` plus some gaussian noise. The optional vector `names` contains descriptive names for `x` and `y`.

```
x, y, names = load_line()
```

6.1.3 Noisy Sin Example

This refers to a static pre-defined toy dataset. In order to generate a noisy sin using some parameters take a look at *noisy_sin()*.

load_sin() → Tuple
Loads an artificial example dataset for a noisy sin. It is particularly useful to explain under- and overfitting.

Returns

The vector `x` contains equally spaced points between 0 and 2π. The vector `y` contains `sin(x)` plus some gaussian noise. The optional vector `names` contains descriptive names for `x` and `y`.

```
x, y, names = load_sin()
```

6.1.4 Noisy Polynome Example

This refers to a static pre-defined toy dataset. In order to generate a noisy polynome using some parameters take a look at *noisy_poly()*.

load_poly() → Tuple
Loads an artificial example dataset for a noisy quadratic function.

Returns

It is particularly useful to explain under- and overfitting. The vector `x` contains 50 points between 0 and 4. The vector `y` contains `2.6 * x^2 + .8 * x` plus some gaussian noise. The optional vector `names` contains descriptive names for `x` and `y`.

```
x, y, names = load_poly()
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

7.1 Acknowledgements

7.2 LICENSE

[FISHER1936] Fisher, Ronald A. “The use of multiple measurements in taxonomic problems.” Annals of eugenics 7.2 (1936): 179-188.

C

E

L

N

R