MLDataPattern.jl Documentation

Release 0.1

Christof Stocker, Tom Breloff, and others

Nov 30, 2020

Contents

This package represents a community effort to provide a native and generic Julia implementation for commonly used **data access pattern** in Machine Learning. Most notably it provides a number of pattern for *shuffling*, *partitioning*, and *resampling* data sets of various types and origin. At its core, the package was designed around the key requirement of allowing any user-defined type to serve as a custom data source and/or access pattern in a first class manner.

In contrast to other data-related Julia packages, the focus of MLDataPattern is specifically on functionality utilized in a machine learning context. As such, it is a part of the JuliaML ecosystem.

CHAPTER 1

Where to begin?

If this is the first time you consider using MLDataPattern for your machine learning related experiments or packages, make sure to check out the "Getting Started" section. It will provide a very condensed overview of all the topics outlined below. If you are looking to perform some specific task, then take a look at "How to ...?", which lists some of most common scenarios and links to the appropriate places that should guide you on how to approach these scenarios using the functionality provided by this or other packages.

1.1 Getting Started

Typical machine learning experiments require a lot of rather mundane but error prone data handling glue code. One particularly interesting category of data handling functionality - and the sole focus of this package - are what we call **data access pattern**. These "pattern" include *subsetting*, *resampling*, *iteration*, and *partitioning* of various types of data sets.

MLDataPattern was designed around the core requirment of providing first class support for user-defined data sources. This idea is based on the assumption that the data source a user is working with, is likely of some very user-specific custom type. That said, we also put a lot of attention into first class support for the most commonly used data containers, such as `Array`.

In this section we will provide a condensed overview of the package functionality. We will cover the topics in about the same relative order as the main documentation. To keep this overview concise, we will not discuss anything here in detail. We will, however, link to the appropriate places. Please take a look at the corresponding sections for more information about a function or concept.

1.1.1 Installation

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

```
Pkg.add("MLDataPattern")
```

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("MLDataPattern")
```

1.1.2 Overview

If there is one requirement that almost all machine learning experiments have in common, it is that they have to interact with "data" in one way or the other. After all, the goal is for a program to learn from the implicit information contained in that data. Consequently, it is of no surprise that over time a number of particularly useful pattern emerged for how to utilize this data effectively. For instance, we learned that we should leave a subset of the available data out of the training process in order to spot and subsequently prevent over-fitting.

In the context of this package we differentiate between two categories of data sources based on some useful properties. A "data source", by the way, is simply any Julia type that can provide data. We need not be more precise with this definition, since it is of little practical consequence. The definitions that matter are for the two sub-categories of data sources that this package can actually interact with: **Data Containers** and **Data Iterators**. These abstractions will allow us to interact with many different types of data using a coherent and non-invasive interface.

Data Container For a data source to belong in this category it needs to be able to provide two things:

1. The total number of observations N, that the data source contains.
2. A way to query a specific observation or sequence of observations. This must be done using indices, where every observation has a unique index $i \in I$ assigned from the set of indices $I = \{1, 2, ..., N\}$.

Data Iterator To belong to this group, a data source must implement Julia's iterator interface. The data source may or may not know the total amount of observations it can provide, which means that knowing N is not necessary.

The key requirement for a iteration-based data source is that every iteration consistently returns either a single observation or a batch of observations.

The more flexible of the two categories are what we call data containers. A good example for such a type is a plain Julia `Array`. Well, almost. To be considered a data container, the type has to implement the required interface. In particular, a data container has to implement the functions *`getobs()`* and *`nobs()`*. For convenience both of those implementations are already provided for `Array` out of the box. Thus on package import the type `Array` becomes a data container type. For more details on the required interface take a look at the section on *Data Container*.

Let us take a look at a few examples to get a feeling of how one can use this library. This package is registered in the Julia package ecosystem. Once installed the package can be imported as usual.

```
using MLDataPattern
```

Consider the following toy feature matrix `X`, which has 2 rows and 6 columns. We can use *`nobs()`* to query the number of observations it contains, and *`getobs()`* to query one or more specific observation(s).

```
julia> X = rand(2, 6)
2×6 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996

julia> nobs(X)
6

julia> getobs(X, 2) # query the second observation
2-element Array{Float64,1}:
 0.933372
```

(continues on next page)

(continued from previous page)

```
 0.522172

julia> getobs(X, [4, 1]) # create a batch with observation 4 and 1
2×2 Array{Float64,2}:
 0.0443222  0.226582
 0.722906   0.504629
```

As you may have noticed, the two functions make a pretty strong assumption about how to interpret the shape of `X`. In particular, they assume that each column denotes a single observation. This may not be what we want. Given that `X` has two dimensions that we could assign meaning to, we should have the opportunity to choose which dimension enumerates the observations. After all, we can think of `X` as a data container that has 6 observations with 2 features each, or as a data container that has 2 observations with 6 features each. To allow for that choice, all relevant functions accept the optional parameter `obsdim`. For more information take a look at the section on *Observation Dimension*.

```
julia> nobs(X, obsdim = 1)
2

julia> getobs(X, 2, obsdim = 1)
6-element Array{Float64,1}:
 0.504629
 0.522172
 0.0997825
 0.722906
 0.245457
 0.000341996
```

Every data container can be subsetted manually using the low-level function *`datasubset()`*. Its signature is identical to *`getobs()`*, but instead of copying the data it returns a lazy subset. A lot of the higher-level functions use *`datasubset()`* internally to provide their functionality. This allows for delaying the actual data access until the data is actually needed. For arrays the returned subset is in the form of a `SubArray`. For more information take a look at the section on *Data Subsets*.

```
julia> datasubset(X, 2)
2-element SubArray{Float64,1,Array{Float64,2},Tuple{Colon,Int64},true}:
 0.933372
 0.522172

julia> datasubset(X, [4, 1])
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.0443222  0.226582
 0.722906   0.504629

julia> datasubset(X, 2, obsdim = 1)
6-element SubArray{Float64,1,Array{Float64,2},Tuple{Int64,Colon},true}:
 0.504629
 0.522172
 0.0997825
 0.722906
 0.245457
 0.000341996
```

Note that a data subset doesn't strictly have to be a true "subset" of the data set. For example, the function *`shuffleobs()`* returns a lazy data subset, which contains exactly the same observations, but in a randomly permuted order.

```
julia> shuffleobs(X)
2×6 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.0443222  0.812814  0.226582  0.11202      0.505208   0.933372
 0.722906   0.245457  0.504629  0.000341996  0.0997825  0.522172
```

Since this function is non-deterministic, it raises the question of what to do when our data set is made up of multiple variables. It is not uncommon, for example, that the targets of a labeled data set are stored in a separate `Vector`. To support such a scenario, all relevant functions also accept a `Tuple` as the data argument. If that is the case, then all elements of the given tuple will be processed in the exact same manner. The return value will then again be a tuple with the individual results. As you can see in the following code snippet, the observation-link between `x` and `y` is preserved after the shuffling. For more information about grouping data containers in a `Tuple`, take a look at the section on *Tuples and Labeled Data*.

```
julia> x = collect(1:6);

julia> y = [:a, :b, :c, :d, :e, :f];

julia> xs, ys = shuffleobs((x, y))
([6,1,4,5,3,2],Symbol[:f,:a,:d,:e,:c,:b])
```

A common requirement in a machine learning experiment is to split the data set into a training and a test portion. While we could already do this manually using *`datasubset()`*, this package also provides a high-level convenience function *`splitobs()`*.

```
julia> y1, y2 = splitobs(y, at = 0.6)
(Symbol[:a,:b,:c,:d],Symbol[:e,:f])
```

As we can see in the example above, the function *`splitobs()`* performs a static "split" of the given data at the relative position `at`, and returns the result in the form of two data subsets. It is also possible to specify multiple fractions, which will cause the function to perform additional splits.

```
julia> y1, y2, y3 = splitobs(y, at = (0.5, 0.3))
(Symbol[:a,:b,:c],Symbol[:d,:e],Symbol[:f])
```

Of course, a simple static split isn't always what we want. In most situations we would rather partition the data set into two disjoint subsets using random assignment. We can do this by combining *`splitobs()`* with *`shuffleobs()`*. Since neither of which copies actual data we do not pay any significant performance penalty for nesting "subsetting" functions.

```
julia> y1, y2 = splitobs(shuffleobs(y), at = 0.6)
(Symbol[:c,:e,:f,:a],Symbol[:b,:d])

julia> y1, y2, y3 = splitobs(shuffleobs(y), at = (0.5, 0.3))
(Symbol[:b,:f,:e],Symbol[:d,:a],Symbol[:c])
```

It is also possible to call *`splitobs()`* with two data containers grouped in a `Tuple`. While this is especially useful for working with labeled data, neither implies the other. That means that one can use tuples to group together unlabeled data, or have a labeled data container that is not a tuple (see *Labeled Data Container* for some examples). For instance, since the function *`splitobs()`* performs a static split, it doesn't actually care if the given `Tuple` describes a labeled data set. In fact, it makes no difference.

```
julia> X = rand(2, 6)
2×6 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996
```

(continues on next page)

(continued from previous page)

```
julia> y = ["a", "b", "b", "b", "b", "a"]
6-element Array{String,1}:
 "a"
 "b"
 "b"
 "b"
 "b"
 "a"

julia> (X1, y1), (X2, y2) = splitobs((X, y), at = 0.6);

julia> y1, y2
(String["a","b","b","b"],String["b","a"])
```

On the other hand, some functions require the presence of targets to perform their respective tasks. In such a case, it is always assumed that the last tuple element contains the targets. An alternative to *splitobs()* that is explicitly for labeled data is *stratifiedobs()*, which tries to preserve the class distribution. The following example shows that both, `y1` and `y2`, contain twice as much `"b"` as `"a"`, just like `y` does.

```
julia> (X1, y1), (X2, y2) = stratifiedobs((X, y), p = 0.5);

julia> y1, y2
(String["a","b","b"],String["b","b","a"])
```

Other functions that deal with supervised data sets are *undersample()* and *oversample()*, which can be used to re-sample a labeled data container in such a way, that the resulting class distribution is uniform.

```
julia> undersample(y)
4-element SubArray{String,1,Array{String,1},Tuple{Array{Int64,1}},false}:
 "a"
 "b"
 "b"
 "a"

julia> Xnew, ynew = undersample((X, y), shuffle = false)
([0.226582 0.933372 0.812814 0.11202; 0.504629 0.522172 0.245457 0.000341996],
 String["a","b","b","a"])

julia> Xnew, ynew = oversample((X, y), shuffle = true)
([0.11202 0.933372 ... 0.505208 0.0443222; 0.000341996 0.522172 ... 0.0997825 0.
↪722906],
 String["a","b","a","a","b","a","b","b"])
```

If need be, all functions that require a labeled data container accept a target-extraction-function as an optional first parameter. If such a function is provided, it will be applied to each observation individually. In the following example the function `indmax` will be applied to each column slice of `Y` in order to derive a class label, which is then used for down-sampling. For more information take a look at the section on *Labeled Data Container*.

```
julia> Y = [1. 0. 0. 0. 0. 1.; 0. 1. 1. 1. 1. 0.]
2×6 Array{Float64,2}:
 1.0  0.0  0.0  0.0  0.0  1.0
 0.0  1.0  1.0  1.0  1.0  0.0

julia> Xnew, Ynew = undersample(indmax, (X, Y));

julia> Ynew
```

(continues on next page)

(continued from previous page)

```
2×4 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 1.0  0.0  0.0  1.0
 0.0  1.0  1.0  0.0
```

This package also provides functions to perform re-partitioning strategies. These result in vector-like views that can be iterated over, in which each element is a different partition of the original data. Note again that all partitions are just lazy subsets, which means that no data is copied. For more information take a look at *Repartitioning Strategies*.

```
julia> x = collect(1:10);

julia> folds = kfolds(x, k = 5)
5-element FoldsView(::Array{Int64,1}, ::Array{Array{Int64,1},1}, ::Array{UnitRange
↪{Int64},1}, ObsDim.Last()) with element type Tuple{SubArray{Int64,1,Array{Int64,1},
↪Tuple{Array{Int64,1}},false},SubArray{Int64,1,Array{Int64,1},Tuple{UnitRange{Int64}}
↪,true}}:
 ([3,4,5,6,7,8,9,10],[1,2])
 ([1,2,5,6,7,8,9,10],[3,4])
 ([1,2,3,4,7,8,9,10],[5,6])
 ([1,2,3,4,5,6,9,10],[7,8])
 ([1,2,3,4,5,6,7,8],[9,10])

julia> train, val = folds[1] # access first fold
([3,4,5,6,7,8,9,10],[1,2])
```

Such "views" also exist for other purposes. For example, the function `obsview()` will create a decorator around some data container, that makes the given data container appear as a vector of individual observations. This "vector" can then be indexed into or iterated over.

```
julia> X = rand(2, 6)
2×6 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996

julia> ov = obsview(X)
6-element obsview(::Array{Float64,2}, ObsDim.Last()) with element type SubArray
↪{Float64,1,Array{Float64,2},Tuple{Colon,Int64},true}:
 [0.226582,0.504629]
 [0.933372,0.522172]
 [0.505208,0.0997825]
 [0.0443222,0.722906]
 [0.812814,0.245457]
 [0.11202,0.000341996]
```

Similarly, the function `batchview()` creates a decorator that makes the given data container appear as a vector of equally sized mini-batches.

```
julia> bv = batchview(X, size = 2)
3-element batchview(::Array{Float64,2}, 2, 3, ObsDim.Last()) with element type␣
↪SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 [0.226582 0.933372; 0.504629 0.522172]
 [0.505208 0.0443222; 0.0997825 0.722906]
 [0.812814 0.11202; 0.245457 0.000341996]
```

A third but conceptually different kind of view is provided by `slidingwindow()`. This function is particularly useful for preparing sequence data for various training tasks. For more information take a look at the section on *Data Views*.

```
julia> data = split("The quick brown fox jumps over the lazy dog")
9-element Array{SubString{String},1}:
 "The"
 "quick"
 "brown"
 "fox"
 "jumps"
 "over"
 "the"
 "lazy"
 "dog"

julia> A = slidingwindow(i->i+2, data, 2, stride=1)
7-element slidingwindow(::##9#10, ::Array{SubString{String},1}, 2, stride = 1) with␣
↪element type Tuple{...}:
 (["The", "quick"], "brown")
 (["quick", "brown"], "fox")
 (["brown", "fox"], "jumps")
 (["fox", "jumps"], "over")
 (["jumps", "over"], "the")
 (["over", "the"], "lazy")
 (["the", "lazy"], "dog")

julia> A = slidingwindow(i->[i-2:i-1; i+1:i+2], data, 1)
5-element slidingwindow(::##11#12, ::Array{SubString{String},1}, 1) with element type␣
↪Tuple{...}:
 (["brown"], ["The", "quick", "fox", "jumps"])
 (["fox"], ["quick", "brown", "jumps", "over"])
 (["jumps"], ["brown", "fox", "over", "the"])
 (["over"], ["fox", "jumps", "the", "lazy"])
 (["the"], ["jumps", "over", "lazy", "dog"])
```

Aside from data containers, there is also another sub-category of data sources, called **data iterators**, that can not be indexed into. For example the following code creates an object that when iterated over, continuously and indefinitely samples a random observation (with replacement) from the given data container.

```
julia> iter = RandomObs(X)
RandomObs(::Array{Float64,2}, ObsDim.Last())
 Iterator providing Inf observations
```

To give a second example for a data iterator, the type *RandomBatches* generates randomly sampled mini-batches for a fixed size. For more information on that topic, take a look at the section on *Data Iterators*.

```
julia> iter = RandomBatches(X, size = 10)
RandomBatches(::Array{Float64,2}, 10, ObsDim.Last())
 Iterator providing Inf batches with size 10

julia> iter = RandomBatches(X, count = 50, size = 10)
RandomBatches(::Array{Float64,2}, 10, 50, ObsDim.Last())
 Iterator providing 50 batches with size 10
```

Let us round out this introduction by taking a look at a "hello world" example (with little explanation) to get a feeling for how to combine the various functions of this package in a typical ML scenario.

```
# X is a matrix; Y is a vector
X, Y = rand(4, 150), rand(150)
```

(continues on next page)

(continued from previous page)

```
# The iris dataset is ordered according to their labels,
# which means that we should shuffle the dataset before
# partitioning it into training- and test-set.
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)

    for epoch = 1:100
        # 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
end
```

In the above code snippet, 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 a data subset for any type of data container.

Furthermore both, `batch_X` and `batch_Y`, will be the same instances each iteration with only their values changed. In other words, they both are 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 side-effects 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.

1.1.3 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 links to places that explain how this or a related package can be utilized to solve them.

- *Create a lazy data subset of some data.*
- *Shuffle the observations of a data container.*
- *Split data into train/test subsets.*
- *Split labeled data into train/test subsets.*
- *Group multiple variables together an treat as a single data set.*
- *Support my own custom user-define data container type.*
- *Over- or undersample an imbalanced labeled data set.*
- *Repartition a data container using a k-folds scheme.*
- *Iterate over my data one observation or batch at a time.*
- *Prepare sequence data such as text for supervised learning.*

1.1.4 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/MLDataPattern.jl

CHAPTER 2

Introduction and Motivation

If you are new to Machine Learning in Julia, or are simply interested in how and why this package works the way it works, feel free to take a look at the following documents. There we discuss the problem of data-partitioning itself and what challenges it entails. Further we will provide some insight on how this package approaches the task conceptually.

2.1 Background and Motivation

In this section we discuss what data partitioning entails and how one could go about approaching this task efficiently when performing it manually. Furthermore, we will outline some of the pitfalls one might encounter when doing so, which will lead to the motivation of the design decisions that this package follows.

When it comes to subsetting/partitioning **index-based** data into individual samples (which we call **observations**), or groups of samples (which we will refer to as **batches** or mini-batches) the problem at hand really breaks down to one core task: **keeping track of indices**.

To get a better understanding of what exactly we mean by "index-based data" and "tracking indices", let us together explore how one would typically approach data partitioning scenarios without the use of external packages (i.e. by getting our hands dirty and coding it ourselves!). We will do so using a number of different but commonly used forms of data storage such as *matrices* and *data frames*.

Warning: This section and its sub-sections serve soley as example to explain the underyling problem of partitioning/subsetting data and further to motivate the solution provided by this package. As such this section is **not** intended as a guide on how to apply this package.

2.1.1 Two Kinds of Data Sources

In the context of this package, we differentiate between two "kinds" of data sources, which we will call **iteration-based** and **index-based** .

Iteration-based (aka Data iterator) To belong to this group, a data source must implement the iterator interface. It may or may not know the total amount of observations it can provide, which means that knowing N is not necessary.

The key requirement for a iteration-based data source is that every iteration either returns a single observation or a batch of observations.

These kind of data sources are primarily used for either streaming data, continuous resampling, or for large/remote data sets where even storing the indices requires too much memory.

Index-based (aka Data Container) For a data source to belong in this category it needs to be able to provide two things:

1. The total number of observations N, that the data source contains.
2. A way to query a specific observation or set of observations. This must be done using indices, where every observation has a unique index $i \in I$ assigned from the set of indices $I = \{1, 2, ..., N\}$.

We will go into more detail about data sources and their differences in later sections. The key takeaway from this little discussion here is that these two kinds of data sources offer distinct challenges and need to be reasoned with differently.

For the rest of this document we will focus on working with **index-based** data.

2.1.2 A Manual Solution for Arrays

Matrices and other (multi-dimensional) arrays are one of the most commonly used data storage containers in Machine Learning. As such, it is quite likely that you will find (or have already found) yourself in the position of working with such data sooner or later.

Let's say you are interested in working with the Iris data set in order to test some clustering or classification algorithm that you are working on. The package `MLDataUtils` provides a convenience function `load_iris` for loading the data set in array form. Calling this function will give us two variables, the feature-matrix `X` and the target-vector of labels `Y`.

```
julia> using MLDataUtils

julia> X, Y = load_iris();
```

The first variable, `X`, contains all our **features**, sometimes called *independent variables* or *predictors*. In this case each column of the matrix corresponds to a single **observation**, or *sample*. Each observation in `X` thus has 4 features each. These features represent some quantitative information known about the corresponding observation, which for the sake of keeping this document concise, is about the extend to which we will discuss their meaning in this tutorial.

```
julia> X
4×150 Array{Float64,2}:
 5.1  4.9  4.7  4.6  5.0  5.4  4.6  ...  6.8  6.7  6.7  6.3  6.5  6.2  5.9
 3.5  3.0  3.2  3.1  3.6  3.9  3.4       3.2  3.3  3.0  2.5  3.0  3.4  3.0
 1.4  1.4  1.3  1.5  1.4  1.7  1.4       5.9  5.7  5.2  5.0  5.2  5.4  5.1
 0.2  0.2  0.2  0.2  0.2  0.4  0.3       2.3  2.5  2.3  1.9  2.0  2.3  1.8
```

The second variable, `Y`, denotes the **labels** (also often called *classes* or *categories*) of each observation. These terms are usually used in the context of predicting categorical variables, such as we do in this example. The more general term for `Y`, which also includes the case of numerical outcomes, is **targets**, *responses*, or *dependent variables*.

```
julia> Y
150-element Array{String,1}:
 "setosa"
```

(continues on next page)

(continued from previous page)

```
 "setosa"
 "setosa"

 "virginica"
 "virginica"
 "virginica"
```

Together, `X` and `Y` represent our data set. Both variables contain 150 observations and the individual elements of the two variables are linked together through the corresponding observation-index. For example, the following code snippet shows how to access the 30-th observation of the data set.

```
julia> X[:, 30], Y[30]
([4.7,3.2,1.6,0.2],"setosa")
```

This link is an important detail that we need to keep in mind when thinking about how to partition our data set into subsets. The main lesson here is that whatever kind of sub-setting strategy we apply to one of the variables we need to apply the exact same sub-setting operation to the other one as well.

Now that we have our full data set we could consider splitting it into two differently sized subsets: a **training set** and a **test set**.

One naive and dangerous approach to achieve this is to do a "static" split, i.e. use the first n observations as training set and the remaining observations as test set. I say dangerous because this strategy makes a strong assumption that may not be true for the data we are working with (and in fact it is not true for the Iris data set). But more on that later.

To perform a static split we first need to decide how many observations we want in our training set and how many observations we would like to hold out on and put in our test set. It is often more convenient to think in terms of proportions instead of absolute numbers. Let's say we decide on using 80% of our data for training. To split our data set in such a way, we first need to derive which elements of `X` and `Y` we need assign to each subset in order to accomplish this exact effect.

```
julia> idx_train = 1:floor(Int, 0.8 * 150)
1:120

julia> idx_test = (floor(Int, 0.8 * 150) + 1):150
121:150
```

As we can see, we made sure that the two ranges do not overlap, implying that our two subsets will be disjoint. At this point we can use these ranges as indices to subset our variables into a training and a test portion.

```
julia> X_train, Y_train = X[:, idx_train], Y[idx_train];
julia> size(X_train)
(4,120)

julia> X_test, Y_test = X[:, idx_test], Y[idx_test];
julia> size(X_test)
(4,30)
```

Note: To put this into perspective: In order to perform this type of static split using the provided functions of this package, one would type the following code:

```
(X_train,Y_train), (X_test,Y_test) = splitobs((X,Y), at = 0.8)
```

For more information take a look at the documentation for the function *`splitobs()`*.

So far so good. For many data sets, this approach would actually work pretty fine. However - as we teased before - performing static splits is not necessarily a good idea if you are not sure that both your resulting subsets (individually!) would end up being representative of the complete data set or population under study.

The concrete issue in our current example is that the iris data set has structure in the order of its observations. In fact, the data set is ordered according to their labels. The first 50 observations all belong to the class `setosa`, the next 50 to `versicolor`, and the last 50 observation to `virginica`. Knowing that piece of trivia it is now plain to see that our supposed test set only contains observation that belong to the class `virginica`.

```
julia> Y_test
30-element Array{String,1}:
 "virginica"
 "virginica"
 "virginica"

 "virginica"
 "virginica"
 "virginica"
```

As a consequence our prediction results would not give us good estimates and chances are that any output we get from our model is completely nonsensical.

Tip: While it surely depends on the situation, as a rough guide we would advise to only use static splits in one of the following two situations:

1. You are *absolutely confident* that the order of the observations in your data set is *random*.
2. You are working with a data set for which there is a convention to use the last n observations as a test set or validation set.

Well, so we saw that a static split would not be a good idea for this data set. What we really want in our situation is a random assignment of each observation to one (and only one) of the two subsets. Turns out we can quite conveniently randomize the order of our observations by using the function `shuffle`.

```
julia> idx = shuffle(1:150)
150-element Array{Int64,1}:
  56
  41
 146

  90
   5
  13
```

The naive thing to do now would be to first create a shuffled version of our full data set using `X[:,idx]` and `Y[idx]` and then do a static split on the new shuffled version. That, however, would in general be quite inefficient as we would copy the data set around unnecessarily a few times before even using it for training our model. The data set usually takes up a lot more memory than just the indices, and if we think about it, we will see that reasoning with the indices is all we really need to do in order to accomplish our partitioning strategy.

Instead of first shuffling the whole data set, let us just perform a static split on `idx`, similar to how we initially did on the data directly. In other words we perform our static sub-setting on the indices in `idx` instead of the observations in data. This is already hinting to what we meant at the beginning of this document with "keeping track of indices", since this concept of index-accumulation is quite powerful.

```
julia> idx_train = idx[1:floor(Int, 0.8 * 150)]
120-element Array{Int64,1}:
  56
  41

 121
   7

julia> idx_test = idx[(floor(Int, 0.8 * 150) + 1):150]
30-element Array{Int64,1}:
 102
  92

   5
  13
```

Using these new training- and test indices we can now construct our two data subsets as we did before, but this time we end up with randomly assigned observations for both.

```
julia> Y_test
30-element Array{String,1}:
 "virginica"
 "versicolor"

 "setosa"
 "setosa"
```

Very well! Now we have a training set and a test set. In many situations we may want to consider further sub-setting of our training set before feeding the subsets into some learning algorithm.

In a typical scenario we would be inclined to split our newly created training set into a smaller training set and a validation set, the later of which we would like to use to test the impact of our hyper-parameters on the prediction quality of our model. And if additionally we employ a stochastic learning algorithm, chances are that we also want to chunk our training data into equally sized mini-batches before feeding those individually into the training procedure.

Even though this is starting to sound rather complex, it turns out that all we really need to do is keep track of our indices properly. In other words, all these sub-setting of sub-sets can be done by just accumulating indices. The following code snippet shows how this could be achieved if implemented manually.

```
X, Y = load_iris()

# trainingset: 100 obs
# validationset: 20 obs
# testset: 30 obs
n_cv    = 120
n_train = 100

# randomly assign observations to either CV set or test set
# the CV set will later be divided into training and validation set
idx = shuffle(1:150)
idx_cv   = idx[1:n_cv]
idx_test = idx[(n_cv + 1):150]

# we will perform 10 different partitions of the CV set into
# a training and validation portion to get a better estimate
# NOTE: This is just a very rudimentary resampling strategy for
#       the sake of keeping this example simple.
```

(continues on next page)

(continued from previous page)

```
for i = 1:10
    # each iteration we shuffle around the CV indices so that
    # a static split into training and validation set will be
    # the same as a random assignment
    shuffle!(idx_cv)
    idx_train = idx_cv[1:n_train]
    idx_val   = idx_cv[(n_train+1):n_cv]

    # iterate over our training set in 20 batches of batch-size 5
    for j = 1:20
        idx_batch = idx_train[(1:5) + (j*5-5)]

        # Now we actually allocate the current batch of data
        # that we need for our computation in this step.
        X_batch = X[:, idx_batch]
        Y_batch = Y[idx_batch]

        # ... train some model on current batch here ...
    end
end
```

I would argue that this code is still quite readable and we managed to delay accessing and sub-setting of our data set to the latest possible moment. Also note how we only copy the portion of the data that we actually need at that iteration.

The main point of this exercise is to show that nesting data access pattern can be reduced to just keeping track of indices. This is the core design principle that the access pattern of MLDataPattern follow.

Note: To put this into perspective: In order to perform this type of partitioning scheme using the provided functions of this package, one would type the following code:

```
cv, test = splitobs(shuffleobs((X,Y), at = 0.8)

for i = 1:10
    train, val = splitobs(shuffleobs(cv), at = 0.84)

    # iterate over our training set in 20 batches of batch-size 5
    for (X_batch, Y_batch) in eachbatch(train, 5)
        # ... train some model on current batch here ...
    end
end
```

For more information take a look at the documentation for the functions *splitobs()*, *shuffleobs()*, and *eachbatch()* respectively.

While this is already a decent enough implementation, we could further reduce our memory footprint by using views. We should not forget that that even if we only copy indices, we still copy around memory.

```
X, Y = load_iris()

# same as before
n_cv    = 120
n_train = 100

# instead of static splits create views into idx
idx = shuffle(1:150)
```

(continues on next page)

(continued from previous page)

```
idx_cv   = view(idx, 1:n_cv)
idx_test = view(idx, (n_cv + 1):150)

# preallocate batch buffers. We will re-use them in every
# iteration to avoid temporary arrays
X_batch = zeros(Float64, 4, 5)
Y_batch = Y[1:5]

# We can create our training and validation views outside the loop,
# as their elements will be mutated when we shuffle idx_cv
idx_train = view(idx_cv, 1:n_train)
idx_val   = view(idx_cv, (n_train+1):n_cv)

for i = 1:10
    # this little trick will randomly assign observations to
    # either training set or validation set in each iteration
    shuffle!(idx_cv)

    for j = 1:20
        idx_batch = view(idx_train, (1:5) + (j*5-5))

        # copy the current batch of interest into a proper
        # array that is a continuous block of memory
        copy!(X_batch, view(X, :, idx_batch))
        # to be fair it makes less difference for an array
        # of strings, but you get the idea.
        copy!(Y_batch, view(Y, idx_batch))

        # .. train some model on current batch here ...
    end
end
```

In this version of the code we did quite a lot of micro-optimization, which at least on paper yields a cleaner solution to our task. While probably improving our performance a little, it did not really help readability of our code however. And if we end up with a bug somewhere we may have a nasty time deducing which little "trick" does not do what we thought it would.

Warning: These kind of hand-crafted micro-optimizations, while fun to think about, can be quite error prone. In some situations they may not even turn out to have been worth the effort when measuring its influence on the training time of your model. Keep that in mind when tinkering on a project. Premature optimization without profiling can cost a lot of valueable time and energy.

Now to the good part. MLDataPattern tries to do these kind of performance tricks for you in certain situations (specifically when working directly with *DataSubset*). So if it makes sense, our provided pattern try to avoid allocating unnecessary index-vectors. Naturally, one will always be able to hand craft some better optimized solution for some special use-case such as this one, but most of the time just avoiding common pitfalls will get you 80% of the way. With an interesting enough problem the other 20% of performance-gain you could achieve by dwelling on this issue would likely be negligible in relation to the training time of your learning algorithm.

2.1.3 Array Dimension for Observations

Before we move on from our array example to a data frame, let us briefly think about the "observation dimension" of some array. Let us consider the Iris data set again.

```
julia> X, Y = load_iris();

julia> size(X)
(4,150)
```

The variable `X` is our feature `Matrix{Float64}`, which in Julia is a typealias for a two dimensional array `Array{Float64,2}`. As such the variable has two dimensions that we can assign meaning to.

So far we acted on the convention that the first dimension encodes our features, and the second dimension encodes our observations. However, there is no law that dictates that this is the right way around. In fact it is much more common in the literature as well as other languages to have the first dimension encode the observations and the second dimension denote the features. This would also be much more relatable to how we organize some data in a spreadsheet.

Note: There is a good reason that you will often find the convention to use the last array dimension to encode the observations when working with Julia. This has to do with how Julia arrays access their memory. For more information on this topic take a look at the corresponding section in the Julia documentation

---

There have been many discussions on which convention is more useful and/or efficient, but the only answer you will find here is a humble **it depends on what you are doing**.

Consider the following scenario. Let's say we would again like to work with the Iris dataset, but this time we use the RDatasets package to load it. This will give us the same data, but in a quite different data-storage type.

```
julia> using RDatasets

julia> iris = dataset("datasets", "iris")
150×5 DataFrames.DataFrame
| Row | SepalLength | SepalWidth | PetalLength | PetalWidth | Species     |
|-----|-------------|------------|-------------|------------|-------------|
| 1   | 5.1         | 3.5        | 1.4         | 0.2        | "setosa"    |
| 2   | 4.9         | 3.0        | 1.4         | 0.2        | "setosa"    |
| 3   | 4.7         | 3.2        | 1.3         | 0.2        | "setosa"    |
| 4   | 4.6         | 3.1        | 1.5         | 0.2        | "setosa"    |

| 147 | 6.3         | 2.5        | 5.0         | 1.9        | "virginica" |
| 148 | 6.5         | 3.0        | 5.2         | 2.0        | "virginica" |
| 149 | 6.2         | 3.4        | 5.4         | 2.3        | "virginica" |
| 150 | 5.9         | 3.0        | 5.1         | 1.8        | "virginica" |
```

There are two common ways of how to go about using such a data frame for some Machine Learning purposes:

1. Using a formula to compute a model-matrix and work with that. This is a typical approach for when one wants to use models that need numerical features, such as linear regression. By using a formula we can transform the categorical features to numerical ones using so-called dummy variables.
2. Using the data frame directly. Some models, such as decision trees, can deal with categorical features themself and don't require the features in a numerical form.

Before we dive into the second scenario, let us consider building a model matrix. This will give us a motivating example to deal with different conventions for the observation dimension.

Without any explanation that does it justice, let us create a feature matrix `X` from the data frame `iris` using the following code snippet:

```
julia> X = ModelMatrix(ModelFrame(@formula(Species ~ SepalLength + SepalWidth +␣
↪PetalLength + PetalWidth), iris)).m
150×5 Array{Float64,2}:
```

(continues on next page)

(continued from previous page)

```
 1.0  5.1  3.5  1.4  0.2
 1.0  4.9  3.0  1.4  0.2
 1.0  4.7  3.2  1.3  0.2
 1.0  4.6  3.1  1.5  0.2

 1.0  6.3  2.5  5.0  1.9
 1.0  6.5  3.0  5.2  2.0
 1.0  6.2  3.4  5.4  2.3
 1.0  5.9  3.0  5.1  1.8
```

Notice two things. First, we now have a feature matrix `X` for which the first dimension (i.e. the rows) denotes the observations. Secondly, we ended up with 5 features for each observation, while in our previous example he had 4. This is because by default the model matrix is augmented with a constant variable that models can use to fit an intercept to. But that need not trouble us right now. The main point is that different tasks often have different conventions, and ideally we would like to have tools that can adapt to the current situation.

So how would this change of convention be reflected in our sub-setting strategy? Well, everywhere we previously wrote `X[:, indices]`, we would now write `X[indices, :]`. This looks like a simple enough change, but it has the consequence that the reuse already written partitioning code can be rather limited without some more coding effort. And even then, what if next time we work with 3 or 4 dimensional arrays (e.g. image data)? Generalizing this concept requires careful considerations.

Note: To put this into perspective: In order to be able to diverge from the convention of using the last array dimension as observation, all relevant methods of this package have an optional parameter `obsdim`, which can be specified either as a positional and type-stable argument, or as a convenient keyword argument

```
train, test = splitobs(X, obsdim = 1)
train, test = splitobs(X, obsdim = :first)
train, test = splitobs(X, ObsDim.First())
```

For more information take a look at the section on *Observation Dimension*.

2.1.4 Generalizing to Other Data

So far we have discussed how to implement a solution to the task of partitioning some data that is in array form. We also showed that it is feasible to consider supporting different conventions for which dimension to use to denote the individual observations.

Now, what if we would like to work with data that is not in array form, such as data frames or any other kind of data container, really. Well, if we look back at the code snippets we have written so far, we will see that we haven't actually specified any type- or structure requirement of the learning algorithm we are interested in. Indeed, we haven't said much about any learning algorithm at all, only that it expects the data in mini-batches. Instead we focused on how to represent our array-like data-subset and even considered to buffer it efficiently by preallocating the subset storage.

Whatever kind of partitioning scheme we code, we would like it to be agnostic about our learning algorithm. What it should really care about is the type of data storage it is working with and how to communicate with it. Ideally we would like to abstract whatever information we need from our data and whatever action we need to perform with our data.

Turns out we only need our data container to expose two things:

1. How many observation the data contains.
2. A way to access the observations of a given index or indices.

Let's consider data in the form of a data frame. We can query the total number of observations using `nrow(iris)`, since each row contains a single observation. Further we can access the observations of some given indices `idx` using `iris[idx, :]`. That is all that is needed to make our first code snippet from the array example work with data frames (we leave the proof of this as an exercise). However, there are a few things to note.

- When we access the observations of a given index we get a `DataFrame` in return. This makes sense for the data we are working with. Our learning algorithm may or may not support working with data frames, but that is not the responsibility of the partitioning logic.
- Notice how no buffering of the mini-batches would occur in this case, as each access to a `getindex` of `iris` would create a new data frame. That said, we can't do much better here because the lack of efficient buffering is a property of the type of data we are working with.

Great! At this point we know how to partition any data set that provides a way to query the number of observation it contains, and has a method available to access observations of specific indices. That does not free us from the burden of **tracking the indices**, however.

This is where MLDataPattern comes in.

2.2 Package Design

In a heterogeneous field, such as Machine Learning, one quickly finds himself/herself collaborating with very smart people of quite different educational backgrounds. This is as much a privilege and opportunity as it can be time-consuming to reach a consensus on how to design, name, and implement functionality. Naturally, there were quite a lot of discussions to be had and disagreements to be settled in order to even reach the current state of MLDataPattern. That said, some details are not yet set into stone, but a moving target.

We welcome anyone interested in contributing ideas and/or code to our community, but would ask you to use this document in order to catch up on the current status of the discussion before making any assertion about how something should be done. This section is our attempt to summarize our position on design issues and to justify why some of the maybe more controversial design decisions were made to be as they are.

2.2.1 Design Principles

While some design goals are arguably generic no-brainer for any software project, we would still like to take the time to write down the key principles and opportunities that we identified while devising and implementing this package.

Julia First

As the name **JuliaML** subtly hints, the mission of our organization is to design and implement Machine Learning functionality in Julia itself. We believe that the design of the language allows us to experiment with API design ideas that may not be feasible or sensible in other languages that suffer from the two-language problem much more significantly. Naturally, that is not as black or white as it sounds, since even Julia itself out-sources certain computation to BLAS (which is a good thing!). As a rough guide: unless there is a really compelling argument, we only consider code that is written solely in Julia to be merged into the JuliaML ecosystem.

Data Agnostic

One of the interesting strong points of the Julia language is its rich and developer-friendly type system. User-created types are as much first class as any other language-provided one. We recognize this as a quite unique opportunity in modern scientific computing and want to make sure that working with custom types is not penalized when using basic functionality such as data partitioning. Thus we made it a key design priority to make as little assumptions as possible about the data at hand. For instance, we do not require custom data-source-types to share a common super-type.

Type Stable

The impact of type-stable code on its performance is not necessarily a clean cut issue. That said, you do not want to have type-inference problems in some inner loop of your algorithm. As such we consider it of key importance that all core functions offer a type-stable API. Notice that this has the consequence that this API must be specified using positional and dispatch-friendly arguments. This can be unintuitive at times, because the ordering of the arguments does not always offer itself to some meaningful convention or interpretation.

Convenient

While efficiency is important, we can sometimes be prone to overthink and indeed overestimate the impact of certain factors to the overall performance of our code. Type stability is one such factor. While compile-time dispatch and clever inlining can be very impactful for a computational kernel, it need not make a significant performance difference for top-level user-facing functions. These - more often than not - out-source the actual computation to some type-stable functions anyways. Yet, usability can suffer significantly if the function signature is unintuitive just because it has been over-engineered solely for the purpose of being type-stable and to avoid keyword arguments.

We tackle this issue by what we consider a middle ground solution. While all our core functions offer a type-stable API, which may in general be less intuitive to use, we also provide a keyword-based convenience method for most. Usually those keyword arguments are designed to be more tolerant with the parameter values and types they accept. However, this can come at the price of poisoning the type-inference of the calling scope.

Extensible

If you find yourself working on an unusual problem, chances are that the standard methods or patterns implemented by this package just don't work or only partially work for your use-case. Not all interesting problems, and certainly not those on the forefront of science, lend themselves to the same kind of treatment. One example could be that you need to implement some special way of sub-sampling your data. It is important to us that one can do such a thing in a sensible way and also still be able to use to rest of the package.

The situation that we judge as probably the most common, though, will be that users will want to work with their own special data containers. Therefore we put our core priority on making sure that doing so is as simple and non-disruptive as possible. Thus we settled on the solution of using duck-typing for custom data containers for the sole reason to avoid influencing the overall design-decisions of your experiment with some super-type requirement.

Furthermore, all the functions and (abstract) types, that are necessary in order for you to be able to provide support for your own custom data-source-type, are defined in a special light-weight package called LearnBase. This way package developers need not pollute their `REQUIRE` file with heavy dependencies just to support the JuliaML ecosystem.

Well Tested

While test coverage can give a rough estimate of how much effort was spend in testing your code, it is still just a proxy variable when it comes to test quality. We do not have a solution to this problem, but we put a large emphasis on testing the actual functionality of our code. As such we can also only consider pull requests that provide sensible and meaningful tests for their proposed changes (so coverage alone won't cut it).

Cross-Community

The initial authors of this package consider it of importance to work through aesthetic-based disagreements, and so to converge towards common solutions to the underlying problems where possible. Machine Learning is a field that crosses over many disciplines and we should try to make use of this opportunity to learn from each other where we can. So if you came across this package and found that it doesn't address your specific use case in the way you would have expected it to, let us know! Maybe we can converge to a common solution.

That said, please keep in mind that it may not always be feasible to please everyone at all times. In such cases we should try to break the corresponding issue down into sub-issues that provide a more promising ground for actionable changes or refactors.

2.2.2 Design Overview

This section will serve as a documentation of how and why specific parts of MLDataPattern was designed the way it is. As such it is *not* a user's guide, but instead a discussion intended to inform potential contributors and users of why things are the way they are currently.

Support for Custom Data Container

We identified quite early in our design discussions, that we wanted to support custom data-container-types as first class citizen in our data access pattern. Consequently, we had to carefully think about what kind of functionality and information any data-source-type must expose in order to achieve this in a clean and efficient manner. Luckily we found that this can be reduced to surprisingly little, as subsetting/partitioning of data really just breaks down to keeping track of indices, and doesn't actually involve the data until the very end (see *Background and Motivation* for a thorough discussion of this).

Furthermore, we wanted to make sure that the decision to opt-in to our ecosystem had as little impact to the overall design of the user code as possible. This had the consequence of not being able to impose a common super-type for data containers. Additionally, we could not rely on `Base` functions, such as `size`, to be implemented for the data at hand. Worse, we could not be confident that (even if implemented) these methods would consistently have the same second-hand interpretation in terms of what denotes the *number of observations*.

Thus we decided to define custom functions with singular interpretation for these purposes. This has a price, however. If a user would like to provide support for his/her custom data-source-type, he/she would need to add at least some JuliaML dependency in order to define methods for the required functions. To keep this dependency reasonable small, we created a light-weight package called LearnBase. The sole purpose of this package is to define common types and functions used through the JuliaML ecosystem. Thus, to opt-in to the ecosystem with your custom package, the LearnBase dependency is all that you will need to accomplish that (if it isn't then you likely found a bug!).

Representing Data Subsets

As we mentioned before, as long as we can somehow keep track of the indices, we don't actually require the data source to offer a lot of special functionality. The question that remained, though, is how to track the indices in a sensible and non-intrusive manner. When in doubt, we try to follow the Julia design by example. Consider the `SubArray` type. In our current context, we can think about it as really just a special case implementation for a data container decorator that keeps track of the indices (especially since the release of 0.5).

We will call an object that connects some data container to some subset-indices a **Subset**. We decided that it would be preferable to allow data containers to specify their own type of subset. For example, a `SubArray` would be a good choice as a subset for some `Matrix`. See *Support for Custom Types* for more information on how to provide a custom subset type for your data container.

To keep user-effort manageable, we provide a generic subset implementation for those types that do not want to implement their own special version. In other words: Unless a custom subset-type is provided, a subset of some given data will be represented by a type called *`DataSubset`*. The main purpose for the existence of *`DataSubset`* - or any special data subset for that matter - is two-fold:

1. To **delay the evaluation** of a sub-setting operation until an actual batch of data is needed.
2. To **accumulate subsetting indices** 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 hard-drive or some remote location. In such a scenario one wants to load only the required data only when it is actually needed.

What about Streaming Data?

So far we talked about data as if it were an universal truth that it can be split somewhere or subsetted somehow. This need not be true for all kinds of data we are interested in working with.

This package differentiates between two kinds of data sources that we will call **iteration-based** (represented as *Data Iterator*), and **index-based** (represented as *Data Container*) respectively. None is the superset of the other, but a user type can be both. This also implies that none require a type to have some specific super-type.

Data Iterator A data iterator is really just the same as a plain Julia iterator that need not (but may) know how many elements it can provide. It also makes no guarantees about being able to be sub-setted, so there is no contract that states that a data iterator must implement a function that allows to query an observation of some specific index.

Each element must either be a single observation or a batch of observations; the choice is up to the data iterator. That said it is important that all provided elements are of the same type and of the same structure (e.g. batch size).

There is no hard distinction between a data iterator that provides the data itself, or a data iterator that just iterates over some other data iterator/container in some manner. For example the data iterator `RandomBatches` iterates over randomly sampled batches of the data container that you pass to it in its constructor.

Data Container A data container is any type that knows how many observations it represents (exposed via `nobs()`) and implements a method for `getobs()` that allows to query individual observations or batches of observations.

There is no contract that states `getobs()` must return some specific type. What it returns is up to the data container. The only requirement is that it is consistent. A single observation should always have the same type and structure, as should a batch of some specific size. Take a look at the section on *Data Container* for more information about the interface and requirements.

A data container need not also be a data iterator! There is no contract that iterating over a data container makes sense in terms of its observations. For example: iterating over a matrix will not iterate over its observations, but instead over each individual element of the matrix.

Any data container can be promoted to be a data iterator as well as a data container by boxing it into a `DataView`, such as `BatchView` or `ObsView`. See the section on *Data Views* for more information.

Tuples and Labeled Data

We made the decision quite early in the development cycle of this package to give `Tuple` special semantics. More specifically, we use tuples to tie together different data sources on a per-observation basis.

All the access-pattern provided by this packages can be called with data sources or tuples of data sources. For the later to work we need to understand the assumptions made when using `Tuple`.

1. All elements of the Tuple must contain the same total number of observations
2. If the data set as a whole contains targets, these must be part of the **last** element of the tuple.

Consider the following toy problem. Let's say we have a numeric feature vector `x` with three observations. Furthermore we have a separate target vector `y` with the three corresponding targets.

```
julia> x = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> y = [:a,:b,:c]
3-element Array{Symbol,1}:
 :a
 :b
 :c
```

Naturally we think of these two data sources as one data set. That means that we require the access pattern to treat them as such. For example if you want to shuffle your data set, you can't just shuffle `x` and `y` independently, because that would break the connection between observations.

```
# !!WRONG!!
julia> shuffleobs(x), shuffleobs(y)
([1,3,2],Symbol[:b,:a,:c])
```

This is why the access pattern provided by this package allow for `Tuple` arguments. The functions will assume that the elements of the tuple are linked by observation index and make sure that all operations performed on the data preserves the per-observation link.

```
# correct
julia> shuffleobs((x,y))
([1,3,2],Symbol[:a,:c,:b])
```

The second assumption we mentioned only concerns supervised data (i.e. data where each observation has a corresponding target). Simply put, if there are any targets, they must be contained in the last tuple element. All the access pattern provided by this package build on that convention.

```
julia> targets((x,y))
3-element Array{Symbol,1}:
 :a
 :b
 :c
```

CHAPTER 3

Using MLDataPattern.jl

The main design principle behind this package is based on the assumption that the data source a user is working with, is likely of some 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`.

The first topic we will cover is about **data containers**. These represent a large subgroup of data sources, that all know how many observations they contain, as well as how to access specific observation(s). As such they are the most flexible kind of data sources and will thus be at the heart of most of the subsequent sections. To start off, we will discuss what makes some type a data container and what that term entails.

3.1 Data Container

We have hinted in previous sections that we differentiate between two "kinds" of data sources, which we called *iteration-based* and *index-based* respectively. Of main interest in this section are index-based data sources, which we will henceforth refer to as **Data Container**. For a data source to qualify as such, it must at the very least be able to provide the following information:

1. The total number of observations N, that the data source contains.
2. A way to query a specific observation or set of observations. This must be done using indices, where every observation has a unique index $i \in I$ assigned to it from the set of indices $I = \{1, 2, ..., N\}$.

If a data source implements the required interface to be considered a data container, a lot of additional much more complex functionality comes for free. Yet the required interface is rather unobtrusive and simple to implement.

- What makes a Julia type a data container are the implemented functions. That means that any custom type can be marked as a data container by simply implementing the required interface. This methodology is often called "duck typing". In other words, there is no abstract type that needs to be sub-typed. This fact makes the interface much less intrusive and allows package developers to opt-in more easily, without forcing them to make any architectural compromises.
- There is no requirement that the actual observations of a data container are stored in the working memory. Instead the data container could very well just be an interface to a remote data storage that requests the data on demand when queried.

- A data container can - but need not - be the data itself. For example a Julia `Array` is both data, as well as data container. That means that querying specific observations of that array will again return an array. On the other hand, if the data container is a custom type that simply serves as an interface to some remote data set, then the type of the data container is distinct from the type of the data (which is likely an array) it returns.

We will spend the rest of this document on discussing data containers in all its details. First, we will provide a rough overview of how the interface looks like. After that, we will take a closer look at every single function individually, and even see some code examples showing off their behaviour.

3.1.1 Interface Overview

For any Julia type to be considered a data container it must implement a minimal set of functions. All of these functions are defined in a small utility package called LearnBase.jl. This means that in order to implement the interface for some custom type, one has to import that package first. More importantly, it implies that one does **not** need to depend on `MLDataPattern.jl` itself. This allows package developers to keep dependencies at a minimum, while still taking part in the JuliaML ecosystem.

There are only two methods that *must* be implemented for every data container. In other words, implementing these two methods is sufficient and necessary for a type to be considered a data container.

Required methods	Brief description
`nobs(data, [obsdim])`	Returns the total number of observations in `data`
`getobs(data, idx, [obsdim])`	Returns the observation(s) from `data` indexed by `idx`

Aside from the required interface, there are a number of optional methods that can be implemented. The main motivation to provide these methods as well for a data container, is that they can allow for a significant boost in performance in some situations.

Optional methods	Brief description
`getobs(data, [obsdim])`	Returns all observations contained in `data`
`getobs!(buf, data, [idx], [obsdim])`	Inplace version of `getobs(data, idx, obsdim)` using `buf`
`gettargets(data, idx, [obsdim])`	Returns the target(s) for the observation(s) in `data` at `idx`
`datasubset(data, idx, obsdim)`	Returns an object representing a lazy subset of `data` at `idx`

Out of the box, this package implements the full data container interface for all subtypes of `AbstractArray`. Furthermore, `Tuple` can be used to link multiple data containers together, and thus are considered quasi data container. They are accepted everywhere data containers are expected, but they do have very special semantics in the context of this package. For more information about how `Tuple` are interpreted, take a look at *Tuples and Labeled Data*.

3.1.2 Number of Observations

Every data container must be able to report how many observations it contains and can provide. To that end it must implement the function *`nobs()`*. We will see that for some data containers the meaning of "observations" can be ambiguous and depend on a user convention. For such cases it is possible to specify an additional argument, that denotes the observation dimension.

`nobs` (*data*[, *obsdim*]) → Int

Return the total number of observations that the given *data* container can provide.

The optional parameter *obsdim* can be used to specify which dimension denotes the observations, if that concept makes sense for the type of *data*.

Parameters

- **data** – The object representing a data container.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns The number of observations in *data* as an Integer

We hinted before that `nobs()` is already implemented for any subtype of `AbstractArray`. This is true for arrays of arbitrary order, even higher order arrays (e.g. images).

Let's start simple and take some random vector `y` as an example. In the case of a vector (i.e. an one-dimensional array), it is assumed that each element is an observation.

```
julia> y = rand(5)
5-element Array{Float64,1}:
 0.542858
 0.28541
 0.613669
 0.217321
 0.018931

julia> nobs(Y)
5
```

If there is more than one array dimension, all but the observation dimension are implicitly assumed to be features (i.e. part of that observation). This implies that for an array, the individual observations have to be explicitly laid out along a single dimension.

```
julia> X = rand(2,5)
2×5 Array{Float64,2}:
 0.175347  0.61498   0.621127   0.0697848  0.454302
 0.196735  0.283014  0.0961759  0.94303    0.584028

julia> nobs(X)
5
```

As you can see, the default assumption is that the last array dimension enumerates the observations. This can be overwritten by explicitly specifying the `obsdim`.

```
julia> nobs(X, ObsDim.First())
2

julia> nobs(X, obsdim = :first)
2

julia> nobs(X, obsdim = 1)
2
```

Note how `obsdim` can either be provided using a type-stable positional argument from the namespace `ObsDim`, or by using a more flexible and convenient keyword argument. We will discuss observation dimensions in more detail in a later section.

3.1.3 Query Observation(s)

At some point in our machine learning pipeline, we need access to specific parts of the "actual data" in our data container. That is, we need the data in a form where an algorithm can *efficiently* process it. This package does not

impose any requirement on how this "actual data" must look like. Every author behind some custom data container can make this decision him-/herself. In reality, it depends on what type the algorithm one is working with expects (`Array` is in general a good choice). Providing a reasonable type is the responsibility of the data container. To that end, every data container must implement a method for the function *`getobs()`*.

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

Return the observation(s) in *data* that correspond to the given index/indices in *idx*. Note that *idx* can be of type `Int` or `AbstractVector`. Both options must be supported.

The returned observation(s) should be in the form intended to be passed as-is to some learning algorithm. There is no strict requirement that dictates what form or type that is. We do, however, expect it to be consistent for *idx* being an integer, as well as *idx* being an abstract vector, respectively.

Parameters

- **`data`** – The object representing a data container.
- **`idx`** – Optional. The index or indices of the observation(s) in *data* that should be returned. Can be of type `Int` or some subtype `AbstractVector{Int}`. Defaults to `1:nobs(data,obsdim)`
- **`obsdim`** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns The actual observation(s) in *data* at *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 fulfill.

Just like for *`nobs()`*, this package natively provides a *`getobs()`* implementation for any subtype of `AbstractArray`. This is again true for arrays of arbitrary order.

```
julia> X = rand(2,5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> getobs(X, 2) # single observation at index 2
2-element Array{Float64,1}:
 0.933372
 0.522172

julia> getobs(X, [1,3,5]) # batch of three observations
2×3 Array{Float64,2}:
 0.226582  0.505208   0.812814
 0.504629  0.0997825  0.245457
```

There are a few subtle but very important details about the above code worth pointing out:

- Notice how the return type of `getobs(::Array, ::Int)` is different from the return type of `getobs(::Array, ::Vector)`. This is allowed and encouraged, because these methods perform conceptually different operations. The first method returns a single observation, while the later returns a batch of observations. The main requirement is that the return type stays consistent for each.
- You may ask yourself why `getobs(::Array, ...)` returns an `Array` instead of a more conservative `SubArray`. This is intentional. The idea behind *`getobs()`* is to be called *once* just shortly before the data is passed to some learning algorithm. That means that we do care deeply about runtime performance aspects at that point, which includes memory locality. This also means that *`getobs()`* is **not** intended for subsetting or partitioning data; use *`datasubset()`* for that (which does return a `SubArray`).

- The type `Array` is both, data container and data itself. This need not be the case in general. For example, you could implement a special type of data container called `MyContainer` that returns an `Array` as its data when the method `getobs(::MyContainer, ...)` is called.

We mentioned before that the default assumption is that the last array dimension enumerates the observations. This can be overwritten by explicitly specifying the `obsdim`. To visualize what we mean, let us consider the following 3-d array as some example data container.

```
julia> X = rand(2,3,4)
2×3×4 Array{Float64,3}:
[:, :, 1] =
 0.226582  0.933372  0.505208
 0.504629  0.522172  0.0997825

[:, :, 2] =
 0.0443222  0.812814  0.11202
 0.722906   0.245457  0.000341996

[:, :, 3] =
 0.380001  0.841177  0.810857
 0.505277  0.326561  0.850456

[:, :, 4] =
 0.478053  0.44701   0.677372
 0.179066  0.219519  0.746407
```

Now what if we are interested in the observation with the index `1`. There are different interpretations of what that could mean. The following code shows the three possible choices for this example.

```
julia> getobs(X, 1) # defaults to ObsDim.Last()
2×3 Array{Float64,2}:
 0.226582  0.933372  0.505208
 0.504629  0.522172  0.0997825

julia> getobs(X, 1, obsdim = 2)
2×4 Array{Float64,2}:
 0.226582  0.0443222  0.380001  0.478053
 0.504629  0.722906   0.505277  0.179066

julia> getobs(X, 1, obsdim = 1)
3×4 Array{Float64,2}:
 0.226582  0.0443222  0.380001  0.478053
 0.933372  0.812814   0.841177  0.44701
 0.505208  0.11202    0.810857  0.677372

julia> getobs(X, 1, ObsDim.First()) # same as above but type-stable
3×4 Array{Float64,2}:
 0.226582  0.0443222  0.380001  0.478053
 0.933372  0.812814   0.841177  0.44701
 0.505208  0.11202    0.810857  0.677372
```

At this point it is worth to again (and maybe redundantly) point out two facts, that we have already established when introducing *`nobs()`*:

- If there is more than one array dimension, all but the observation dimension are implicitly assumed to be features (i.e. part of that observation). This implies that for an array, the individual observations have to be explicitly laid out along a single dimension.
- Note how `obsdim` can either be provided using a type-stable positional argument from the namespace `ObsDim`,

or by using a more flexible and convenient keyword argument. We will discuss observation dimensions in more detail in a later section.

It is also possible to link multiple different data containers together on an per-observation level. To do that, simply put all the relevant data container into a single `Tuple`, before passing it to *`getobs()`* (or other functions that expect a data container). The return value will then be a `Tuple` of the same length, with the resulting data in the same tuple-order.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> y = rand(4)
4-element Array{Float64,1}:
 0.812814
 0.245457
 0.11202
 0.000341996

julia> getobs((X,y), 2) # single observation at index 2
([0.933372,0.522172],0.24545709827626805)

julia> getobs((X,y), [2,4]) # batch of 2 observations
([0.933372 0.0443222; 0.522172 0.722906], [0.245457,0.000341996])
```

It is worth pointing out, that the tuple elements (i.e. data container) need not be of the same type, nor of the same shape. You can observe this in the code above, where `X` is a `Matrix` while `y` is a `Vector`. Note, however, that all tuple elements must be data containers themselves. Furthermore, they all must contain the same exact number of observations. This is required, even if the requested observation-index would be in-bounds for each data container individually.

```
julia> getobs((rand(3), rand(4)), 2)
ERROR: DimensionMismatch("all data container must have the same number of observations
↪")
[...]
```

When grouping data containers in a `Tuple`, it is also possible to specify multiple `obsdim` for each data container (if need be). Note that if `obsdim` is specified as a `Tuple`, then it needs to have the same number of elements as the `Tuple` of data containers.

```
julia> getobs((X,y), 2, obsdim = :last)
([0.933372,0.522172],0.24545709827626805)

julia> getobs((X,y), 2, obsdim = (2,1))
([0.933372,0.522172],0.24545709827626805)

julia> getobs((X,y), 2, ObsDim.Last())
([0.933372,0.522172],0.24545709827626805)

julia> getobs((X,y), 2, (ObsDim.Last(),ObsDim.Last()))
([0.933372,0.522172],0.24545709827626805)

julia> getobs((X',y), 2, (ObsDim.First(),ObsDim.Last())) # note the transpose
([0.933372,0.522172],0.24545709827626805)
```

Aside from the main signature for *`getobs()`*, it is also possible to call it without specifying any observation index/indices.

```
julia> X = rand(2,5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> getobs(X)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457
```

You may ask yourself what the purpose of this particular method is. It is particularly useful for converting a data container into the actual data that it represents. In contrast to calling `getobs(X, 1:nobs(X))`, `getobs(X)` will not cause any memory allocation if the given data `X` already is an `Array`. In other words, its main purpose is for a user to be able to call `X = getobs(mysubset)` right before passing `X` to some learning algorithm. This should make sure that `X` is not a `SubArray` or *DataSubset* anymore, without causing overhead in case `mysubset` already is an `Array` (in which case `X === mysubset`).

```
julia> X = rand(2,5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> @assert getobs(X) === X # will NOT copy

julia> Xv = view(X, :, :) # just to create a SubArray
2×5 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Colon},true}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> getobs(Xv) # will copy and return a new array
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457
```

By default `getobs(X, obsdim)` will call `getobs(data, 1:nobs(data,obsdim), obsdim)` for any type of `data` that does not provide a custom method for it. If you are a package author and your type has a more efficient (or conservative) way to return the complete data set, you need to implement this method yourself.

So far we have only discussed how to query observation(s) without any regard for preallocation of the underlying memory. To achieve competitive performance, however, it can be very crucial to reuse memory if at all possible for the given data. For that purpose we provide a mutating variant of *getobs()* called *getobs!()*.

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

Write the observation(s) from *data* that correspond to the given index/indices in *idx* into *buffer*. Note that *idx* can be of type `Int` or `AbstractVector`. Both options should be supported.

Inplace version of *getobs()* using the preallocated *buffer*. 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. This in turn can significantly improve the memory footprint of various data access pattern.

Unless specifically implemented for the type of *data*, it defaults to returning `getobs(data, idx, obsdim)`, in which case *buffer* is ignored.

Parameters

- **buffer** – The preallocated storage to copy the given observations of *data* into. *Note:* The type and structure should be equivalent to the return value of the corresponding *getobs()* call, since this is how *buffer* is preallocated by some higher-level functions.

- **data** – The object representing a data container.
- **idx** – Optional. The index or indices of the observation(s) in *data* that should be written into *buffer*. Can be of type `Int` or some subtype `AbstractVector{Int}`.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns Either the mutated *buffer* if preallocation is supported by *data*, or the result of calling *getobs()* otherwise.

```
julia> batch = Matrix{Float64}(2,4) # allocate buffer

julia> data = rand(2,10)
2×10 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  ...  0.841177  0.810857  0.478053
 0.504629  0.522172  0.0997825  0.722906       0.326561  0.850456  0.179066

julia> getobs!(batch, data, [1,3,4,6]) # write 4 observations into batch
2×4 Array{Float64,2}:
 0.226582  0.505208   0.0443222  0.11202
 0.504629  0.0997825  0.722906   0.000341996
```

Note that in contrast to typical mutating functions, *getobs!()* does not always actually use `buffer` to store the result. This is because some types of data container may not support the concept of preallocation, in which case the default implementation will ignore `buffer` and just return the result of calling *getobs()* instead. This controversial design decision was made for the sake of compatibility. This way, higher-level functions such as *eachobs()* can benefit from preallocation if supported by `data`, but will still work for data container that do not support it.

3.1.4 Sample Observation(s)

Aside from requesting specific observations, we also allow to sample observations at random. To that end we provide a convenience function *randobs()*.

randobs (*data*[, *n*][, *obsdim*])

Sample a random observation or a batch of *n* random observations from data. The sampling is performed with replacement.

Parameters

- **data** – The object representing a data container.
- **n** (*Int*) – Optional. The number of observations to sample. If omitted a single observation is returned. Note that omitting is not equivalent to setting `n = 1`. The latter will return a batch with just one observation in it.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Just like for *nobs()* and *getobs()*, this package natively provides a *randobs()* implementation for any subtype of `AbstractArray`. This is again true for arrays of arbitrary order.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
```

(continues on next page)

(continued from previous page)

```
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> randobs(X, 3) # batch of observations
2×3 Array{Float64,2}:
 0.505208   0.933372  0.0443222
 0.0997825  0.522172  0.722906

julia> randobs(X) # single observation
2-element Array{Float64,1}:
 0.505208
 0.0997825

julia> randobs(X, 1) # different to above
2×1 Array{Float64,2}:
 0.0443222
 0.722906

julia> randobs(X', obsdim = 1) # note the transpose
2-element Array{Float64,1}:
 0.226582
 0.504629
```

Similar to *getobs()*, you can again use a `Tuple` to link multiple data containers on a per-observation level.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> y = rand(4)
4-element Array{Float64,1}:
 0.812814
 0.245457
 0.11202
 0.000341996

julia> randobs((X,y)) # single observation
([0.933372,0.522172],0.24545709827626805)

julia> randobs((X,y), 2) # batch of 2 observations
([0.0443222 0.505208; 0.722906 0.0997825], [0.000341996,0.11202])
```

For *randobs()* it is very important to use tuples in this case (in contrast to *getobs()*, where it was optional). This is because here it is crucial that every involved data container samples the same observation index/indices. Otherwise the link would be broken and the resulting observations do not correspond to each other anymore.

```
# WARNING: Wrong code! This is not equivalent to above
julia> randobs(X), randobs(y)
([0.933372,0.522172],0.00034199581283611 56)
```

3.1.5 Observation Dimension

By now we have seen multiple examples for a data container, where there was no clear type-level convention for what exactly denotes an observation. This is primarily the case for the rather important family of data container,

`AbstractArray`.

To see another concrete example, let us consider the following random matrix `X`. This variable will serve as our toy feature matrix.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906
```

Now that we see `X` before us, think about the following questions: What is the number of features, and what is the number of observations? It is the opinion of the package authors, that the correct answer is "it depends", because there are two schools of thought that both have their merits.

1. The typical convention in Statistics, as well as many frameworks, would be that `X` has 2 observations with 4 features each. This convention makes sense and is intuitive, because one could easily see this matrix as a data table with 2 rows and 4 column. Furthermore, this convention is used in a lot of text books and Machine Learning classes.
2. On the other hand, one could be compelled to say that this matrix `X` has 4 observations with 2 features each. This convention is particularly useful for the Julia language, because Julia stores the arrays in column-major order. This means that if we interpret each column as single observation, then all features of a single observation are right next to each other in memory. Making good use of this fact can have a big influence on performance (see the corresponding section of the official documentation).

We decided quite early in the design process, that we want to support both interpretations in a generic way. Furthermore, we also wanted to support data container that don't have the concept of "dimensionality" (i.e. where it is clear for the type what an observation is). To that end, all relevant functions allow for an optional parameter `obsdim`, which can usually be specified as either a keyword argument or a positional argument.

The following two code snippets show different ways to access the first observation of some example feature matrix `X`. In the first snippet we assume that each row of `X` represents an observation.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> getobs(X, 1, obsdim = 1)
4-element Array{Float64,1}:
 0.226582
 0.933372
 0.505208
 0.0443222

julia> getobs(X, 1, obsdim = :first)
4-element Array{Float64,1}:
 0.226582
 0.933372
 0.505208
 0.0443222

julia> getobs(X, 1, ObsDim.First())
4-element Array{Float64,1}:
 0.226582
 0.933372
 0.505208
 0.0443222
```

The second code snippet assumes that each column of `X` represents an observation

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> getobs(X, 1)
2-element Array{Float64,1}:
 0.226582
 0.504629

julia> getobs(X, 1, obsdim = 2)
2-element Array{Float64,1}:
 0.226582
 0.504629

julia> getobs(X, 1, obsdim = :last)
2-element Array{Float64,1}:
 0.226582
 0.504629

julia> getobs(X, 1, ObsDim.Last())
2-element Array{Float64,1}:
 0.226582
 0.504629
```

We can see that the default assumption for arrays is that the last dimension enumerates the observations. Furthermore, it is worth pointing out that while type-unstable, the keyword argument `obsdim` is more convenient to use than the (type-stable) positional API. This is the main reason both alternatives exist, because functionality wise they are identical. The positional argument is intended to be used by code that cares about type stability, while the keyword argument is mainly provided for end-user convenience. Note that in the REPL, the performance impact of using the keyword argument is negligible.

All possible values for the positional API are contained in the namespace `ObsDim`, which itself is provided by the package `LearnBase.jl`.

class `ObsDim.`**Undefined**
: Default value for most data sources. It represents the fact that the concept of an observation dimension is not defined for the given data.

  Can usually be ommited.

class `ObsDim.`**First**
: Defines that the first dimension denotes the observations

class `ObsDim.`**Constant{DIM}**
: Defines that the dimension DIM denotes the observations

class `ObsDim.`**Last**
: Defines that the last dimension denotes the observations

Once we understand what data containers are and how they can be interacted with, we can introduce more interesting behaviour on top of them. The most enabling of them all is the idea of a **data subset**. A data subset is in essence just a lazy representation of a specific sequence of observations from a data container, the sequence itself being another data container. What that means and why that is useful will be discussed in detail in the following section.

3.2 Data Subsets

It is a common requirement in Machine Learning related experiments to partition some data set in one way or the other. At its essence, data partitioning can be thought of as a process that assigns observations to one or more subsets of the original data. This abstraction is also true for other important and widely used data access pattern in machine learning (e.g. over- and under-sampling of a labeled data set).

In other words, the core problem that needs to be addressed efficiently, is how to create and represent a **data subset** in a generic way. Once we can subset arbitrary index-based data, more complicated tasks such as data *partitioning*, *shuffling*, or *resampling* can be expressed through data subsetting in a coherent manner.

Before we move on, let us quickly clarify what exactly we mean when we talk about a data "subset". We don't think about the term "subset" in the mathematical sense of the word. Instead, when we attempt to subset some data source, what we are really interested in, is a representation (aka. subset) of a specific sequence of observations from the original data source. We specify which observations we want to be part of this subset, by using observation-indices from the set $I = \{1, 2, ..., N\}$. Here N is the total number of observations in our data source. This interpretation of "subset" implies the following:

1. We can only subset data sources that are considered data container. Furthermore, a subset of a data container is again considered a data container.
2. When specifying a subset, the order of the requested observation-indices matter. That means that different index permutations will cause conceptually different "subsets".
3. A subset can contain the same exact observation for an arbitrary number of times (including zero). Furthermore, an observation can be part of multiple distinct subsets.

We will spend the rest of this document discussing how to use this package to create data subsets and how to interact with them. After introducing the basics, we will go over the multiple high-level functions that create data subsets for you. These include splitting your data into train and test portions, shuffling your data, and resampling your data using a k-folds partitioning scheme.

3.2.1 Subsetting a Data Container

We have seen before that when confronted with a **data container**, nesting various subsetting operations really just breaks down to keeping track of the observation-*indices*. This in turn is much cheaper than copying observation-*values* around needlessly (see *Background and Motivation* for an in-depth discussion).

Ideally, when we "subset" a data container, what we want is a lazy representation of that subset. In other words, we would like to avoid copying the values of our data set around until we actually need it. To that end, we provide the function `datasubset()`, which tries to choose the most appropriate type of subset for the given data container.

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

Returns a lazy subset of the observations in *data* that correspond to the given index/indices in *idx*. No data will be copied except of the indices

This function is similar to calling the constructor for `DataSubset`, with the main difference that `datasubset()` will return a `SubArray` if the type of *data* is an `Array` or `SubArray`. Furthermore, this function can be extended for custom types of *data* that also want to provide their own subset-type.

The returned subset will in general not be of the same type as the underlying observations it represents. If you want to query the actual observations corresponding to the given indices in their true form, use `getobs()` instead.

Parameters

- **`data`** – The object representing a data container.

- **idx** – Optional. The index or indices of the observation(s) in *data* that should be part of the subset. Can be of type `Int` or some subtype `AbstractVector{Int}`. Defaults to `1:nobs(data,obsdim)`
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns An object representing a lazy subset of *data* for the observation-indices in *idx*. The type of the return value depends on the type of *data*.

Out of the box, this package provides *custom* support for all subtypes of `AbstractArray`. With the exception of sparse arrays, we represent all subsets of arrays in the form of a `SubArray`. To give a concrete example of what we mean, let us consider the following random matrix `X`. We will think about it as a small data set that has 4 observations with 2 features each.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> datasubset(X, 2) # single observation at index 2
2-element SubArray{Float64,1,Array{Float64,2},Tuple{Colon,Int64},true}:
 0.933372
 0.522172

julia> datasubset(X, [2,4]) # batch of 2 observations
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.933372  0.0443222
 0.522172  0.722906
```

If there is more than one array dimension, all but the observation dimension are implicitly assumed to be features (i.e. part of that observation). As you can see in the example above, the default assumption is that the last array dimension enumerates the observations. This can be overwritten by explicitly specifying the `obsdim`. In the following code snippet we treat `X` as a data set that has 2 observations with 4 features each.

```
julia> datasubset(X, 2, ObsDim.First())
4-element SubArray{Float64,1,Array{Float64,2},Tuple{Int64,Colon},true}:
 0.504629
 0.522172
 0.0997825
 0.722906

julia> datasubset(X, 2, obsdim = 1)
4-element SubArray{Float64,1,Array{Float64,2},Tuple{Int64,Colon},true}:
 0.504629
 0.522172
 0.0997825
 0.722906
```

Note how `obsdim` can either be provided using a type-stable positional argument from the namespace `ObsDim`, or by using a more flexible and convenient keyword argument. For more take a look at *Observation Dimension*.

Remember that every data subset - which includes `SubArray` - is again a fully qualified data container. As such, it supports both *nobs()* and *getobs()*.

```
julia> mysubset = datasubset(X, [2,4]) # batch of 2 observations
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.933372  0.0443222
 0.522172  0.722906

julia> nobs(mysubset)
2

julia> getobs(mysubset)
2×2 Array{Float64,2}:
 0.933372  0.0443222
 0.522172  0.722906
```

Because a `SubArray` is also a data container, it can be subsetted even further by using *`datasubset()`* again. The result of which will be a new `SubArray` into the original data container `X`. As such it will use the accumulated indices of both subsetting steps. In other words, while subsetting operations can be nested, they will be combined into a single layer (i.e. you don't want a subset of a subset of a subset represented as nested types)

```
julia> datasubset(mysubset, 1) # will still be a view into X
2-element SubArray{Float64,1,Array{Float64,2},Tuple{Colon,Int64},true}:
 0.933372
 0.522172
```

It is also possible to link multiple different data containers together on an per-observation level. This way they can be subsetted as one coherent unit. To do that, simply put all the relevant data container into a single `Tuple`, before passing it to *`datasubset()`* (or any other function that expect a data container). The return value will then be a `Tuple` of the same length, with the resulting data subsets in the same tuple position.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> y = rand(4)
4-element Array{Float64,1}:
 0.812814
 0.245457
 0.11202
 0.000341996

julia> datasubset((X,y), 2) # single observation at index 2
([0.933372,0.522172],0.24545709827626805)

julia> Xs, ys = datasubset((X,y), [2,4]) # batch of 2 observations
([0.933372 0.0443222; 0.522172 0.722906], [0.245457,0.000341996])

julia> Xs
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.933372  0.0443222
 0.522172  0.722906

julia> ys
2-element SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}:
 0.245457
 0.000341996
```

It is worth pointing out, that the tuple elements (i.e. data container) need not be of the same type, nor of the same

shape. You can observe this in the code above, where `X` is a `Matrix` while `y` is a `Vector`. Note, however, that all tuple elements must be data containers themselves. Furthermore, they all must contain the same exact number of observations. This is required, even if the requested observation-index would be in-bounds for each data container individually.

```
julia> datasubset((rand(3), rand(4)), 2)
ERROR: DimensionMismatch("all data container must have the same number of observations
↪")
[...]
```

When grouping data containers in a `Tuple`, it is of course possible to specify the `obsdim` for each data container. If all data container share the same observation dimension, it suffices to specify it once.

```
julia> Xs, ys = datasubset((X,y), [2,4], obsdim = :last);

julia> Xs
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.933372  0.0443222
 0.522172  0.722906

julia> ys
2-element SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}:
 0.245457
 0.000341996
```

Note that if `obsdim` is specified as a `Tuple`, then it needs to have the same number of elements as the `Tuple` of data containers.

```
julia> Xs, ys = datasubset((X,y), [2,4], obsdim = (2,1));

julia> Xs
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.933372  0.0443222
 0.522172  0.722906

julia> ys
2-element SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}:
 0.245457
 0.000341996
```

Multiple `obsdim` can of course also be specified using type-stable positional arguments.

```
julia> Xs, ys = datasubset((X',y), [2,4], (ObsDim.First(),ObsDim.Last())); # note the␣
↪transpose

julia> Xs
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Array{Int64,1},Colon},false}:
 0.933372   0.522172
 0.0443222  0.722906

julia> ys
2-element SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}:
 0.245457
 0.000341996
```

3.2.2 The DataSubset Type

So far we have only considered subsetting data container of type `Array`. However, what if we want to subset some other data container that does not implement the `AbstractArray` interface? Naturally, we can't just use `SubArray` to represent those subsets. For that reason we provide a generic type *DataSubset*, that serves as the default subset type for every data container that does not implement their own methods for *datasubset()*.

class DataSubset

Used as the default type to represent a subset of some arbitrary data container. Its main task is to keep track of which observation-indices the subset spans. As such it is designed in a way that makes sure that subsequent subsettings are accumulated without needing to access the actual data.

The main purpose for the existence of *DataSubset* is to delay data-access and -movement until an actual batch of data (or single observation) is needed for some computation. This is particularly useful when the data is not located in memory, but on the hard drive or some remote location. In such a scenario one wants to load the required data only when needed.

DataSubset (*data*[, *idx*][, *obsdim*]) → DataSubset

Create an instance of *DataSubset* that will represent a lazy subset of the observations in *data* corresponding to the given index/indices in *idx*. No data will be copied except of the indices.

If *data* is a *DataSubset*, then the indices of the subset will be combined with *idx* and consequently an accumulated *DataSubset* will be created and returned.

In general we advice to use *datasubset()* instead of calling *DataSubset()* directly. This is because *datasubset()* will only invoke *DataSubset()* if there is no alternative choice of subset-type known for the given *data*.

Parameters

- **data** – The object representing a data container.
- **idx** – Optional. The index or indices of the observation(s) in *data* that should be part of the subset. Can be of type `Int` or some subtype `AbstractVector{Int}`. Defaults to `1:nobs(data,obsdim)`
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

The type *DataSubset* can be used to represent a subset of any type of data container. This even includes arrays, which we have seen provide their own special type of subset.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> DataSubset(X, 2) # single observation at index 2
DataSubset(::Array{Float64,2}, ::Int64, ObsDim.Last())
 1 observations

julia> DataSubset(X, [2, 4]) # batch of 2 observations
DataSubset(::Array{Float64,2}, ::Array{Int64,1}, ObsDim.Last())
 2 observations
```

As you can see, a *DataSubset* does not tell you a lot of information about the observations it represents. The reason for this is that it was designed around the requirement of not needlessly accessing actual data unless requested using

getobs(). That said, remember that every data subset is also a fully qualified data container. As such, it supports both *nobs()* and *getobs()*.

```
julia> mysubset = DataSubset(X, [2, 4]) # batch of 2 observations
DataSubset(::Array{Float64,2}, ::Array{Int64,1}, ObsDim.Last())
 2 observations

julia> nobs(mysubset)
2

julia> getobs(mysubset) # request the data it represents
2×2 Array{Float64,2}:
 0.933372  0.0443222
 0.522172  0.722906
```

The real strength of the *DataSubset* type (or any data subset really), is that it can be subsetted even further. The result of which will be a new *DataSubset* into the original data container `X` that uses the accumulated indices. In other words, while subsetting operations can be nested, they will be combined into a single layer (i.e. you don't want a subset of a subset of a subset represented as nested types)

```
julia> mysubset2 = DataSubset(mysubset, 2) # second observation of mysubset
DataSubset(::Array{Float64,2}, ::Int64, ObsDim.Last())
 1 observations

julia> getobs(mysubset2) # request the data it represents
2-element Array{Float64,1}:
 0.0443222
 0.722906
```

As you can see in the example above, *DataSubset* also stores the utilized `obsdim`. Because we are using an `Array` as example data container, the default assumption is that the last array dimension enumerates the observations. This can be overwritten by explicitly specifying the `obsdim`. As always, the `obsdim` can be specified in a type-stable manner using a positional argument, or by using a more convenient keyword argument.

```
julia> mysubset = DataSubset(X', 2, obsdim = 1) # note the transpose
DataSubset(::Array{Float64,2}, ::Int64, ObsDim.Constant{1}())
 1 observations

julia> getobs(mysubset)
2-element Array{Float64,1}:
  0.933372
  0.522172
```

It is worth pointing out that *DataSubset* remembers the specified `obsdim`, which means that it is not required to specify it again for subsequent data access pattern. In contrast to this, a `SubArray` does not have the means to remember it, and as such one needs to specify the `obsdim` every time.

It is also possible to link multiple different data containers together on an per-observation level. This way they can be subsetted as one coherent unit. To do that, simply put all the relevant data container into a single `Tuple`, before passing it to *DataSubset()*.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> y = rand(4)
```

(continues on next page)

(continued from previous page)

```
4-element Array{Float64,1}:
 0.812814
 0.245457
 0.11202
 0.000341996

julia> Xs, ys = DataSubset((X,y), [2,4]); # batch of 2 observations
(DataSubset(::Array{Float64,2}, ::Array{Int64,1}, ObsDim.Last())
  2 observations,
 DataSubset(::Array{Float64,1}, ::Array{Int64,1}, ObsDim.Last())
  2 observations)

julia> getobs(Xs)
2×2 Array{Float64,2}:
 0.933372  0.0443222
 0.522172  0.722906

julia> getobs(ys)
2-element Array{Float64,1}:
 0.245457
 0.000341996
```

Note something subtle but important in the code snippet above. The constructor *DataSubset()* does not return a *DataSubset* when it is called with a tuple of data containers. Instead, it maps the constructor onto each data container individually. Thus if we invoke *DataSubset()* with a `Tuple`, it will return a `Tuple` of *DataSubset*.

3.2.3 Support for Custom Types

We have seen in the previous section what the type *DataSubset* is, and why it exists. We also mentioned that an end-user does not usually need to work with the constructor *DataSubset()* directly. Instead, we recommended to always just use *datasubset()* instead.

You may ask yourself right now why we were using this *DataSubset* type in the first place. After all, we saw that calling the function *datasubset()* gave us a more convenient `SubArray` to work with. Well, as we hinted before, not every data container can be expected to be a subtype of `AbstractArray`. To get a better understanding of why we care about this, let us together explore the implications on a couple of commonly used data sources that are available in the Julia package ecosystem.

Example: DataFrames.jl

Note: If you are using MLDataUtils.jl then support for `DataFrame` is already provided for you.

Let's consider a type of data source that is very different to an `Array`; a `DataFrame` from the DataFrames.jl package. By default, a `DataFrame` is not a data container, because it does not implement the required interface. We can change that however.

```
julia> using DataFrames, LearnBase

julia> LearnBase.getobs(df::DataFrame, idx) = df[idx,:]

julia> StatsBase.nobs(df::DataFrame) = nrow(df)
```

With those two methods defined, every `DataFrame` is a fully qualified data container. This means that it can now be subsetted.

```
julia> df = DataFrame(x1 = rand(4), x2 = rand(4))
4×2 DataFrames.DataFrame
│ Row │ x1       │ x2        │
├─────┼──────────┼───────────┤
│ 1   │ 0.226582 │ 0.505208  │
│ 2   │ 0.504629 │ 0.0997825 │
│ 3   │ 0.933372 │ 0.0443222 │
│ 4   │ 0.522172 │ 0.722906  │

julia> mysubset = datasubset(df, [2,4])
DataSubset(::DataFrames.DataFrame, ::Array{Int64,1})
 2 observations

julia> getobs(mysubset)
2×2 DataFrames.DataFrame
│ Row │ x1       │ x2        │
├─────┼──────────┼───────────┤
│ 1   │ 0.504629 │ 0.0997825 │
│ 2   │ 0.522172 │ 0.722906  │
```

Notice how we used *`datasubset()`* here, instead of invoking the *`DataSubset()`* constructor directly. This is the recommended way of creating data subsets. The main difference is, that *`datasubset()`* will try to choose the most appropriate type to represent a subset for the given container, while the constructor will always use *`DataSubset`*. For this example we did not specify any special kind of data subset for `DataFrame`, and thus the default *`DataSubset`* is used.

Example: DataTables.jl

Another good example for a custom data source are `DataTable` from the DataTables.jl package. This rather new, table-like type is advertised as the "future of working with tabular data in Julia". To make it more interesting after the `DataFrame` example, we will also make use of a native view-type called `SubDataTable`, which is a perfect candidate for a custom data subset type.

Not unlike `DataFrame`, a `DataTable` is by default not a data container, because it does not implement the required interface. We will again change that. In contrast to before, however, we will also implement a custom method for *`datasubset()`*.

```
julia> using DataTables, LearnBase

julia> StatsBase.nobs(dt::AbstractDataTable) = nrow(dt)

julia> LearnBase.getobs(dt::AbstractDataTable, idx) = dt[idx,:]

julia> LearnBase.datasubset(dt::AbstractDataTable, idx, ::ObsDim.Undefined) = view(dt,
↪ idx)
```

It is worth pointing out that it is a current limitation that any custom method for *`datasubset()`* must also include the third parameter `obsdim` (even if it is undefined).

Now that we have the required interface implemented, every `DataTable` is regarded as a fully qualified data container. In contrast to the `DataFrame` example, it even has its own custom type for representing a data subset (Note that we could also do the same thing for `DataFrame` using the type `SubDataFrame`).

```
julia> dt = DataTable(x1 = rand(4), x2 = rand(4))
4×2 DataTables.DataTable
| Row | x1       | x2        |
|-----|----------|-----------|
| 1   | 0.226582 | 0.505208  |
| 2   | 0.504629 | 0.0997825 |
| 3   | 0.933372 | 0.0443222 |
| 4   | 0.522172 | 0.722906  |

julia> mysubset = datasubset(dt, [2, 4])
2×2 DataTables.SubDataTable{Array{Int64,1}}
| Row | x1       | x2        |
|-----|----------|-----------|
| 1   | 0.504629 | 0.0997825 |
| 2   | 0.522172 | 0.722906  |

julia> datasubset(mysubset, 2) # subsetting a subset
1×2 DataTables.SubDataTable{Array{Int64,1}}
| Row | x1       | x2       |
|-----|----------|----------|
| 1   | 0.522172 | 0.722906 |

julia> getobs(mysubset)
2×2 DataTables.DataTable
| Row | x1       | x2        |
|-----|----------|-----------|
| 1   | 0.504629 | 0.0997825 |
| 2   | 0.522172 | 0.722906  |
```

One may ask why we go through this trouble, if we could just use `Base.view` instead. Aside from the observation dimension aspect when working with arrays, there are good reason for having such a neutral interface. After all, a data subset is just a means to an end. We will see in the following sections how higher-level functions can create various data subsets in much more useful ways than us just calling *`datasubset()`* ourselves. So once some data source supports the data container interface, all the high-level functionality that we will spend the rest of this document on, comes with it for free.

3.2.4 Shuffling a Data Container

A vastly under-appreciated duty of any Machine Learning framework is shuffling a data set (or parts of a data set). Shuffling the order of the observations before training a model on that data set is important for various practical and well known reasons. We still call it under-appreciated, however, because it is easy to implement "shuffling" inefficiently. That in turn can influence a lot of dependent functionality; especially if big data sets are involved. For example, it is not unusual that the shuffling is performed very early in the ML pipeline. Depending on the design of the framework, this could cause a lot of unnecessary data movement.

In this package we follow the simple idea, that the "shuffling" of a data set should be performed on an indices level, and not an observation level. What that means is that instead of copying or mutating the actual data, we simply create a lazy "subset" of that data using shuffled indices. As a consequence, the actual data remains untouched by the process until *`getobs()`* is called. In other words, while the resulting subset points to the same observations, it has the order of the indices shuffled. The function that implements this functionality is called *`shuffleobs()`*.

`shuffleobs`(*data*[, *obsdim*])

Return a "subset" of *data* that spans the same exact observations, but has the order of those observations permuted.

The values of *data* itself are not copied. Instead only the indices are shuffled. This function calls

datasubset() to accomplish that, which means that the return value is likely of a different type than data.

Parameters

- **data** – The object representing a data container.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

This is where we will start to see the subtle beauty of the package design. We have previously discussed in some detail how to interact (and subset) data containers such as `Array`, `DataTable`, and `DataFrame`. Let us now take a look at what it means to "shuffle" each of those. First, we will consider a plain Julia `Array`.

```
julia> X = rand(2,4)
2×4 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> X_shuf = shuffleobs(X) # each column is an observation
2×4 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.933372  0.505208   0.0443222  0.226582
 0.522172  0.0997825  0.722906   0.504629

julia> getobs(X_shuf) # copy into a Array
2×4 Array{Float64,2}:
 0.933372  0.505208   0.0443222  0.226582
 0.522172  0.0997825  0.722906   0.504629

julia> shuffleobs(X, obsdim = 1) # each row is an observation
2×4 SubArray{Float64,2,Array{Float64,2},Tuple{Array{Int64,1},Colon},false}:
 0.504629  0.522172  0.0997825  0.722906
 0.226582  0.933372  0.505208   0.0443222
```

As we can see, *shuffleobs()* returns a `SubArray` instead of an `Array`. As such, it still points at the data in `X`. To get the actual data as a proper `Array` (e.g. for memory locality) we can use *getobs()* on the result. Also note how the result of *shuffleobs()* depends on the specified `obsdim`. This is because we just want to permute the order of the observations, not the features.

Next we will take a look at what happens when we call *shuffleobs()* with a `DataTable`. Note that for this to work it is required that the data container interface is implemented (which we did as an exercise in *Example: DataTables.jl*)

```
julia> dt = DataTable(x1 = rand(4), x2 = rand(4))
4×2 DataTables.DataTable
| Row | x1       | x2        |
|-----|----------|-----------|
| 1   | 0.226582 | 0.505208  |
| 2   | 0.504629 | 0.0997825 |
| 3   | 0.933372 | 0.0443222 |
| 4   | 0.522172 | 0.722906  |

julia> dt_shuf = shuffleobs(dt)
4×2 DataTables.SubDataTable{Array{Int64,1}}
| Row | x1       | x2        |
|-----|----------|-----------|
| 1   | 0.933372 | 0.0443222 |
| 2   | 0.504629 | 0.0997825 |
```

(continues on next page)

(continued from previous page)

```
| 3   | 0.226582 | 0.505208 |
| 4   | 0.522172 | 0.722906 |

julia> getobs(dt_shuf)
4×2 DataTables.DataTable
| Row | x1       | x2       |
|-----|----------|----------|
| 1   | 0.933372 | 0.0443222|
| 2   | 0.504629 | 0.0997825|
| 3   | 0.226582 | 0.505208 |
| 4   | 0.522172 | 0.722906 |
```

Note how the actual code did not change much, even though `DataTables` are quite different to `Array`. We can again observe how *`shuffleobs()`* did not return a new `DataTable`, but instead a lazy view in the form of a `SubDataTable`.

To mix it up a little, let us take a look at a data container that does not provide its own type of data subset; a `DataFrame`. Note that for the following code to work, it is required that the data container interface is implemented (which we did as an exercise in *Example: DataFrames.jl*)

```
julia> df = DataFrame(x1 = rand(4), x2 = rand(4))
4×2 DataFrames.DataFrame
| Row | x1       | x2       |
|-----|----------|----------|
| 1   | 0.226582 | 0.505208 |
| 2   | 0.504629 | 0.0997825|
| 3   | 0.933372 | 0.0443222|
| 4   | 0.522172 | 0.722906 |

julia> df_shuf = shuffleobs(df)
DataSubset(::DataFrames.DataFrame, ::Array{Int64,1})
 4 observations

julia> getobs(df_shuf)
4×2 DataFrames.DataFrame
| Row | x1       | x2       |
|-----|----------|----------|
| 1   | 0.933372 | 0.0443222|
| 2   | 0.504629 | 0.0997825|
| 3   | 0.226582 | 0.505208 |
| 4   | 0.522172 | 0.722906 |
```

Admittedly, the result of *`shuffleobs()`* does not look as intuitive or information in this example. It does however do its job perfectly, which is avoiding data access. This property of *`DataSubset`* is particularly useful if our data container is some interface to a big remote data set. In such a case we would like to avoid loading any data until we really need it.

Aside from a common interface for different data types, the real power of using *`shuffleobs()`* is in linking multiple data containers together on an per-observation level. This way they can be shuffled as one coherent unit. To do that, simply put all the relevant data container into a single `Tuple`, before passing it to *`shuffleobs()`*. For example, let's say that our features are contained in a `DataTable` and the targets stored in a separate `Vector`.

```
julia> dt = DataTable(x1 = rand(4), x2 = rand(4))
4×2 DataTables.DataTable
| Row | x1       | x2       |
|-----|----------|----------|
```

(continues on next page)

(continued from previous page)

```
| 1   | 0.226582 | 0.505208  |
| 2   | 0.504629 | 0.0997825 |
| 3   | 0.933372 | 0.0443222 |
| 4   | 0.522172 | 0.722906  |

julia> y = rand(4)
4-element Array{Float64,1}:
 0.812814
 0.245457
 0.11202
 0.000341996

julia> df_shuf, y_shuf = shuffleobs((dt, y))
(4×2 DataTables.SubDataTable{Array{Int64,1}}
| Row | x1       | x2        |
|-----|----------|-----------|
| 1   | 0.504629 | 0.0997825 |
| 2   | 0.933372 | 0.0443222 |
| 3   | 0.522172 | 0.722906  |
| 4   | 0.226582 | 0.505208  |,[0.245457,0.11202,0.000341996,0.812814])

julia> typeof(y_shuf)
SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}
```

As we can see, the observations in `dt` and `y` are both shuffled in the same manner. Thus the per-observation link is preserved and we can continue to treat it as a single data set.

3.2.5 Splitting into Train and Test

Some data preparation tasks, such as partitioning the data set into a training-, (validation-,) and test-set, are often performed offline or sometimes even predefined by a third party (e.g. the initial authors of a benchmark data set). That said, it is useful to efficiently and conveniently be able to split a given data set into differently sized subsets. For that purpose, this package provides a function called *`splitobs()`*. As the name subtly hints, this function does not shuffle the content, but instead performs a static split at the relative position specified in `at`.

To begin with, *`splitobs()`* provides a method to pre-compute a partition that is applicable to any data set of some fixed size.

splitobs (*n*[, *at = 0.7*]) → Tuple

Compute the indices for two disjoint subsets and return them as a tuple of two ranges. The first range will span the first *at* fraction of possible indices, while the second range will cover the rest. These indices are applicable to any data container of size *n*.

Parameters

- **n** (*Integer*) – Total number of observations to compute the partition indices for.
- **at** (*AbstractFloat*) – Optional. The fraction of observations that should be in the first subset. Must be in the interval (0,1). Can be specified as positional or keyword argument. Defaults to 0.7 (i.e. 70% of the observations in the first subset).

The following code snippet will pre-compute the subset indices for a training- and a test portion of some data set that has 100 observations in it. The training indices will cover 70% of the observations, while the test indices will cover the other 30%

```
julia> train_idx, test_idx = splitobs(100, at = 0.7)
(1:70,71:100)
```

These pre-computed indices could then be used to create the subsets of some data container manually. Naturally, most of the time it would be much more convenient to just specify the data and have the function do all the work. To then end we provide a more convenient method for *splitobs()* as well.

splitobs (*data*[, *at = 0.7*][, *obsdim*]) → Tuple

Split the given *data* into two disjoint subsets and returns them as a `Tuple`. The first subset contains the fraction *at* of observations in *data*, and the second subset contains the rest.

Note that this function will perform the splits statically and thus not perform any shuffling or sampling. If you want to perform a random assignment of observations to the subsets, you can use the function in combination with *shuffleobs()*.

Parameters

- **data** – The object representing a data container.
- **at** (*AbstractFloat*) – Optional. The fraction of observations that should be in the first subset. Must be in the interval (0,1). Can be specified as positional or keyword argument. Defaults to 0.7 (i.e. 70% of the observations in the first subset).
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Let's consider an example feature matrix `X` in the form of an `Array`, which has 8 observations with 2 features each.

```
julia> X = rand(2, 8)
2×8 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202      0.380001  0.841177
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996  0.505277  0.326561
```

We can split this data container into two subsets by calling *splitobs()* with the desired relative split point. If `at` is specified as a floating point number, then the return-value will be a `Tuple` with two elements (i.e. subsets), in which the first subset contains the fraction of observations specified by `at` and the second subset contains the rest.

In the following code the first subset `train` will contain the first 60% of the observations and the second subset `test` the rest. Note how we can provide the split point `at` as either a type-stable positional argument, or as a more descriptive keyword argument.

```
julia> train, test = splitobs(X, at = 0.6); # or splitobs(X, 0.6)

julia> train
2×5 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> test
2×3 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.11202      0.380001  0.841177
 0.000341996  0.505277  0.326561
```

It is worth pointing out explicitly, that *splitobs()* works for any type that implements the data container interface. The following code shows a concrete example using a `DataTable` (see *Example: DataTables.jl* to make the following code work).

```
julia> dt = DataTable(x1 = rand(4), x2 = rand(4))
4×2 DataTables.DataTable
| Row | x1        | x2        |
```

(continues on next page)

(continued from previous page)

```
├─────┼──────────┼──────────┤
│ 1   │ 0.226582 │ 0.505208 │
│ 2   │ 0.504629 │ 0.0997825│
│ 3   │ 0.933372 │ 0.0443222│
│ 4   │ 0.522172 │ 0.722906 │

julia> train, test = splitobs(dt, at = 0.8);

julia> train
3×2 DataTables.SubDataTable{UnitRange{Int64}}
│ Row │ x1       │ x2       │
├─────┼──────────┼──────────┤
│ 1   │ 0.226582 │ 0.505208 │
│ 2   │ 0.504629 │ 0.0997825│
│ 3   │ 0.933372 │ 0.0443222│

julia> test
1×2 DataTables.SubDataTable{UnitRange{Int64}}
│ Row │ x1       │ x2       │
├─────┼──────────┼──────────┤
│ 1   │ 0.522172 │ 0.722906 │
```

Naturally, *splitobs()* also supports the optional parameter `obsdim`, which is especially useful for arrays. It can be specified as either a positional argument, or as a keyword argument. See *Observation Dimension* for more information.

```
julia> train, test = splitobs(X', at = 0.6); # note the transpose

julia> train
5×2 SubArray{Float64,2,Array{Float64,2},Tuple{UnitRange{Int64},Colon},false}:
 0.226582   0.504629
 0.933372   0.522172
 0.505208   0.0997825
 0.0443222  0.722906
 0.812814   0.245457

julia> test
3×2 SubArray{Float64,2,Array{Float64,2},Tuple{UnitRange{Int64},Colon},false}:
 0.11202   0.000341996
 0.380001  0.505277
 0.841177  0.326561
```

It is also possible to call *splitobs()* with multiple data container wrapped in a `Tuple`, which all must have the same number of total observations. This will link the data containers together on a per-observation basis. Consider the following example feature-matrix `X` and the corresponding target vector `y`. Note how both data container have 8 observations.

```
julia> X = rand(2,8)
2×8 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202      0.380001  0.841177
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996  0.505277  0.326561

julia> y = rand(8)
8-element Array{Float64,1}:
 0.810857
 0.850456
```

(continues on next page)

(continued from previous page)

```
 0.478053
 0.179066
 0.44701
 0.219519
 0.677372
 0.746407
```

We can pass both data containers to *splitobs()* using a tuple to group them together. The result of calling the function will still be a tuple just like in the examples we have seen so far.

```
julia> train, test = splitobs((X, y), at = 0.6);
```

Unlike previous examples, however, both `train` and `test` will themselves be tuples as well. Their elements and order will correspond to the elements of the given data container tuple passed to *splitobs()* (here `(X, Y)`). We can see this explicitly by splatting their elements into variables.

```
julia> (x_train,y_train), (x_test,y_test) = splitobs((X, y), at = 0.6); # same but␣
↪splat

julia> x_train
2×5 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> y_train
5-element SubArray{Float64,1,Array{Float64,1},Tuple{UnitRange{Int64}},true}:
 0.810857
 0.850456
 0.478053
 0.179066
 0.44701

julia> x_test
2×3 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.11202      0.380001  0.841177
 0.000341996  0.505277  0.326561

julia> y_test
3-element SubArray{Float64,1,Array{Float64,1},Tuple{UnitRange{Int64}},true}:
 0.219519
 0.677372
 0.746407
```

As we can see in all previous examples, the function performs a static split and not a random assignment. This may not always be what we really want. For that purpose, this package provides a function called *shuffleobs()*, which we introduced in an earlier section. Using *shuffleobs()* in combination with *splitobs()* will result in a random assignment of observations to the data partitions.

```
julia> train, test = splitobs(shuffleobs(X), at = 0.6);

julia> train
2×5 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.841177  0.812814  0.226582  0.11202      0.933372
 0.326561  0.245457  0.504629  0.000341996  0.522172

julia> test
```

(continues on next page)

(continued from previous page)

```
2×3 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.0443222  0.380001  0.505208
 0.722906   0.505277  0.0997825
```

So far we have only considered how to partition one or more data container into exactly two disjoint data subsets. The function *splitobs()* allows to partition into an arbitrary amount of subsets, however. To partition the given data into N subsets, you simply need to specify a tuple of $N-1$ fractions. The sum of all fractions must be in the interval (0,1).

splitobs(*data*, *at*[, *obsdim*]) → NTuple

Split the given *data* into multiple disjoint subsets with sizes proportional to the value(s) of *at*.

Note that this function will perform the splits statically and thus not perform any randomization. The function creates a `NTuple` of data subsets in which the first $N-1$ elements/subsets contain the fraction of observations from *data* that is specified by the values in *at*. The last tuple element will then contain the rest of the data.

Parameters

- **data** – The object representing a data container.
- **at** (*Tuple*) – Tuple of fractions. All elements must be positive and their sum must be in the interval (0,1).
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Creating more than two data subsets is particularly convenient for creating an additional validation set. In the following example `train` will contain the first 50% of the observations, `val` will have the next 40%, and `test` the last 10%.

```
julia> X = rand(2,8)
2×8 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202      0.380001  0.841177
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996  0.505277  0.326561

julia> train, val, test = splitobs(X, at = (0.5, 0.4));

julia> train
2×4 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.226582  0.933372  0.505208   0.0443222
 0.504629  0.522172  0.0997825  0.722906

julia> val
2×3 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.812814  0.11202      0.380001
 0.245457  0.000341996  0.505277

julia> test
2×1 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.841177
 0.326561
```

While the ability to partitioning a data set this way is very useful, a fixed validation set is rarely the best approach for estimating a model's performance on the held-out test set. In the *Repartitioning Strategies* we will introduce various alternative, including k-folds. These usually allow for a more effective use of the training data.

By this point we know what data containers and data subsets are. In particular, we discussed how we can split our data container into disjoint subsets. We have even seen how we can use tuples to link multiple data container together on a

per-observation level. While we mentioned that this is particularly useful for labeled data, we did not really elaborate on what that means. In order to change that, we will spend the next section solely on working with data containers that have **targets**. This will put us into the realm of supervised learning. We will see how we can work with labeled data containers and what special functionality is available for them.

3.3 Labeled Data Container

Depending on the domain-specific problem and the data one is working with, a data source may or may not contain what is known as **targets**. A target (singular) is a piece of information about a single observation that represents the desired output (or correct answer) for that specific observation. If targets are involved, then we find ourselves in the realm of supervised learning. This includes both, classification (predicting categorical output) and regression (predicting real-valued output).

Dealing with targets in a generic way was quite a design challenge for us. There are a lot of aspects to consider and issues to address in order to achieve an extensible and flexible package architecture, that can deal with labeled- and unlabeled data sources equally well.

- Some data container may not contain any targets or even understand what targets are.
- There are labeled data sources that are not considered *Data Container* (like many data iterators). A flexible package design needs a reasonably consistent API for both cases.
- The targets can be in a different data container than the features. For example it is quite common to store the features in a matrix X, while the corresponding targets are stored in a separate vector $\vec{y}$.
- For some data container, the targets are an intrinsic part of the observations. Furthermore, it might be the case that every data set has its own convention concerning which part of an observation represents the target. An example for such a data container is a `DataFrame` where one column denotes the target. The name/index of the target-column depends on the concrete data set, and is in general different for each `DataFrame`. In other words, this means that for some data containers, the type itself does not know how to access a target. Instead it has to be a user decision.
- There are scenarios, where a data container just serves as an interface to some remote data set, or a big data set that is stored on the disk. If so, it is likely the case, that the targets are not part of the observations, but instead part of the data container metadata. An example would be a data container that represents a nested directory of images in the file system. Each sub-directory would then contain all the images of a single class. In that scenario, the targets are known from the directory names and could be part of some metadata. As such, it would be far more efficient if the data container can make use of this information, instead of having to load an actual image from the disk just to access its target. Remember that targets are not only needed during training itself, but also for data partitioning and resampling.

The implemented target logic is in some ways a bit more complex than the *`getobs()`* logic. The main reason for this is that while *`getobs()`* is solely designed for data containers, we want the target logic to seamlessly support a wide variety of data sources and data scenarios. In this document, however, we will only focus on data sources that are considered labeled *Data Container*. Note that in the context of this package, a **labeled data container** is a data container that contains *targets*; be it categorical or continuous.

3.3.1 Query Target(s)

The first question one may ask is: "Why would the access pattern need to *extract* the targets out of some data container?". After all, it would be simpler to just pass the targets as an additional parameter to any function that needs them. In fact, that is pretty much how almost all other ML frameworks handle labeled data. The reason why we diverge from this tradition is two-fold.

1. The set of access pattern that work on labeled data is really just a superset of the set of access pattern that work on unlabeled data. So by doing it our way, we avoid duplicate code.
2. The second (and more important) reason is that we decided that there is really no convincing argument for restricting the user input to either be in the form of one variable (unlabeled data), or two variables (for labeled data). In fact, we wanted to allow the same variable to contain the features as well as targets. We also wanted to allow users to work with mutliple data sources that don't contain any targets at all.

To that end we provide the function `targets()`. It can be used to query all the, well, targets of some given labeled data container or data subset.

targets (*data*[, *obsdim*])

Query the concrete targets from *data* and return them.

This function is eager in the sense that it will always call `getobs()` unless a custom method for `LearnBase.gettargets` (see later) is implemented for the type of *data*. This will make sure that actual values are returned (in contrast to placeholders such as `DataSubset` or `SubArray`).

In other words, the returned values must be in the form intended to be passed as-is to some resampling strategy or learning algorithm.

Parameters

- **data** – The object representing a labeled data container.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

In some cases we will see that invoking `targets()` just seems to return the given data container unchanged. The reason for this is simple. What `targets()` tries to do is return the portion of the given data container that corresponds to the targets. The function assumes that there *must* be targets of some sorts (otherwise, why would you call a function called "targets"?). If there is no decision to be made (e.g. there is only a single vector to begin with), then the function simply returns the result of `getobs()` for the given data.

```
julia> targets([1,2,3,4])
4-element Array{Int64,1}:
 1
 2
 3
 4
```

The above edge-case isn't really that informative for the main functionality that `targets()` provides. The more interesting behaviour can be seen for custom types and/or tuples. More specifically, if the given data is a `Tuple`, then the convention is that the last element of the tuple contains the targets and the function is recursed once (and only once).

```
julia> targets(([1,2], [3,4]))
2-element Array{Int64,1}:
 3
 4

julia> targets(([1,2], ([3,4], [5,6])))
([3,4],[5,6])
```

What this shows us is that we can use tuples to create a labeled data container out of two simple data containers. This is particularly useful when working with arrays. Considering the following situation, where we have a feature matrix `X` and a corresponding target vector `y`.

```
julia> X = rand(2, 5)
2×5 Array{Float64,2}:
 0.987618  0.365172  0.306373  0.540434  0.805117
 0.801862  0.469959  0.704691  0.405842  0.014829

julia> y = [:a, :a, :b, :a, :b]
5-element Array{Symbol,1}:
 :a
 :a
 :b
 :a
 :b

julia> targets((X, y))
5-element Array{Symbol,1}:
 :a
 :a
 :b
 :a
 :b
```

You may have noticed from the signature of *targets()*, that there is no parameter for passing indices. This is no accident. The purpose of *targets()* is not subsetting, it is to extract the targets; no more, no less. If you wish to only query the targets of a subset of some data container, you can use *targets()* in combination with *datasubset()*.

```
julia> targets(datasubset((X, y), 2:3))
2-element Array{Symbol,1}:
 :a
 :b
```

If the type of the data itself is not sufficient information to be able to extract the targets, one can specify a target-extraction-function `fun` that is to be applied to each observation. This function must be passed as the first parameter to *targets()*.

targets (*fun*, *data*[, *obsdim*]) → Vector

Extract the concrete targets from the observations in *data* by applying *fun* on each observation individually. The extracted targets are returned as a `Vector`, which preserves the order of the observations from *data*.

Parameters

- **fun** – A callable object (usually a function) that should be applied to each observation individually in order to extract or compute the target for that observation.
- **data** – The object representing a labeled data container.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

A great example for a data source, that stores the features and the targets in the same manner, is a `DataFrame`. There is no clear convention what column of the table denotes the targets; it depends on the data set. As such, we require a data-specific target-extraction-function. Consider the following example using a toy `DataFrame` (see *Example: DataFrames.jl* to make the following code work). For this particular data frame we know that the column `:y` contains the targets.

```
julia> df = DataFrame(x1 = rand(5), x2 = rand(5), y = [:a,:a,:b,:a,:b])
5×3 DataFrames.DataFrame
```

(continues on next page)

(continued from previous page)

```
| Row | x1        | x2       | y |
|-----|-----------|----------|---|
| 1   | 0.176654  | 0.821837 | a |
| 2   | 0.0397664 | 0.894399 | a |
| 3   | 0.390938  | 0.29062  | b |
| 4   | 0.582912  | 0.509047 | a |
| 5   | 0.407289  | 0.113006 | b |

julia> targets(row->row.y, df)
5-element Array{Symbol,1}:
 :a
 :a
 :b
 :a
 :b
```

Another use-case for specifying an extraction function, is to discretize some continuous regression targets. We will see later, when we start discussing higher-level functions, how this can be useful in order to over- or under-sample the data set (see *oversample()* or *undersample()*).

```
julia> targets(x -> (x > 0.7), rand(6))
6-element Array{Bool,1}:
  true
 false
  true
 false
  true
  true
```

Note that if this optional first parameter (i.e. the extraction function) is passed to *targets()*, it will always be applied to the observations, and **not** the container. In other words, the first parameter is applied to each observation individually and not to the data as a whole. In general this means that the return type changes drastically, even if passing a no-op function.

```
julia> X = rand(2, 3)
2×3 Array{Float64,2}:
 0.105307   0.58033   0.724643
 0.0433558  0.116124  0.89431

julia> y = [1 3 5; 2 4 6]
2×3 Array{Int64,2}:
 1  3  5
 2  4  6

julia> targets((X,y))
2×3 Array{Int64,2}:
 1  3  5
 2  4  6

julia> targets(x->x, (X,y))
3-element Array{Array{Int64,1},1}:
 [1,2]
 [3,4]
 [5,6]
```

We can see in the above example, that the default assumption for an `Array` of higher order is that the last array dimension enumerates the observations. The optional parameter `obsdim` can be used to explicitly overwrite that

default. If the concept of an observation dimension is not defined for the type of `data`, then `obsdim` can simply be omitted.

```
julia> X = [1 0; 0 1; 1 0]
3×2 Array{Int64,2}:
 1  0
 0  1
 1  0

julia> targets(argmax, X, obsdim=1)
3-element Array{Int64,1}:
 1
 2
 1

julia> targets(argmax, X, ObsDim.First())
3-element Array{Int64,1}:
 1
 2
 1
```

Note how `obsdim` can either be provided using type-stable positional arguments from the namespace `ObsDim`, or by using a more flexible and convenient keyword argument. See *Observation Dimension* for more information on that topic.

3.3.2 Iterate over Targets

In some situations, one only wants to *iterate* over the targets, instead of querying all of them at once. In those scenarios it would be beneficial to avoid the allocation temporary memory all together. To that end we provide the function *`eachtarget()`*, which returns a `Base.Generator`.

`eachtarget`(*[fun]*, *data*[, *obsdim*]) → Generator

Return a `Base.Generator` that iterates over all targets in *data* once and in the right order. If *fun* is provided it will be applied to each observation in data.

Parameters

- **`fun`** – Optional. A callable object (usually a function) that should be applied to each observation individually in order to extract or compute the target for that observation.
- **`data`** – The object representing a labeled data container.
- **`obsdim`** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

The function *`eachtarget()`* behaves very similar to *`targets()`*. For example, if you pass it a `Tuple` of data container, then it will assume that the last tuple element contains the targets.

```
julia> iter = eachtarget(([1,2], [3,4]))
Base.Generator{UnitRange{Int64},MLDataPattern.##79#80{2,Tuple{Array{Int64,1},Array
↪{Int64,1}},Tuple{LearnBase.ObsDim.Last,LearnBase.ObsDim.Last}}}(MLDataPattern.#79,
↪1:2)

julia> collect(iter)
2-element Array{Int64,1}:
```

(continues on next page)

(continued from previous page)

```
 3
 4
```

The one big difference to *targets()* is that *eachtarget()* will always iterate over the targets one observation at a time, regardless whether or not an extraction function is provided.

```
julia> iter = eachtarget([1 3 5; 2 4 6])
Base.Generator{UnitRange{Int64},MLDataPattern.##72#73{Array{Int64,2},LearnBase.ObsDim.
↪Last}}(MLDataPattern.#72,1:3)

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

julia> targets([1 3 5; 2 4 6]) # as comparison
2×3 Array{Int64,2}:
 1  3  5
 2  4  6
```

Of course, it is also possible to work with any other type of data source that is considered a *Data Container*. Consider the following example using a toy `DataFrame` (see *Example: DataFrames.jl* to make the following code work). For this particular data frame we will assume that the column `:y` contains the targets.

```
julia> df = DataFrame(x1 = rand(5), x2 = rand(5), y = [:a,:a,:b,:a,:b])
5×3 DataFrames.DataFrame
│ Row │ x1        │ x2       │ y │
├─────┼───────────┼──────────┼───┤
│ 1   │ 0.176654  │ 0.821837 │ a │
│ 2   │ 0.0397664 │ 0.894399 │ a │
│ 3   │ 0.390938  │ 0.29062  │ b │
│ 4   │ 0.582912  │ 0.509047 │ a │
│ 5   │ 0.407289  │ 0.113006 │ b │

julia> iter = eachtarget(row->row.y, df)
Base.Generator{MLDataPattern.ObsView{MLDataPattern.DataSubset{DataFrames.DataFrame,
↪Int64,LearnBase.ObsDim.Undefined},...

julia> collect(iter)
5-element Array{Symbol,1}:
 :a
 :a
 :b
 :a
 :b
```

Just like for *targets()*, the optional parameter `obsdim` can be used to specify which dimension denotes the observations, if that concept makes sense for the type of the given data.

```
julia> X = [1 0; 0 1; 1 0]
3×2 Array{Int64,2}:
 1  0
 0  1
 1  0
```

(continues on next page)

(continued from previous page)

```
julia> iter = eachtarget(argmax, X, obsdim = 1)
Base.Generator{MLDataPattern.ObsView{SubArray{Int64,1,Array{Int64,2},Tuple{Int64,
↪Colon},true},Array{Int64,2},LearnBase.ObsDim.Constant{1}},...

julia> collect(iter)
3-element Array{Int64,1}:
 1
 2
 1
```

3.3.3 Support for Custom Types

Any labeled data container has the option to customize the behaviour of *targets()*. The emphasis here is on "option", because it is not required by the interface itself. Aside from leaving the default behaviour, there are two ways to customize the logic behind *targets()*.

1. Implement `LearnBase.gettargets` for the **data container** type. This will bypasses the function *getobs()* entirely, which can significantly improve the performance.
2. Implement `LearnBase.gettarget` for the **observation** type, which is applied on the result of *getobs()*. This is useful when the observation itself contains the target.

Let us consider two example scenarios that benefit from implementing custom methods. The first one for `LearnBase.gettargets`, and the second one for `LearnBase.gettarget`. Note again that these functions are internal and only intended to be *extended* by the user (and **not** called). A user should not use them directly but instead always call *targets()* or *eachtarget()*.

Example 1: Custom File-Based Data Source

Let's say you want to write a custom data container that describes a directory on your hard-drive. Each sub-directory is expected to contain a set of large images that belong to a single class (the directory name). This kind of data container only loads the images itself if they are actually needed (so on *getobs()*). The targets, however, would technically be available in the memory at all times, since it is part of the metadata.

To "simulate" such a scenario, let us define a dummy type that represents the idea of such a data container for which each observation is expensive to access, but where the corresponding targets are available in some member variable.

```
using LearnBase

immutable DummyDirImageSource
    targets::Vector{String}
end

LearnBase.getobs(::DummyDirImageSource, i) = error("expensive computation triggered")

StatsBase.nobs(data::DummyDirImageSource) = length(data.targets)
```

Naturally, we would like to avoid calling *getobs()* if at all possible. While we can't avoid calling *getobs()* when we actually need the data, we could avoid it when we only require the targets (for example for data partitioning or resampling). This is because in this example, the targets are part of the metadata that is always loaded. We can make use of this fact by implementing a custom method for `LearnBase.gettargets`.

```
LearnBase.gettargets(data::DummyDirImageSource, i) = data.targets[i]
```

By defining this method, the function *targets()* can now query the targets efficiently by looking them up in the member variable. In other words it allows to provide the targets of some observation(s) without ever calling *getobs()*. This even works seamlessly in combination with *datasubset()*.

```
julia> source = DummyDirImageSource(["malign", "benign", "benign", "malign", "benign
↪"])
DummyDirImageSource(String["malign","benign","benign","malign","benign"])

julia> targets(source)
5-element Array{String,1}:
 "malign"
 "benign"
 "benign"
 "malign"
 "benign"

julia> targets(datasubset(source, 3:4))
2-element Array{String,1}:
 "benign"
 "malign"
```

Note however, that calling *targets()* with a target-extraction-function will still trigger *getobs()*. This is expected behaviour, since the extraction function is intended to "extract" the target from each actual observation (i.e. the result of *getobs()*).

```
julia> targets(x->x, source)
ERROR: expensive computation triggered
```

Example 2: Symbol Support for DataFrames.jl

`DataFrame` are a kind of data container for which the targets are as much part of the data as the features are (in contrast to Example 1). Furthermore, each observation is itself also a `DataFrame`. Before we start, let us implement the required *Data Container* interface.

```
using DataFrames, LearnBase

LearnBase.getobs(df::DataFrame, idx) = df[idx,:]

StatsBase.nobs(df::DataFrame) = nrow(df)
```

Here we are fine with *getobs()* being called, since we need to access the actual `DataFrame` anyway. However, we still need to specify which column actually describes the features. This can be done generically by specifying a target-extraction-function.

```
julia> df = DataFrame(x1 = rand(5), x2 = rand(5), y = [:a,:a,:b,:a,:b])
5×3 DataFrames.DataFrame
| Row | x1       | x2        | y |
|-----|----------|-----------|---|
| 1   | 0.226582 | 0.0997825 | a |
| 2   | 0.504629 | 0.0443222 | a |
| 3   | 0.933372 | 0.722906  | b |
| 4   | 0.522172 | 0.812814  | a |
| 5   | 0.505208 | 0.245457  | b |

julia> targets(row->row.y, df)
5-element Array{Symbol,1}:
```

(continues on next page)

(continued from previous page)

```
 :a
 :a
 :b
 :a
 :b
```

Alternatively, we could also implement a convenience syntax by overloading `LearnBase.gettarget`.

```
LearnBase.gettarget(col::Symbol, df::DataFrameRow) = df[col]
```

This now allows us to call `targets(:y, dataframe)`. While not strictly necessary in this case, it can be quite useful for special types of observations, such as `ImageMeta`.

```
julia> targets(:y, df)
5-element Array{Symbol,1}:
 :a
 :a
 :b
 :a
 :b
```

We could even implement the default assumption, that the last column denotes the targets unless otherwise specified.

```
LearnBase.gettarget(df::DataFrameRow) = df[end]
```

Note that this might not be a good idea for a `DataFrame` in particular. The purpose of this exercise is solely to show what is possible.

```
julia> targets(df)
5-element Array{Symbol,1}:
 :a
 :a
 :b
 :a
 :b
```

3.3.4 Stratified Sampling

In a supervised learning scenario, in which we are usually confronted with a labeled data set, we have to be considerate of the distribution of the targets. That is, how likely is it to observe some given target-value (e.g. an observation labeled "malignant") without conditioning on the features.

It is important to be aware of the class distribution for a couple of different reason, one of which is data partitioning. Usually a good idea is to make sure that we actively try to preserve the class distribution for every data subset. This will help to make sure that the data subsets are similar in structure and more likely to be representative of the full data set.

Consider the following target vector `y`. Note how there are only two elements of value `:b`. If we just use random assignment to partition the data set, then chances are that in some cases one subset does not contain any element of value `b`. This kind of effect becomes less frequent as the size of the data set increases.

```
julia> y = [:a, :a, :a, :a, :b, :b];

julia> splitobs(shuffleobs(y), 0.5)
(Symbol[:b,:b,:a],Symbol[:a,:a,:a])
```

To perform partitioning using stratified sampling without replacement, this package provides the function *stratifiedobs()*.

stratifiedobs(*[fun]*, *data*[, *p*][, *shuffle*][, *obsdim*])

Partition the data into multiple disjoint subsets proportional to the value(s) of *p*. The observations are assignmed to a data subset using stratified sampling without replacement. These subsets are then returned as a Tuple of subsets, where the first element contains the fraction of observations of data that is specified by the first float in *p*.

Parameters

- **fun** – Optional. A callable object (usually a function) that should be applied to each observation individually in order to extract or compute the label for that observation.
- **data** – The object representing a labeled data container.
- **p** – Optional. The fraction of observations that should be in the first subset. Must be in the interval (0,1). Can be specified as positional or keyword argument, as either a single float or a tuple of floats. Defaults to 0.7 (i.e. 70% of the observations in the first subset).
- **shuffle** (*bool*) – Optional. Determines if the resulting data subsets will be shuffled after their creation. If `false`, then all the observations will be clustered together accoring to their class label in each subset. Note that this has nothing to do with random assignment to some data subset, it only inluences the order of observation in each subset individually. Defaults to `true`.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Let's consider the following toy data vector `y`, which contains a total of 9 observations. Notice how each value of `y` is either `:a` or `:b`, which means we have a binary classification problem. We can also see that the data set has twice as many observations for `:a` as it has for `:b`.

```
y = [:a, :a, :a, :a, :a, :a, :b, :b, :b]
```

We have seen before that using *splitobs()* can result in a partition where one subset contains all the `:b`, while the other one contains only `:a`. In contrast to this, *stratifiedobs()* will try to make sure that both subsets are both appropriately distributed. More concretely, if `p` is a `Float64`, then the return-value of *stratifiedobs()* will be a tuple with two elements (i.e. subsets), in which the first element contains the fraction of observations specified by `p` and the second element contains the rest. In the following code the first subset `train` will contain around 70% of the observations and the second subset `test` the rest.

```
julia> train, test = stratifiedobs(y, p = 0.7)
(Symbol[:b,:a,:a,:a,:a,:b],Symbol[:a,:a,:b])

julia> test
3-element SubArray{Symbol,1,Array{Symbol,1},Tuple{Array{Int64,1}},false}:
 :a
 :a
 :b
```

Notice how both subsets contain twice as much `:a` as `:b`, just like `y` does. Furthermore, it is worth pointing out how `test` (and `train` for that matter) is a `SubArray`. As such, it is just a view into the corresponding original variable `y`. The motivation for this behaviour is to avoid data movement until *getobs()* is called.

Recall how I said explicitly that *stratifiedobs()* will "try to make sure" that the distribution is preserved. This is because it is not always possible to preserve the class distribution. If that is the case, the function will try to have each subset contain at least one of each class, even if that does not reflect the distribution appropriately.

```
julia> train, test = stratifiedobs(y, p = 0.8)
(Symbol[:b,:a,:b,:a,:a,:a],Symbol[:a,:b])
```

It is also possible to specify multiple fractions for `p`. If `p` is a `Tuple` of `Float64`, then additional subsets will be created. This can be useful to create an additional validation set.

```
julia> train, val, test = stratifiedobs(y, p = (0.3, 0.3))
(Symbol[:b,:a,:a],Symbol[:b,:a,:a],Symbol[:a,:b,:a])
```

It is also possible to call *`stratifiedobs()`* with multiple data arguments as tuple, which all must have the same number of total observations. Note that if data is a tuple, then it will be assumed that the last element of the tuple contains the targets.

```
train, test = stratifiedobs((X, y), p = 0.7)
(X_train,y_train), (X_test,y_test) = stratifiedobs((X, y), p = 0.7)
```

If the type of the data is not sufficient information to be able to extract the targets, one can specify a target-extraction-function `fun`, that is to be applied to each individual observation. The behaviour when specifying or omitting `fun` is equivalent to its behaviour for *`eachtarget()`*, because that is the function that is used internally by *`stratifiedobs()`*.

A good example for a type that requires the parameter `fun` is a `DataTable`. Consider the following toy data container where we know that the column `:y` contains the targets (see *Example: DataTables.jl* to make the following code work).

```
julia> dt = DataTable(x1 = rand(6), x2 = rand(6), y = [:a,:b,:b,:b,:b,:a])
6×3 DataTables.DataTable
| Row | x1        | x2          | y  |
|-----|-----------|-------------|----|
| 1   | 0.226582  | 0.0443222   | :a |
| 2   | 0.504629  | 0.722906    | :b |
| 3   | 0.933372  | 0.812814    | :b |
| 4   | 0.522172  | 0.245457    | :b |
| 5   | 0.505208  | 0.11202     | :b |
| 6   | 0.0997825 | 0.000341996 | :a |

julia> train, test = stratifiedobs(row->row[1,:y], dt)
(3×3 DataTables.SubDataTable{Array{Int64,1}}
| Row | x1        | x2          | y  |
|-----|-----------|-------------|----|
| 1   | 0.505208  | 0.11202     | :b |
| 2   | 0.522172  | 0.245457    | :b |
| 3   | 0.0997825 | 0.000341996 | :a |,
3×3 DataTables.SubDataTable{Array{Int64,1}}
| Row | x1        | x2          | y  |
|-----|-----------|-------------|----|
| 1   | 0.226582  | 0.0443222   | :a |
| 2   | 0.933372  | 0.812814    | :b |
| 3   | 0.504629  | 0.722906    | :b |)
```

The optional parameter `obsdim` can be used to specify which dimension denotes the observations, if that concept makes sense for the type of data. For instance, consider the following toy data set where the targets `Y` are now a one-of-k encoded matrix. In such a case we would like the be able to re-sample without first having to convert `Y` to a different class-encoding.

```
# 2 imbalanced classes in one-of-k encoding
julia> Y = [1 0; 1 0; 1 0; 1 0; 0 1; 0 1]
```

(continues on next page)

(continued from previous page)

```
6×2 Array{Int64,2}:
 1  0
 1  0
 1  0
 1  0
 0  1
 0  1
```

Here we could use the function `argmax` to discretize the individual target vectors on the fly. Remember that the target-extraction-function is applied on each individual observation in `Y`. Since `Y` is a matrix, each observation is a vector slice. Here we use `obsdim` to specify that each row is an observation.

```
julia> train, test = stratifiedobs(argmax, X, p = 0.5, obsdim = 1)
([1 0; 1 0; 0 1], [0 1; 1 0; 1 0])
```

3.3.5 Under- and Over-Sampling

It is not uncommon in a classification setting, that we find ourselves working with an *imbalanced data set*. We call a labeled data set **imbalanced**, if it contains more observations of some class(es) than for the other(s). Training on such a data set can pose a significant challenge for many commonly used algorithms; especially if the difference in the class frequency is large.

There are different conceptual approaches for dealing with imbalanced data. A quite simple but popular strategy that works for *data containers*, is to either under- or over-sample it according to the class distribution. What that means is that the data container is re-sampled in such a way, that the class distribution in the resulting data container is approximately uniform.

This package provides two functions to re-sample an imbalanced data container; the first of which is called *`undersample()`*. When under-sampling a data container, it will be down-sampled in such a way, that each class has about as many observations in the resulting subset, as the least represented class has in the original data container.

`undersample`(*[fun]*, *data*[, *shuffle*][, *obsdim*])

Generate a class-balanced version of *data* by down-sampling its observations in such a way that the resulting number of observations will be the same number for every class. This way, all classes will have as many observations in the resulting data subset as the smallest class has in the given (i.e. original) *data* container.

Parameters

- **`fun`** – Optional. A callable object (usually a function) that should be applied to each observation individually in order to extract or compute the label for that observation.
- **`data`** – The object representing a labeled data container.
- **`shuffle`** (*`bool`*) – Optional. Determines if the resulting data will be shuffled after its creation. If `false`, then all the observations will be in their original order. Defaults to `false`.
- **`obsdim`** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns A down-sampled but class-balanced version of *data* in the form of a lazy data subset. No data is copied until *`getobs()`* is called.

Let's consider the following toy data set, which consists of a feature matrix `X` and a corresponding target vector `y`. Both variables are data containers with 6 observations.

```
julia> X = rand(2, 6)
2×6 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996

julia> y = ["a", "b", "b", "b", "b", "a"]
6-element Array{String,1}:
 "a"
 "b"
 "b"
 "b"
 "b"
 "a"
```

As we can see, the target of each observation is either `"a"` or `"b"`, which means we have a binary classification problem. We can also see that the data set has twice as many observations for `"b"` as it has for `"a"`. Thus we consider it imbalanced.

We can down-sample our toy data set by passing it to *`undersample()`*. In order to tell the function that these two data containers should be treated as a one data set, we have to group them together using a `Tuple`. This will cause *`undersample()`* to return a `Tuple` of equal length and ordering.

```
julia> X_bal, y_bal = undersample((X, y));

julia> X_bal
2×4 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.226582  0.933372  0.0443222  0.11202
 0.504629  0.522172  0.722906   0.000341996

julia> y_bal
4-element SubArray{String,1,Array{String,1},Tuple{Array{Int64,1}},false}:
 "a"
 "b"
 "b"
 "a"
```

Note two things in the code above.

1. Both, `X_bal` and `y_bal`, are of type `SubArray`. As such, they are just views into the corresponding original variable `X` or `y`. The motivation for this behaviour is to avoid data movement until *`getobs()`* is called.
2. The order of the observations in the resulting data subsets is the same as in the original data containers. This is no accident. If that behaviour is undesired, you can pass `shuffle = true` to the function.

If the type of the data is not sufficient information to be able to extract the targets, one can specify a target-extraction-function `fun`, that is to be applied to each individual observation. The behaviour when specifying or omitting `fun` is equivalent to its behaviour for *`eachtarget()`*, because that is the function that is used internally by *`undersample()`*.

A good example for a type that requires the parameter `fun` is a `DataTable`. Consider the following toy data container where we know that the column `:y` contains the targets (see *Example: DataTables.jl* to make the following code work).

```
julia> dt = DataTable(x1 = rand(6), x2 = rand(6), y = [:a,:b,:b,:b,:a])
6×3 DataTables.DataTable
| Row | x1        | x2          | y   |
```

(continues on next page)

(continued from previous page)

```
├─────┼──────────┼──────────┼─────┤
│ 1   │ 0.226582 │ 0.0443222 │ :a  │
│ 2   │ 0.504629 │ 0.722906  │ :b  │
│ 3   │ 0.933372 │ 0.812814  │ :b  │
│ 4   │ 0.522172 │ 0.245457  │ :b  │
│ 5   │ 0.505208 │ 0.11202   │ :b  │
│ 6   │ 0.0997825 │ 0.000341996 │ :a  │

julia> undersample(row->row[1,:y], dt)
4×3 DataTables.SubDataTable{Array{Int64,1}}
│ Row │ x1       │ x2          │ y   │
├─────┼──────────┼──────────┼─────┤
│ 1   │ 0.226582 │ 0.0443222 │ :a  │
│ 2   │ 0.504629 │ 0.722906  │ :b  │
│ 3   │ 0.522172 │ 0.245457  │ :b  │
│ 4   │ 0.0997825 │ 0.000341996 │ :a  │
```

Of course, under-sampling the larger classes has the consequence of decreasing the total size of the training set. After all, this approach effectively discards perfectly usable training examples for the sake of having a balanced data set. Alternatively, one can also achieve a balanced class distribution by over-sampling the smaller classes instead. To that end, we provide the function `oversample()`. While this function effectively increases the apparent size of the given data container, it does use the same exact observations multiple times.

oversample([*fun*], *data*[, *fraction*][, *shuffle*][, *obsdim*])

Generate a re-balanced version of *data* by repeatedly sampling existing observations in such a way that every class will have at least *fraction* times the number observations of the largest class. This way, all classes will have a minimum number of observations in the resulting data set relative to what largest class has in the given (i.e. original) *data*.

Parameters

- **fun** – Optional. A callable object (usually a function) that should be applied to each observation individually in order to extract or compute the label for that observation.
- **data** – The object representing a labeled data container.
- **fraction** (*Real*) – Optional. Minimum number of observations (as a fraction relative to the largest class) that every class should have. Defaults to `1`, which implies completely balanced.
- **shuffle** (*bool*) – Optional. Determines if the resulting data will be shuffled after its creation. If `false`, then all the repeated samples will be together at the end, sorted by class. Defaults to `true`.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns An up-sampled version of *data* in the form of a lazy data subset. No data is copied until `getobs()` is called.

Let us again consider the toy data set from before, which consists of a feature matrix `X` and a corresponding target vector `y`. Both variables are data containers with 6 observations.

```
julia> X = rand(2, 6)
2×6 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202
```

(continues on next page)

(continued from previous page)

```
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996

julia> y = ["a", "b", "b", "b", "b", "a"]
6-element Array{String,1}:
 "a"
 "b"
 "b"
 "b"
 "b"
 "a"
```

We previously "balanced" this data set by down-sampling it. To show you an alternative, we can also up-sample it by repeating observations for the under-represented class `a`. You may notice that this time the order of the observations will *not* be preserved; it will even be shuffled. If that behaviour is undesired, you can specify `shuffle = false`.

```
julia> X_bal, y_bal = oversample((X, y));

julia> X_bal
2×8 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.11202      0.812814  0.226582  0.11202      0.933372 0.0443222  0.226582  0.505208
 0.000341996  0.245457  0.504629  0.000341996  0.522172 0.722906   0.504629  0.0997825

julia> y_bal
8-element SubArray{String,1,Array{String,1},Tuple{Array{Int64,1}},false}:
 "a"
 "b"
 "a"
 "a"
 "b"
 "b"
 "a"
 "b"
```

Similar to *`undersample()`* it is also possible to specify a target-extraction-function `fun`, that is to be applied to each observation individually. Consider the following toy `DataTable`, which we also used as an example data container to demonstrate *`undersample()`*. For this particular data table we know that the column `:y` contains the targets (see *Example: DataTables.jl* to make the following code work).

```
julia> dt = DataTable(x1 = rand(6), x2 = rand(6), y = [:a,:b,:b,:b,:a])
6×3 DataTables.DataTable
| Row | x1        | x2          | y  |
|-----|-----------|-------------|----|
| 1   | 0.226582  | 0.0443222   | :a |
| 2   | 0.504629  | 0.722906    | :b |
| 3   | 0.933372  | 0.812814    | :b |
| 4   | 0.522172  | 0.245457    | :b |
| 5   | 0.505208  | 0.11202     | :b |
| 6   | 0.0997825 | 0.000341996 | :a |

julia> oversample(row->row[1,:y], dt)
8×3 DataTables.SubDataTable{Array{Int64,1}}
| Row | x1        | x2          | y  |
|-----|-----------|-------------|----|
| 1   | 0.226582  | 0.0443222   | :a |
| 2   | 0.505208  | 0.11202     | :b |
| 3   | 0.0997825 | 0.000341996 | :a |
```

(continues on next page)

(continued from previous page)

```
| 4 | 0.504629 | 0.722906    | :b |
| 5 | 0.933372 | 0.812814    | :b |
| 6 | 0.226582 | 0.0443222   | :a |
| 7 | 0.0997825| 0.000341996 | :a |
| 8 | 0.522172 | 0.245457    | :b |
```

While primarily intended for data container types, such as `DataTable`, it is also useful for discretizing continuous regression targets. Let's say you have a regression problem, where you know that you have a small but important cluster of observations with a particularly low target value. Given that this cluster is under-represented, it could very well cause a model to neglect those observations in order to improve its performance on the rest of the data.

```
julia> X = rand(2, 6)
2×6 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996

julia> y = [0.1, 0.95, 0.8, 0.9, 1.1, 0.11];
```

As can be seen in the code above, most observations have a target value around `1`, while just a small group of observations have a target value around `0.1`. In such a situation, you could use the parameter `fun` to categorize the targets in such a way, that will cause the under-represented "category" to be up-sampled (or down-sampled) accordingly.

```
julia> X_bal, y_bal = oversample(yi -> yi > 0.2, (X, y));

julia> y_bal
8-element SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}:
 0.11
 1.1
 0.1
 0.11
 0.95
 0.9
 0.1
 0.8
```

While the above example is a bit arbitrary, it highlights the possibility that the functions *`undersample()`* and *`oversample()`* can also be used to re-sample data container with continuous targets.

A more common scenario would be when working with targets in the form of a `Matrix`. For instance, consider the following toy data set where the targets `Y` are now a one-of-k encoded matrix. In such a case we would like the be able to re-sample without first having to convert `Y` to a different class-encoding.

```
julia> X = rand(2, 6)
2×6 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814  0.11202
 0.504629  0.522172  0.0997825  0.722906   0.245457  0.000341996

julia> Y = [1. 0. 0. 0. 0. 1.; 0. 1. 1. 1. 1. 0.]
2×6 Array{Float64,2}:
 1.0  0.0  0.0  0.0  0.0  1.0
 0.0  1.0  1.0  1.0  1.0  0.0
```

Here we could use the function `argmax` to discretize the individual target vectors on the fly. Remember that the target-extraction-function is applied on each individual observation in `Y`. Since `Y` is a matrix, each observation is a vector slice.

```
julia> X_bal, Y_bal = oversample(argmax, (X, Y));

julia> X_bal
2×8 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 0.226582  0.11202      0.11202      0.505208   0.226582  0.0443222  0.933372  0.
↪812814
 0.504629  0.000341996  0.000341996  0.0997825  0.504629  0.722906   0.522172  0.
↪245457

julia> Y_bal
2×8 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
 1.0  1.0  1.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0  1.0  1.0  1.0
```

Now that we have covered all the basics, we can start to discuss some of the more advanced topics. A particularly important aspect of modern Machine Learning is what is known as *model selection*. Most of the time, this boils down to choosing appropriate hyper-parameters for the model one is working with. To avoid subtle problems in this selection process, and to reduce variance of the performance estimates, it is quite common to employ some kind of **repartitioning strategy** on the training data. Of course, the partitioning itself is just one part of such a model selection process, since we still have to somehow compute and compare the performance. However, it is an important step that is needed to make the most of the available data. So important in fact, that we will spend a whole section on it.

3.4 Repartitioning Strategies

Most non-trivial machine learning experiments require some form of model tweaking *prior* to training. A particularly common scenario is when the model (or algorithm) has hyper parameters that need to be specified manually. The process of searching for suitable hyper parameters is a sub-task of what we call *model selection*.

If model selection is part of the experiment, then it is quite likely that a simple train/test split will not be effective enough to achieve results that are representative for new, unseen data. The reason for this is subtle, but very important. If the hyper parameters are chosen based on how well the corresponding model performs on the test set, then information about the test set is actively fed back into the model. This is because the test set is used several times *and* decisions are made based on what was observed. In other words: the test set participates in an aspect of the training process, namely the model selection. Consequently, the results on the test set become less representative for the expected results on new, unseen data. To avoid causing this kind of manual overfitting, one should instead somehow make use of the training set for such a model selection process, while leaving the test set out of it completely. Luckily, this can be done quite effectively using by a repartitioning strategy, such as a k-folds, to perform cross-validation.

We will start by discussing the terminology that is used throughout this document. More importantly, we will define how the various terms are interpreted in the context of this package. The rest of this document will then focus on how these concepts are implemented and exposed to the user. There we will start by introducing some low-level helper methods for computing the required subset-assignment indices. We will then use those "assignments" to motivate a type called `FoldsView`, which can be configured to represent almost any kind of repartitioning strategy for a given data container. After discussing those basics, we will introduce the high-level methods that serve as a convenience layer around `FoldsView` and the low-level functionality.

3.4.1 Terms and Definitions

Before we dive into the provided functionality, let us quickly discuss some terminology. A few of the involved terms are often used quite casually in conversations, and thus easy to mix up. In general that doesn't cause much confusion, but since parts of this document are concerned with low-level functionality, we deem it important that we share the same wording.

- When we have multiple disjoint subsets of the same data container (or tuple of data containers), we call the grouping of those subsets a **partition**. That is, a partition is a particular outcome of assigning the observations from some data container to multiple disjoined subsets. In contrast to the formal definition in mathematics, we do allow the same observation to occur multiple times in the *same* subset.

  For instance the function `splitobs()` creates a single partition in the form of a tuple. More concretely, the following code snippet creates a partition with two subsets from a given toy data-vector that has 5 observations.

```
julia> partition = splitobs([1,2,3,4,5], at = 0.6)
([1,2,3],[4,5])
```

- In the context of this package, a **repartitioning strategy** describes a particular "system" for reassigning the observations of a data container (or tuple of data containers) to a training subset and a validation subset *multiple times*. So in contrast to a simple train/validation split, the data isn't just partitioned once, but in multiple different configurations. In other words, the result of a repartitioning strategy are multiple different partitions of the same data. We use the term "repartitioning strategy" instead of "resampling strategy" to emphasize that the subsets of each partition are disjoint.

  An example for performing a really simply repartitioning strategy would be to create a sequences of random train/validation partitions of some given data. The following code snippet computes 3 partitions (which are also often referred to as *folds*) for such a strategy on a random toy data-vector `y` that has 5 observations in it.

```
julia> y = rand(5);

julia> folds = [splitobs(shuffleobs(y), at = 0.6) for i in 1:3]
3-element Array{Tuple{SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},
↪false},SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}},1}:
 ([0.933372,0.522172,0.505208],[0.504629,0.226582])
 ([0.226582,0.504629,0.505208],[0.522172,0.933372])
 ([0.505208,0.504629,0.933372],[0.226582,0.522172])
```

- The result of a repartitioning strategy can be described through a sequences of *subset assignment indices*, or short **assignments**. An assignment (singular) describes a partition that is valid for any data container of size N by using indices from the set $\{1, 2, ..., N\}$. For instance, if a single partition should consist of two subsets, then the corresponding assignment (singular), is made up of two vectors of indices, each vector describing the content of one subset. Because of this, it is also fair to think about the result of a repartitioning strategy as two sequences, one for the *training assignments* and a corresponding sequence for the *validation assignments*.

  To give a concrete example of such assignment sequences, consider the result of calling `kfolds(6, 3)` (see code below). It will compute the training assignments `train_idx` and the corresponding validation assignments `val_idx` for a 3-fold repartitioning strategy that is applicable to any data container that has 6 observations in it.

```
julia> train_idx, val_idx = kfolds(6, 3)
([[3,4,5,6],[1,2,5,6],[1,2,3,4]], [[1,2],[3,4],[5,6]])

julia> train_idx # sequence of training assignments
3-element Array{Array{Int64,1},1}:
 [3,4,5,6]
 [1,2,5,6]
 [1,2,3,4]

julia> val_idx # sequence of validation assignments
3-element Array{Array{Int64,1},1}:
 [1,2]
 [3,4]
 [5,6]
```

- The result of applying a sequence of assignments to some data container (or tuple of data containers) is a sequence of **folds**. In the context of this package the term "fold" is almost interchangeable with "partition". In contrast to a partition, however, the term "fold" implies that there exist more than one.

  For instance, let us consider manually applying the assignments (which we have computed above) to some random toy data-vector `y` of appropriate length 6.

```
julia> y = rand(6)
6-element Array{Float64,1}:
 0.226582
 0.504629
 0.933372
 0.522172
 0.505208
 0.0997825

julia> folds = map((t,v)->(view(y,t),view(y,v)), train_idx, val_idx)
3-element Array{Tuple{SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},
↪false},SubArray{Float64,1,Array{Float64,1},Tuple{UnitRange{Int64}},true}},1}:
 ([0.933372,0.522172,0.505208,0.0997825],[0.226582,0.504629])
 ([0.226582,0.504629,0.505208,0.0997825],[0.933372,0.522172])
 ([0.226582,0.504629,0.933372,0.522172],[0.505208,0.0997825])
```

Naturally, the above code snippets just serve as examples to motivate the problem. This package implements a number of functions that provide the necessary functionality in a more intuitive and convenient manner.

3.4.2 Computing K-Folds Indices

A particularly popular validation scheme for model selection is *k-fold cross-validation*; the first step of which is dividing the data set into k roughly equal-sized parts. Each model is fit k times, while each time a different part is left out during training. The left out part instead serves as a validation set, which is used to compute the metric of interest. The validation results of the k trained model-instances are then averaged over all k folds and reported as the performance for the particular set of hyper parameters.

Before we go into details about the partitioning or, later, the validation aspects, let us first consider how to compute the underlying representation. In particular how to compute the **assignments** that can then be used to create the folds. For that purpose we provide a helper method for the function `kfolds()`.

kfolds(*n*[, *k* = *5*]) → Tuple

Compute the train/validation assignments for *k* partitions of *n* observations, and return them in the form of two vectors. The first vector contains the sequence of training assignments (i.e. the indices for the training subsets), and the second vector the sequence of validation assignments (i.e. the indices for the validation subsets).

Each observation is assigned to a validation subset once (and only once). Thus, a union over all validation assignments reproduces the full range `1:n`. Note that there is no random placement of observations into subsets, which means that adjacent observations are likely part of the same subset.

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

Parameters

- **n** (*Integer*) – Total number of observations to compute the folds for.
- **k** (*Integer*) – Optional. The number of folds to compute. A general rule of thumb is to use either `k = 5` or `k = 10`. Must be within the range `2:n`. Defaults to `k = 5`.

Returns A `Tuple` of two `Vector`. Both vectors are of length k, where each element is also a vector. The first vector represents the sequence of training assignments, and the second vector the sequence of validation assignments.

Invoking *`kfolds()`* with an integer as first parameter - as outlined above - will compute the assignments for a k-folds repartitioning strategy. For instance, the following code will compute the sequences of training- and validation assignments for 10 observations and 4 folds.

```
julia> train_idx, val_idx = kfolds(10, 4); # 10 observations, 4 folds

julia> train_idx
4-element Array{Array{Int64,1},1}:
 [4,5,6,7,8,9,10]
 [1,2,3,7,8,9,10]
 [1,2,3,4,5,6,9,10]
 [1,2,3,4,5,6,7,8]

julia> val_idx
4-element Array{UnitRange{Int64},1}:
 1:3
 4:6
 7:8
 9:10
```

As we can see, there is no actual data set involved yet. We just computed assignments that are applicable to *any* data set that has exactly 10 observations in it. The important thing to note here is that while the indices in `train_idx` overlap, the indices in `val_idx` do not, and further, all 10 observation-indices are part of one (and only one) element of `val_idx`.

3.4.3 Computing Leave-Out Indices

A different way to think about a k-folds repartitioning strategy is in terms of the size of each validation subset. Instead of specifying the number of folds directly, we specify how many observations we would like to be in each validation subset. While the resulting assignments are equivalent to the result of some particular k-folds scheme, it is sometimes referred to as *leave-p-out partitioning*. A particularly common version of which is leave-one-out, where we set the validation subset size to 1 observation.

`leaveout` (n[, *size* = *1*]) $\rightarrow$ Tuple

Compute the train/validation assignments for `k  n/size` repartitions of n observations, and return them in the form of two vectors. The first vector contains the sequence of training assignments (i.e. the indices for the training subsets), and the second vector the sequence of validation assignments (i.e. the indices for the validation subsets).

Each observation is assigned to the validation subset once (and only once). Furthermore, each validation subset will have either *size* or *size* + 1 observations assigned to it.

Note that there is no random placement of observations into subsets, which means that adjacent observations are likely part of the same subset.

Parameters

- **`n`** (*`Integer`*) – Total number of observations to compute the folds for.
- **`size`** (*`Integer`*) – Optional. The desired number of observations in each validation subset. Defaults to `size = 1`.

Returns A `Tuple` of two `Vector`. Both vectors are of queal length, where each element is also a vector. The first vector represents the sequence of training assignments, and the second vector the sequence of validation assignments.

Invoking *leaveout()* with an integer as first parameter will compute the sequence of assignments for a k-folds repartitioning strategy. For example, the following code will assign the indices of 10 observations to as many partitions as it takes such that every validation subset contains approximately 2 observations.

```
julia> train_idx, val_idx = leaveout(10, 2);

julia> train_idx
5-element Array{Array{Int64,1},1}:
 [3,4,5,6,7,8,9,10]
 [1,2,5,6,7,8,9,10]
 [1,2,3,4,7,8,9,10]
 [1,2,3,4,5,6,9,10]
 [1,2,3,4,5,6,7,8]

julia> val_idx
5-element Array{UnitRange{Int64},1}:
 1:2
 3:4
 5:6
 7:8
 9:10
```

Just like before, there is no actual data set involved here. We simply computed assignments that are applicable to *any* data set that has exactly 10 observations in it. Note that for the above example the result is equivalent to calling `kfolds(10, 5)`.

3.4.4 The FoldsView Type

So far we focused on just computing the sequence of assignments for various repartition strategies, without any regard to an actual data set. Instead, we just specified the total number of observations. Naturally that is only one part of the puzzle. What we really care about after all, is the repartitioning of an actual data set. To that end we provide a type called *FoldsView*, which associates a *data container* with a given sequence of assignments.

FoldsView <: DataView <: AbstractVector

A vector-like representation of applying a repartitioning strategy to a specific data container. It is used to associate a data container with appropriate assignments, and will act as a lazy view, that allows the data to be treated as a sequence of folds. As such it does not copy any data.

FoldsView is a subtype of `AbstractArray` and as such supports the appropriate interface. Each individual element (accessible via `getindex`) is a tuple of two subsets of the data container; a training- and a validation subset.

data

The object describing the data source of interest. Can be of any type as long as it implements the *Data Container* interface.

train_indices

Vector of integer vectors containing the sequences of assignments for the *training* subsets. This means that each element of this vector is a vector of observation-indices valid for `data`. The length of this vector must match `val_indices`, and denotes the number of folds.

val_indices

Vector of integer vectors containing the sequences of assignments for the *validation* subsets. This means that each element of this vector is a vector of observation-indices valid for `data`. The length of this vector must match `train_indices`, and denotes the number of folds.

obsdim

If defined for the type of data, `obsdim` can be used to specify which dimension of `data` denotes the

observations. Should be `ObsDim.Undefined` if not applicable.

The purpose of *`FoldsView`* is to apply a precomputed sequence of assignments to some data container in a convenient manner. By itself, *`FoldsView`* is agnostic to any particular repartitioning- or resampling strategy. Instead, the assignments, `train_indices` and `val_indices`, need to be precomputed by such a strategy and then passed to *`FoldsView()`* with a concrete data container. The resulting object can then be queried for its individual folds using `getindex`, or alternatively, simply iterated over.

`FoldsView`(*data*, *train_indices*, *val_indices*[, *obsdim*]) → FoldsView

Create a *`FoldsView`* for the given *data* container. The number of folds is denoted by the length of *train_indices*, which must be equal to the length of *val_indices*.

Note that the number of observations in *data* is expected to match the number of observations that the given assignments were designed for.

Parameters

- **`data`** – The object representing a data container.
- **`train_indices`** (*`AbstractVector`*) – Vector of integer vectors. It denotes the sequence of training assignments (i.e. the indices of the training subsets).
- **`val_indices`** (*`AbstractVector`*) – Vector of integer vectors. It denotes the sequence of validation assignments (i.e. the indices of the validation subsets)
- **`obsdim`** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

To get a better feeling of how exactly *`FoldsView`* works, let us consider the following toy data container `X`. We will generate this data in such a way, that it is easy to see where each observation ends up after applying our partitioning strategy. To keep it simple let's say it has 10 observations with 2 features each.

```
julia> X = hcat(1.:10, 11.:20)' # generate toy data
2×10 Array{Float64,2}:
  1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0
 11.0  12.0  13.0  14.0  15.0  16.0  17.0  18.0  19.0  20.0
```

First we need to compute appropriate assignments that are applicable to our data container `X`. Ideally these assignments should follow some repartitioning strategy. For this example we will use *`kfolds()`*, which we introduced in a previous section. In particular we will compute the sequence of assignments for a 5-fold repartitioning.

```
julia> train_idx, val_idx = kfolds(10, 5);

julia> train_idx
5-element Array{Array{Int64,1},1}:
 [3,4,5,6,7,8,9,10]
 [1,2,5,6,7,8,9,10]
 [1,2,3,4,7,8,9,10]
 [1,2,3,4,5,6,9,10]
 [1,2,3,4,5,6,7,8]

julia> val_idx
5-element Array{UnitRange{Int64},1}:
 1:2
 3:4
 5:6
 7:8
 9:10
```

Now that we have appropriate assignments, we can use *FoldsView* to apply those to our data container X. Note that since *FoldsView* is designed to act as a "view", it won't actually copy any data from X, instead each "fold" will be a tuple of two SubArray into X.

```
julia> folds = FoldsView(X, train_idx, val_idx)
5-fold MLDataPattern.FoldsView of 10 observations:
  data: 2×10 Array{Float64,2}
  training: 8 observations/fold
  validation: 2 observations/fold
  obsdim: last

julia> train, val = folds[2]; # access second fold

julia> train
2×8 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
  1.0   2.0   5.0   6.0   7.0   8.0   9.0  10.0
 11.0  12.0  15.0  16.0  17.0  18.0  19.0  20.0

julia> val
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
  3.0   4.0
 13.0  14.0
```

As we can see in the above example, each element of folds is a tuple of two data subsets. More specifically, since our data container X is an Array, each tuple element is a SubArray into some part of X.

Similar to most other functions defined by this package, you can use the optional parameter obsdim to specify which dimension of data denotes the observations. If that concept does not make sense for the type of data it can simply be omitted. For example, the following code shows how we could work with a transposed version of X, where the first dimension enumerates the observations.

```
julia> folds = FoldsView(X', train_idx, val_idx, obsdim=1) # note the transpose
5-fold MLDataPattern.FoldsView of 10 observations:
  data: 10×2 Array{Float64,2}
  training: 8 observations/fold
  validation: 2 observations/fold
  obsdim: first

julia> train, val = folds[2]; # access second fold

julia> train
8×2 SubArray{Float64,2,Array{Float64,2},Tuple{Array{Int64,1},Colon},false}:
  1.0  11.0
  2.0  12.0
  5.0  15.0
  6.0  16.0
  7.0  17.0
  8.0  18.0
  9.0  19.0
 10.0  20.0

julia> val
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{UnitRange{Int64},Colon},false}:
 3.0  13.0
 4.0  14.0
```

It is also possible to link multiple different data containers together on an per-observation level. This way they can be repartitioned as one coherent unit. To do that, simply put all the relevant data container into a single Tuple, before

passing it to *FoldsView()*.

```
julia> y = collect(1.:10) # generate a toy target vector
10-element Array{Float64,1}:
  1.0
  2.0
  3.0

  8.0
  9.0
 10.0

julia> folds = FoldsView((X, y), train_idx, val_idx); # note the tuple

julia> (train_x, train_y), (val_x, val_y) = folds[2]; # access second fold

julia> val_x
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
  3.0   4.0
 13.0  14.0

julia> val_y
2-element SubArray{Float64,1,Array{Float64,1},Tuple{UnitRange{Int64}},true}:
 3.0
 4.0
```

It is worth pointing out, that the tuple elements (i.e. data container) need not be of the same type, nor of the same shape. You can observe this in the code above, where `X` is a `Matrix` while `y` is a `Vector`. Note, however, that all tuple elements must be data containers themselves. Furthermore, they all must contain the same exact number of observations.

While it is useful and convenient to be able to access some specific fold using the `getindex` syntax sugar (e.g. `folds[2]`), *FoldsView* can also be iterated over (just like any other `AbstractVector`). In fact, this is the main intention behind its design, because it allows you to conveniently loop over all folds.

```
julia> for (X_train, X_val) in FoldsView(X, train_idx, val_idx)
           println(X_val) # do something useful here instead
       end
[1.0 2.0; 11.0 12.0]
[3.0 4.0; 13.0 14.0]
[5.0 6.0; 15.0 16.0]
[7.0 8.0; 17.0 18.0]
[9.0 10.0; 19.0 20.0]
```

So far we showed how to use the low-level API to perform a repartitioning strategy on some data container. This was a two-step process. First we had to compute the assignments, and then we had to apply those assignment to some data container using the type *FoldsView*. In the rest of this document we will see how to do the same tasks in just one single step by using the high-level API.

3.4.5 K-Folds for Data Container

Let us revisit the idea behind a k-folds repartitioning strategy, which we introduced in the beginning of this document. Conceptually, k-folds divides the given data container into k roughly equal-sized parts. Each part will serve as validation set once, while the remaining parts are used for training at that stage. This results in k different partitions of the same data.

We have already seen how to compute the assignments of a k-folds scheme manually, and how to apply those to a

data container using the type *FoldsView*. We can do both those steps in just one single swoop by passing the data container to *kfolds()* directly.

kfolds (*data*[, *k* = 5][, *obsdim*]) → FoldsView

Repartition a *data* container *k* times using a *k*-folds strategy and return the sequence of folds as a lazy *FoldsView*. The resulting *FoldsView* can then be indexed into or iterated over. Either way, only data subsets are created. That means that no actual data is copied until *getobs()* is invoked.

In the case that the number of observations in *data* is not dividable by the specified *k*, the remaining observations will be evenly distributed among the parts. Note that there is no random assignment of observations to parts, which means that adjacent observations are likely part of the same validation subset.

Parameters

- **data** – The object representing a data container.
- **k** (*Integer*) – Optional. The number of folds to compute. Can be specified as positional argument or as keyword argument. A general rule of thumb is to use either `k = 5` or `k = 10`. Must be within the range `2:nobs(data)`. Defaults to `k = 5`.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

To visualize what exactly *kfolds()* does, let us consider the following toy data container `X`. We will generate this data in such a way, that makes it easy to see where each observation ends up after we apply the partitioning strategy to it. To keep it simple let's say it has 10 observations with 2 features each.

```
julia> X = hcat(1.:10, 11.:20)' # generate toy data
2×10 Array{Float64,2}:
  1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0
 11.0  12.0  13.0  14.0  15.0  16.0  17.0  18.0  19.0  20.0
```

Now that we have a data container to work with, we can pass it to the function *kfolds()* to create a view of the data that lets us treat it as a sequence of distinct partitions/folds.

```
julia> folds = kfolds(X, k = 5)
5-fold MLDataPattern.FoldsView of 10 observations:
  data: 2×10 Array{Float64,2}
  training: 8 observations/fold
  validation: 2 observations/fold
  obsdim: last
```

We can now query any individual fold using the typical indexing syntax. For instance, the following code snippet shows the training- and validation subset of the third fold.

```
julia> train, val = folds[3]; # access third fold

julia> train
2×8 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},false}:
  1.0   2.0   3.0   4.0   7.0   8.0   9.0  10.0
 11.0  12.0  13.0  14.0  17.0  18.0  19.0  20.0

julia> val
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
  5.0   6.0
 15.0  16.0
```

Note how `train` and `val` are of type `SubArray`, which means that their content isn't actually a copy from `X`. Instead, they serve as a view into the original data container `X`. For more information about on that topic take a look at *Data Subsets*.

If instead of a view you would like to have the folds as actual `Array`, you can use *`getobs()`* on the *`FoldsView`*. This will trigger *`getobs()`* on each subset and return the result as a `Vector`.

```
julia> getobs(folds) # output reformated for readability
5-element Array{Tuple{Array{Float64,2},Array{Float64,2}},1}:
 ([3.0 4.0 ... 9.0 10.0; 13.0 14.0 ... 19.0 20.0], [1.0  2.0; 11.0 12.0])
 ([1.0 2.0 ... 9.0 10.0; 11.0 12.0 ... 19.0 20.0], [3.0  4.0; 13.0 14.0])
 ([1.0 2.0 ... 9.0 10.0; 11.0 12.0 ... 19.0 20.0], [5.0  6.0; 15.0 16.0])
 ([1.0 2.0 ... 9.0 10.0; 11.0 12.0 ... 19.0 20.0], [7.0  8.0; 17.0 18.0])
 ([1.0 2.0 ... 7.0  8.0; 11.0 12.0 ... 17.0 18.0], [9.0 10.0; 19.0 20.0])

julia> fold_3 = getobs(folds, 3)
([1.0 11.0; 2.0 12.0; ... ; 9.0 19.0; 10.0 20.0], [5.0 15.0; 6.0 16.0])

julia> typeof(fold_3)
Tuple{Array{Float64,2},Array{Float64,2}}
```

You can use the optional parameter `obsdim` to specify which dimension of data denotes the observations. It can be specified as positional argument (which is type-stable) or as a more convenient keyword argument. For instance, the following code shows how we could work with a transposed version of `X`, where the first dimension enumerates the observations.

```
julia> folds = kfolds(X', 5, ObsDim.First()); # equivalent to below, but typesable

julia> folds = kfolds(X', k = 5, obsdim = 1) # note the transpose
5-fold MLDataPattern.FoldsView of 10 observations:
  data: 10×2 Array{Float64,2}
  training: 8 observations/fold
  validation: 2 observations/fold
  obsdim: first
```

It is also possible to call *`kfolds()`* with multiple data containers wrapped in a `Tuple`. Note, however, that all data containers must have the same total number of observations. Using a tuple this way will link those data containers together on a per-observation basis.

```
julia> y = collect(1.:10) # generate a toy target vector
10-element Array{Float64,1}:
  1.0
  2.0
  3.0

  8.0
  9.0
 10.0

julia> folds = kfolds((X, y), k = 5); # note the tuple

julia> (train_x, train_y), (val_x, val_y) = folds[2]; # access second fold
```

For more information and additional examples on what you can do with the result of *`kfolds()`*, take a look at *The FoldsView Type*.

3.4.6 Leave-Out for Data Container

Recall how we motivated leave-p-out as a different way to think about k-folds. Instead of specifying the number of folds k directly, we specify how many observations of the given data container should be in each validation subset.

Similar to `kfolds()`, we provide a method for `leaveout()` that allows it to be invoked with a data container. This method serves as a convenience layer that will return an appropriate `FoldsView` of the given data for you.

leaveout (*data*[, *size* = *1*][, *obsdim*]) → FoldsView

Repartition a *data* container using a k-fold strategy, where `k` is chosen in such a way, that each validation subset of the computed folds contains roughly *size* observations. The resulting sequence of folds is then returned as a lazy `FoldsView`, which can be index into or iterated over. Either way, only data subsets are created. That means no actual data is copied until `getobs()` is invoked.

Parameters

- **data** – The object representing a data container.
- **size** (*Integer*) – Optional. The desired number of observations in each validation subset. Can be specified as positional argument or as keyword argument. Defaults to `size = 1`, which results in a "leave-one-out" partitioning.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Let us again consider the toy feature-matrix `X` from before. We can pass it to the function `leaveout()` to create a view of the data. This "view" is represented as a `FoldsView` which lets us treat it is as a sequence of distinct partitions/folds.

```
julia> X = hcat(1.:10, 11.:20)' # generate toy data
2×10 Array{Float64,2}:
  1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0
 11.0  12.0  13.0  14.0  15.0  16.0  17.0  18.0  19.0  20.0

julia> folds = leaveout(X, size = 2)
5-fold MLDataPattern.FoldsView of 10 observations:
  data: 2×10 Array{Float64,2}
  training: 8 observations/fold
  validation: 2 observations/fold
  obsdim: last
```

We can now query any individual fold using the typical indexing syntax. Additionally, the function `leaveout()` supports all the signatures of `kfolds()`. For more information and additional examples on what you can do with the result of `leaveout()`, take a look at *The FoldsView Type*.

A different kind of partitioning-need arises from the fact that the interesting data sets are increasing in size as the scientific community continues to improve the state-of-the-art in Machine Learning. While "too much" data is a nice problem to have, bigger data sets also pose 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 for many algorithms to iterate over a given dataset in mini-batches, or even just one observation at a time.

The way this package approaches the topic of data iteration is complex enough that it deserves two parts. In the first part we will introduce a few special data iterators, which we call **data views**, that will allow us to perform such iteration-pattern conveniently for data containers. In fact, they are more than "just" data iterators; they are proper vectors. As such, they also serve as a tool to "view" a data container from a specific aspect: As a vector of observations, or a vector of batches. Thus these views know how many observations they contain, and how to query specific parts of the data.

Furthermore, we also provide a convenient way to partition a long sequence of data (e.g. some text or time-series) into a vector of smaller sequences. This is done using a sliding window approach that supports custom strides and even a self-labeling function. What that means, and how that works will be discussed in the next section.

3.5 Data Views

There exist a wide variety of machine learning algorithms, each of which unique in their own way. Yet, there are clear similarities concerning how these algorithms utilize the data set during model training. In fact, most algorithms belong to at least one of the following categories.

- Some algorithms use the whole training data in every iteration of the training procedure. If that is the case, then it is not necessary to partition or otherwise prepare the data any further.
- In contrast to this, many of the modern algorithms prefer to process the training data piece by piece using smaller chunks. These "chunks" are usually of some fixed size and are commonly referred to as *mini batches*.
- Yet another (but overlapping) group of algorithms processes the training data just one single observation at a time.

What we can learn from this is that regardless of the concrete algorithm and all its details, it is quite likely that at some point during a machine learning experiment, we need to iterate over the training data in a particular way. Most of the time we either iterate over the training data one observation at a time, or one mini-batch at at time.

This package provides multiple approaches to accomplish such a functionality. This "multiple" is necessary, because there are different forms of data sources, that have very different properties. Recall how we differentiated between data container and data iterators. In this document, however, we will solely focus on data sources that are considered *Data Container*.

3.5.1 As Vector of Observations

One could be inclined to believe, that in order to iterate over some data container one observation at a time, it should suffice to just iterate over the data container itself. While this may work as expected for some data container types, it will not work in general.

Consider the following two example data containers, the matrix `X` and the vector `y`. While both are different types of data container, we will make each one store exactly 5 observations.

```
julia> X = rand(2, 5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> y = rand(5)
5-element Array{Float64,1}:
 0.11202
 0.000341996
 0.380001
 0.505277
 0.841177
```

When we iterate over `y`, it turns out we also iterate over each observation. So in other words, for a `Vector` it would work to just iterate over the data container itself, if our goal is to process it one observation at a time.

```
julia> foreach(println, y)
0.11201971329803984
```

(continues on next page)

(continued from previous page)

```
0.0003419958128361156
0.3800005018641015
0.5052774551949404
0.8411766784932724
```

On the other hand, when we iterate over `X`, we iterate over all individual array elements, and *not* over what we consider to be the observations. In general that is the behaviour we want for a `Array`, but it is not in line with our domain interpretation of a data container.

```
julia> foreach(println, X)
0.22658190197881312
0.5046291972412908
0.9333724636943255
0.5221721267193593
0.5052080505550971
0.09978246027514359
0.04432218813798294
0.7229058081423172
0.8128138585478044
0.24545709827626805
```

This means, that we need a more general approach for iterating over a data container one observation at a time. Recall how data containers have the nice property of knowing how many observations they contain, and how to access each individual observation. Because of this we need not even limit ourselves to just iteration here, instead we could just create a new "view". For that purpose we provide the type `ObsView`, which can be used to treat any data container as a vector-like representation of that data container, where each vector element corresponds to a single observation.

`ObsView <: DataView <: AbstractVector`

Lazy representation of a data container as a vector of individual observations.

Any data access is delayed until `getindex` is called, and even `getindex` returns the result of *`datasubset()`* which in general avoids data movement until *`getobs()`* is invoked.

`obsview`(*data*[, *obsdim*]) → ObsView

Create a `ObsView` for the given *data* container. It will serve as a vector-like view into *data*, where every element of the vector points to a single observation in *data*.

Parameters

- **`data`** – The object representing a data container.
- **`obsdim`** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Let us consider our toy matrix `X` again, which we will interpret as containing 5 observations with 2 features each. This time we pass it to *`obsview()`* before iterating over it. Notice how the resulting `ObsView` will look like a vector of vectors. As we will see from the type, each element of the `ObsView` `ov` is just a `SubArray` into `X`. As such, no data from `X` is copied.

```
julia> X = rand(2, 5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> ov = obsview(X)
5-element obsview(::Array{Float64,2}, ObsDim.Last()) with element type SubArray
↪{Float64,1,Array{Float64,2},Tuple{Colon,Int64},true}:
```

(continues on next page)

(continued from previous page)

```
 [0.226582,0.504629]
 [0.933372,0.522172]
 [0.505208,0.0997825]
 [0.0443222,0.722906]
 [0.812814,0.245457]

julia> ov[2] # access second observation
2-element SubArray{Float64,1,Array{Float64,2},Tuple{Colon,Int64},true}:
 0.933372
 0.522172

julia> foreach(println, ov) # now we iterate over observation
[0.226582,0.504629]
[0.933372,0.522172]
[0.505208,0.0997825]
[0.0443222,0.722906]
[0.812814,0.245457]
```

If there is more than one array dimension, all but the observation dimension are implicitly assumed to be features (i.e. part of that observation). As we have seen with `X` in the example above, the default assumption is that the last array dimension enumerates the observations. This can be overwritten by explicitly specifying the `obsdim`. In the following code snippet we treat `X` as a data set that has 2 observations with 5 features each.

```
julia> X = rand(2, 5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> ov = obsview(X, obsdim = 1)
2-element obsview(::Array{Float64,2}, ObsDim.Constant{1}()) with element type SubArray
↪{Float64,1,Array{Float64,2},Tuple{Int64,Colon},true}:
 [0.226582,0.933372,0.505208,0.0443222,0.812814]
 [0.504629,0.522172,0.0997825,0.722906,0.245457]

julia> ov = obsview(X, ObsDim.First()); # same as above but type-stable
```

Similarly, we can also call *`obsview()`* with our toy vector `y`. Recall that a `Vector` is just an `Array` with only one dimension. This example will help demonstrate how an `ObsView` handles data container that are already in a vector-like form.

```
julia> y = rand(5)
5-element Array{Float64,1}:
 0.11202
 0.000341996
 0.380001
 0.505277
 0.841177

julia> ov = obsview(y)
5-element obsview(::Array{Float64,1}, ObsDim.Last()) with element type SubArray
↪{Float64,0,Array{Float64,1},Tuple{Int64},false}:
 0.11202
 0.000341996
 0.380001
 0.505277
 0.841177
```

(continues on next page)

(continued from previous page)

```
julia> ov[2] # access second observation
0-dimensional SubArray{Float64,0,Array{Float64,1},Tuple{Int64},false}:
0.000341996
```

On first glance, the result of indexing into `ov` may seem unintuitive. Why does it return a 0-dimensional `SubArray` instead of simply the value? The main reason for this behaviour is that we try to avoid data movement unless *`getobs()`* is called. Until that point, we just create subsets into the original data container.

```
julia> getobs(ov[2])
0.0003419958128361156

julia> getobs(ov)
5-element Array{Float64,1}:
 0.11202
 0.000341996
 0.380001
 0.505277
 0.841177

julia> getobs(ov, 2)
0.0003419958128361156
```

You may have noted in all the examples so far, that creating an `ObsView` preserves the order of the observations. This is of course on purpose and the desired behaviour. However, since `ObsView` is commonly used as an iterator, one may be inclined to prefer iterating over the data in a random order. To do so, simply combine the functions *`obsview()`* and *`shuffleobs()`*.

```
julia> ov = obsview(shuffleobs(y))
5-element obsview(::SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false},␣
↪ObsDim.Last()) with element type SubArray{Float64,0,Array{Float64,1},Tuple{Int64},
↪false}:
 0.505277
 0.11202
 0.841177
 0.380001
 0.000341996

julia> ov = shuffleobs(obsview(y)) # also possible
5-element obsview(::SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false},␣
↪ObsDim.Last()) with element type SubArray{Float64,0,Array{Float64,1},Tuple{Int64},
↪false}:
 0.505277
 0.380001
 0.000341996
 0.841177
 0.11202
```

It is also possible to link multiple different data containers together on an per-observation level. To do that, simply put all the relevant data container into a single `Tuple`, before passing it to *`obsview()`*. Each element of the resulting `ObsView` will then be a `Tuple`, with the resulting observation in the same tuple position.

```
julia> ov = obsview((X, y))
5-element obsview(::Tuple{Array{Float64,2},Array{Float64,1}}, (ObsDim.Last(),ObsDim.
↪Last())) with element type Tuple{SubArray{Float64,1,Array{Float64,2},Tuple{Colon,
↪Int64},true},SubArray{Float64,0,Array{Float64,1},Tuple{Int64},false}}:
```

(continues on next page)

(continued from previous page)

```
 ([0.226582,0.504629],0.11202)
 ([0.933372,0.522172],0.000341996)
 ([0.505208,0.0997825],0.380001)
 ([0.0443222,0.722906],0.505277)
 ([0.812814,0.245457],0.841177)

julia> ov[2] # access second observation
([0.933372,0.522172],0.000341996)

julia> typeof(ov[2])
Tuple{SubArray{Float64,1,Array{Float64,2},Tuple{Colon,Int64},true},SubArray{Float64,0,
↪Array{Float64,1},Tuple{Int64},false}}
```

It is worth pointing out, that the tuple elements (i.e. data container), that are passed to *obsview()*, need not be of the same type, nor of the same shape. You can observe this in the code above, where `X` is a `Matrix` while `y` is a `Vector`. Note, however, that all tuple elements must be data containers themselves. Furthermore, they all must contain the same exact number of observations.

3.5.2 As Vector of Batches

Another common use case is to iterate over the given data set in small equal-sized chunks. These chunks are usually referred to as *mini-batches*.

Not unlike `ObsView`, this package provides a vector-like type called `BatchView`, that can be used to treat any data container as a vector of equal-sized batches.

BatchView <: DataView <: AbstractVector

Lazy representation of a data container as a vector of batches. Each batch will contain an equal amount of observations in them. In the case that the number ob observations is not dividable by the specified (or inferred) batch-size, the remaining observations will be ignored.

Any data access is delayed until `getindex` is called, and even `getindex` returns the result of *datasubset()* which in general avoids data movement until *getobs()* is invoked.

batchview(*data*[, *size*|*maxsize*][, *count*][, *obsdim*]) → BatchView

Create a `BatchView` for the given *data* container. It will serve as a vector-like view into *data*, where every element of the vector points to a batch of *size* observations from *data*. The number of batches and the batch-size can be specified using (keyword) parameters *count* and *size* (or alternatively *maxsize*).

In the case that the size of the dataset is not dividable by the specified (or inferred) *size*, the remaining observations will be ignored. If *maxsize* is provided instead of *size*, then the next dividable size will be used such that no observations are ignored.

Parameters

- **data** – The object representing a data container.
- **size** (*Integer*) – Optional. The exact number of observations in each batch.
- **maxsize** (*Integer*) – Optional alternative to *size*. The maximal number of observations in each batch, such that all observations get used.
- **count** (*Integer*) – Optional. The number of batches that should be used. This will also we the length of the return value.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-

stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Consider the following toy data-matrix `X`, which we will interpret as containing a total of 5 observations, where each observation consists of 2 features.

```
julia> X = rand(2, 5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457
```

Using a prime number for the total number of observations makes this data container a particularly interesting example for using *`batchview()`*. Unless we choose a batch-size of `1` or `5`, there is no way to iterate the whole data in terms of equally-sized batches. `BatchView` deals with such edge cases by ignoring the excess observations with an informative message.

```
julia> bv = batchview(X, size = 2)
INFO: The specified values for size and/or count will result in 1 unused data points
2-element batchview(::Array{Float64,2}, 2, 2, ObsDim.Last()) with element type␣
↪SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 [0.226582 0.933372; 0.504629 0.522172]
 [0.505208 0.0443222; 0.0997825 0.722906]

julia> bv = batchview(X, maxsize = 2)
5-element batchview(::Array{Float64,2}, 1, 5, ObsDim.Last()) with element type␣
↪SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 [0.226582; 0.504629]
 [0.933372; 0.522172]
 [0.505208; 0.0997825]
 [0.0443222; 0.722906]
 [0.812814; 0.245457]
```

You can query the size of each batch by using the function `batchsize()` on any `BatchView`.

```
julia> batchsize(bv)
2
```

Similar to `ObsView`, a `BatchView` acts like a vector and can be used as such. The one big difference to the former is that each element is now a batch of `X` instead of a single observation.

```
julia> bv[2] # access second batch
2×2 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.505208   0.0443222
 0.0997825  0.722906
```

Naturally, *`batchview()`* also supports the optional parameter `obsdim`, which can be used to specify which dimension denotes the observation. If that concept of dimensionality does not make sense for the given data container, then `obsdim` can simply be omitted.

```
julia> bv = batchview(X', size = 2, obsdim = 1) # note the transpose
INFO: The specified values for size and/or count will result in 1 unused data points
2-element batchview(::Array{Float64,2}, 2, 2, ObsDim.Constant{1}()) with element type␣
↪SubArray{Float64,2,Array{Float64,2},Tuple{UnitRange{Int64},Colon},false}:
 [0.226582 0.504629; 0.933372 0.522172]
 [0.505208 0.0997825; 0.0443222 0.722906]
```

So far we used the parameter `size` to explicitly specify how many observation we want to be in each batch. Alternatively, we can also use the parameter `count` to specify the total number of batches that we would like to use.

```
julia> bv = batchview(X, count = 4)
INFO: The specified values for size and/or count will result in 1 unused data points
4-element batchview(::Array{Float64,2}, 1, 4, ObsDim.Last()) with element type␣
↪SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 [0.226582; 0.504629]
 [0.933372; 0.522172]
 [0.505208; 0.0997825]
 [0.0443222; 0.722906]

julia> bv[2] # access second batch
2×1 SubArray{Float64,2,Array{Float64,2},Tuple{Colon,UnitRange{Int64}},true}:
 0.933372
 0.522172
```

Note how in the above example, the inferred batch-size is `1`. Arguably, this makes the resulting `BatchView`, `bv`, appear very similar to an `ObsView`. The big difference, though, is that `BatchView` preserves the shape on indexing. Consequently, each element of `bv` is a subtype of `AbstractMatrix` and not `AbstractVector`.

It is also possible to call *`batchview()`* with multiple data containers wrapped in a `Tuple`. Note, however, that all data containers must have the same total number of observations. Using a tuple this way will link those data containers together on a per-observation basis.

```
julia> y = rand(5)
5-element Array{Float64,1}:
 0.11202
 0.000341996
 0.380001
 0.505277
 0.841177

julia> bv = batchview((X, y))
INFO: The specified values for size and/or count will result in 1 unused data points
2-element batchview(::Tuple{Array{Float64,2},Array{Float64,1}}, 2, 2, (ObsDim.Last(),
↪ObsDim.Last())) with element type Tuple{SubArray{Float64,2,Array{Float64,2},Tuple
↪{Colon,UnitRange{Int64}},true},SubArray{Float64,1,Array{Float64,1},Tuple{UnitRange
↪{Int64}},true}}:
([0.226582 0.933372; 0.504629 0.522172], [0.11202,0.000341996])
([0.505208 0.0443222; 0.0997825 0.722906], [0.380001,0.505277])

julia> bv[2]
([0.505208 0.0443222; 0.0997825 0.722906], [0.380001,0.505277])
```

3.5.3 As Vector of Sequences

Time series data, or more generally *sequence data*, often requires a special type of preparation in order to work with it in a machine learning experiment. The big difference to "normal" data is that in sequence data the observations are not independent from each other. For example if you think of a piece of text as a sequence of words (i.e. each word is an observation), you'll notice that there is an inherent order in the data.

```
julia> data = split("The quick brown fox jumps over the lazy dog")
9-element Array{SubString{String},1}:
 "The"
 "quick"
 "brown"
 "fox"
```

(continues on next page)

(continued from previous page)

```
 "jumps"
 "over"
 "the"
 "lazy"
 "dog"
```

Unlabeled Windows

There are some common scenarios when working with sequence data such as the above. Before we think about more practical use cases that require labeled windows, let us quickly consider the case that we would like to process our sequence data in chunks of equally sized windows.

slidingwindow(*data*, *size*[, *stride*][, *obsdim*])

Return a vector-like view of the *data* for which each element is a fixed size "window" of *size* adjacent observations. By default these windows are not overlapping.

Parameters

- **data** – The object representing a data container.
- **size** (*Integer*) – The number of observations in each window.
- **stride** (*Integer*) – Optional. The step size between the starting observation of susequent windows. Defaults to *size*.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Its worth pointing out that only complete windows are included in the output. This implies that it is possible for excess observations to be omitted from the view. The following code snippet shows an example that partitions 22 observation into 4 windows, where the last two observations are omitted.

```
julia> A = slidingwindow(1:22, 4)
5-element slidingwindow(::UnitRange{Int64}, 4) with element type SubArray{Int64,1,
↪UnitRange{Int64},Tuple{UnitRange{Int64}},true}:
 [1, 2, 3, 4]
 [5, 6, 7, 8]
 [9, 10, 11, 12]
 [13, 14, 15, 16]
 [17, 18, 19, 20]
```

Note that the values of the given data are not actually copied. Instead the function *datasubset()* is called when `getindex` is invoked. To actually get a copy of the data at some window use the function *getobs()*.

```
julia> A[2]
4-element SubArray{Int64,1,UnitRange{Int64},Tuple{UnitRange{Int64}},true}:
 5
 6
 7
 8

julia> getobs(A, 2)
4-element Array{Int64,1}:
 5
 6
```

(continues on next page)

(continued from previous page)

```
 7
 8
```

Up to this point the behaviour may be very reminiscent of a *batchview()*, but this is where the similarities end. The optional parameter `stride` can be used to specify the distance between the start elements of each adjacent window. By default the stride is equal to the window size.

```
julia> A = slidingwindow(1:22, 4, stride=2)
10-element slidingwindow(::UnitRange{Int64}, 4, stride = 2) with element type SubArray
↪{Int64,1}:
 [1, 2, 3, 4]
 [3, 4, 5, 6]
 [5, 6, 7, 8]
 [7, 8, 9, 10]
 [9, 10, 11, 12]
 [11, 12, 13, 14]
 [13, 14, 15, 16]
 [15, 16, 17, 18]
 [17, 18, 19, 20]
 [19, 20, 21, 22]

julia> A = slidingwindow(data, 4, stride=2)
3-element slidingwindow(::Array{SubString{String},1}, 4, stride = 2) with element␣
↪type SubArray{...}:
 ["The", "quick", "brown", "fox"]
 ["brown", "fox", "jumps", "over"]
 ["jumps", "over", "the", "lazy"]
```

Labeled Windows

Now that we have seen the general idea of `slidingwindow`, let us consider a more practical variation of it. A conceptually simple use case may be that we want to predict the next word in a sentence given all the words that came before it (e.g. for autocompletion).

An interesting aspect of sequence prediction is that we can transform an unlabeled sequence into a number of labeled sub-sequences. Let's again use our original (unlabeled) data for this.

```
julia> data = split("The quick brown fox jumps over the lazy dog")
9-element Array{SubString{String},1}:
 "The"
 "quick"
 "brown"
 "fox"
 "jumps"
 "over"
 "the"
 "lazy"
 "dog"
```

If we were to train a model that given two words should predict the next word, we would need to rearrange our `data` quite a bit. To make this process more convenient we provide a custom method for `slidingwindow` that expects an target-index function `f` as first parameter.

slidingwindow (*f*, *data*, *size*[, *stride*][, *excludetarget*][, *obsdim*])

Return a vector-like view of the *data* for which each element is a tuple of two elements:

1. A fixed size "window" of *size* adjacent observations. By default these windows are not overlapping. This can be changed by explicitly specifying a *stride*.
2. A single target (or vector of targets) for the window. The content of the target(s) is defined by the label-index function *f*.

Parameters

- **f** (*Function*) – A unary function that takes the index of the first observation in the current window and should return the index (or indices) of the associated target(s) for that window.
- **data** – The object representing a data container.
- **size** (*Integer*) – The number of observations in each window.
- **stride** (*Integer*) – Optional. The step size between the starting observation of susequent windows. Defaults to *size*.
- **excludetarget** (*Bool*) – Should a target index returned by *f* also occur in the window, then setting this to `true` will make sure that such elements are removed from the window. Defaults to `false`.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Note that only complete and in-bound windows are included in the output, which implies that it is possible for excess observations to be omitted from the resulting view.

```
julia> A = slidingwindow(i->i+2, data, 2, stride=1)
7-element slidingwindow(::##9#10, ::Array{SubString{String},1}, 2, stride = 1) with␣
↪element type Tuple{...}:
 (["The", "quick"], "brown")
 (["quick", "brown"], "fox")
 (["brown", "fox"], "jumps")
 (["fox", "jumps"], "over")
 (["jumps", "over"], "the")
 (["over", "the"], "lazy")
 (["the", "lazy"], "dog")

julia> A = slidingwindow(i->i-1, data, 2, stride=1)
7-element slidingwindow(::##11#12, ::Array{SubString{String},1}, 2, stride = 1) with␣
↪element type Tuple{...}:
 (["quick", "brown"], "The")
 (["brown", "fox"], "quick")
 (["fox", "jumps"], "brown")
 (["jumps", "over"], "fox")
 (["over", "the"], "jumps")
 (["the", "lazy"], "over")
 (["lazy", "dog"], "the")
```

As hinted above, it is also allowed for `f` to return a vector of indices. This can be useful for emulating techniques such as skip-gram.

```
julia> A = slidingwindow(i->[i-2:i-1; i+1:i+2], data, 1)
5-element slidingwindow(::##11#12, ::Array{SubString{String},1}, 1) with element type␣
↪Tuple{...}:
 (["brown"], ["The", "quick", "fox", "jumps"])
```

(continues on next page)

(continued from previous page)

```
 (["fox"], ["quick", "brown", "jumps", "over"])
 (["jumps"], ["brown", "fox", "over", "the"])
 (["over"], ["fox", "jumps", "the", "lazy"])
 (["the"], ["jumps", "over", "lazy", "dog"])
```

Should it so happen that the targets overlap with the features, then the affected observation(s) will be present in both. To change this behaviour one can set the optional parameter `excludetarget = true`. This will remove the target(s) from the feature window.

```
julia> slidingwindow(i->i+2, data, 5, stride = 1, excludetarget = true)
5-element slidingwindow(::##17#18, ::Array{SubString{String},1}, 5, stride = 1) with␣
↪element type Tuple{...}:
 (["The", "quick", "fox", "jumps"], "brown")
 (["quick", "brown", "jumps", "over"], "fox")
 (["brown", "fox", "over", "the"], "jumps")
 (["fox", "jumps", "the", "lazy"], "over")
 (["jumps", "over", "lazy", "dog"], "the")
```

While these data views are also data iterators, the inverse is not true. In the following section we will introduce a number of **data iterators**, that don't make *any* other promises than, well, iteration. As such, they may not know how many observations they can provide, nor have the means to access specific observations. Consequently, these data iterators are not data containers. We will see how that is useful, and also how some of them are actually created using a data container as input.

3.6 Data Iterators

We hinted a few times before that we differentiate between two kinds of data sources, namely *data containers* and *data iterators*. We also briefly mentioned that a data source can either be one, both, or neither of the two. So far, though, we solely focused on data containers, and a special kind of data container / data iterator hybrid that we called *data views*. If we free ourselves from the notion that a data source has to know how many observations it can provide, or that it has to understand the concept of "accessing a specific observation", it opens up a lot of new functionality that would otherwise be infeasible.

In this document we will finally introduce those types of data iterators, that do not make any other guarantees than what the name implies: **iteration**. As such, they may not know how many observation they can provide, or even understand what an observation-index should be. One could ask what we could possibly gain with such a type over the already introduced - and seemingly more knowledgeable - data views. The answer is: **They address different problems**. A very common and illustrative task, that these data iterators are uniquely suited for, is continuous random sampling from a data container.

3.6.1 Randomly sample Observations

We previously introduced a type called `ObsView`, which we showed can be used to convert any data container to a data iterator. As such, it makes it possible to, well, iterate over the data container one observation at a time. Additionally, an `ObsView` also behaves like a vector, in that it allows to use `getindex` to query a specific observation. By combining `ObsView` with *`shuffleobs()`*, we were also able to iterate over all the observations from a data container in a random order.

A different approach to iterating over a data container one observation after another, is to continuously sample a single observation from it. Per definition that means that the process of determining the "next" observation is random. Thus, indexing a specific observation of that iterator is ill defined. Therefore data iterators in general only guarantee that they

can be used as a Julia iterator; every additional functionality is optional. One such type data iterator that this package provides is called *RandomObs*.

RandomObs <: ObsIterator

A decorator type that transforms a data containers into a data iterator. Each iteration produces a randomly sampled observation from the given data container (with replacement).

Note that each iteration returns the result of a *datasubset()*, which means that any data movement is delayed until *getobs()* is called.

RandomObs (*data*[, *count*][, *obsdim*]) → RandomObs

Create an iterator that generates *count* randomly sampled observations from the given *data* container. In the case *count* is not provided, it will generate random samples indefinitely.

Parameters

- **data** – The object representing a data container.
- **count** (*Integer*) – Optional. The number of randomly sampled observations that the iterator will generate before stopping. If omitted, the iterator will generate randomly sampled batches forever.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Consider the following toy data vector `x` that has 5 observations. We will use simple values to make it easy to see where each observation ends up.

```
julia> x = collect(1.0:5)
5-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

Because `x` is a `Vector` it is considered a data container. Thus we can pass it to *RandomObs()*. If we specify a `count` (i.e. limit the number of samples to generate), we can use `collect` on it.

```
julia> iter = RandomObs(x, count = 10)
RandomObs(::Array{Float64,1}, 10, ObsDim.Last())
 Iterator providing 10 observations

julia> xnew = collect(iter)
10-element Array{SubArray{Float64,0,Array{Float64,1},Tuple{Int64},false},1}:
 4.0
 4.0
 1.0
 5.0
 2.0
 5.0
 1.0
 2.0
 1.0
 5.0

julia> xnew[1]
```

(continues on next page)

(continued from previous page)

```
0-dimensional SubArray{Float64,0,Array{Float64,1},Tuple{Int64},false}:
 4.0
```

Notice two things in the code above.

- The observations from `x` are sampled randomly with replacement. That means the same observation can occur in `xnew` once, multiple times, or not at all.
- Each sampled observation is actually a lazy subset (i.e. a `SubArray`) into the original data container `x`. To get the underlying data you need to use *`getobs()`* on the result.

The constructor parameter `count` is optional and can be omitted. If that is the case, then the resulting iterator will continue to sample random observations forever, or until interrupted.

```
julia> iter = RandomObs(x)
RandomObs(::Array{Float64,1}, ObsDim.Last())
 Iterator providing Inf observations

julia> collect(iter) # can't collect infinite iterator
ERROR: MethodError: no method matching _collect(::UnitRange{Int64}, ::MLDataPattern.
↪RandomObs{SubArray{Float64,0,Array{Float64,1},Tuple{Int64},false},Array{Float64,1},
↪LearnBase.ObsDim.Last,Base.IsInfinite}, ::Base.HasEltype, ::Base.IsInfinite)

julia> collect(take(iter, 5))
5-element Array{SubArray{Float64,0,Array{Float64,1},Tuple{Int64},false},1}:
 4.0
 4.0
 1.0
 5.0
 2.0
```

Similar to an `ObsView`, it is also possible to use a `Tuple` to group data containers together on a per-observation level. This will cause each iteration to return a `Tuple` of equal length and ordering.

```
julia> y = [:a, :b, :c, :d, :e];

julia> iter = RandomObs((x, y), count = 5)
RandomObs(::Tuple{Array{Float64,1},Array{Symbol,1}}, 5, (ObsDim.Last(),ObsDim.Last()))
 Iterator providing 5 observations

julia> collect(iter)
5-element Array{Tuple{SubArray{Float64,0,Array{Float64,1},Tuple{Int64},false},SubArray
↪{Symbol,0,Array{Symbol,1},Tuple{Int64},false}},1}:
 (4.0,:d)
 (4.0,:d)
 (1.0,:a)
 (5.0,:e)
 (2.0,:b)
```

In case of skewed class distributions we offer an alternative iterator called `BalancedObs`, which samples from each label uniformly.

```
julia> y = [:a, :a, :a, :a, :a, :a, :a, :a, :b, :b];

julia> iter = BalancedObs((1:10, y), count = 6)
BalancedObs(::Tuple{UnitRange{Int64},Array{Symbol,1}}, 6, (ObsDim.Last(), ObsDim.
↪Last()))
```

(continues on next page)

(continued from previous page)

```
 Iterator providing 6 observations

julia> collect(iter)
6-element Array{Tuple{SubArray{Int64,0,UnitRange{Int64},Tuple{Int64},false},SubArray
↪{Symbol,0,Array{Symbol,1},Tuple{Int64},false}},1}:
 (10, :b)
 (4, :a)
 (9, :b)
 (7, :a)
 (8, :a)
 (9, :b)
```

3.6.2 Randomly sample Mini-Batches

Similarly to `BatchView`, an object of type *`RandomBatches`* can be used as an iterator that produces a mini-batch of fixed size in each iteration. In contrast to `BatchView`, however, *`RandomBatches`* generates completely random mini-batches, in which the containing observations are generally not adjacent to each other in the original dataset.

`RandomBatches <: BatchIterator`

A decorator type that transforms a data container into a data iterator, that on each iteration returns a batch of fixed size containing randomly sampled observation from the given data container (with replacement).

Each iteration returns the result of calling *`datasubset()`*, which means that any data movement is delayed until *`getobs()`* is called.

`RandomBatches`(*data*[, *size*][, *count*][, *obsdim*]) → RandomBatches

Create an iterator that generates *count* randomly sampled batches from the given *data* container using a batch-size of *size*. In the case *count* is not provided, it will generate random batches indefinitely.

Parameters

- **`data`** – The object representing a data container.
- **`size`** (*`Integer`*) – Optional. The batch-size of each batch. I.e. the number of randomly sampled observations in each batch.
- **`count`** (*`Integer`*) – Optional. The number of randomly sampled batches that the iterator will generate before stopping. If omitted, the iterator will generate randomly sampled observations forever.
- **`obsdim`** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Consider our simple toy data vector `x` again, that we used before to motivate *`RandomObs`*.

```
julia> x = collect(1.0:5)
5-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

Because `x` is considered a data container, it can be used to produce random batches with *`RandomBatches`*. We can use the parameter `size` to specify how many observations each mini-batch should contain. If we also specify a `count` (i.e. limit the number of mini-batches to generate), we can use `collect` on the result.

```
julia> iter = RandomBatches(x, size = 3, count = 10)
RandomBatches(::Array{Float64,1}, 3, 10, ObsDim.Last())
 Iterator providing 10 batches of size 3

julia> collect(iter)
10-element Array{SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false},1}:
 [4.0,4.0,1.0]
 [5.0,2.0,5.0]
 [1.0,2.0,1.0]
 [5.0,2.0,4.0]
 [1.0,1.0,2.0]
 [2.0,5.0,2.0]
 [3.0,2.0,1.0]
 [2.0,5.0,4.0]
 [1.0,2.0,4.0]
 [5.0,5.0,2.0]
```

The constructor parameter `count` is optional and can be omitted. If that is the case, then the resulting iterator will continue to sample random mini-batches forever, or until interrupted.

```
julia> iter = RandomBatches(x, size = 3)
RandomBatches(::Array{Float64,1}, 3, ObsDim.Last())
 Iterator providing Inf batches of size 3

julia> collect(iter) # can't collect infinite iterator
ERROR: MethodError: no method matching _collect(::UnitRange{Int64}, ::MLDataPattern.
↪RandomBatches{SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false},Array
↪{Float64,1},LearnBase.ObsDim.Last,Base.IsInfinite}, ::Base.HasEltype, ::Base.
↪IsInfinite)

julia> collect(take(iter, 5))
5-element Array{SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false},1}:
 [4.0,4.0,1.0]
 [5.0,2.0,5.0]
 [1.0,2.0,1.0]
 [5.0,2.0,4.0]
 [1.0,1.0,2.0]
```

Because the utilized data container `x` is a vector, each mini-batch is a one-dimensional `SubArray` (i.e. a lazy subset into `x`). The type of each mini-batch depends on the given data container. For example if we instead use a feature *matrix* `X`, each mini-batch would be a two-dimensional `SubArray`.

```
julia> X = rand(2, 5)
2×5 Array{Float64,2}:
 0.226582  0.933372  0.505208   0.0443222  0.812814
 0.504629  0.522172  0.0997825  0.722906   0.245457

julia> iter = RandomBatches(X, size = 3, count = 10)
RandomBatches(::Array{Float64,2}, 3, 10, ObsDim.Last())
 Iterator providing 10 batches of size 3

julia> collect(iter)
10-element Array{SubArray{Float64,2,Array{Float64,2},Tuple{Colon,Array{Int64,1}},
↪false},1}:
 [0.226582 0.933372 0.933372; 0.504629 0.522172 0.522172]
 [0.812814 0.933372 0.505208; 0.245457 0.522172 0.0997825]
 [0.933372 0.226582 0.933372; 0.522172 0.504629 0.522172]
```

(continues on next page)

(continued from previous page)

```
 [0.812814 0.0443222 0.226582; 0.245457 0.722906 0.504629]
 [0.933372 0.0443222 0.812814; 0.522172 0.722906 0.245457]
 [0.812814 0.933372 0.0443222; 0.245457 0.522172 0.722906]
 [0.226582 0.933372 0.226582; 0.504629 0.522172 0.504629]
 [0.0443222 0.812814 0.505208; 0.722906 0.245457 0.0997825]
 [0.226582 0.812814 0.812814; 0.504629 0.245457 0.245457]
 [0.812814 0.812814 0.0443222; 0.245457 0.245457 0.722906]
```

It is also possible to link multiple different data containers together on an per-observation level. This way they can be sampled from as one coherent unit. To do that, simply put all the relevant data container into a single `Tuple`, before passing it to *`RandomBatches()`*. This will cause each iteration to return a `Tuple` of equal length and ordering.

```
julia> y = [:a, :b, :c, :d, :e];

julia> iter = RandomBatches((x, y), size = 3, count = 5)
RandomBatches(::Tuple{Array{Float64,1},Array{Symbol,1}}, 3, 5, (ObsDim.Last(),ObsDim.
↪Last()))
 Iterator providing 5 batches of size 3

julia> collect(iter)
5-element Array{Tuple{SubArray{Float64,1,Array{Float64,1},Tuple{Array{Int64,1}},false}
↪,SubArray{Symbol,1,Array{Symbol,1},Tuple{Array{Int64,1}},false}},1}:
 ([4.0,4.0,1.0],Symbol[:d,:d,:a])
 ([5.0,2.0,5.0],Symbol[:e,:b,:e])
 ([1.0,2.0,1.0],Symbol[:a,:b,:a])
 ([5.0,2.0,4.0],Symbol[:e,:b,:d])
 ([1.0,1.0,2.0],Symbol[:a,:a,:b])
```

The fact that the observations within each mini-batch are randomly sampled has an important consequences. Because observations are sampled with replacement, 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 noticeable impact.

3.6.3 The BufferGetObs Type

You may have noticed that all the data iterators and data views, *`RandomObs`*, *`RandomBatches`*, `ObsView`, and `BatchView`, return a lazy data subset for every iteration. This is useful in general, because it avoids data access and memory allocation until the user makes a conscious decision to do so by calling *`getobs()`*. That said, in many use cases it would be convenient if we could tell a data iterator (or data view) to return the actual data in each iteration, instead of a lazy subset. To that end, this package provides a special iterator decorator that is itself an iterator (just "iterator"; it is not a "data iterator") called *`BufferGetObs`*.

`class BufferGetObs`

A stateful iterator that decorates an inner `iterator`. When iterated over the type stores the output of `next(iterator,state)` into a `buffer` using `getobs!(buffer, ...)`. Depending on the type of data provided by `iterator` this may be more memory efficient than `getobs(...)`. In the case of array data, for example, this allows for cache-efficient processing of each element without allocating a temporary array.

Note that not all types of data support buffering, because it is the developers's choice to opt-in and implement a custom *`getobs!()`*. For those types that do not provide a custom *`getobs!()`*, the `buffer` will be ignored and the result of `getobs(...)` returned.

`BufferGetObs`(*iterator*[, *buffer*]) → BufferGetObs

Parameters

- **iterator** – Some type that implements the iterator pattern, and for which every generated element supports *getobs()*
- **buffer** – Optional. If the elements of *iterator* support *getobs!()*, then this buffer is used as temporary storage on every iteration. Defaults to the result of *getobs()* on the first element of *iterator*.

Let us take a look at an example where *BufferGetObs* shines. Consider the following toy feature matrix `X` that contains 5 observation with 3 features each. Notice how in this example each row denotes a single observation.

```
julia> X = rand(5, 3)
5×3 Array{Float64,2}:
 0.226582  0.0997825  0.11202
 0.504629  0.0443222  0.000341996
 0.933372  0.722906   0.380001
 0.522172  0.812814   0.505277
 0.505208  0.245457   0.841177
```

Given that arrays in Julia are in column-major order, the features of each observations are not a continuous block of memory. This fact by itself need not be an issue. For example, if we would want to iterate over the data container one observation at a time, we could still use *obsview()* without noticing any obvious differences.

```
julia> ov = obsview(X, obsdim = 1)
5-element obsview(::Array{Float64,2}, ObsDim.Constant{1}()) with element type SubArray
↪{Float64,1,Array{Float64,2},Tuple{Int64,Colon},true}:
 [0.226582,0.0997825,0.11202]
 [0.504629,0.0443222,0.000341996]
 [0.933372,0.722906,0.380001]
 [0.522172,0.812814,0.505277]
 [0.505208,0.245457,0.841177]

julia> ov[2] # access second observation
3-element SubArray{Float64,1,Array{Float64,2},Tuple{Int64,Colon},true}:
 0.504629
 0.0443222
 0.000341996
```

On the other hand, if need to interact with some C library, which requires us to pass to it a proper continuous array, then we can't just use this `SubArray` as it is. Luckily, we could just use *getobs()* on each subset and pass the resulting `Array` to the C library.

```
for xv in obsview(X, obsdim = 1)
    x = getobs(xv)
    # pass x to some c library
end
```

The remaining annoyance with the above code is that it allocates temporary memory on each iteration. In a performance critical inner loop this is undesired and could have a significant influence on the performance. To avoid that problem, we can preallocate a buffer and reuse it in every iteration with *getobs!()*.

```
x = Vector{Float64}(3)
for xv in obsview(X, obsdim = 1)
    getobs!(x, xv)
    # pass x to some c library
end
```

This should give us pretty good performance. This pattern is so common, however, that this package provides a convenience implementation for it, namely *BufferGetObs*.

```
for x in BufferGetObs(obsview(X, obsdim = 1), Vector{Float64}(3))
    # pass x to some c library
end
```

The nice thing about using *BufferGetObs* is that it doesn't even require us to manually provide a preallocated buffer. If omitted, *BufferGetObs* simply reuses the result of *getobs()* from the first element.

```
for x in BufferGetObs(obsview(X, obsdim = 1))
    # pass x to some c library
end
```

Furthermore, because it is so common to use *BufferGetObs* in combination with either `ObsView` or `BatchView`, we provide convenience functions for both. More concretely, the functions *eachobs()* and *eachbatch()* simply translate to `BufferGetObs(ObsView(...))` and `BufferGetObs(BatchView(...))` respectively.

eachobs (*data*[, *obsdim*]) → BufferGetObs

Iterate over *data* one observation at a time using `ObsView`. In contrast to `ObsView`, each iteration returns the result of *getobs()* (i.e. actual data). If supported by the type of *data*, a buffer will be preallocated and reused every iteration for memory efficiency.

Parameters

- **data** – The object representing a data container.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns The result of `BufferGetObs(ObsView(data, obsdim))`

eachbatch (*data*[, *size*][, *count*][, *obsdim*]) → BufferGetObs

Iterate over *data* one batch at a time using `BatchView`. In contrast to `BatchView`, each iteration returns the result of *getobs()* (i.e. actual data). If supported by the type of data, a buffer will be preallocated and reused for memory efficiency.

The (constant) batch-size can be either provided directly using *size* or indirectly using *count*, which derives the size based on *nobs()*. In the case that the size of the *data* is not dividable by the specified (or inferred) *size*, the remaining observations will be ignored.

Parameters

- **data** – The object representing a data container.
- **size** (*Integer*) – Optional. The number of observations in each batch.
- **count** (*Integer*) – Optional. The number of batches that should be used. This will also we the length of the return value.
- **obsdim** – Optional. If it makes sense for the type of *data*, then *obsdim* can be used to specify which dimension of *data* denotes the observations. It can be specified in a type-stable manner as a positional argument, or as a more convenient keyword parameter. See *Observation Dimension* for more information.

Returns The result of `BufferGetObs(BatchView(data, size, count, obsdim))`

CHAPTER 4

Indices and tables

4.1 LICENSE

The MLDataPattern.jl package is licensed under the **MIT "Expat" License**

see LICENSE.md in the Github repository.

- genindex
- modindex
- search

Index