Contributing to `smartdata`

Ignacio Cordón

2019-01-07

The package `smartdata` aims to provide an useful common interface for a collection of machine learning packages. In other words, it was thought to ease the application of algorithms, since there are plenty of packages in R dedicated to machine learning topics. In addition to this, it offers a pipe operator, through `magrittr` package, which makes possible to have a nice and convenient workflow of operations.

Since the purpose of this package is to integrate as much packages as possible, all contributions are welcomed. The goal of this vignette is to describe as easily as possible the general structure of the package, so that making contributions gets easier for an external programmer.

General structure

The package covers the following topics, each one with its respective `.R` file in the folder `R/` and with an associated function:

Topic	File	Wrapper
Oversampling	`oversampling.R`	`oversample`
Instance selection	`instanceSelection.R`	`instance_selection`
Feature selection	`featureSelection.R`	`feature_selection`
Normalization	`normalization.R`	`normalize`
Discretization	`discretization.R`	`discretize`
Space transformation	`spaceTransformation.R`	`space_transformation`
Outliers	`outliers.R`	`clean_outliers`
Noise	`noise.R`	`clean_noise`
Missing values	`missingValues.R`	`impute_missing`

Naming conventions

Inside the package, everything is coded using `CamelCase` style, but the API for the functions callable from outside the package, once loaded, is named using `snake_case` style. The reason for the former convention is to ensure compatibility with `Tidyverse` packages, which use that naming convention.

`smartdata.R`

The file `R/smartdata.R` contains the description of the package, some imports needed for the correct functioning of the software, the definition of the documentation function `which_options`, which describes the parameters and options available for each method, and the `preprocess` mappings. To illustrate the purpose of `which_options` and the `preprocess` functions, here are some examples:

```
which_options("instance_selection")
#> Possible methods are: 'CNN', 'ENN', 'multiedit', 'FRIS'
```

```
which_options("instance_selection", "multiedit")
#> For more information do: ?class::multiedit
#> Parameters for multiedit are:
#>   * k: Number of neighbors used in KNN
#>        Default value: 1
#>   * num_folds: Number of partitions the train set is split in
#>                Default value: 3
#>   * null_passes: Number of null passes to use in the algorithm
#>                  Default value: 5
```

`preprocess` is an `S3` method which assigns to an object `task` the name of the package the method is in as a class, and calls the function which resolves the task (which is going to be another `S3` method which different for each single package):

```
preprocess <- function(task){
  UseMethod("preprocess")
}

preprocess.instanceSelection <- function(task){
  class(task) <- instSelectionPackages[[task$method]]$pkg

  doInstSelection(task)
}
```

{wrapper}.R

Inside each `{wrapper}.R` file, we will find, at the top, a declaration of the available methods for that preprocessing with information about the package they come from (`pkg` slot), and the original name of the method in that package (`map` slot). For example, for `instanceSelection.R`:

```
instSelectionPackages <- list(
  "CNN" = list(
    pkg = "unbalanced",
    map = "ubCNN"
  ),
  "ENN" = list(
    pkg = "unbalanced",
    map = "ubENN"
  ),
  "multiedit" = list(
    pkg       = "class"
  ),
  "FRIS" = list(
    pkg  = "RoughSets",
    map  = "IS.FRIS.FRST"
  )
)
```

An absent `map` slot means the method in the origin package coincides with the method name.

From the information above, we know `CNN` method derives from the `unbalanced` package

and the `ubCNN` function inside it.

Names of valid methods will be contained in a similar variable:

```
instSelectionMethods <- names(instSelectionPackages)
```

For each method, there should be a declared variable `arg.{method}` which would contain information for each of the accepted arguments for that method (`default` values in case those arguments can be omitted when calling the function, a `check` function to ensure the parameter is passed correctly, and a string of information about the parameter, `info`, which will be shown if `which_options` is called with the names of the wrapper and the method). Specifically, package `checkmate` has been used to provide verbose and compact check functions (curried in the first argument with `Curry` method from `functional` package, i.e. the same as writing `function(x) { qexpect(x, rules = "foo") }`). As example:

```
args.multiedit <- list(
  k = list(
    check = Curry(qexpect, rules = "X1[1,Inf)", label = "k"),
    info = "Number of neighbors used in KNN",
    default = 1
  ),
  num_folds = list(
    check = Curry(qexpect, rules = "X1[1,Inf)", label = "num_folds"),
    info = "Number of partitions the train set is split in",
    default = 3,
    map = "V"
  ),
  null_passes = list(
    check = Curry(qexpect, rules = "X1[1,Inf)", label = "null_passes"),
    info = "Number of null passes to use in the algorithm",
    default = 5,
    map = "I"
  )
)
```

If an argument has a missing `default` slot, then a value must be provided for it when the function is called.

There should be an `S3` method which evaluates the wrapper for methods included in a given package. As an example, let's observe the method that evaluates instance selection using methods from `unbalanced`:

```
doInstSelection.unbalanced <- function(task){
  callArgs <- eval(parse(text = paste("args.", task$method, sep = "")))
  callArgs <- mapArguments(task$args, callArgs)
  classAttr <- task$classAttr
  classIndex <- task$classIndex
  dataset <- task$dataset

  method <- mapMethod(instSelectionPackages, task$method)

  # CNN and ENN need minority class as 1, and majority one as 0
  minorityClass <- whichMinorityClass(dataset, classAttr)
  minority <- whichMinority(dataset, classAttr)
```

```
  old_levels <- levels(dataset[, classIndex])
  new_levels <- old_levels
  new_levels[old_levels == minorityClass] <- 1
  new_levels[old_levels != minorityClass] <- 0
  levels(dataset[, classIndex]) <- as.numeric(new_levels)

  callArgs <- c(list(X = dataset[, -classIndex],
                     Y = dataset[, classIndex], verbose = FALSE),
                callArgs)
  result <- do.call(method, callArgs)
  result <- cbind(result$X, result$Y)
  # Assign original classAttr name to class column
  names(result)[classIndex] <- classAttr
  # Retrieve original levels for class
  levels(result[, classIndex]) <- old_levels
  # Reset rownames
  rownames(result) <- c()

  result
}
```

A reason to use such an structure (resolve each call grouping methods per origin package instead of treating each method differently) is that methods coming from the same package usually need similar adjustments (for example, order of columns).

Finally, the correspoding wrapper will:

- Make some basic checks (`dataset` is a `data.frame`, `class_attr` exists in the dataset as a column).
- Create a `preprocessingTask` with the dataset, the type of the preprocessing, the method to use, the `classAttr` (note the `CamelCase` style), and further arguments for the method that will be stored as a variable `args` inside the task.
- Call the `preprocess` function
- Output the result

As an example, the `instance_selection` wrapper:

```
instance_selection <- function(dataset, method, class_attr = "Class", ...){
  classAttr <- class_attr
  checkDataset(dataset)
  checkDatasetClass(dataset, classAttr)

  method <- matchArg(method, instSelectionMethods)

  # Perform instance selection
  task <- preprocessingTask(dataset, "instanceSelection", method, classAttr, ...)
  dataset <- preprocess(task)

  dataset
}
```

utils.R

The file `utils.R` contains helpers to assist in the coding process. Most useful ones are:

- `checkDataset`: given a variable returns whether its class is `data.frame`.
- `checkDatasetClass`: checks whether a certain class variable is present in a `data.frame`
- `mapArguments`: used inside `doInstSelection.unbalanced` for example, it matches a list of arguments (present in `...` argument passed to the corresponding wrapper) against `args.CNN` or `args.ENN`, in the case of `unbalanced` package, maps its names to the original names the origin package needs and substitutes default arguments.
- `mapMethod`: given a packages list and a name of method, returns the function to evaluate (that is, returns `pkg::method` function, that can be evaluated with `do.call` and a list of arguments).