mlrCPO Internals

This file is written in markdown and should be found in the `info` directory; a compiled `.pdf` version is also supplied in the same directory.

The following describes the internal design of `mlrCPO`. Package names, file names, and object names are in monospace: `Classname`; functions are monospace with parentheses: `fun()`; exported functions are followed by an asterisk: `exportedFun()*`; list slots are monospace, prepended with a dollar sign: `$slot`.

Overview

`mlrCPO` builds on the `mlr` package and adds flexible preprocessing operator objects. Please make sure you are familiar with the user interface, by reading the vignette, the R help pages, and possibly going through the tutorial.

Coding Style

To fit in with the rest of *mlr-org*, it follows the same code style guide. A subset of this style is checked by lintr automatically during tests. To run lintr during tests, the github version needs to be installed. Use the `quicklint` tool in the `tools` directory to run lint on only the files that have changed with respect to the `master` branch.

Class Structure

The central object of `mlrCPO` is the `CPO` with the following lifecycle and subclasses:

```
CPOConstructor ---------> CPO ---------------> CPOTrained
                         /   \                   /   \
                        /     \                 /     \
             CPOPrimitive   CPOPipeline  CPORetrafo  CPOInverter
```

The `CPOConstructor` is a function that is called to create a `CPO`, examples are `cpoPca` and `cpoScale`. `CPO` is the object representing a specific operation, completely with hyperparameters. `CPOTrained` represents the "retafo" operation that can be retrieved with `retrafo()*` or `inverter()*` from a preprocessed data object, or from a trained model.

```
CPO -------------------> CPOLearner -------------------> CPOModel
```

When a `CPO` gets attached to an `mlr Learner`, a `CPOLearner` object is created. The trained model of this learner has the class `CPOModel`.

CPOConstructor

`CPOConstructor` is created in `makeCPOGeneral()` which is called by `makeCPO()*` or `makeCPOExtended()*`, or similar functions, all defined in `makeCPO.R`. A `CPOConstructor` is an R *function* that takes all the `CPO`'s arguments, as well as the `affect.*`, `export`, and `id` special arguments and assembles a `CPO` object. The body of each `CPOConstructor` is the same and can be found in `makeCPO.R`, starting at

```
funbody = quote({
```

CPO

A `CPO` can either be "primitive" or "compound".

The primitive `CPO` has the additional class `CPOPrimitive` and is at the heart of `mlrCPO` functionality. It is defined and documented in `makeCPO.R` starting at

```
cpo = makeS3Obj(c("CPOPrimitive", "CPO"),
```

Besides much meta-information, the primitive `CPO` stores the parameter set as `$par.set` and `$unexported.par.set`, as well as the parameter values as `$par.vals` and `$unexported.pars`. The trafo and retrafo operations are functions stored in the `$trafo` and `$retrafo` slots.

A compound `CPO` can be created from two `CPO`s by composing them using the `%>>%` operator, which calls `composeCPO()*` (which can also be called directly). It has the additional class `CPOPipeline`, and is defined in `composeCPO()*`. Compound `CPO`s have a tree structure: Each compound `CPO` has a slot `$first` and `$second`, referencing two child `CPO`s (which may be compound *or* primitive) in the order in which they are applied. Otherwise `CPOPipeline` objects are relatively lightweight, they store meta-information computed from the child objects (e.g. a name referencing both children, and common properties), and parameter values.

When the hyperparameter of a `CPOPipeline` are changed, the child nodes are not modified; instead, the changed parameter values are stored in the root node. The parameter values of the child nodes are only updated when the `CPO` is actually called.

CPOTrained

`CPOTrained` objects are created in `makeCPOTainedBasic`, which is called by `makeCPORetrafo` and `makeCPOInverter`, which are both called by `callCPO.CPOPrimitive`. `CPOTrained` objects are thus created whenever data is fed into a `CPO`, be it by using `%>>%`, by calling `applyCPO()*` directly, or by training a `CPOLearner`. A `CPOTrained` object is relatively lean; it mostly points to the `CPOPrimitive` object that helped create it, contains a `$state` slot which contains the control object or retrafo function created by the `$trafo` call, and "shapeinfo" about the data shape (feature names and types) used when calling the trafo.

A `CPOTrained` can be a "retrafo", an "inverter", or both. A retrafo has the `CPORetrafo` class and is used to re-apply a transformation that was "trained" on a dataset. An inverter has the `CPOInverter` class, it concerns only "Target Operating `CPO`s". It is created whenever a target operating `CPO` is applied to a dataset and gives the user the possibility of inverting the prediction performed with the transformed dataset. If the target operating `CPO` is "stateless", the resulting inverter can be used on *any* prediction, if it is not "stateless", the inverter can only be used on predictions made with the resulting transformed dataset. Since the inverters resulting from "stateless" target operating transformations can be used on any new data, they are retrieved with the `retrafo()*` call, they are so called "hybrids" (i.e. both retrafo and inverter). The `CPOInverter` specific to the prediction of an individual processed dataset is retrieved using `inverter()*`.

`CPOTrained` objects are linked lists, linked by the `$prev.retrafo` slot. The `callCPO()` call generates both `CPORetrafo` and `CPOInverter` linked lists; they are stored as attributes of the resulting data by `applyCPO.CPO()*`. `retrafio()*` and `inverter()*` are both relatively dumb functions which retrieve these object attributes.

`CPOLearner`

The `CPOLeaner` is created using `mlr`'s `makeBaseWrapper()*` functionality, in `attachCPO()*`. Whenever another `CPO` is attached to a `CPOLearner`, the Learner is not wrapped again, instead the attached `CPO` is extended. The `CPOLearner` also has properties and hyperparameters that are extended / modified according to the `CPO`. Whenever the hyperparameters of a `CPOLearner` are changed, the attached `CPO` (and its `$par.vals` slot) is not modified; instead, the `CPO` is modified upon invocation of `trainLearner.CPOLearner()*`.

When the `train()` method is called with a `CPOLearner`, the resulting model has two `CPOTrained` attached: the `CPORetrafo` and the `CPOInverter` chains. These are used on newly arriving data, and on the resulting prediction, respectively.

`NULLCPO`

The `NULLCPO` object has a special place in `mlrCPO`: It is the neutral element of the `%>>%` operator and stands for an "empty" cpo. All functions pertaining to it are in `NULLCPO.R`.

File Overview

As of writing of this document, there are 21 `.R` files in the `R` directory that make up the `mlrCPO` package. They are listed here, in approximate order of dependence, and with a short description. The most important files are described in more detail in Functionality.

Core Files

These files are the core of `CPO` inner workings.

File Name	Description
`makeCPO.R`	`makeCPO()*` and related functions, for definition of `CPOConstructors`
`callCPO.R`	Invocation of `CPO` trafo and retrafo functions, and creation of `CPOTrained`
`FormatCheck.R`	Checking of input and output data conformity to `CPO` "properties", and uniformity of data between trafo and retrafo
`CPO_operators.R`	Composition and splitting of `CPO` and `CPOTrained` objects, as well as getters and setters
`CPOLearner.R`	Everything `CPOLearner`-related: Attachment of `CPO` to `Learner`, training and prediction
`RetrafoState.R`	Retrieval of the retrafo state, and re-creation of a `CPORetrafo` from it
`inverter.R`	Framework for of `CPO` inverter functionality

Auxiliary Files

These files give auxiliary functions and boilerplate.

File Name	Description
`doublecaret.R`	`%>>%`-operator
`attributes.R`	`retrafo()*` and `inverter()*` functions that access object attributes
`print.R`	Printing of `CPO` objects
`parameters.R`	Auxiliary functions that check parameter feasibility and overlap
`generics.R`	Definition of generic functions and their `.default` implementations.
`NULLCPO.R`	`NULLCPO` object and all related functions
`listCPO.R`	Listing of present `CPO`s
`CPOAuxiliary.R`	Helper functions

File Name	Description
zzz.R	Package initialization and import of external packages

CPO Definition Files

These files contain concrete CPO implementations.

File Name	Description
CPO_meta.R	cpoMultiplex and cpoCase
CPOCbind.R	cpoCbind and its special printing function
CPO_concrete.R	General data manipulation CPOs
CPO_filterFeatures.R	Feature filter CPOs
CPO_impute.R	Imputation CPOs

Functionality

CPO Creation (makeCPO.R)

CPO creation happens in makeCPO.R. Actual creation happens in makeCPOGeneral(), which gets called with different values depending on which makeCPOXXX()* is called by the user. Besides general checking of parameter validity, the $trafo and $retrafo functions are created. If they are given as special NSE blocks (just curly braces without function headers), makeFunction creates the necessary function headers, otherwise the given headers are checked.

The .dataformat and .dataformat.factor.with.ordered variables are internally put together into one $datasplit slot of CPO. For this, the "split" value of .dataformat is translated to either "most" (.dataformat.factor.with.ordered == TRUE) or "all", and the "factor" value is translated to "onlyfactor" if .dataformat.factor.with.ordered is FALSE.

All .trafo.types are handled as one would expect, with the exception of the "trafo.returns.control" trafo type. It is emulated by wrapping the user provided functions into a "trafo.returns.data" conforming stub.

CPO Invocation (callCPO.R)

Invocation of CPO happens recursively by the callCPO() function: If called with a CPOPipeline, the given data is first transformed by the $first, then by the $second slot (which in turn may be CPOPipeline objects). If called with a CPOPrimitive, the necessary data and property checks (See Format Check) are performed and the $trafo slot is called, and the CPORetrafo and CPOInverter chains are constructed. Both of them are linked lists; therefore, `callCPO()` takes a `prev.retrafo` and a `prev.inverter` argument, to which the newly created retrafo and inverter objects are prepended.

Invocation of a `CPOTrained` happens in `callCPOTrained()`. It respects the linked-list nature of `CPOTrained` and recursively invocates possible `$prev.inverter` objects associated with the `CPOTrained`. For target operating `CPO`s, it goes on to call `callCPO()`, otherwise it works similarly to `callCPO()`, except calling the `$retrafo` slot of the associated `CPOPrimitive`, and only constructing a `CPOInverter` chain.

Format Check (`FormatCheck.R`)

`FormatCheck.R` is a central part of `CPO` that does checking of input and output data for property adherence, checking that input data to retrafo conforms to input data to the corresponding trafo invocation, and conversion of data depending on `.dataformat` values for a given CPO.

Data preparation and post-processing is done by the functions `prepareTrafoInput()`, `prepareRetrafoInput()`, `handleTrafoOutput()`, and `handleRetrafoOutput()`. Each of these takes the data, desired properties, and possibly shape-info (data feature column type info), checks the data validity, and returns the converted data according to the `.dataformat`.

`FormatCheck.R` also hosts the important `compositeProperties()` function which calculates the properties of a compound `CPO` given the properties of its constituents, after checking if these properties are compatible at all.

CPO Operators (`CPO_operators.R`)

`CPO` and `CPOTrained` composition and splitting mostly happens as they should happen for functional data structures (trees and linked lists, respectively). Composition also checks properties and computes certain aggregate metavalues (e.g. of properties or parameter values) that are stored at the root of the tree or first element of the linked list, respectively.

`CPO` getters and setters are mostly dumb accessors, with the exception of `setCPOId()*`, which actually changes the `$par.set` and `$par.vals` slots to respect the name changes of the parameters.

`CPO` Learner Attachment (`CPOLearner.R`)

`CPOLearner`s are created using the `makeBaseWrapper()` `mlr` functionality. To prevent deep nesting of learners, if a `CPO` is attached to a `CPOLearner`, the `Learner` is *not* wrapped another time, instead the already attached `CPO` is extended by the new CPO. A tricky part of extending a Learner with a CPO is the calculation of resulting learner properties, which is performed by compositeCPOLearnerProps(). The predict type of a CPOLearner respects the $predict.type slot of the attached CPO(s); this translation is done by setPredictType.CPOLearner().

Training of learners done using the makeChainModel() mlr function, which contains the child learner's model and also has the CPORetrafo created by application of the CPO. Prediction then only needs to apply this retrafo to the new data, run the model, and apply the inverter created by application of the retrafo.

Retrafo State (RetrafoState.R)

The "retrafo state" is mostly the $state slot of the CPORetrafo object, with additional information about the data in the $shapeinfo.input and $shapeinfo.output slots. For functional CPOs, the retrafo function's environment is turned into a list (with the $cpo.retrafo pointing to that function), for object based CPOs, the $state object is saved to the retrafo state's $control slot, next to the hyperparameter values. makeRetrafoFromState()* creates a bare CPO object from the given constructor and puts in the state information.

Inverter functionality (inverter.R)

invert()*'s main task is to call invertCPO() which calls the $retrafo slot of a CPO, but a major challenge is to convert between prediction formats. Predictions not always carry meta-information about the task type they were generated for, and they are not always in a uniform format (varying between being a vector, a matrix, a data.frame, etc.). Some helper functions take over the tasks of prediction format validation and normalization.

The %>>%-Operator (doublecaret.R)

The %>>% operator is syntactic sugar for the applyCPO, composeCPO, and attachCPO functions defined in callCPO.R, CPO_operators.R, and attachCPO.R.

retrafo()* and inverter()* (attributes.R)

Both retrafo()* and inverter()* are very lightweight functions that only access the respective attributes of a data object. If the data object has no such attribute, a NULLCPO is returned, instead of NULL, so that y %>>% retrafo(x) works even when no retrafo is present. An exception is made for CPOModel: It retrieves the `CPORetrafo` generated while training a `CPOLearner`; the generic is found in `CPOLearner.R`.

NULLCPO (`NULLCPO.R`)

The `NULLCPO` object is implemented by implementing all generics for it, and have them do the respective no-op.

`CPO` listing (`listCPO.R`)

A `CPO` is registered in a global variable `CPOLIST` by calling `registerCPO` with the respective descriptive items. To have definition and documentation relatively close, `registerCPO` should be called right after the definition of an internal `CPO`. The `listCPO` function then only creates a `data.frame` from this list.