concrete-vignette

David Chen

2022-08-23

```
library(concrete)
library(data.table)
```

formatArguments()

DataTable

DataTable should be a data.frame or data.table, each row containing one subject's information (longitudinal data not supported). DataTable must include the following columns:

- EventTime: the time to observed failure (or censoring) event.
- EventType: the type of event, encoded as integers with 0 being reserved for right-censoring.
- Treatment: the intervention variable - binary is fully supported while numeric and multinomial interventions may be supported in the future.

DataTable may include an 'ID' column (potentially for stratified cross-validation, etc) and any number of column containing baseline covariates.

```
obs <- as.data.table(survival::pbc)
set.seed(0)
obs[, trt := sample(0:1, length(trt), replace = TRUE)]
obs[, stage := sample(1:4, length(stage), replace = TRUE)]
obs <- obs[, c("id", "time", "status", "trt", "age", "albumin", "sex", "stage")]
head(obs)
```

```
##    id time status trt      age albumin sex stage
## 1:  1  400      2   1 58.76523    2.60   f     1
## 2:  2 4500      0   0 56.44627    4.14   f     1
## 3:  3 1012      2   1 70.07255    3.48   m     3
## 4:  4 1925      2   0 54.74059    2.54   f     2
## 5:  5 1504      1   0 38.10541    3.53   f     2
## 6:  6 2503      2   1 66.25873    3.98   f     4
```

The EventTime column must be positive numeric (e.g. obs$time), the EventType column must be non-negative integers (e.g. obs$status), and the Treatment column must be binary (support for multinomial and continuous treatment variables is in the testing stage).

This data is passed into concrete as:

```
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id")
```

Intervention

The desired intervention is specified by a pair of functions: an 'intervention' function which outputs desired treatment *assignments* and a 'g.star' function which outputs desired treatment *probabilities*. In the simple case of a 2-armed trial where the desired treatment assignments are either to assign everyone the treatment (i.e. trt = 1) or to assign everyone to a control (i.e. trt = 0), the functions are available as `concrete::ITT`. ITT is a list of two desired counterfactual interventions: "A==1" details an the intervention where everyone is assigned treatment, and "A==0" details an intervention where everyone is assigned control.

```
ITT <- makeITT()
str(ITT, give.attr = FALSE)
```

```
## List of 2
##  $ A==1:List of 2
##   ..$ intervention:function (ObservedTreatment, Covariates)
##   ..$ g.star      :function (Treatment, Covariates)
##  $ A==0:List of 2
##   ..$ intervention:function (ObservedTreatment, Covariates)
##   ..$ g.star      :function (Treatment, Covariates)
```

The intervention function takes as inputs the vector of observed treatment assignments and data.table of covariates, and outputs a vector of desired treatment assignments. For "A==1" the intervention function returns a vector of 1s.

```
ITT[["A==1"]]$intervention
```

```
## function(ObservedTreatment, Covariates) {
##         IntervenedAssignment <- rep_len(1, length(ObservedTreatment))
##         return(IntervenedAssignment)
##     }
## <bytecode: 0x56214f9b5c30>
## <environment: 0x56214f9c40c8>
```

The 'g.star' function takes as inputs the vector of treatment assignments and data.table of covariates, and outputs a vector of desired treatment probabilities for the provided vector of treatment assignments. In "A==1", the desired intervention is to assign everyone to treatment (i.e. trt = 1) with 100% probability and to control with 0% probability and the corresponding g.star function reflects this, returning 1 if the treatment assignment is 1 and 0 if the treatment assignment is 0.

```
ITT[["A==1"]]$g.star
```

```
## function(Treatment, Covariates) {
##         IntervenedProbability <- as.numeric(Treatment == 1)
##         return(IntervenedProbability)
##     }
## <bytecode: 0x56214f9b6fb8>
## <environment: 0x56214f9c40c8>
```

For "A==0" the intervention function returns a vector of 0s and the treatment assignment probabilities are flipped so that a treatment assignment of 0 is given 100% probability while treatment assignments of 1 are given 0% probability.

```
ITT[["A==0"]]
```

```
## $intervention
## function(ObservedTreatment, Covariates) {
##         IntervenedAssignment <- rep_len(0, length(ObservedTreatment))
##         return(IntervenedAssignment)
##     }
```

```
## <bytecode: 0x56214f9be090>
## <environment: 0x56214f9c40c8>
##
## $g.star
## function(Treatment, Covariates) {
##         IntervenedProbability <- as.numeric(Treatment == 0)
##         return(IntervenedProbability)
##     }
## <bytecode: 0x56214f9bf418>
## <environment: 0x56214f9c40c8>
```

This method of specifying treatment assignments allows for more flexible dynamic and stochastic treatment rules.

```
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT)
```

Targets

The continuous-time TMLE for survival outcomes implemented in concrete estimates treatment specific risk for targeted events at targeted times.

Target Event(s) For instance, in the `pbc` dataset, there are 3 possible values of "status": 0 for censored, 1 for transplant, and 2 for death. In concrete 0 is similarly reserved to indicate the presence of censoring, while failure events can be encoded as any positive integer. If we are interested in looking at the risk of transplant and death jointly in this pbc dataset, this can be specified as:

```
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2)
```

If no input is provided for TargetEvent, the formatArguments will use target all observed failure events by default.

Target Time(s) Target times should be restricted to the time range in which failure events are observed, since estimating event risks after the point in time where all individuals are censored is purely extrapolation. To discourage this behaviour, formatArguments() will return an error if target time is after the last observed failure event time. If no TargetTime is probided, then `concrete` will target the last observed event time.

```
ExtrapolationTime <- unique(obs[status > 0, max(time)]) + 1
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = ExtrapolationTime)
```

```
## Error in concrete:::getTargetTime(TargetTime = unique(obs[status > 0,  :
##   TargetTime must not target times after which all individuals are Censored, 4191
```

```
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500)
```

```
## Categorical covariates detected: DataTable will be 1-hot encoded.
```

```
## Model input missing. An example template will be returned but should be amended to suit your applica
```

CVArg

`concrete` uses `origami` to specify cross-validation folds, specifically the function `origami::make_folds()`. If no input is provided to the formatArguments(CVArg=) argument, concrete will use origami to implement a simple 10-fold cross-validation scheme. For how to specify more sophisticated cross-validation schemes, see this brief vignette or this detailed chapter on using origami from the tlverse handbook.

```
library(origami)
# If the CVArg argument is NULL, concrete uses a simple 10-fold CV as the default specification, i.e.
CVArgs <- list(n = ncol(obs), fold_fun = folds_vfold, cluster_ids = NULL, strata_ids = NULL)

ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = NULL)

ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = CVArgs)

# For different number of folds, simply add the `V = ` argument, e.g.
CV5Fold <- list(n = ncol(obs), V = 5L, fold_fun = folds_vfold, cluster_ids = NULL, strata_ids = NULL)
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = CV5Fold)
```

Model

TMLE requires initial estimation of some parts of the observed data distribution; for continuous-time TMLE of survival and absolute risks, we require estimates of the treatment propensity score and conditional hazards for each event and censoring type. The formatArguments(Model =) argument is how `concrete` accepts model specifications for estimating those parameters. Inputs into the Model argument must be named lists with one entry for the 'Treatment' variable, and for each of the event type (and censoring). The list element corresponding to the 'Treatment' variable must be named as the variable name, and the list elements corresponding to each event type must be named as the numeric value of the event type (with "0" being reserved for censoring)

```
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = NULL, Model = NULL)
str(ConcreteArgs[["Model"]], give.attr = FALSE)
```

```
## List of 4
##  $ trt: chr "SL.glmnet"
##  $ 0  :List of 1
##   ..$ model1:Class 'formula'  language Surv(time, status == 0) ~ .
##  $ 1  :List of 1
##   ..$ model1:Class 'formula'  language Surv(time, status == 1) ~ .
##  $ 2  :List of 1
##   ..$ model1:Class 'formula'  language Surv(time, status == 2) ~ .
```

Treatment Models Propensity scores for treatment assignment are estimated using the Superlearner stacked ensemble machine learning algorithm, using either the *SuperLearner* package (PropScoreBackend =

"Superlearner") or the *sl3* package (PropScoreBackend = "sl3"). Detailed instructions for how to specify models using *SuperLearner* can be found in the package vignette or ?SuperLearner::SuperLearner documentation. If using formatArguments(PropScoreBackend = "SuperLearner), `concrete` passes the 'Model' specification for the 'Treatment' variable into SuperLearner(SL.library =). Below we demonstrate some examples of how to specify treatment models using the"SuperLearner" backend.

```
library(SuperLearner)

# use Superlearner::listWrappers() to show the available models. For additional models see https://gith

# simple example
SLModel <- c("SL.glmnet", "SL.bayesglm", "SL.xgboost", "SL.polymars")
# example with screening
SLModel <- list(c("SL.ranger", "screen.corRank"), c("SL.glmnet", "All", "screen.randomForest"),
                c("SL.bayesglm", "screen.glmnet"), "SL.polymars")

ConcreteArgs[["Model"]][["trt"]] <- SLModel
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = NULL, Model = ConcreteArgs[["Model"]],
                                PropScoreBackend = "SuperLearner")
```

To use the *sl3* package as the backend for estimating treatment propensity score, Chapter 6 in the tlverse handbook provides an in depth explanation for how to specify the desired library of models. Below we show a simple example of specifying a set of models and then passing them into `concrete`.

```
library(sl3)
# use sl3::sl3_list_learners() to show the available models. Use sl3_list_learners(properties = ) to li
sl3glmnet <- Lrnr_glmnet$new()
sl3hal <- Lrnr_hal9001$new()
sl3dbarts <- Lrnr_dbarts$new()

sl3Model <- Stack$new(sl3glmnet, sl3hal, sl3dbarts)
ConcreteArgs[["Model"]][["trt"]] <- sl3Model

ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = NULL, Model = ConcreteArgs[["Model"]],
                                PropScoreBackend = "sl3")
```

Models for Event and Censoring Hazards For estimating the necessary conditional hazards, `concrete` currently relies on Cox models implemented by *survival::coxph*. A library of Cox models can be used, which are used for a discrete Superlearner selector based on cross-validated pseudo-likelihood loss. Examples of how to specify models for estimating conditional hazards with `concrete` are shown below

```
ConcreteArgs[["Model"]][["0"]] <- list("model1" = Surv(time, status == 0) ~ trt + age:sex,
                                       "model2" = Surv(time, status == 0) ~ .)
ConcreteArgs[["Model"]][["1"]] <- list(Surv(time, status == 1) ~ .,
                                       ~ trt + age)
ConcreteArgs[["Model"]][["2"]] <- "."
```

```
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
```

```r
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = NULL, Model = ConcreteArgs[["Model"]],
                                PropScoreBackend = "SuperLearner", HazEstBackend = "coxph")
```

```
## Categorical covariates detected: DataTable will be 1-hot encoded.
## Superlearner model specifications are not checked here. The input must be a valid argument into the
## Cox model specifications have been renamed where necessary to reflect changed covariate names. Model
## The left hand side of the cox formula for Model[[1]][[2]] has been corrected to Surv(time, status ==
## Cox model specifications have been renamed where necessary to reflect changed covariate names. Model
## The left hand side of the cox formula for Model[[2]][[1]] has been corrected to Surv(time, status ==
## Cox model specifications have been renamed where necessary to reflect changed covariate names. Model
```

TMLE Update Parameters

MaxUpdateIter is an integer that controls the maximum number of small steps along the universal least favorable path for one-step tmle. OneStepEps is a positive number that controls the size of the small steps for one-step tmle, which is shrunk by factors of 2 whenever a step would increase the norm of the efficient influence function. MinNuisance is a positive number less than 1 that determines the lower bounding the nuisance parameters, essentially decreasing variance at the cost of introducing bias. Recommend to keep this value small, but even better would be to ask questions about regimes that are better supported in data.

```r
ConcreteArgs <- formatArguments(DataTable = obs, EventTime = "time", EventType = "status",
                                Treatment = "trt", ID = "id",
                                Intervention = ITT, TargetEvent = 1:2, TargetTime = (3:7)*500,
                                CVArg = NULL, Model = ConcreteArgs[["Model"]],
                                PropScoreBackend = "SuperLearner", HazEstBackend = "coxph",
                                MaxUpdateIter = 100, OneStepEps = 1, MinNuisance = 0.05)
```

```
## Categorical covariates detected: DataTable will be 1-hot encoded.
## Superlearner model specifications are not checked here. The input must be a valid argument into the
```

doConcrete

getOutput