Adding Custom Sampling Functions

Jenna Reps

2021-12-16

Contents

1 Introduction

This vignette describes how you can add your own custom function for sampling the target population in the Observational Health Data Sciencs and Informatics (OHDSI) `PatientLevelPrediction` package. This vignette assumes you have read and are comfortable with building single patient level prediction models as described in the `BuildingPredictiveModels` vignette.

We invite you to share your new sample functions with the OHDSI community through our GitHub repository.

2 Sample Function Code Structure

To make a sampling function that can be used within PatientLevelPrediction you need to write two different functions. The 'create' function and the 'implement' function.

The 'create' function, e.g., create<SampleFunctionName>, takes the parameters of the sample 'implement' function as input, checks these are valid and outputs these as a list of class 'sampleSettings' with the 'fun' attribute specifying the 'implement' function to call.

The 'implement' function, e.g., implement<SampleFunctionName>, must take as input: * trainData - a list containing: - covariateData: the plpData$covariateData restricted to the training patients - labels: a data frame that contain rowId (patient identifier) and outcomeCount (the class labels) - folds: a data.frame that contains rowId (patient identifier) and index (the cross validation fold) * sampleSettings - the output of your create<SampleFunctionName>

The 'implement' function can then do any manipulation of the trainData (such as undersampling or oversampling) but must output a trainData object containing the covariateData, labels and folds for the new training data sample.

3 Example

Let's consider the situation where we wish to take a random sample of the training data population. To make this custom sampling function we need to write the 'create' and 'implement' R functions.

3.1 Create function

Our random sampling function will randomly sample `n` patients from the trainData. Therefore, the inputs for this are: * `n` an integer/double specifying the number of patients to sample * `sampleSeed` an integer/double specifying the seed for reproducibility

```r
createRandomSampleSettings <- function(
                     n = 10000,
                     sampleSeed = sample(10000,1)
                     ){

  # add input checks
  checkIsClass(n, c('numeric','integer'))
  checkHigher(n,0)
  checkIsClass(sampleSeed, c('numeric','integer'))

  # create list of inputs to implement function
  sampleSettings <- list(
    n = n,
    sampleSeed  = sampleSeed
    )

  # specify the function that will implement the sampling
  attr(sampleSettings, "fun") <- "implementRandomSampleSettings"

  # make sure the object returned is of class "sampleSettings"
  class(sampleSettings) <- "sampleSettings"
  return(sampleSettings)

}
```

We now need to create the 'implement' function `implementRandomSampleSettings()`

3.2 Implement function

All 'implement' functions must take as input the trainData and the sampleSettings (this is the output of the 'create' function). They must return a trainData object containing the covariateData, labels and folds.

In our example, the `createRandomSampleSettings()` will return a list with 'n' and 'sampleSeed'. The sampleSettings therefore contains these.

```r
implementRandomSampleSettings <- function(trainData, sampleSettings){

  n <- sampleSetting$n
  sampleSeed <- sampleSetting$sampleSeed

  if(n > nrow(trainData$labels)){
    stop('Sample n bigger than training population')
  }

  # set the seed for the randomization
  set.seed(sampleSeed)

  # now implement the code to do your desired sampling

  sampleRowIds <- sample(trainData$labels$rowId, n)

  sampleTrainData <- list()

  sampleTrainData$labels <- trainData$labels %>%
    dplyr::filter(.data$rowId %in% sampleRowIds) %>%
    dplyr::collect()

  sampleTrainData$folds <- trainData$folds %>%
    dplyr::filter(.data$rowId %in% sampleRowIds) %>%
    dplyr::collect()

  sampleTrainData$covariateData <- Andromeda::andromeda()
  sampleTrainData$covariateData$covariateRef <-trainData$covariateData$covariateRef
  sampleTrainData$covariateData$covariates <- trainData$covariateData$covariates %>% dplyr::filter(.data

  #update metaData$populationSize
  metaData <- attr(trainData$covariateData, 'metaData')
  metaData$populationSize = n
  attr(sampleTrainData$covariateData, 'metaData') <- metaData

  # make the cocvariateData the correct class
  class(sampleTrainData$covariateData) <- 'CovariateData'

  # return the updated trainData
  return(sampleTrainData)
}
```

4 Acknowledgments

Considerable work has been dedicated to provide the `PatientLevelPrediction` package.

```
citation("PatientLevelPrediction")
```

```
##
## To cite PatientLevelPrediction in publications use:
##
## Reps JM, Schuemie MJ, Suchard MA, Ryan PB, Rijnbeek P (2018). "Design and implementation of a
## standardized framework to generate and evaluate patient-level prediction models using
## observational healthcare data." _Journal of the American Medical Informatics Association_,
## *25*(8), 969-975. <URL: https://doi.org/10.1093/jamia/ocy032>.
##
## A BibTeX entry for LaTeX users is
##
##   @Article{,
##     author = {J. M. Reps and M. J. Schuemie and M. A. Suchard and P. B. Ryan and P. Rijnbeek},
##     title = {Design and implementation of a standardized framework to generate and evaluate patient-
##     journal = {Journal of the American Medical Informatics Association},
##     volume = {25},
##     number = {8},
##     pages = {969-975},
##     year = {2018},
##     url = {https://doi.org/10.1093/jamia/ocy032},
##   }
```

Please reference this paper if you use the PLP Package in your work:

Reps JM, Schuemie MJ, Suchard MA, Ryan PB, Rijnbeek PR. Design and implementation of a standardized framework to generate and evaluate patient-level prediction models using observational healthcare data. J Am Med Inform Assoc. 2018;25(8):969-975.

This work is supported in part through the National Science Foundation grant IIS 1251151.