Adding Custom Patient-Level Prediction Algorithms

Jenna Reps, Martijn J. Schuemie, Patrick B. Ryan, Peter R. Rijnbeek

2018-10-04

Contents

1 Introduction

This vignette describes how you can add your own custom algorithms in the Observational Health Data Sciencs and Informatics (OHDSI) `PatientLevelPrediction` package. This allows you to fully leverage the OHDSI PatientLevelPrediction framework for model development and validation. 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 algorithms with the OHDSI community through our GitHub repository.

2 Algorithm Code Structure

Each algorithm in the package should be implemented in its own <Name>.R file, e.g. KNN.R, containing a set<Name> function and a fit<Name> function. Furthermore, a corresponding predict function in predict.R is needed (if there isn't one available that would work, see example at the end of the document). We will now describe each of these functions in more detail below.

2.1 Set

The set<Name> is a function that takes as input the different hyper-parameter values to do a grid search when training. The output of the functions needs to be a list as class `modelSettings` containing:

- param - all the combinations of the hyper-parameter values input
- model - a string specifying what function to call to fit the model
- name - a string containing the name of the model.

For example, if you were adding a model called madeUp that has two hyper-parameters then the set function should be:

```
setMadeUp <- function(a=1, b=2, seed=NULL){
  # add input checks here...

  # now create list of all combinations:
  result <- list(model='fitMadeUp', # this will be called to train the made up model
                 param= split(expand.grid(a=a,
                                          b=b,
                                          seed=ifelse(is.null(seed),'NULL', seed)),
                              1:(length(a)*length(b)  )),
                 name='Made Up Algorithm'
  )
  class(result) <- 'modelSettings'

  return(result)
}
```

2.2 Fit

This function should train your custom model for each parameter entry, pick the best parameters and train a final model for that setting.

The fit<Model> should have as inputs:

- population - the study popualation the model is being developed on
- plpData - the plpData object
- param - the hyper-parameters as a list of all combinations
- quiet - T or F indicating whether to output progress
- outcomeId - the outcome id
- cohortId - the target population id

The fit function should return a list of class `plpModel` with the following objects:

- model - a trained model
- modelSettings - a list containing the model and input param
- trainCVAac - a value with the train AUC value
- hyperParamSearch - a dataframe with the hyperparameter grid and corresponding AUCs
- metaData - the metaData from the plpData object
- populationSettings - the settings used to create the population and define the time-at-risk
- outcomeId - the outcomeId being predicted
- cohortId - the cohortId corresponding to the target cohort
- varImp - a dataframe with the covaraites and a measure of importance
- trainingTime - how long it took to develop/evaluate the model
- covariateMap - if the plpData are converted to a matrix for model compatibility this tells us what covariate each row in the matrix correpsonds to and is need when implementing the model on new data

The plpModle returned by fit also has a type attribute, this points to the predict function, for example `attr(result, 'type') <- 'madeup'` means when the model is applied to new data, the 'predict.madeup' function in Predict.R is called. if this doesnt exist, then the model will fail. Another attribute is the predictionType `attr(result, 'predictionType') <- 'binary'` this is currently not needed but may be important in the future when we expand to regression or multiclass classification.

For example:

```
fitMadeUp <- function(population, plpData, param, quiet=F,
                      outcomeId, cohortId, ...){

  # **************** code to train the model here
  # trainedModel <- this code should apply each hyper-parameter using the cross validation
  #                 then pick out the best hyper-parameter setting
  #                 and finally fit a model on the whole train data using the
  #                 optimal hyper-parameter settings
  # ****************

  # construct the standard output for a model:
  result <- list(model = trainedModel,
                 modelSettings = list(model='made_up', modelParameters=param),
                 trainCVAuc = NULL,
                 hyperParamSearch = hyperSummary,
                 metaData = plpData$metaData,
                 populationSettings = attr(population, 'metaData'),
                 outcomeId=outcomeId,# can use populationSettings$outcomeId?
                 cohortId=cohortId,
                 varImp = NULL,
                 trainingTime=comp,
                 covariateMap=result$map
  )
  class(result) <- 'plpModel'
  attr(result, 'type') <- 'madeup'
  attr(result, 'predictionType') <- 'binary'
  return(result)

}
```

You could make the fitMadeUp function cleaner by adding helper function in the MadeUp.R file that are called by the fit function. As the end of the fit function specified `attr(result, 'type') <- 'madeup'` we also need to make sure there is a `predict.madeup` function in Predict.R:

2.3 Predict

The prediction function takes as input the plpModel returned by fit, a population and corresponding plpData. It returns a data.frame with the columns:

- rowId - the id for each person in the population
- value - the predicted risk from the plpModel

If the population contains the columns outcomeCount and indexes, then these are also in the output.

For example:

```
predict.madeup <- function(plpModel,population, plpData, ...){

  # ************* code to do prediction for each rowId in population
  # prediction <- code to do prediction here returning columns: rowId
  #               and value (predicted risk)
  #***************

  prediction <- merge(population, prediction, by='rowId')
  prediction <- prediction[,colnames(prediction)%in%c('rowId','outcomeCount',
```

```
                                                    'indexes', 'value')]
  attr(prediction, "metaData") <- list(predictionType = "binary")
  return(prediction)

}
```

3 Algorithm Example

Below a fully functional algorithm example is given, however we highly recommend you to have a look at the available algorithms in the package.

3.1 Set

```
setMadeUp <- function(a=1, b=2, seed=NULL){
  # check a is valid positive value
  if(missing(a)){
    stop('a must be input')
  }
  if(!class(a)%in%c('numeric','integer')){
    stop('a must be numeric')
  }
  if(a < 0){
    stop('a must be positive')
  }
  # check b is numeric
  if(missing(b)){
    stop('b must be input')
  }
  if(!class(b)%in%c('numeric','integer')){
    stop('b must be numeric')
  }

  # now create list of all combinations:
  result <- list(model='fitMadeUp',
                 param= split(expand.grid(a=a,
                                          b=b,
                                          seed=ifelse(is.null(seed),'NULL', seed)),
                              1:(length(a)*length(b)  )),
                 name='Made Up Algorithm'
  )
  class(result) <- 'modelSettings'

  return(result)


}
```

3.2 Fit

```
fitMadeUp <- function(population, plpData, param, quiet=F,
                      outcomeId, cohortId, ...){
    if(!quiet)
    writeLines('Training Made Up model')

  if(param[[1]]$seed!='NULL')
    set.seed(param[[1]]$seed)

    # check plpData is coo format:
  if(!'ffdf'%in%class(plpData$covariates) )
    stop('This algorithm requires plpData in coo format')

  metaData <- attr(population, 'metaData')
  if(!is.null(population$indexes))
    population <- population[population$indexes>0,]
  attr(population, 'metaData') <- metaData

  # convert data into sparse R Matrix:
  result <- toSparseM(plpData,population,map=NULL)
  data <- result$data

  data <- data[population$rowId,]

  # set test/train sets (for printing performance as it trains)
  if(!quiet)
    writeLines(paste0('Training made up model on train set containing ', nrow(population),
                      ' people with ',sum(population$outcomeCount>0), ' outcomes'))
  start <- Sys.time()

  #============== STEP 1 ======================================
  # pick the best hyper-params and then do final training on all data...
  writeLines('train')
  datas <- list(population=population, data=data)
  param.sel <- lapply(param, function(x) do.call(made_up_model, c(x,datas)  ))
  hyperSummary <- do.call(rbind, lapply(param.sel, function(x) x$hyperSum))
  hyperSummary <- as.data.frame(hyperSummary)
  hyperSummary$auc <- unlist(lapply(param.sel, function(x) x$auc))
  param.sel <- unlist(lapply(param.sel, function(x) x$auc))
  param <- param[[which.max(param.sel)]]

  # set this so you do a final model train
  param$final=T

  writeLines('final train')
  trainedModel <- do.call(made_up_model, c(param,datas)  )$model

  comp <- Sys.time() - start
  if(!quiet)
    writeLines(paste0('Model Made Up trained - took:',  format(comp, digits=3)))

  # construct the standard output for a model:
```

```
  result <- list(model = trainedModel,
                 modelSettings = list(model='made_up', modelParameters=param),
                 trainCVAuc = NULL,
                 hyperParamSearch = hyperSummary,
                 metaData = plpData$metaData,
                 populationSettings = attr(population, 'metaData'),
                 outcomeId=outcomeId,# can use populationSettings$outcomeId?
                 cohortId=cohortId,
                 varImp = NULL,
                 trainingTime=comp,
                 covariateMap=result$map
  )
  class(result) <- 'plpModel'
  attr(result, 'type') <- 'madeup'
  attr(result, 'predictionType') <- 'binary'
  return(result)

}
```

3.3 Helpers

In the fit model a helper function `made_up_model` is called, this is the function that trains a model given the data and population (where the popualtion contains a column outcomeCount corresponding to the outcome). Both the data and population are ordered the same way:

```
made_up_model <- function(data, population,
                          a=1,b=1, final=F, ...){

  writeLines(paste('Training Made Up model with ',length(unique(population$indexes)),
                   ' fold CV'))
  if(!is.null(population$indexes) && final==F){
    index_vect <- unique(population$indexes)
    perform <- c()

    # create prediction matrix to store all predictions
    predictionMat <- population
    predictionMat$value <- 0
    attr(predictionMat, "metaData") <- list(predictionType = "binary")

    for(index in 1:length(index_vect )){
      writeLines(paste('Fold ',index, ' -- with ', sum(population$indexes!=index),
                       'train rows'))
      model <- madeup::model(x = data[population$indexes!=index,],
                             y= population$outcomeCount[population$indexes!=index],
                                  a=a, b=b)

      pred <- stats::predict(model, data[population$indexes==index,])
      prediction <- population[population$indexes==index,]
      prediction$value <- pred
      attr(prediction, "metaData") <- list(predictionType = "binary")
      aucVal <- computeAuc(prediction)
      perform <- c(perform,aucVal)
```

```
      # add the fold predictions and compute AUC after loop
      predictionMat$value[population$indexes==index] <- pred

     }
    ##auc <- mean(perform) # want overal rather than mean
    auc <- computeAuc(predictionMat)

    foldPerm <- perform
  } else {
    model <- madeup::model(x= data,
                           y= population$outcomeCount,
                           a=a,b=b)

    pred <- stats::predict(model, data)
    prediction <- population
    prediction$value <- pred
    attr(prediction, "metaData") <- list(predictionType = "binary")
    auc <- computeAuc(prediction)
    foldPerm <- auc
  }

  result <- list(model=model,
                 auc=auc,
                 hyperSum = unlist(list(a = a, b = b, fold_auc=foldPerm))
  )
  return(result)
}
```

3.4 Predict

The final step is to create a predict function for the model. This gets added to the predict.R file. In the example above the type `attr(result, 'type') <- 'madeup'` was madeup, so a `predict.madeup` function is required to be added into the predict.R. The predict function needs to take as input the plpModel returned by the fit function, the population to apply the model on and the plpData specifying the covariates of the population.

```
predict.madeup <- function(plpModel,population, plpData, ...){
  result <- toSparseM(plpData, population, map=plpModel$covariateMap)
  data <- result$data[population$rowId,]
  prediction <- data.frame(rowId=population$rowId,
                           value=stats::predict(plpModel$model, data)
                           )

  prediction <- merge(population, prediction, by='rowId')
  prediction <- prediction[,colnames(prediction)%in%
                           c('rowId','outcomeCount','indexes', 'value')] # need to fix no index issue
  attr(prediction, "metaData") <- list(predictionType = "binary")
  return(prediction)

}
```

As the madeup model uses the standard R prediction, it has the same prediction function as xgboost, so we could have not added a new prediction function and instead made the type of the result returned by

fitMadeUpModel to `attr(result, 'type') <- 'xgboost'`.

4 Acknowledgments

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

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

```
##
##   Jenna Reps, Martijn J. Schuemie, Marc A. Suchard, Patrick B.
##   Ryan and Peter R. Rijnbeek (2018). PatientLevelPrediction:
##   Package for patient level prediction using data in the OMOP
##   Common Data Model. R package version 2.0.5.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {PatientLevelPrediction: Package for patient level prediction using data in the OMOP Comm
## Model},
##     author = {Jenna Reps and Martijn J. Schuemie and Marc A. Suchard and Patrick B. Ryan and Peter R
##     year = {2018},
##     note = {R package version 2.0.5},
##   }
```

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.