Custom patient-level prediction algorithms

Jenna Reps

2018-05-15

Contents

1 Introduction

The `PatientLevelPrediction` package enables data extraction, model building, and model evaluation using data from databases that are translated into the Observational Medical Outcomes Partnership Common Data Model (OMOP CDM).

This vignette describes how you can add custom algorithms to the list of available algorithms in the `PatientLevelPrediction` package. This would allow you to fully leverage the OHDSI PatientLevelPrediction model development process with your own favourite algorithm.

Of course, we invite you to share your new algorithms with the community through the GitHub repository.

2 General Structure

To add a custom classifier to the package you need to add the set, the fit functions into a R file with the model name. You also need to ensure there is a corresponding predict function in predict.R. For example, if you were to make a made up model, then in MadeUp.R you would add the following models:

2.1 set

The setNewModel is a function that takes as input the different hyper-parameter values to do a grid search when training the model. The output of the model 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 call madeUp that had two hyper-parameters then the set function would 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 calle 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

fitNewModel this function takes as input: * 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

then it trains a model for each param entry, picks the best param entry and trains a final model for that setting. The fit function returns 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.

The fit shell is:

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

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

```
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 R Model Example

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
  #TODO - how to incorporate indexes?

  # 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 wit
  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 I specified calling `made_up_model`, 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 tpye of the result returned by fitMadeUpModel to `attr(result, 'type') <- 'xgboost'`.