Adding Custom Patient-Level Prediction Algorithms

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

2021-12-16

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, a fit<Name> function and a predict<Name> function. Occasionally the fit and prediction functions may be reused (if using an R classifier see RClassifier.R or if using a scikit-learn classifier see SklearnClassifier.R). 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
- fitFunction - a string specifying what function to call to fit the model

The param object can have a setttings attribute containing any extra settings. For example to specify the model name and the seed used for reproducibility:

```r
attr(param, 'settings') <- list(
  seed = 12,
  modelName = "Special classifier"
  )
```

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

```r
setMadeUp <- function(a=c(1,4,10), b=2, seed=NULL){
  # add input checks here...

  param <- split(
    expand.grid(
      a=a,
      b=b
    ),
    1:(length(a)*length(b))
    )

  attr(param, 'settings') <- list(
    modelName = "Made Up",
    requiresDenseMatrix = TRUE,
    seed = seed
    )

  # now create list of all combinations:
  result <- list(
    fitFunction = 'fitMadeUp', # this will be called to train the made up model
    param = param
  )
  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:

- trainData - a list containing the covariateData, labels and folds for the training population
- param - the hyper-parameters as a list of all combinations
- search - the type of hyper-parameter search
- analysisId - an identifier for the analysis

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

- model - a trained model (or location of the model if it is not an R object)
- prediction - a data.frame object with the trainData$labels plus an extra column with the name 'value' corresponding to the predicted risk of having the outcome during the time-at-risk.
- settings - a list containing:
  - plpDataSettings - the plpData settings e.g., attr(trainData, "metaData")$plpDataSettings

  - covariateSettings - the covariate settings e.g., attr(trainData, “metaData”)$covariateSettings
  - populationSettings - the population settings e.g., attr(trainData, “metaData”)$populationSettings,
  - featureEngineering - the feature engineering settings e.g., attr(trainData*covariateData*, ”*metaData*”)featureEngineer
  - tidyCovariates - the preprocessing settings e.g., attr(trainData*covariateData*, ”*metaData*”)tidyCovariateDataSetting
  - requireDenseMatrix - does the model require a dense matrix? e.g., attr(param, ‘settings’)$requiresDenseMatrix,
  - modelSettings = a list containing: model (model name), param (the hyper-parameter search list), finalModelParameters (the final model hyper-parameters), extraSettings (any extra settings)
  - splitSettings - the split settings e.g., attr(trainData, “metaData”)$splitSettings,
  - sampleSettings - the sample settings e.g., attr(trainData, “metaData”)$sampleSettings
- trainDetails - a list containing:
  - analysisId - the identifier for the analysis
  - cdmDatabaseSchema - the database used to develop the model
  - outcomeId - the outcome id
  - cohortId - the target population id
  - attrition - the attrition
  - trainingTime - how long it took to train the model
  - trainingDate - date of model training
  - hyperParamSearch - the hyper-parameter search used to train the model
- covariateImportance - a data.frame containing the columns ‘covariateId’, ‘covariateValue’ (the variable importance) and ‘columnId’ (the column number that the variable need to be mapped to when implementing the model)

In additon the plpModel requires two attributes:

- predictionFunction - the name of the function used to make predictions
- modelType - whether the model is ‘binary’ or ‘survival’

For example `attr(result, 'predictionFunction') <- 'madeupPrediction'` means when the model is applied to new data, the ‘madeupPrediction’ function is called to make predictions. If this doesnt exist, then the model will fail. The other attribute is the modelType `attr(result, 'modelType') <- 'binary'` this is needed when evaluating the model to ensure the correct evaluation is applied. Currently the evaluation supports ‘binary’ and ‘survival’ modelType.

Note: If a new modelType is desired, then the evalaution code within PatientLevelPrediction must be updated to specify how the new type is evaluated. This requires making edits to PatientLevelPrediction and then making a pull request to the PatientLevelPrediction github. The evaluation cannot have one off customization because the evaluation must be standardized to enable comparison across similar models.

A full example of a custom ‘binary’ classifier fit function is:

```r
fitMadeUp <- function(trainData, param, search, analysisId){

  # **************** code to train the model here
  # trainedModel <- this code should apply each hyper-parameter combination
  # (param[[i]]) using the specified search (e.g., 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
  # ****************

  # **************** code to apply the model to trainData
  # prediction <- code to apply trainedModel to trainData
  # ****************

  # **************** code to get variable importance (if possible)
  # varImp <- code to get importance of each variable in trainedModel
  # ****************


  # construct the standard output for a model:
  result <- list(model = trainedModel,
                 prediction = prediction, # the train and maybe the cross validation predictions for th
    settings = list(
      plpDataSettings = attr(trainData, "metaData")$plpDataSettings,
      covariateSettings = attr(trainData, "metaData")$covariateSettings,
      populationSettings = attr(trainData, "metaData")$populationSettings,
      featureEngineering = attr(trainData$covariateData, "metaData")$featureEngineering,
      tidyCovariates = attr(trainData$covariateData, "metaData")$tidyCovariateDataSettings,
      requireDenseMatrix = attr(param, 'settings')$requiresDenseMatrix,
      modelSettings = list(
        model = attr(param, 'settings')$modelName, # the model name
        param = param,
        finalModelParameters = param[[bestInd]], # best hyper-parameters
        extraSettings = attr(param, 'settings')
      ),
      splitSettings = attr(trainData, "metaData")$splitSettings,
      sampleSettings = attr(trainData, "metaData")$sampleSettings
    ),

    trainDetails = list(
      analysisId = analysisId,
      cdmDatabaseSchema = attr(trainData, "metaData")$cdmDatabaseSchema,
      outcomeId = attr(trainData, "metaData")$outcomeId,
      cohortId = attr(trainData, "metaData")$cohortId,
      attrition = attr(trainData, "metaData")$attrition,
      trainingTime = timeToTrain, # how long it took to train the model
      trainingDate = Sys.Date(),
      hyperParamSearch = hyperSummary # the hyper-parameters and performance data.frame
    ),
    covariateImportance = merge(trainData$covariateData$covariateRef, varImp, by='covariateId') # add v
  )
  class(result) <- 'plpModel'
  attr(result, 'predictionFunction') <- 'madeupPrediction'
  attr(result, 'modelType') <- '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 (for example a function to run cross validation). It is important to ensure there is a valid prediction function (the one specified by `attr(result, 'predictionFunction') <- 'madeupPrediction'` is `madeupPrediction()`) as specified below.

2.3 Predict

The prediction function takes as input the plpModel returned by fit, new data and a corresponding cohort. It returns a data.frame with the same columns as cohort but with an additional column:

- value - the predicted risk from the plpModel for each patient in the cohort

For example:

```
madeupPrediction <- function(plpModel, data, cohort){

  # ************* code to do prediction for each rowId in cohort
  # predictionValues <- code to do prediction here returning the predicted risk
  #               (value) for each rowId in cohort
  #**************

  prediction <- merge(cohort, predictionValues, by='rowId')
  attr(prediction, "metaData") <- list(modelType = attr(plpModel, 'modelType'))
  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 (see GradientBoostingMachine.R for the set function, RClassifier.R for the fit and prediction function for R classifiers).

3.1 Set

```
setMadeUp <- function(a=c(1,4,6), b=2, seed=NULL){
  # add input checks here...

  if(is.null(seed)){
    seed <- sample(100000,1)
  }

  param <- split(
    expand.grid(
      a=a,
      b=b
    ),
    1:(length(a)*length(b))
    )

  attr(param, 'settings') <- list(
    modelName = "Made Up",
    requiresDenseMatrix = TRUE,
    seed = seed
    )

  # now create list of all combinations:
  result <- list(
    fitFunction = 'fitMadeUp', # this will be called to train the made up model
    param = param
  )
  class(result) <- 'modelSettings'

  return(result)
}
```

3.2 Fit

```r
fitMadeUp <- function(trainData, param, search, analysisId){

  # set the seed for reproducibility
  set.seed(attr(param, 'settings')$seed)

  # add folds to labels:
  trainData$labels <- merge(trainData$labels, trainData$folds, by= 'rowId')
  # convert data into sparse R Matrix:
  mappedData <- toSparseM(trainData,map=NULL)
  matrixData <- mappedData$dataMatrix
  labels <- mappedData$labels
  covariateRef <- mappedData$covariateRef

  #============= STEP 1 ======================================
  # pick the best hyper-params and then do final training on all data...
  writeLines('Cross validation')
  param.sel <- lapply(
    param,
    function(x){
      do.call(
        made_up_model,
        list(
          param = x,
          final = F,
          data = matrixData,
          labels = labels
          )
      )
      }
    )
  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))
  bestInd <- which.max(param.sel)

  #get cross val prediction for best hyper-parameters
  prediction <- param.sel[[bestInd]]$prediction
  prediction$evaluationType <- 'CV'

  writeLines('final train')
  finalResult <- do.call(
    made_up_model,
    list(
      param = param[[bestInd]],
      final = T,
      data = matrixData,
      labels = labels
      )
    )

  trainedModel <- finalResult$model
  # prediction risk on training data:
  finalResult$prediction$evaluationType <- 'Train'

  # get CV and train prediction
  prediction <- rbind(prediction, finalResult$prediction)

  varImp <- covariateRef %>% dplyr::collect()
  # no feature importance available
  vqrImp$covariateValue <- 0

 timeToTrain <- Sys.time() - start

  # construct the standard output for a model:
  result <- list(model = trainedModel,
                 prediction = prediction,
    settings = list(
      plpDataSettings = attr(trainData, "metaData")$plpDataSettings,
      covariateSettings = attr(trainData, "metaData")$covariateSettings,
      populationSettings = attr(trainData, "metaData")$populationSettings,
      featureEngineering = attr(trainData$covariateData, "metaData")$featureEngineering,
      tidyCovariates = attr(trainData$covariateData, "metaData")$tidyCovariateDataSettings,
      requireDenseMatrix = attr(param, 'settings')$requiresDenseMatrix,
      modelSettings = list(
        model = attr(param, 'settings')$modelName, # the model name
        param = param,
        finalModelParameters = param[[bestInd]], # best hyper-parameters
        extraSettings = attr(param, 'settings')
      ),
      splitSettings = attr(trainData, "metaData")$splitSettings,
      sampleSettings = attr(trainData, "metaData")$sampleSettings
    ),

    trainDetails = list(
      analysisId = analysisId,
      cdmDatabaseSchema = attr(trainData, "metaData")$cdmDatabaseSchema,
      outcomeId = attr(trainData, "metaData")$outcomeId,
      cohortId = attr(trainData, "metaData")$cohortId,
      attrition = attr(trainData, "metaData")$attrition,
      trainingTime = timeToTrain, # how long it took to train the model
      trainingDate = Sys.Date(),
      hyperParamSearch = hyperSummary # the hyper-parameters and performance data.frame
    ),
    covariateImportance = varImp
  )
  class(result) <- 'plpModel'
  attr(result, 'predictionFunction') <- 'madeupPrediction'
  attr(result, 'modelType') <- '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, labels and hyper-parameters.

```
made_up_model <- function(param, data, final=F, labels){

  if(final==F){
    # add value column to store all predictions
    labels$value <- rep(0, nrow(labels))
    attr(labels, "metaData") <- list(modelType = "binary")

    foldPerm <- c() # this holds CV aucs
    for(index in 1:max(labels$index)){
      model <- madeup::model(
        x = data[labels$index!=index,], # remove left out fold
        y = labels$outcomeCount[labels$index!=index],
        a = param$a,
        b = param$b
      )

      # predict on left out fold
      pred <- stats::predict(model, data[labels$index==index,])
      labels$value[labels$index==index] <- pred

      # calculate auc on help out fold
      aucVal <- computeAuc(labels[labels$index==index,])
      foldPerm<- c(foldPerm,aucVal)
    }
    auc <- computeAuc(labels) # overal AUC

  } else {
    model <- madeup::model(
      x = data,
      y = labels$outcomeCount,
      a = param$a,
      b = param$b
      )

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

  result <- list(
    model = model,
    auc = auc,
    prediction = labels,
    hyperSum = c(a = a, b = b, fold_auc = foldPerm)
  )

  return(result)
}
```

3.4 Predict

The final step is to create a predict function for the model. In the example above the predeiction function `attr(result, 'predictionFunction') <- 'madeupPrediction'` was madeupPrediction, so a `madeupPrediction` function is required when applying the model. The predict function needs to take as input the plpModel returned by the fit function, new data to apply the model on and the cohort specifying the patients of interest to make the prediction for.

```
madeupPrediction <- function(plpModel, data , cohort){

  if(class(data) == 'plpData'){
    # convert
    matrixObjects <- toSparseM(
      plpData = data,
      cohort = cohort,
      map = plpModel$covariateImportance %>%
        dplyr::select(.data$columnId, .data$covariateId)
    )

    newData <- matrixObjects$dataMatrix
    cohort <- matrixObjects$labels

  }else{
    newData <- data
  }

  if(class(plpModel) == 'plpModel'){
    model <- plpModel$model
  } else{
    model <- plpModel
  }

  cohort$value <- stats::predict(model, data)

  # fix the rowIds to be the old ones
  # now use the originalRowId and remove the matrix rowId
  cohort <- cohort %>%
    dplyr::select(-.data$rowId) %>%
    dplyr::rename(rowId = .data$originalRowId)

  attr(cohort, "metaData") <- list(modelType = attr(plpModel, 'modelType'))
  return(cohort)

}
```

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 predictionFunction of the result returned by fitMadeUpModel to `attr(result, 'predictionFunction') <- 'predictXgboost'`.

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.