PatientLevelPrediction: Extra Methods

Jenna Reps, Peter Rijnbeek

February 24, 2016

Contents

1 Introduction **1**

2 Default Model Example **2**
2.1 Loading Libraries . . . 2
2.2 Extracting Data . . . 3
2.3 Censoring Data . . . 6
2.4 Describing Data . . . 6
2.5 Model Development . . . 6
2.5.1 Model Settings . . . 7
2.5.2 Feature settings . . . 7
2.5.3 Putting it together . . . 8
2.6 Evaluation . . . 8

3 Custom Feature Selection **10**

4 Custom Classifier **14**
4.1 Custom train function . . . 14
4.2 Custom predict function . . . 16

5 Default Methods/Models **18**

1 Introduction

This document details the various functionalities of the Extra Methods branch of Patient Level Prediction package. The first section gives a step by step example of predicting whether somebody smokes using data in the OMOP CDM structure by using the previous 365 days worth of records; logistic regression, gradient boosting machines and neural network models are demonstrated. In the second section an example of integrating a customized feature selection method is explained and the final section described how to include customized classifiers.

The Patient Level Prediction packageextra methods package comes with a range of default feature processing and classification functions. The functions and their parameters are displayed in Table 2.

The prediction process follows the general flow:

1. extract data - getPlpData()
2. censor (filter) data - censorPlpData()
3. develop the model - developModel2()
   - split data into test/train sets
   - apply any feature engineering/selection
   - apply classifier training
   - evaluate model on test set
4. compare models - comparePlp

The developModel2() function consists of a feature selection/engineering step followed by a classifier training step and ends with an evaluation of the model on the test set. There are various feature selection/engineering function and classifier functions that come with the Patient Level Prediction package, but the framework has been developed to readily enable users to incorporate their own functions for each step. This document serves as a guide as to how additional methods can be incorporated by specifying the required structure of the methods and providing example code.

2 Default Model Example

The first step is extracting the Patient Level Prediction packagedata. This requires setting up a database connection and creating cohort/outcome tables. You also need to initiate the h2o cluster if you are using any h2o models (and this requires downloading h2o from: http://www.h2o.ai/) and I recommend setting the fftempdir to a directory on your computer with plenty of space.

2.1 Loading Libraries

Load the following libraries:

```
library(DatabaseConnector)
library(OhdsiRTools)
library(SqlRender)
library(ffbase)
library(PatientLevelPrediction)
library(h2o)
library(BigKnn)
library(caret)
```

```
# general initial settings '50g' means 50GB - you may need to reduce
options(fftempdir = "drive:/FFtemp")
h2o.init(nthreads = -1, max_mem_size = '50g')
```

2.2 Extracting Data

To set up a database connection for the user 'JoeDoe' with a password of 'password1' on the server 'server name' with a work schema (must have read/write access) of 'Work.dbo' using parallel data warehouse:

```
dbms <- "pdw"
user <- JoeDoe
pw <- password1
server <- "server name"
port <- 1
oracleTempSchema <- NULL
cdmVersion <- 5


connectionDetails <- DatabaseConnector::createConnectionDetails(dbms = dbms,
                                                                server = server,
                                                                user = user,
                                                                password = pw,
                                                                port = port)
conn <- DatabaseConnector::connect(connectionDetails)
```

The following SQL code will create a combined cohort/outcome table for smoker (people with concept id 2101895 recorded) and non-smoker (people with concept id 2101897 recorded) with the cohort start date being the first recording date minus 1 (this is due to the covariate creation in Patient Level Prediction packageusing the cohort start date). Smoking has conceptId 2 and non-smoking has conceptId 1; the data are stored in table 'smokeall_yourDataBase' in the schema 'scratch.dbo'.

```
# all people with smoking recorded (concept: 2101895)
sql <- "select * into scratch.dbo.smokeall_yourDataBase
       from (select person_id subject_id,
         dateadd(day, -1, OBSERVATION_DATE) as cohort_start_date,
          OBSERVATION_DATE as cohort_end_date,
          case when observation_concept_id ='2101895' then 2 else 1 end
              cohort_concept_id,
row_number() over (partition by person_id order by OBSERVATION_DATE asc) rn
          from yourDataBase.dbo.observation
          where observation_concept_id in (2101895, 2101897)) temp
          where temp.rn=1;"
sql <- translateSql(sql, 'sql server','pdw')\$sql
executeSql(conn, sql)
```

With our connection and cohort/outcome tables created, we now need to specify the type of predictive variables we wish to extract from the database using the covariateSettingMain() function:

```
covariateSettingsMain <- PatientLevelPrediction::createCovariateSettings(
useCovariateDemographics = TRUE,
useCovariateDemographicsGender = T,
useCovariateDemographicsRace = T,
useCovariateDemographicsEthnicity = T,
useCovariateDemographicsAge = TRUE,
useCovariateDemographicsYear = F,
useCovariateDemographicsMonth = F,
useCovariateConditionOccurrence = T,
useCovariateConditionOccurrence365d = T,
useCovariateConditionOccurrence30d = T,
useCovariateConditionOccurrenceInpt180d = F,
useCovariateConditionEra = F,
useCovariateConditionEraEver = F,
useCovariateConditionEraOverlap = F,
useCovariateConditionGroup = TRUE,
useCovariateConditionGroupMeddra = TRUE,
useCovariateConditionGroupSnomed = TRUE,
useCovariateDrugExposure = T,
useCovariateDrugExposure365d = T,
useCovariateDrugExposure30d =T,
useCovariateDrugEra = F,
useCovariateDrugEra365d =F ,
useCovariateDrugEra30d = F,
useCovariateDrugEraOverlap = F,
useCovariateDrugEraEver = F,
useCovariateDrugGroup = TRUE,
useCovariateProcedureOccurrence = T,
useCovariateProcedureOccurrence365d = T,
useCovariateProcedureOccurrence30d = T,
useCovariateProcedureGroup = TRUE,
useCovariateObservation = T,
useCovariateObservation365d = T,
useCovariateObservation30d = T,
useCovariateObservationCount365d = T,
useCovariateMeasurement = T,
useCovariateMeasurement365d = TRUE,
useCovariateMeasurement30d = T,
useCovariateMeasurementCount365d = T,
useCovariateMeasurementBelow = F,
useCovariateMeasurementAbove = F,
useCovariateConceptCounts = F,
useCovariateRiskScores = F,
useCovariateRiskScoresCharlson = F,
useCovariateRiskScoresDCSI = F,
useCovariateRiskScoresCHADS2 = F,
useCovariateRiskScoresCHADS2VASc = F,
useCovariateInteractionYear = FALSE,
useCovariateInteractionMonth = FALSE,
excludedCovariateConceptIds = c(),
includedCovariateConceptIds = c(),
deleteCovariatesSmallCount = 50)

covariateSettings <- list(covariateSettingsMain)
```

We can now run getDbPlpData() to extract our data. The data extracted will extract everyone from our previously extracted cohort table and label smokers as having the outcome.

```
# set some more variables
cdmDatabaseSchema <- "yourDataBase.dbo"
workDatabaseSchema <-"Work.dbo"

plpData <- getDbPlpData(connectionDetails = connectionDetails,
                     cdmDatabaseSchema=cdmDatabaseSchema,
                     oracleTempSchema = NULL,
                     cohortDatabaseSchema = workDatabaseSchema,
                     cohortTable = "‘smokeall_yourDataBase"’,
                     cohortIds = c(1,2),
                     washoutWindow = 0,
                     useCohortEndDate = FALSE,
                     windowPersistence = 0,
                     covariateSettings= covariateSettingsMain,
                     outcomeDatabaseSchema = workDatabaseSchema,
                     outcomeTable = "‘smokeall_yourDataBase"’,
                     firstOutcomeOnly = FALSE,
                     excludeHistory=FALSE,
                     outcomeIds = 2,
                     outcomeIdsExclude=NULL,
                     startAdd=0,
                     cdmVersion = cdmVersion)
```

It is a good idea to save the data at this point, this will save the plpData into a folder named ‘smoke_first’ in the working directory.

```
savePlpData(plpData, file.path(getwd(),’smoke_first’))
```

2.3 Censoring Data

The next step is to filter the data based on certain criteria. As we are using the previous 365 days to construct the predictive variables, it seems reasonable to require that each patient included into the prediction has a minimum of 365 days observation prior to the prediction index date.

Load the data:

```
loadPlpData(file.path(getwd(),'smoke_first'))
```

The code to censor patients and filter those with less than 365 days observation and only include people who had a recording of smoker or non-smoker for the first time between 2008-01-01 and 2011-01-01:

```
plpData.censor <- censorPlpData(plpData, minPriorObservation = 365,
                                dateInterval = c('2008-01-01','2011-01-01'))
```

We can explore how many people were removed by using the following code:

```
plpData.censor$metaData$excluded
```

2.4 Describing Data

It is useful to gain in sight into the data and ensure the outcome prevalence is table across time and is consistent with current knowledge:

```
describePlpData(plpData.censor)
```

this is plot the prevalence by year and print summary details.

2.5 Model Development

Once you have the Patient Level Prediction packagedata extracted into R and done any appropriate censoring you can then use the developModel2() function to train and evaluate the model. The developModel2() requires five inputs:

- plpData - the data to train and test the model one.
- modelSettings - a list containing: model - a string with the model name, param - a list of parameters that are used by the model.
- featureSettings - a list containing: method - a string with the feature processing method name, param - a list of parameters that are used by the method.
- type - whether the split the test/train data on patient or year or both.
- validationFraction - the fraction of data used in the test set.

2.5.1 Model Settings

The modelSettings specifies the model to use to train a classifier and the hyper parameters to do a grid search over while training the classifier. The modelSettings class must be set to 'modelSettings'.

To train a lasso logistic regression use the model='lr_lasso' and use the param list to specify the variance to use by lasso logistic regression, also within param the outcomeId specifying what the classifier is prediction should be input and the cohortId is a vector of cohortIds to do the model/prediction on.

```
# lasso logistic regression using no feature selection:
modset_llr <- list(model='lr_lasso',
                   param=list(variance =0.001, cohortId=c(1,2), outcomeId=2))
class(modset_llr) <- 'modelSettings'
```

To train a gradient boosting machine to predict smoking using grid search across the hyper parameters: rsampRate (number of rows to sample in each tree)=0.8, ntrees (number of trees to build) = 100 and 150 and max_depth (max interaction level) = 2, 4 and 5:

```
modset_gbm <- list(model='gbm_plp',
                   param=list(rsampRate=0.8, ntrees=c(100,150),
                       max_depth=c(2,4,5), cohortId=c(1,2), outcomeId=2))
class(modset_gbm) <- 'modelSettings'
```

To train a neural network to predict smoking using grid search across the hyper parameters: size (number of hidden nodes) of 2 and 4, decay (form of regulisation) or 0 and 0.01 and also setting the maxits (the maximum number of interactions while minimizing the objective function):

```
modset_nnet <- list(model='nnet_plp',
                    param=list(size =c(2,4), decay=c(0,0.01),maxits=3000,
                        cohortId=c(1,2), outcomeId=2))
class(modset_nnet) <- 'modelSettings'
```

2.5.2 Feature settings

The feature selection settings is similar to the model setting. The featureSettings is a class of type featureSelection corresponding to a list containing type objects: method and param.

To use a genetic algorithm (GA) for wrapper feature selection set the method to 'wrapperGA' and the parameters for the wrapper just requires varSize (the approximate number of variable you wish to return), iter (the number of iterations for the GA) the outcomeId and cohortId to know what data to try the feature subset on,

```
featSet_wrapper <- list(method='wrapperGA', param=list(cohortId=c(1,2),
    outcomeId=2, varSize=300, iter=1))
class(featSet_wrapper) <- 'featureSettings'
```

To select a specific type of covariate, use the 'filterCovariates' method with the param specifying whether the include or exlcude the covariates of interest and a vector of the covariates of interest. In the example below we only include covariates of analysis type 4 and 101 (demographics and condition occurance 365d prior):

```
featSet_filter <- list(method='filterCovariates', param=list(include=T,
                                                      analysisIds=c(4,101)))
class(featSet_filter) <- 'featureSettings'
```

2.5.3 Putting it together

We can now input the modelSettings and featureSettings into the developModel2() function.

To train a lasso logistic regression with no feature selection using the 'year' type evaluation (80% of data into train set and 20% into validation set, where data is split on year (older date into train, newer data into validation):

```
mod_llr <- developModel2(plpData= plpData.censor,
                      featureSettings = NULL,
                      modelSettings = modset_llr ,
                      type='year')
```

To train a gradient boosting machine with genetic algorithm wrapper feature selection using the 'year' type evaluation (80% of data into train set and 20% into validation set, where data is split on year (older date into train, newer data into validation):

```
mod_gbm <- developModel2(plpData= plpData.censor,
                    featureSettings = featSet_wrapper,
                    modelSettings = modset_gbm,
                    type='year')
```

To train a neural network with specific covariate feature selection using the 'year' type evaluation (80% of data into train set and 20% into validation set, where data is split on year (older date into train, newer data into validation):

```
mod_nnet <- developModel2(plpData= plpData.censor,
                        featureSettings = featSet_filter,
                        modelSettings = modset_nnet,
                        type='year')
```

It is now a good idea to save the models. (TO DO)

2.6 Evaluation

Once the models are developed we can then compare the models by using the comparePlp() function. First you need to load the models then construct a list of each model you wish to include into the comparison,

```
allMods <- list(mod_llr[[1]], mod_gbm[[1]], mod_nnet[[1]])
```

next you simply run,

```
#  Compare the models:
comparePlp(allMods)
```

and the model summaries and plots are returned. You can save this image within R Studios or (in the future I will enable a save location input).

3 Custom Feature Selection

To include a custom feature selection method into the framework you need to create a function to process the data that has as input:

- plpData- the inout data being processed by the method
- param1 - the first parameters of the method (if needed)
- param2 - the second parameter of the method (if needed)
- ...
- paramN - the Nth parameters of the method (if needed)

the output is a list containing:

- covariates- an ffdf of the plp covariate data
- cohorts - an ffdf of the plp cohorts data
- outcomes - an ffdf of the plp outcomes data
- covariateRef - an ffdf of the plp covariate details
- metaData - a list of the original plpData's metaData with the addition objects: featureInclude, featureExclude and featureSet appended to the list.

The additions to the metaData enablea record of the feature selection/engineering that was applied to be stored so that future predictions can implement the same feature processing. The featureInclude and featureExclude objects are vectors containing the features included or excluded respectively. The featureSet object is a list containing the following objects:

- method - a string detailing the function name
- parameter values - all the parameters of the feature selection model
- transform - a function that performs the same feature processing that will be called when the model is applied to new data - this must be included!

For example, the develop a feature selection method that picks a random subset of features n times and calculates an average of the variable importance to pick a final m number of features, named 'varImp':

```
varImp <- function(plpData, sampSize, n, m){

            #clone data to prevent accidentally deleting plpData
        cohorts <-ff::clone(plpData$cohorts)
        outcomes <-ff::clone(plpData$outcomes)
        covariates <- ff::clone(plpData$covariates)
```

```
    # take a random straified sample if data > 50,000 people
    ppl.all <- unique(as.ram(cohorts$rowId))
    if(length(ppl.all)>50000){
 ppl.out <- unique(as.ram(outcomes$rowId))
 ppl.notout <- ppl.all[!ppl.all%in%ppl.out]
 ppl.out<- sample(ppl.out, 50000/length(ppl.all)*length(ppl.out))
 ppl.notout <- sample(ppl.notout,
     50000/length(ppl.all)*length(ppl.notout))
       ppl.all <- c(ppl.out, ppl.notout)
    }
   t <- ffbase::ffmatch(covariates$rowId, table=ff::as.ff(ppl.all))
covariates.samp<- covariates[ffbase::ffwhich(t, !is.na(t)),]


 # Create the non-sparse feature matrix
covMat <- reshape2::dcast(ff::as.ram(covariates.samp),
   rowId~covariateId,
                value.var='covariateValue', fill=0)
    # add label
covMat <- merge(covMat, as.ram(outcomes[,c('rowId','outcomeCount')]),
                    by='rowId', all.x=T)
labs <- rep(1, nrow(covMat))
labs[is.na(covMat$outcomeCount)] <- 0
covMat$outcomeCount <- labs

    #load into h2o
covMat <- h2o::as.h2o(covMat[,-1]) # remove rowId
covMat$outcomeCount <- h2o::as.factor(covMat$outcomeCount)


 # randomly sample sampSize features and train a gbm n times
 varImpModel <- function(...){
               mod <- h2o::h2o.gbm(x=sample(ncol(covMat)-1, sampSize),
                                      y=ncol(covMat),
                                   training_frame=covMat,
                                                       max_depth=4,
                                            ntrees=50, nfolds=5)
                         return(mod@model$variable_importances)
          }
 # get a list of n variable importance data frames
 imps <- lapply(1:n, varImpModel)

 # now merge the n var importance data frames
 varImp <- imps[[1]]
 for(i in 2:n){
      additionImp <- imps[[i]][,c('variable','scaled_importance')]
 colnames(additionImp)[2] <- paste0('imp',i)
 varImp <- merge(varImp, additionImp, by='variable', all=T, fill=NA)

  }

 # now calculate the average importance
    # (sum of var imp/number of times included)
 varScore <- apply(varImp[,-1], 1, mean)
 varImp <- varImp[order(-varScore),]

# find the top m features on average varImp
selFeat <- varImp$variable[1:m]

# now extract the covariates that are in selFeat
t <- ffbase::ffmatch(covariates$covariateId,
                        table=ff::as.ff(as.double(selFeat)))
covariates <- covariates[ffbase::ffwhich(t,!is.na(t)),]

# create the transformData function
transformData <- function(plpData2){
writeLines('transform function running...')
newcovs <- ff::clone(plpData2$covariates)
t <- ffbase::ffmatch(newcovs$covariateId,
                        table=ff::as.ff(as.double(selFeat)))

newcovs<- newcovs[ffbase::ffwhich(t, !is.na(t)),]

metaData <- plpData2$metaData
if(!is.null(metaData$featureInclude) )
 metaData$featureInclude <-
     c(metaData$featureInclude,as.double(selFeat))
if(is.null(metaData$featureInclude) )
 metaData$featureInclude <- as.double(selFeat)


newData <- list(covariates = newcovs,
                cohorts = ff::clone(plpData2$cohorts),
                outcomes = ff::clone(plpData2$outcomes),
                covariateRef = ff::clone(plpData2$covariateRef),
                metaData = metaData
                )
return(newData)
}


# update metaData
# add the covariatesIncluded metadata list of featureSelect
```

```
          metaDataUpdate <- plpData$metaData
          if(!is.null(metaDataUpdate$featureInclude))
            metaDataUpdate$featureInclude <-
                c(metaDataUpdate$featureInclude,as.double(selFeat))
          if(is.null(metaDataUpdate$featureInclude))
            metaDataUpdate$featureInclude <- as.double(selFeat)

          featureSet <- list(method='varImp',
                                  sampSize = sampSize
                             n = n,
                             m = m,
                             transform=transformData)
          if(!is.null(metaDataUpdate$featureSettings))
            metaDataUpdate$featureSettings <-
                list(metaDataUpdate$featureSettings, featureSet )
          if(is.null(metaDataUpdate$featureSettings))
            metaDataUpdate$featureSettings <- featureSet

          result <- list(covariates = covariates,
                         cohorts = cohorts,
                         outcomes = outcomes,
                         covariateRef = ff::clone(plpData$covariateRef),
                         metaData = metaDataUpdate
                         )
          return(result)


}
```

You can then call this function within the Patient Level Prediction packageframework using the featureSetting object,

```
featSet_varImp <- list(method='varImp', param=list(sampSize=40, n=100, m=100))
class(featSet_filter) <- 'featureSettings'
```

4 Custom Classifier

The custom classifier requires creating two functions: the train function and the predict function. The train function has the input:

- plpData- the data used the train the model
- param - a list containing the parameters used by the model
- quiet - a boolean variable stating whether output progression should be printed for the user (quiet=T means no output, quiet=F means write progression details)

The train function outputs a list of class 'plpmodel' and an attribute named 'type' that specifies the type of prediction function. The 'plpModel' list contains:

- model - the model that will be applied using the predict function
- modelLoc - the location of the model if it is saved
- trainAuc - the AUC on the train set (CV)
- trainCalibration - the calibration of the train set (CV)
- modelSettings - a list containing the inputs to the function
- metaData - the plpData metaData
- trainingTime - the time is takes the train the model

The predict function for a plpModel with an attribute of type 'typeName' should be named 'predict.typeName'. Each predict function takes as input:

- plpData - the test data
- plpModel - the model to apply to the test data

The predict function outputs a data frame containing the cohort data cast as a data frame with the model prediction added as a column called 'value'. The prediction must also have an attribute named 'outcomeId' set to the outcomeId being predicted.

4.1 Custom train function

For example, a knn classifier, that I will name 'knn_plp' that uses the knn algorithm to predict the outcome:

```
knn_plp <- function(plpData, param, quiet=T){
  start <- Sys.time()

   # set default k value
   k <- param$k
  if(is.null(k))
    k <- 10
  cohortId <- param$cohortId
  outcomeId <- param$outcomeId
  indexFolder <- param$indexFolder

   #clone data to prevent accidentally deleting plpData
  cohorts <-ff::clone(plpData$cohorts)
  outcomes <-ff::clone(plpData$outcomes)
  covariates <- ff::clone(plpData$covariates)

  # filter the outcome and cohort ids:
  if(!is.null(cohortId)){
    t <- ffbase::ffmatch(cohorts$cohortId, table=ff::as.ff(cohortId))
    ppl<- cohorts$rowId[ffbase::ffwhich(t, !is.na(t))]

    t <- ffbase::ffmatch(covariates$rowId, table=ppl)
    covariates <- covariates[ffbase::ffwhich(t, !is.na(t)),]
  }
  if(!is.null(outcomeId)){
    t <- ffbase::ffmatch(outcomes$outcomeId, table=ff::as.ff(outcomeId))
    outcomes<- outcomes[ffbase::ffwhich(t, !is.na(t)),]
  }

  # format of knn
  outcomes$y <- ff::as.ff(rep(1, length(unique(ff::as.ram(outcomes$rowId)))))

  # add 0 outcome:
  ppl <- as.ram(cohorts$rowId)
  new <- ppl[!ppl%in%unique(ff::as.ram(outcomes$rowId))]
  newOut <- data.frame(rowId=new, outcomeId=-1,outcomeCount=1,timeToEvent=0,y=0)
  outcomes <- as.ffdf(rbind(as.ram(outcomes),newOut))

   # create the model in indexFolder
   BigKnn::buildKnn(outcomes = ff::as.ffdf(outcomes),
                  covariates = ff::as.ffdf(covariates),
                  indexFolder = indexFolder)

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

  result <- list(model = indexFolder,
                modelLoc = indexFolder, # did I actually save this!?
                trainAuc = NULL,
                trainCalibration=NULL,
                modelSettings = list(model='knn',
                                          k=k,
                                                                  cohortId=cohortId,
                                                                  outcomeId=outcomeId,
                                                                  indexFolder=indexFolder),
                 metaData = plpData$metaData,
                 trainingTime=comp
    )
    class(result) <- 'plpModel'
    attr(result, 'type') <- 'knn'
    return(result)
  }
```

the 'knn_plp' function returns the model with an attribute of type 'knn', this is used during prediction by pointing to a function called 'predict.knn' to do the prediction for this model. Therefor, another function called 'predict.knn' must also be written.

4.2 Custom predict function

Now we need to define a prediction function for models with the attribute 'type' of 'knn',

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

prediction <- BigKnn::predictKnn(covariates = plpData$covariates,
                      indexFolder = plpModel$model,
                      k = plpModel$modelSettings$k,
                      weighted = TRUE)

# return the cohorts as a data frame with the prediction added as
# a new column with the column name 'value'
prediction <- merge(ff::as.ram(plpData$cohorts), prediction, by='rowId',
                  all.x=T, fill=0)
attr(prediction, "outcomeId") <- plpModel$modelSettings$outcomeId


return(prediction)
}
```

To put it all together into the frame framework create a modelSettings object,

```
modset_knn <- list(model=knn_plp',
                 param=list(k=25, indexFolder=file.path(getwd(),'knn') )
class(modset_plp) <- 'modelSettings'
```

You can now run the custom model that uses gradient boosting machine average variable importance to select a subset of features and then implements knn on the data with the reduced dimensionality to evaluate by year split using the code,

```
mod_custom <- developModel2(plpData= plpData.censor,
                         featureSettings = featSet_varImp,
                                  modelSettings = modset_knn,
                                  type='year')
```

5 Default Methods/Models

Name	Description	Parameters
'filterCovariates'	Filters covariates	• include: boolean (T= include, F=exclude) • conceptIds: vector of concept_ids to include/exclude • covariateIds: vector of covariate_ids to include/exclude • analysisIds: vector of analysis_ids to include/exclude • quiet: T=suppress messages, F=output messages
'wrapperGA'	Uses genetic algorithm wrapped around model to pick feature subset	• model: the wrapper model • varSize: the approximate number of features returned by the genetic algorithm • iter: the number of iterations of the genetic algorithm
'varImp'	Randomly samples sampSize features per gradient boosting machine n times and then picks the top n features based on average variable importance over the n models	• sampSize: the number of features sampled in each model • n: the number of models trained • m: the number of features returned
'lassolr'	The variables chosen by logistic regression with lasso regularisation	• cohortId: the cohortIds to use in the training set • outcomeId: the outcomeIds to identify an outcome • variance: how regularized the model is • quiet: whether to suppress output messages
'glrm'	Generalised low rank models to find latent topics that combine features	IN PROGRESS

Table 1: The feature processing methods that are default

Method	Name	Parameters
Lasso Logistic Regression	‘lr_lasso’	variance
KNN	‘knn_plp ’	k
Neural network	‘nnet_plp’	size, decay
Gradient boosting machines	‘gbm_plp’	bal, rsampRate, ntrees, csampRate, nbins, max_depth, min_rows, learn_rate
Random forest	‘randomForest_plp’	bal, rsampRate, ntrees, mtries, nbins, max_depth, min_rows
Support Vector Machines	‘svmRadial_plp’	sigma, C

Table 2: The methods that are default