Patient Level Prediction Examples
version 2.0

Jenna Reps and Peter Rijnbeek

October 19, 2016

1 Introduction

The patient level prediction package is a R package that enables users to download data from health observational medical outcomes partnership (OMOP) databases in common data model (CDM) format, constructs features and trains models using various machine learning algorithms. The trained models can then be used to predict an outcome within a user specified cohort during a defined time period.

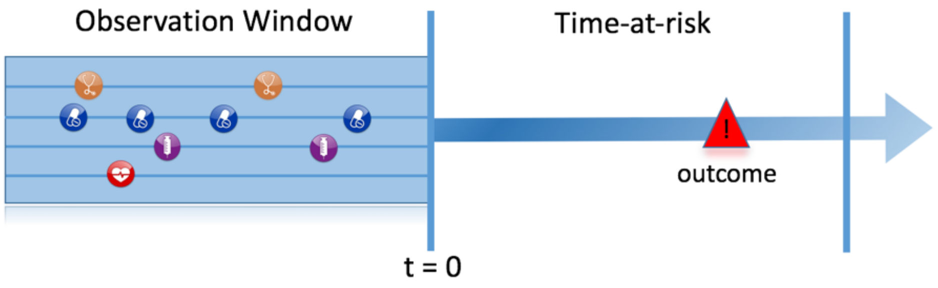


Figure 1: The prediction problem

Figure 1 illustrates the prediction problem we address. Among a population at risk, we aim to predict which patients at a defined moment in time ($t = 0$) will experience some outcome during a time-at-risk. Prediction is done using only information about the patients in an observation window prior to that moment in time.

To develop a model the user needs to take the following steps:

a. create the at risk and outcome cohorts in the CDM

b. extract the patient-level data from the server

c. define the population of interest (e.g., first time outcome)

d. define the time-at-risk (e.g., within 1 year)

e. pick a test/train split (e.g. 20% test: 80% train splitting by time)

f. create model settings

g. fit the model

h. evaluate the model

i. apply the model

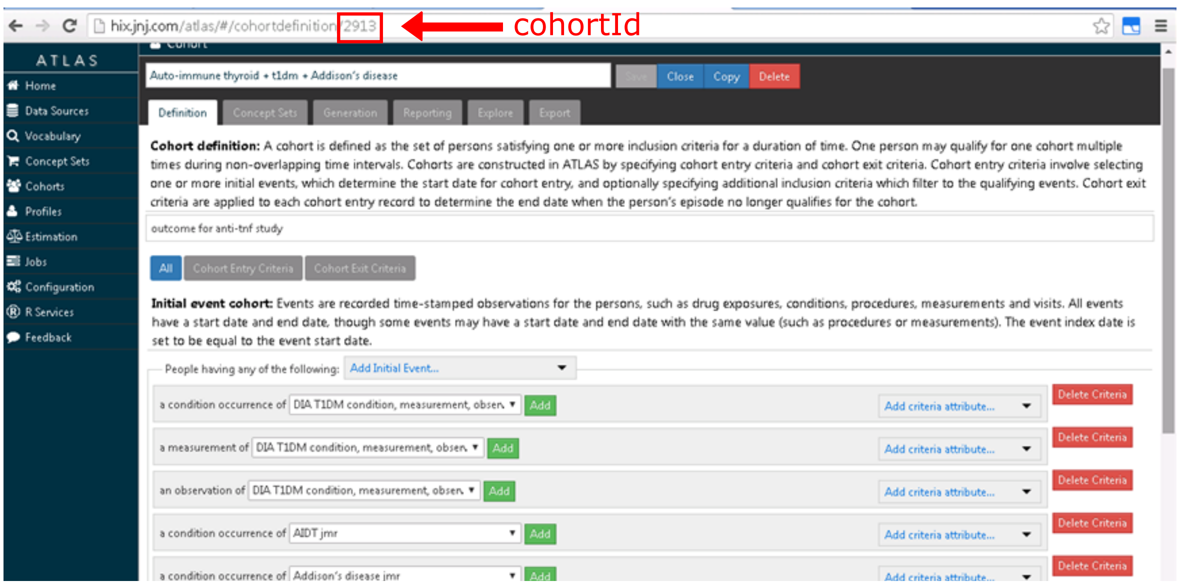
This document describes all these steps in detail

2 Cohort Creation

The first step is the creation of the outcome and at-risk cohorts in the OMOP CDM. The observational and health data sciences & informatics (OHDSI) community has developed a tool named Atlas which can be used to create cohorts based on user inclusion/exclusion criteria. However, occasionally you may need to create your own cohort definition using SQL. Both methods are described below.

2.1 Example using Atlas

The patient level prediction package enables you to use cohorts created in Atlas. Details of the creation of cohorts in Atlas can be found elsewhere. When a cohort is created in Atlas the cohortid (see the link as shown in the figure below) will be required to extract the patient-level data.



Auto-immune thyroid + t1dm + Addison's disease

Cohort definition: A cohort is defined as the set of persons satisfying one or more inclusion criteria for a duration of time. One person may qualify for one cohort multiple times during non-overlapping time intervals. Cohorts are constructed in ATLAS by specifying cohort entry criteria and cohort exit criteria. Cohort entry criteria involve selecting one or more initial events, which determine the start date for cohort entry, and optionally specifying additional inclusion criteria which filter to the qualifying events. Cohort exit criteria are applied to each cohort entry record to determine the end date when the person's episode no longer qualifies for the cohort.

outcome for anti-tnf study

Initial event cohort: Events are recorded time-stamped observations for the persons, such as drug exposures, conditions, procedures, measurements and visits. All events have a start date and end date, though some events may have a start date and end date with the same value (such as procedures or measurements). The event index date is set to be equal to the event start date.

People having any of the following:

a condition occurrence of DIA T1DM condition, measurement, obser.

a measurement of DIA T1DM condition, measurement, obser.

an observation of DIA T1DM condition, measurement, obser.

a condition occurrence of AIDT jmr

a condition occurrence of Addison's disease jmr

Figure 2: Using Atlas to create the cohorts

For the purpose of the example we will assume we have created our risk cohort has with an id of: 2913 and our outcome cohort has with an id of:1913. You can see in figure 2.1 that the number at the end of the url is the cohort_definition_id.

2.2 Example using custom cohort code

Occasionally you may need to create your own cohorts. The cohorts should be provided as data in a table on the server, where the table should have the same structure as the cohort table in the OMOP CDM, meaning it should have the following columns:

- cohort_definition_id (CDM $v5+$), a unique identifier for distinguishing between different types of cohorts, e.g. cohorts of interest and outcome cohorts.
- subject_id, a unique identifier corresponding to the person_id in the CDM.
- cohort_start_date, the start of the time period where we wish to predict the occurrence of the outcome.
- cohort_end_date, which can be used to determine the end of the prediction window. Can be NULL for outcomes.

For example, imagine you have a cohort of pregnant women with a cohort_start_date corresponding to the conception date, a cohort_definition_id of 1, the pregnancy cohort table is called 'pregnancy' and it is, for example, in the 'writeable.dbo' schema. For example, you also have a cohort corresponding to gestational diabetes with the cohort_start_date being the first gestational diabetes record date, the cohort_definition_id is 1, the gestational diabetes table is 'gdm' and it is in the 'madeup.dbo' schema. To run the patient level prediction package using these cohort tables you need to extract the data by referencing the correct tables and databases (in the continuation of this example below we also chose to remove people who have less that 180 days observation prior to cohort_start_date):

2.3 Extracting data

First we need to load the patient level prediction library and set the location where you want to save results:

```
library(DatabaseConnector)
library(FeatureExtraction)
library(PatientLevelPrediction)
wd <- file.path("C:/Users/jreps/Documents", 'patientprediction')
```

Next, set up the database connection:

```
databaseSchema <- 'cdm_optum_v5.dbo'

dbms <- "pdw"
user <- enterusername
pw <- enterpassword
server <- "JRDUSAPSCTL01"
port <- 17001
oracleTempSchema <- NULL
cdmVersion <- 5

connectionDetails <- createConnectionDetails(dbms = dbms,
                                        server = server,
                                        user = user,
                                        password = pw,
                                        port = port)
```

Then specify the covariates/features that will be used by the prediction algorithm. These are created using the FeatureExtraction package. For more information about these settings see the details of the package (https://github.com/OHDSI/FeatureExtraction/blob/master/extras/FeatureExtraction.pdf)

```
# covariate settings:
covSettings <- createCovariateSettings(
useCovariateCohortIdIs1 = FALSE,
useCovariateDemographics = TRUE,
useCovariateDemographicsGender = TRUE,
useCovariateDemographicsRace = TRUE,
useCovariateDemographicsEthnicity = TRUE,
useCovariateDemographicsAge = TRUE,
useCovariateDemographicsYear = TRUE,
useCovariateDemographicsMonth = TRUE,
useCovariateConditionOccurrence = TRUE,
useCovariateConditionOccurrence365d = TRUE,
useCovariateConditionOccurrence30d = FALSE,
useCovariateConditionOccurrenceInpt180d = FALSE,
useCovariateConditionEra = FALSE,
useCovariateConditionEraEver = FALSE,
useCovariateConditionEraOverlap = FALSE,
useCovariateConditionGroup = TRUE,
useCovariateConditionGroupMeddra = TRUE,
useCovariateConditionGroupSnomed = FALSE,
useCovariateDrugExposure = TRUE,
useCovariateDrugExposure365d = TRUE,
useCovariateDrugExposure30d = FALSE,
useCovariateDrugEra = FALSE,
useCovariateDrugEra365d = FALSE,
useCovariateDrugEra30d = FALSE,
useCovariateDrugEraOverlap = FALSE,
useCovariateDrugEraEver = FALSE,
useCovariateDrugGroup = TRUE,
useCovariateProcedureOccurrence = TRUE,
useCovariateProcedureOccurrence365d = TRUE,
useCovariateProcedureOccurrence30d = FALSE,
useCovariateProcedureGroup = TRUE,
useCovariateObservation = TRUE,
useCovariateObservation365d = TRUE,
useCovariateObservation30d = FALSE,
useCovariateObservationCount365d = FALSE,
useCovariateMeasurement = TRUE,
useCovariateMeasurement365d = TRUE,
useCovariateMeasurement30d = FALSE,
useCovariateMeasurementCount365d = FALSE,
useCovariateMeasurementBelow = FALSE,
useCovariateMeasurementAbove = FALSE,
useCovariateConceptCounts = FALSE,
useCovariateRiskScores = TRUE,
useCovariateRiskScoresCharlson = TRUE,
useCovariateRiskScoresDCSI = TRUE,
useCovariateRiskScoresCHADS2 = TRUE,
useCovariateRiskScoresCHADS2VASc = TRUE,
useCovariateInteractionYear = FALSE,
useCovariateInteractionMonth = FALSE,
excludedCovariateConceptIds = c(),
includedCovariateConceptIds = c(),
deleteCovariatesSmallCount = 50)
```

The final step for extracting the data is to using the getPlpData function and input the connection details, the database schema where the Atlas cohorts are stored, the cohort_definition_ids for the cohort and outcome, and the washoutPeriod which is the minimum number of days prior to cohort index date that the person must have been observed to be included into the data and finally input the previously constructed covariate settings.

```
# extract the Atlas create cohort data:
plpData <- getPlpData(connectionDetails,
                      cdmDatabaseSchema=databaseSchema,
                      cohortId=2913,
                                 outcomeIds=1913,
                      outcomeDatabaseSchema = databaseSchema,
                      outcomeTable = 'cohort',
                      cohortDatabaseSchema = databaseSchema,
                      cohortTable = 'cohort',
                      cdmVersion=5,
                      washoutPeriod=365,
                      covariateSettings=covSettings)
```

```
# extract the custom cohort data:
plpData <- getPlpData(connectionDetails,
                      cdmDatabaseSchema=databaseSchema,
                      cohortId=1,
                                 outcomeIds=1,
                      outcomeDatabaseSchema = 'madeup.dbo',
                      outcomeTable = 'gdm',
                      cohortDatabaseSchema = 'writeable.dbo',
                      cohortTable = 'pregnancy',
                      cdmVersion=5,
                      washoutPeriod=180,
                      covariateSettings=covSettings)
```

Now might be a good time to save the data.

```
# save the data:
PatientLevelPrediction::savePlpData(plpData, file=file.path(wd))

# you can load the saved data using the following code:
plpData <- PatientLevelPrediction::loadPlpData(file=file.path(wd))
```

2.4 Create population

To standardize the prediction problem and ensure we are consistently defining cohorts we want to create general cohorts that can be reused in other prediction problems. This means we do not want to create outcome cohorts that are specific to the risk cohort. As a consequence, we need to create a prediction population using the at risk cohort, the outcome cohort and the time at risk (the period of time relative to the at risk cohort dates where we look for the outcome occurrence).
After downloading the data corresponding to the cohorts, we need to define the prediction problem by creating the population (define the prediction period, the required minimum amount of prior observation time and other exclusions). Figure 2.4 illustrates the parameters used to determine whether the outcome occurs for the at risk people.

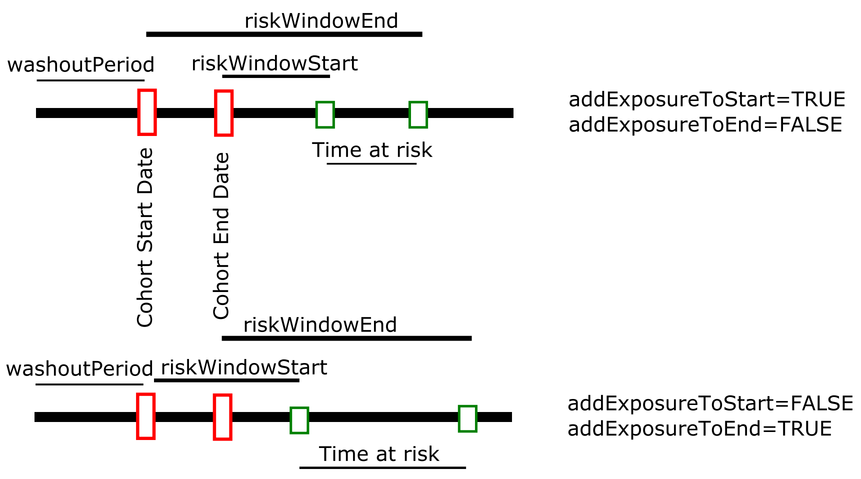


Figure 3: Example of parameters used in createStudyPopulation

```
# to create a population to predict outcome with cohort_definition_if of 1913
# where we only want first time exposure (earliest at risk cohort start date per person), a
    minimum of 365 days of prior observation time (washoutPeriod) and we wish to do the
    prediction between the at risk cohort_start_date + 28 day until the cohort_end_date

population <- createStudyPopulation(plpData, outcomeId = 1913,
                                includeAllOutcomes = T,
                                firstExposureOnly = T,
                                washoutPeriod = 365,
                                    removeSubjectsWithPriorOutcome = F,
                                priorOutcomeLookback=0,
                                                riskWindowStart = 1,
                                requireTimeAtRisk = T,
                                                minTimeAtRisk = 180,
                                addExposureDaysToStart = FALSE,
                                                riskWindowEnd = 28,
                                addExposureDaysToEnd=T)
```

2.5 Create model settings

The package currently supports the following models:

- Logistic regression with regularization
- Gradient boosting machines
- Random forest
- K nearest neighbors
- Naive Bayes

with current development on these methods:

- (comming soon) Neural network
- (coming soon) Deep learning

Each method has its own set of hyper-parameters as shown in Table 2.5.

Method	Parameters
Lasso Logistic Regression	var (starting variance)
Random forest	mtry (number of features in each tree), ntree (number of trees), max_depth (max levels in tree), min_rows (minimum data points in in node), balance (balance class labels)
Gradient boosting machine	ntree (number of trees), max_depth (max levels in tree), min_rows (minimum data points in in node), learning rate
KNN	k (number of neighbours), weighted (weight by inverse frequency)
Naive Bayes	-

The first step of developing the model is to specify the model settings such as the machine learning algorithm and the hyper-parameter search. In the settings function the user can input a range of hyper-parameter values that get used to construct a grid containing every combination of user specified hyper-parameters as a list. This combination of hyper-parameters is then searched during the model training to pick the optimal hyper-parameter combination out of combinations provided. If a user does not specify any hyper-parameter value then the default value is used instead.
For example if we use the following settings for the gradientBoostingMachine: ntrees=c(100,200), max_depth=4 the grid search will investigate the gradient boosting machine algorithm with ntrees=100 and max_depth=4 plus the default settings for other hyper-parameters and ntrees=200 and max_depth=4 plus the default settings for other hyper-parameters. The hyper-parameters that lead to the best cross-validation performance will then be chosen for the final model.
To run a logistic regression with lasso regularization run:

```r
# create the logistic regression settings:
lr_model <- PatientLevelPrediction::setLassoLogisticRegression()
```

To run a gradient boosting model run:

```r
# try gradient boosting machine with grid search testing number of trees of 100,150
# and 200 and max depth of 3,4,5,6 or 10 levels.
gbm_model <- setGradientBoostingMachine(
                       ntrees=c(100,150,200),
                                   max_depth = c(3,4,5,6,10))
```

2.6 Run model development

Once the modelsettings are created we can run the model development using the runPlp function. Below some examples are given.

```
# now run the RunPlp function
# inputting the plpData, population, model settings
# as well as the type of evaluation and test/train split fraction and
# number of folds used during during cross validation
# using a time split (train set is older data and test set is later data)
lr_results <- RunPlp(population, plpData,
                        modelSettings = lr_model,
                        testSplit='time', # this splits by time
                        testFraction=0.25,
                        nfold=2)

# for the gradient boosting (splitting the test/train sets by person - stratified
# by outcome so that the same proportion of people with the outcome are
# in the test and train sets):
gbm_results <- RunPlp(population, plpData,
                         modelSettings = gbm_model,
                         testSplit='person', # this splits by person
                         testFraction=0.25,
                         nfold=2)
```

```
# try random forest with grid search testing number of trees of 100,150
# and 200 and mtry is default of -1 (square root of total number of features).
rf_model <- setRandomForest.(ntrees=c(100,150,200), mtry=-1)
rf_results <- RunPlp(population, plpData,
                  modelSettings = rf_model,
                  testSplit='time',
                  testFraction=0.25,
                  nfold=2),
```

```
# try naive bayes model
# and 200 and mtry is default of -1 (square root of total number of features).
nb_model <- setNaiveBayes()
nb_results <- RunPlp(population, plpData,
                  modelSettings = nb_model,
                  testSplit='time',
                  testFraction=0.25,
                  nfold=2),
```

2.7 Saving/Loading Results

You can save and load a model by running:

```
# save the model to the folder model in the current working directory
savePlpModel(lr_results$model, dirPath=file.path(getwd(),'model'))

# load the model
plpModel <- loadPlpModel(getwd(),'model'))
```

You can load/save the test/train performance by running:

```
# save the results of runPlp in the current working directory
savePlpResult(lr_results, location=file.path(getwd(),'lrmodel'))

# load the results performance
lr_results_saved <- loadPlpResult(getwd(),'lrmodel'))
```

2.8 Viewing Results

The RunPlp() function returns the trained model and the evaluation of the model on the train/test sets. To generate plots run the following code:

```
# plot the results into a folder named plots within the current working directory
plotPlp(lr_results_saved, filename=file.path(getwd(),'plots'))
```

2.8.1 ROC plot

The ROC plot plots the sensitivity against 1-specificity. This shows how well the model is able to discriminate between the people with the outcome and those without. The plot is done by applying th model to the test set (internal validation) and the diagonal dashed line shows the performance of a model that randomly assigns predictions. The area under the ROC plot is in blue.

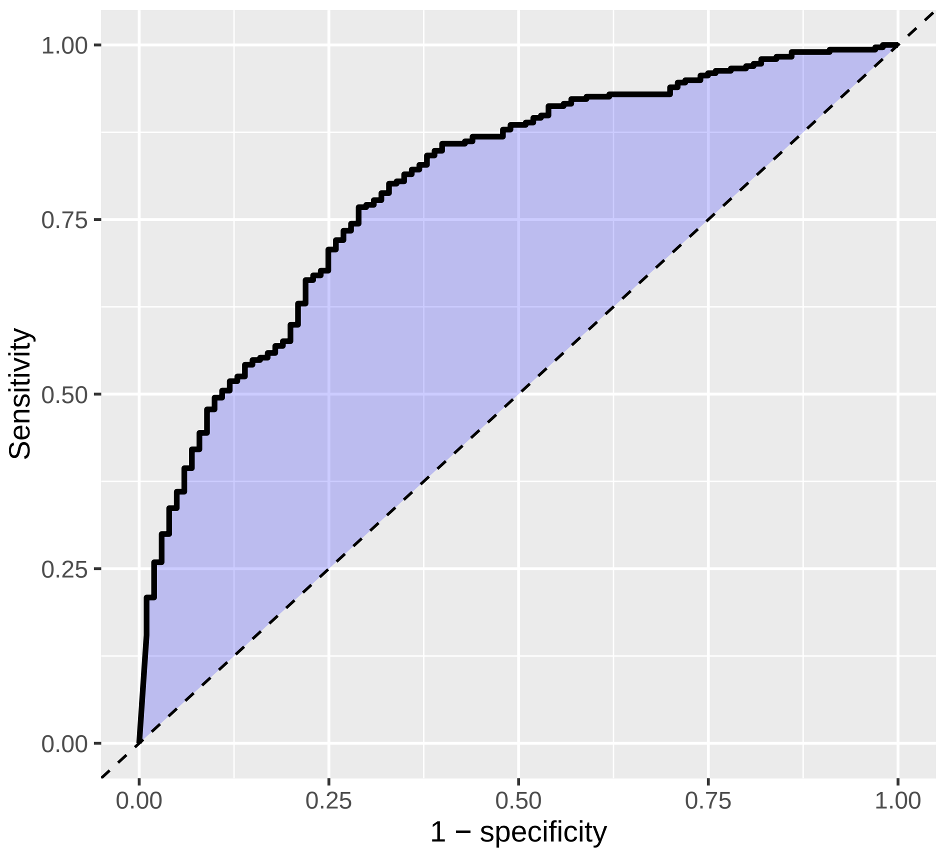


Figure 4: ROC Plot

2.8.2 Calibration plot

The calibration plot plots the observed risk vs the predicted risk for the model's predictions on the test set split into 10 quantiles. The dashed line is the x=y line which would indicate a perfectly calibrated model. The ten (or fewer) dots represent the mean predicted values for each quantile plotted against the observed fraction of people in that quantile who had the outcome (observed fraction) and the straight black line is the linear regression using these 10 plotted quantile mean predicted vs observed fraction points. The two blue straight lines represented the 95% lower and upper confidence intervals for the linear regression.

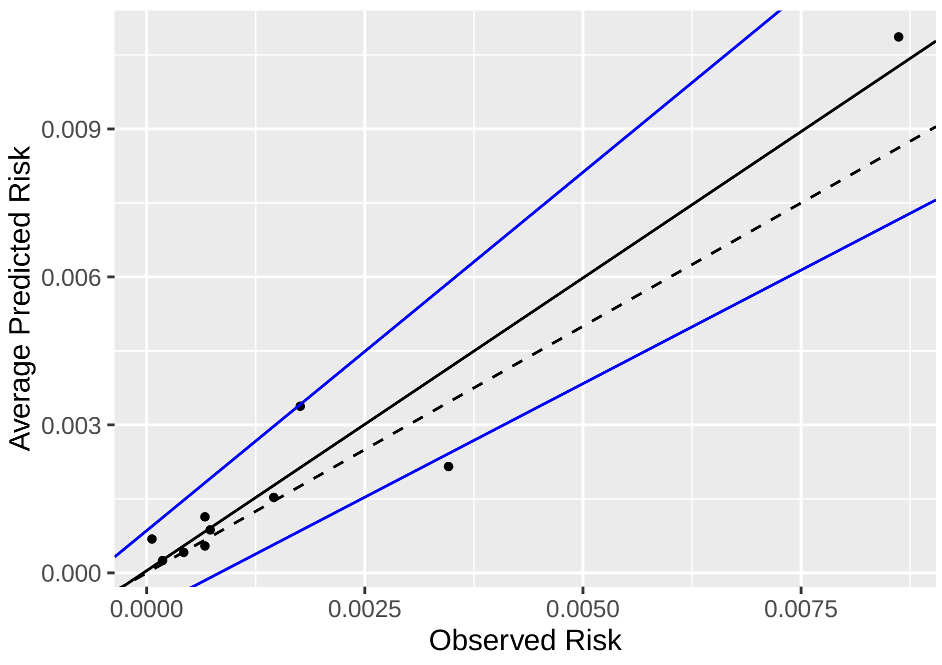


Figure 5: Calibration Plot

2.8.3 Preference distribution plots

The preference distribution plots are the preference score distributions corresponding to i) people in the test set with the outcome (red) and ii) people in the test set without the outcome (blue).

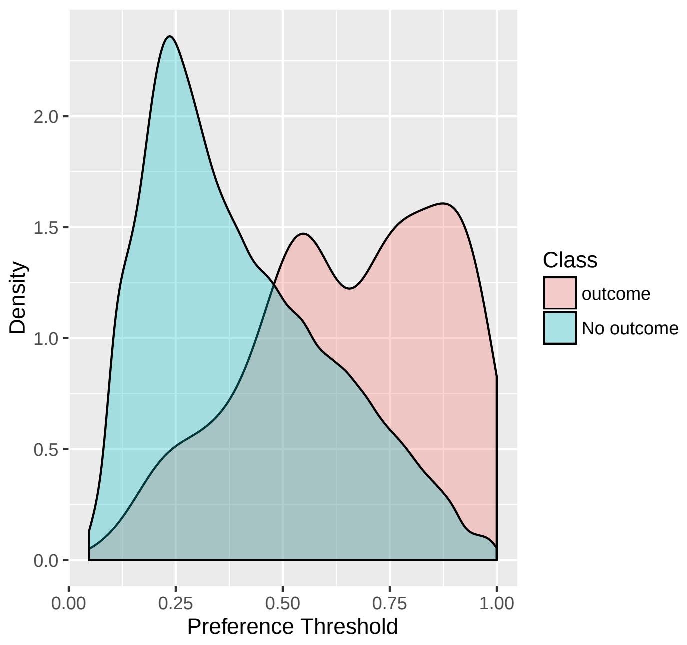


Figure 6: Preference Distributions

2.8.4 Boxplots

The predictionDistribution boxplots are box plots for the predicted risks of the people in the test set with the outcome (class 1: blue) and without the outcome (class 0: red).

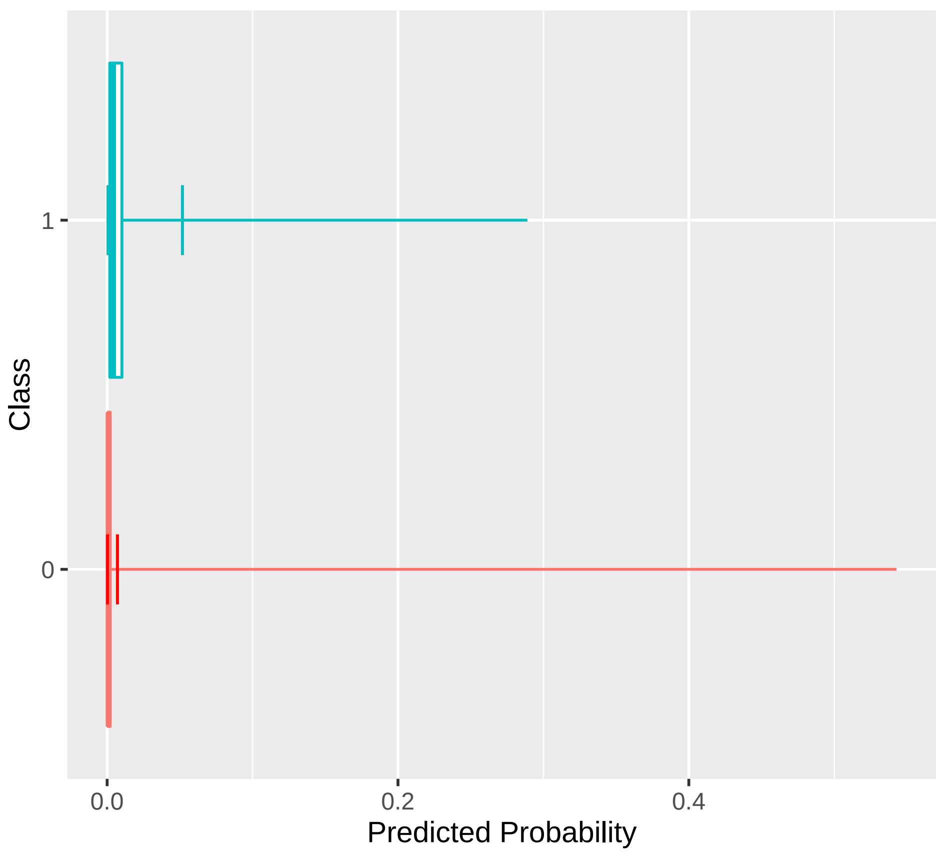


Figure 7: Prediction Box Plots

2.8.5 Test/Train similarity plot

The test train similarity is presented in the generalization plots where the mean covariate values are plotted in the train set vs the test set for people with and without the outcome.

2.8.6 Variable scatterplot

The variable scatter plot plots the mean covariate value for the people with the outcome against the mean covariate value for the people without the outcome. The size and color of the dots corresponds to the importance of the covariates in the trained model.

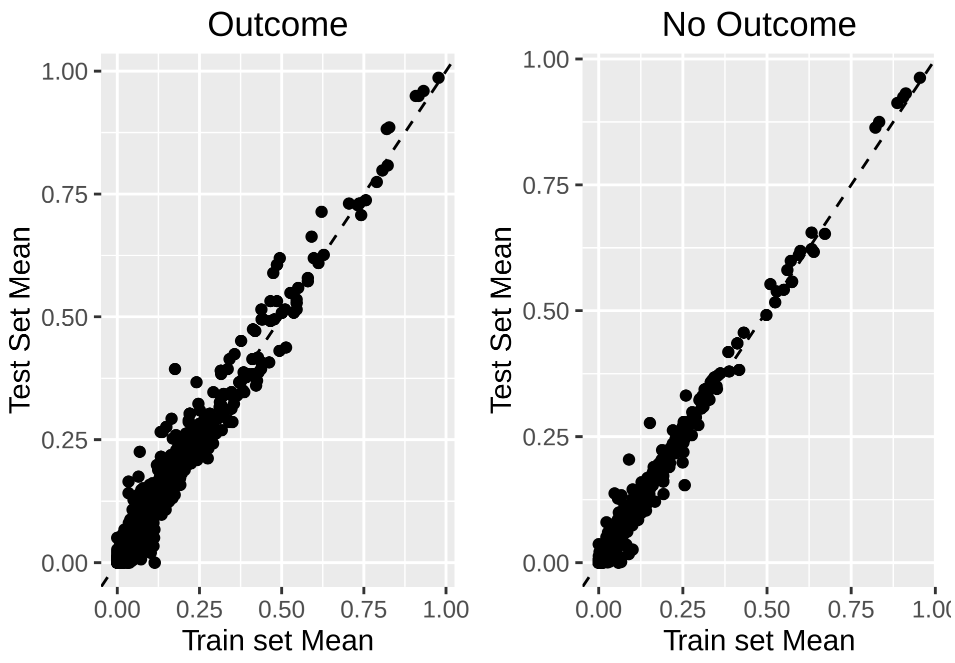


Figure 8: Generalization Plot

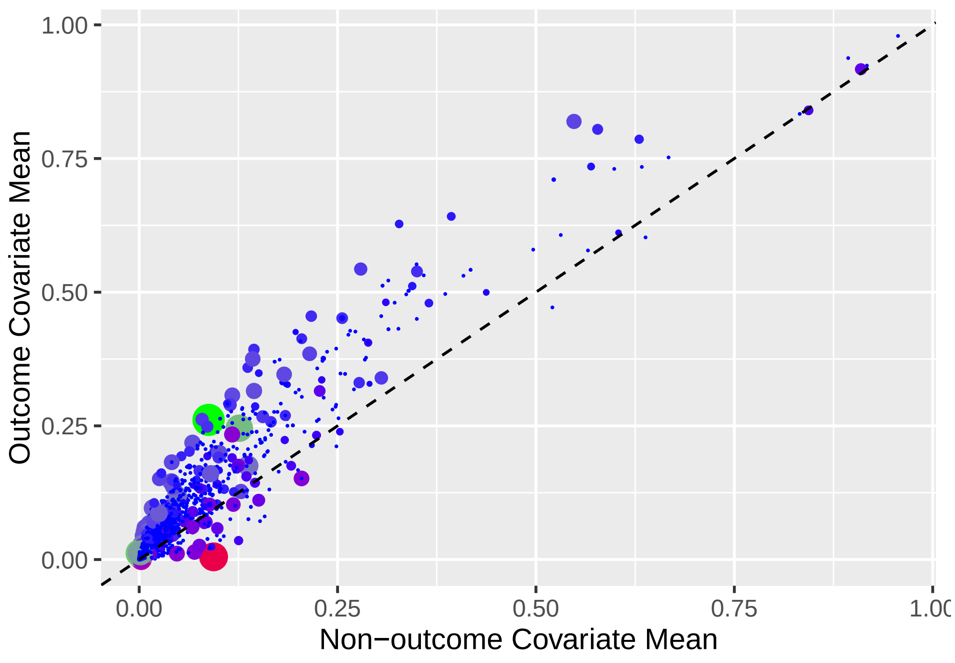


Figure 9: Variable Scatterplot

3 Applying the model

Once you have a model trained using the RunPlp() function, you can then apply the model to new data by running:

```r
# load the trained model:
plpModel <- loadPlpModel(getwd(),'model'))

# load the new plpData and create the population:
plpData <- loadPlpData(getwd(),'data'))
population <- createStudyPopulation(plpData, outcomeId = 1913,
                                firstExposureOnly = T,
                                washoutPeriod = 365,
                                    removeSubjectsWithPriorOutcome = F,
                                priorOutcomeLookback=0,
                                                riskWindowStart = 1,
                                requireTimeAtRisk = T,
                                                minTimeAtRisk = 180,
                                addExposureDaysToStart = FALSE,
                                                riskWindowEnd = 28,
                                addExposureDaysToEnd=T)

# now apply the trained model on the new data:
validationResults <- applyModel(population, plpData, plpModel)

# this will return a list containing the prediction on the population
# and the performance (if the outcome if known for the plpData)
```

4 Extras

4.1 custom machine learning models

There is the flexibility within the package of adding your own machine learning models. You need to specify the model function, the model settings function and the prediction function.
An example template for creating a novel method can be found in the ?.R file within the PatientLevel-Prediction r package.

4.1.1 model setting function

This function takes as input the hyper-parameter values to construct a grid search and returns the settings ready to be used by the model function. The output of this function is a list containing:

- model - a string specifying the name of the model function (e.g., 'lasso_lr' corresponds to the regularized logistic regression)
- param - a list, with each element containing one of the hyper-parameter combinations to test during the grid search.

the output should be of class 'modelSettings' and have an attribute named libSVM with is binary and specifies whether the model requires libSVM format.

4.1.2 model function

To add your own model you need to create a function that takes as input:

- population
- plpData
- param
- index
- quiet
- outcomeId
- cohortId

The plpData data used to train and evaluate the model is restricted to those in the population, param is a list containing the hyper-parameters combinations that will be searched during model training (this gets created by the model settings function you write)
and the output of this function must be a list containing:

- model (the model function name as a string)
- modelSettings (a list of the settings input into the model training)
- trainCVAuc (the cross validation performance on the train set)
- metaData (the plpData metaData)
- populationSettings (the settings of the population)
- outcomeId (being predicted)
- cohortId (which cohort is the prediction run on)
- varImp (variable importance)
- trainingTime (time it took to train the model)

In addition, the result list should have a class of 'plpModel', and attribute type of 'plp' and an attribute predictionType of 'binary':

```
class(result) <- 'plpModel'
attr(result, 'type') <- 'plp'
attr(result, 'predictionType') <- 'binary'
```

For examples of default models see the defaultModels.R code within the patientLevelPrediction package.

4.1.3 prediction function

The prediction function needs to take as input:

- plpModel (the trained model)
- plpData (patient level prediction format data)
- population (patient level prediction population data.frame)
- silent (binary whether the progress should be reported)

The prediction function should then use these inputs to make a prediction for the population using the plpData and plpModel and returns a data.frame containing 'rowId', 'outcomeCount' , 'indexes', 'value' where rowId is the id used to reference a person in the population, outcomeCount is the true label (when applicable), indexes is a column indicating whether a rowId was used in the test/train set or the cross validation fold number (when applicable) where the data points in the test set have a value of -1 and the data points in the train set have their fold id and value is the prediction value between 0 and 1. The output should have the same number of rows as the input population. The data.frame should have an attribute of metaData specifying the predictionType (binary/multiclass):

```
attr(prediction, 'metaData') <- list(predictionType='binary')
```

Examples of prediction functions for the defaul models can be found in the Predict.R file within the PatientLevelPrediction package.