

# Building patient-level predictive models

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

*2018-09-09*

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Installation instructions</b>	<b>2</b>
<b>3</b>	<b>Data extraction</b>	<b>2</b>
3.1	Configuring the connection to the server . . . . .	3
3.2	Preparing the cohort and outcome of interest . . . . .	3
3.3	Extracting the data from the server . . . . .	6
3.4	Saving the data to file . . . . .	7
<b>4</b>	<b>Applying additional inclusion criteria</b>	<b>7</b>
<b>5</b>	<b>Model Development</b>	<b>8</b>
5.1	Defining the model settings . . . . .	8
5.2	Model training . . . . .	8
5.3	Saving and loading . . . . .	10
<b>6</b>	<b>Model Evaluation</b>	<b>11</b>
6.1	ROC plot . . . . .	11
6.2	Calibration plot . . . . .	11
6.3	Smooth Calibration plot . . . . .	11
6.4	Preference distribution plots . . . . .	14
6.5	Box plots . . . . .	14
6.6	Test-Train similarity plot . . . . .	14
6.7	Variable scatter plot . . . . .	16
6.8	Plot Precision Recall . . . . .	17
6.9	Demographic Summary plot . . . . .	19
<b>7</b>	<b>External validation</b>	<b>20</b>
<b>8</b>	<b>Acknowledgments</b>	<b>21</b>

## 1 Introduction

This vignette describes how you can use the `PatientLevelPrediction` package to build patient-level predictive models. The 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).

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:

1. Create the target population and outcome cohorts

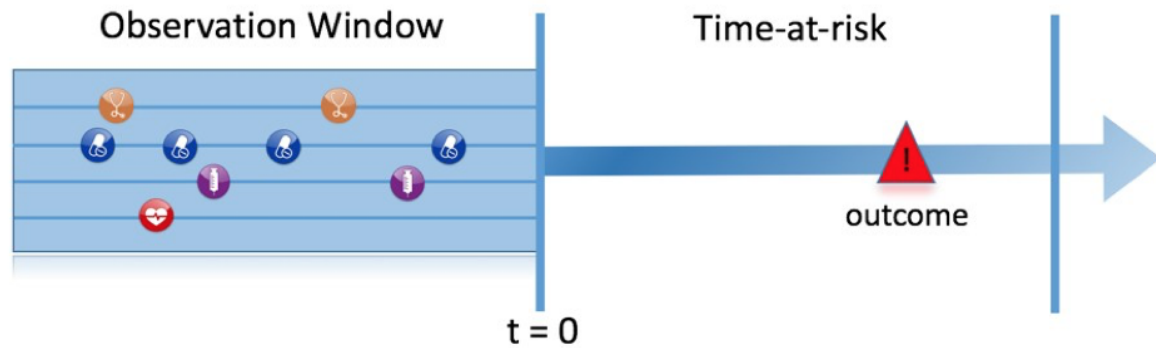


Figure 1: The prediction problem

2. Define the covariates (aka features/independent variables) to use in the model
3. Extract the patient-level data from the server
4. Define the population of interest (this defines the time-at-risk period to create the class labels)
5. Pick a test/train split
6. Create model settings
7. Fit the model
8. Evaluate the model
9. Apply the model

We have selected the well-studied topic of predicting re-hospitalization to walk you through these steps. The model will be developed for a diabetes type 2 population.

## 2 Installation instructions

Before installing the `PatientLevelPrediction` package make sure you have Java available. Java can be downloaded from [www.java.com](http://www.java.com). For Windows users, RTools is also necessary. RTools can be downloaded from CRAN.

Furthermore, a python installation is required for some of the machine learning algorithms. We advise to install Python 3 using Anaconda (<https://www.continuum.io/downloads>). If you are using Mac or Linux make sure the anaconda python is your default python (the python that runs then you type python in the command line) - you may need to add it to your path before the default python. If you edit the path, close and restart R as the updated path will not happen until you re-open R.

The `PatientLevelPrediction` package is currently maintained in a GitHub repository (<https://github.com/OHDSI/PatientLevelPrediction>), and has dependencies on other packages in Github. All of these packages can be downloaded and installed from within R using the `drat` package:

```
r install.packages("drat") drat::addRepo("OHDSI") install.packages("PatientLevelPrediction")
```

Once installed, you can type `library(PatientLevelPrediction)` to load the package.

We recommend testing the installation by running: `checkPlpInstallation()`

## 3 Data extraction

The `PatientLevelPrediction` package requires longitudinal observational healthcare data in the OMOP Common Data Model format. The user will need to specify two things:

1. Time periods for which we wish to predict the occurrence of an outcome. We will call this the **target population cohort of interest** or cohort for short. One person can have multiple time periods, but time periods should not overlap.
2. Outcomes for which we wish to build a predictive model.

The first step in running the `PatientLevelPrediction` is extracting all necessary data from the database server holding the data in the CDM.

### 3.1 Configuring the connection to the server

We need to tell R how to connect to the server where the data are. `PatientLevelPrediction` uses the `DatabaseConnector` package, which provides the `createConnectionDetails` function. Type `?createConnectionDetails` for the specific settings required for the various database management systems (DBMS). For example, one might connect to a PostgreSQL database using this code:

```
“r connectionDetails <- createConnectionDetails(dbms = “postgresql”, server = “localhost/ohdsi”, user = “joe”, password = “supersecret”)
```

```
cdmDatabaseSchema <- “my_cdm_data” cohortsDatabaseSchema <- “my_results” cdmVersion <- “5” ““
```

The last three lines define the `cdmDatabaseSchema` and `cohortsDatabaseSchema` variables, as well as the CDM version. We will use these later to tell R where the data in CDM format live, where we want to create the cohorts of interest, and what version CDM is used. Note that for Microsoft SQL Server, databaseschemas need to specify both the database and the schema, so for example `cdmDatabaseSchema <- “my_cdm_data.dbo”`.

### 3.2 Preparing the cohort and outcome of interest

First we need to define the cohort of persons for which we want to perform the prediction and we need to define the outcomes we want to predict.

The cohort and outcomes are provided as data in a table on the server that has the same structure as the ‘cohort’ table in the OMOP CDM, meaning it should have the following columns:

- `cohort_definition_id`, 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 set equal to the `cohort_start_date` for outcomes.

The observational and health data sciences & informatics (OHDSI) community has developed a tool named ATLAS which can be used to create cohorts based on inclusion criteria. We can also write custom SQL statements against the CDM.

#### 3.2.1 Cohort creation using ATLAS

ATLAS as shown in Figure 2 allows you to define cohorts interactively by specifying cohort entry 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. For the outcome cohort the end date is less relevant. More details on the use of ATLAS can be found on the OHDSI wiki pages.

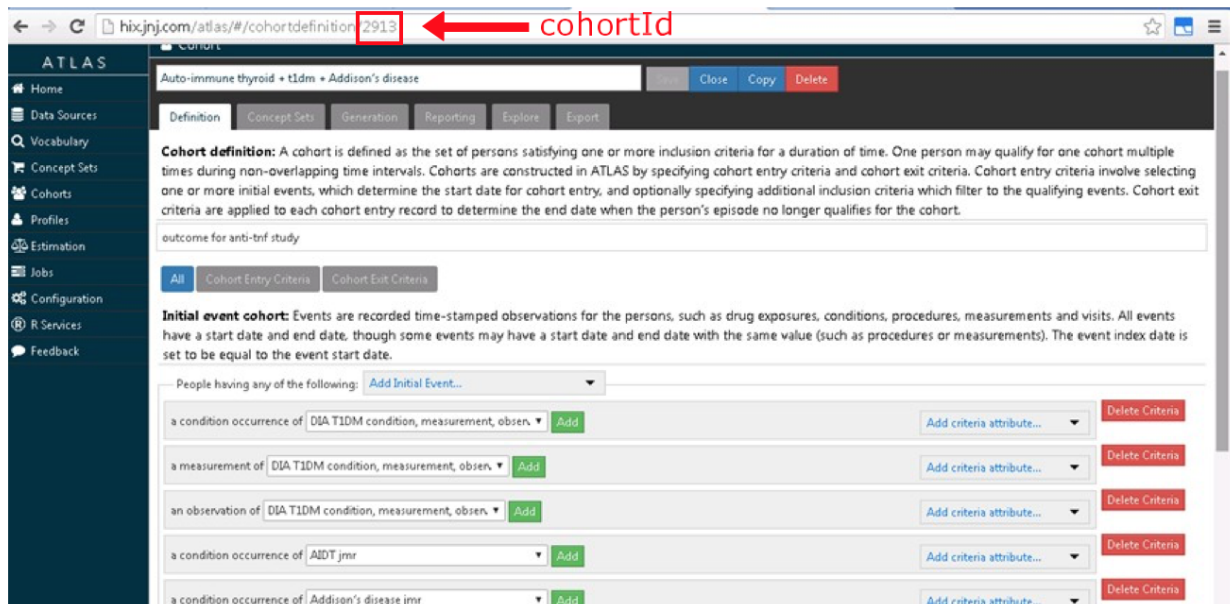


Figure 2: Cohort creation using ATLAS

When a cohort is created in ATLAS the cohortid is needed to extract the data in R. The cohortid can be found in the link as shown in Figure 2.

### 3.2.2 Custom cohort creation

It is also possible to create cohorts without the use of ATLAS. Using custom cohort code (SQL) you can make more advanced cohorts if needed.

For our example study, we need to create the cohort of diabetics that have been hospitalized and have a minimum amount of observation time available before and after the hospitalization. We also need to define re-hospitalizations, which we define as any hospitalizations occurring after the original hospitalization.

For this purpose we have created a file called *HospitalizationCohorts.sql* with the following contents:

```

/*****
File HospitalizationCohorts.sql
*****/
IF OBJECT_ID('@resultsDatabaseSchema.rehospitalization', 'U') IS NOT NULL
DROP TABLE @resultsDatabaseSchema.rehospitalization;

SELECT visit_occurrence.person_id AS subject_id,
MIN(visit_start_date) AS cohort_start_date,
DATEADD(DAY, @post_time, MIN(visit_start_date)) AS cohort_end_date,
1 AS cohort_definition_id
INTO @resultsDatabaseSchema.rehospitalization
FROM @cdmDatabaseSchema.visit_occurrence
INNER JOIN @cdmDatabaseSchema.observation_period
ON visit_occurrence.person_id = observation_period.person_id
INNER JOIN @cdmDatabaseSchema.condition_occurrence
ON condition_occurrence.person_id = visit_occurrence.person_id
WHERE visit_concept_id IN (9201, 9203)

```

```

AND DATEDIFF(DAY, observation_period_start_date, visit_start_date) > @pre_time
AND visit_start_date > observation_period_start_date
AND DATEDIFF(DAY, visit_start_date, observation_period_end_date) > @post_time
AND visit_start_date < observation_period_end_date
AND DATEDIFF(DAY, condition_start_date, visit_start_date) > @pre_time
AND condition_start_date <= visit_start_date
AND condition_concept_id IN (
SELECT descendant_concept_id
FROM @cdmDatabaseSchema.concept_ancestor
WHERE ancestor_concept_id = 201826) /* Type 2 DM */
GROUP BY visit_occurrence.person_id;

INSERT INTO @resultsDatabaseSchema.rehospitalization
SELECT visit_occurrence.person_id AS subject_id,
visit_start_date AS cohort_start_date,
visit_end_date AS cohort_end_date,
2 AS cohort_definition_id
FROM @resultsDatabaseSchema.rehospitalization
INNER JOIN @cdmDatabaseSchema.visit_occurrence
ON visit_occurrence.person_id = rehospitalization.subject_id
WHERE visit_concept_id IN (9201, 9203)
AND visit_start_date > cohort_start_date
AND visit_start_date <= cohort_end_date
AND cohort_definition_id = 1;

```

This is parameterized SQL which can be used by the `SqlRender` package. We use parameterized SQL so we do not have to pre-specify the names of the CDM and result schemas. That way, if we want to run the SQL on a different schema, we only need to change the parameter values; we do not have to change the SQL code. By also making use of translation functionality in `SqlRender`, we can make sure the SQL code can be run in many different environments.

```

library(SqlRender)
sql <- readSql("HospitalizationCohorts.sql")
sql <- renderSql(sql,
  cdmDatabaseSchema = cdmDatabaseSchema,
  cohortsDatabaseSchema = cohortsDatabaseSchema,
  post_time = 30,
  pre_time = 365)$sql
sql <- translateSql(sql, targetDialect = connectionDetails$dbms)$sql

connection <- connect(connectionDetails)
executeSql(connection, sql)

```

In this code, we first read the SQL from the file into memory. In the next line, we replace four parameter names with the actual values. We then translate the SQL into the dialect appropriate for the DBMS we already specified in the `connectionDetails`. Next, we connect to the server, and submit the rendered and translated SQL.

If all went well, we now have a table with the events of interest. We can see how many events per type:

```

sql <- paste("SELECT cohort_definition_id, COUNT(*) AS count",
  "FROM @cohortsDatabaseSchema.rehospitalization",
  "GROUP BY cohort_definition_id")
sql <- renderSql(sql, cohortsDatabaseSchema = cohortsDatabaseSchema)$sql
sql <- translateSql(sql, targetDialect = connectionDetails$dbms)$sql

```

```
querySql(connection, sql)
```

```
## cohort_definition_id count
## 1                    1 527616
## 2                    2 221555
```

### 3.3 Extracting the data from the server

Now we can tell `PatientLevelPrediction` to extract all necessary data for our analysis. This is done using the `FeatureExtractionPackage` available at <https://github.com/OHDSI/FeatureExtraction>. In short the `FeatureExtractionPackage` allows you to specify which features (covariates) need to be extracted, e.g. all conditions and drug exposures. It also supports the creation of custom covariates. For more detailed information on the `FeatureExtraction` package see its vignettes.

```
covariateSettings <- createCovariateSettings(useDemographicsGender = TRUE,
useDemographicsAge = TRUE, useDemographicsAgeGroup = TRUE,
useDemographicsRace = TRUE, useDemographicsEthnicity = TRUE,
useConditionOccurrenceLongTerm = TRUE,
useDrugExposureLongTerm = TRUE,
useProcedureOccurrenceLongTerm = TRUE,
useMeasurementLongTerm = TRUE,
useObservationLongTerm = TRUE,
useDistinctConditionCountLongTerm = TRUE,
useVisitCountLongTerm = TRUE,
longTermStartDays = -365,
endDays = -1)
```

The final step for extracting the data is to run the `getPlpData` function and input the connection details, the database schema where the 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.

```
plpData <- getPlpData(connectionDetails = connectionDetails,
cdmDatabaseSchema = cdmDatabaseSchema,
oracleTempSchema = oracleTempSchema,
cohortDatabaseSchema = cohortsDatabaseSchema,
cohortTable = "rehospitalization",
cohortId = 1,
washoutPeriod = 183,
covariateSettings = covariateSettings,
outcomeDatabaseSchema = cohortsDatabaseSchema,
outcomeTable = "rehospitalization",
outcomeIds = 2,
cdmVersion = cdmVersion)
```

Note that if the cohorts are created in ATLAS the corresponding database schemas need to be selected. There are many additional parameters for the `getPlpData` function which are all documented in the `PatientLevelPrediction` manual. The resulting `plpData` object uses the package `ff` to store information in a way that ensures R does not run out of memory, even when the data are large.

### 3.4 Saving the data to file

Creating the `plpData` object can take considerable computing time, and it is probably a good idea to save it for future sessions. Because `plpData` uses `ff`, we cannot use R's regular save function. Instead, we'll have to use the `savePlpData()` function:

```
savePlpData(plpData, "rehosp_plp_data")
```

We can use the `loadPlpData()` function to load the data in a future session.

## 4 Applying additional inclusion criteria

To completely define the prediction problem the final study population is obtained by applying additional constraints on the two earlier defined cohorts, e.g., a minimum time at risk can be enforced (`requireTimeAtRisk`, `minTimeAtRisk`). In this step it is also possible to redefine the risk window based on the at-risk cohort. For example, if we like the risk window to start 30 days after the at-risk cohort start and end a year later we can set `riskWindowStart = 30` and `riskWindowEnd = 365`. In some cases the risk window needs to start at the cohort end date. This can be achieved by setting `addExposureToStart = TRUE` which adds the cohort (exposure) time to the start date.

In the example below a final population is created using an additional constraint on the washout period, removal of patients with prior outcomes in the year before, and a time at risk definition.

```
population <- createStudyPopulation(plpData,
  outcomeId = 2,
  includeAllOutcomes = TRUE,
  firstExposureOnly = TRUE,
  washoutPeriod = 365,
  removeSubjectsWithPriorOutcome = TRUE,
  priorOutcomeLookback = 365,
  riskWindowStart = 1,
  requireTimeAtRisk = FALSE,
  riskWindowEnd = 365)
```

Note that some of these constraints could also already be applied in the cohort creation step, however, the `createStudyPopulation` function allows you do sensitivity analyses more easily on the already extracted `plpData` from the database.

## 5 Model Development

### 5.1 Defining the model settings

Models	Python	Parameters
Logistic regression with regularization	No	var (starting variance), seed
Gradient boosting machines	No	ntree (number of trees), max depth (max levels in tree), min rows (minimum data points in in node), learning rate, seed
Random forest	Yes	mtry (number of features in each tree), ntree (number of trees), maxDepth (max levels in tree), minRows (minimum data points in in node), balance (balance class labels), seed
K-nearest neighbors	No	k (number of neighbours), weighted (weight by inverse frequency)
Naive Bayes	Yes	none
AdaBoost	Yes	nEstimators (the maximum number of estimators at which boosting is terminated), learningRate (learning rate shrinks the contribution of each classifier by learning_rate. There is a trade-off between learningRate and nEstimators)
Decision Tree	Yes	maxDepth (the maximum depth of the tree), minSamplesSplit, minSamplesLeaf, minImpuritySplit (threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.), seed, classWeight ("Balance" or "None")
Multilayer Perception	Yes	size (the number of hidden nodes), alpha (the l2 regularisation), seed

The table shows the currently implemented algorithms and their hyper-parameters. Some of these algorithms are calling python. In the settings function of the algorithm the user can specify a list of eligible values for each hyper-parameter. All possible combinations of the hyper-parameters are included in a so-called grid search using cross-validation on the training set. If a user does not specify any value then the default value is used instead.

For example, if we use the following settings for the gradientBoostingMachine: ntrees=c(100,200), maxDepth=4 the grid search will apply the gradient boosting machine algorithm with ntrees=100 and maxDepth=4 plus the default settings for other hyper-parameters and ntrees=200 and maxDepth=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.

```
gbmModel <- setGradientBoostingMachine(ntrees = c(100, 200), maxDepth = 4)
lrModel <- setLassoLogisticRegression()
```

### 5.2 Model training

The runPLP function uses the population, plpData, and model settings to train and evaluate the model. Because evaluation using the same data on which the model was built can lead to overfitting, one uses a train-test split of the data or cross-validation. This functionality is build in the runPLP function. We can use



the `testSplit` (person/time) and `testFraction` parameters to split the data in a 75%-25% split and run the patient-level prediction pipeline:

```
lrResults <- runPlp(population,plpData, modelSettings = lrModel, testSplit = 'person',  
testFraction = 0.25, nfold = 2)
```

Under the hood the package will now use the `Cyclops` package to fit a large-scale regularized regression using 75% of the data and will evaluate the model on the remaining 25%. A results data structure is returned containing information about the model, its performance etc.

### 5.3 Saving and loading

You can save and load the model using:

```
savePlpModel(lrResults$model, dirPath = file.path(getwd(), "model"))
plpModel <- loadPlpModel(getwd(), "model")
```

You can save and load the full results structure using:

```
savePlpResult(lrResults, location = file.path(getwd(), "lr"))
lrResults <- loadPlpResult(file.path(getwd(), "lr"))
```

you can interactively view the results by running: `viewPlp(runPlp=lrResults)`.

## 6 Model Evaluation

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

```
plotPlp(lrResults, dirPath = getwd())
```

To run individual plots you can use the following functions:

```
testResults <- lrResult$performanceEvaluationTest

plotSparseRoc(testResults, "sparseROC.pdf")

plotSparseCalibration(testResults, "sparseCalibration.pdf")

plotPreferencePDF(testResults, "preferencePDF.pdf")

plotPredictionDistribution(testResults, "predictionDistribution.pdf")

plotGeneralizability(testResults, "generalizability.pdf")

plotVariableScatterplot(testResults, "variableScatterplot.pdf")

plotPrecisionRecall(testResults, "precisionRecall.pdf")

plotDemographicSummary(testResults, "demographicSummary.pdf")
```

These plots are described in more detail in the following paragraphs.

### 6.1 ROC plot

The ROC plot plots the sensitivity against 1-specificity on the test set. The plot shows how well the model is able to discriminate between the people with the outcome and those without. The dashed diagonal line is the performance of a model that randomly assigns predictions. The higher the area under the ROC plot the better the discrimination of the model.

### 6.2 Calibration plot

The calibration plot shows how close the predicted risk is to the observed risk. The diagonal dashed line thus indicates 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). 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 of the slope of the fitted line.

### 6.3 Smooth Calibration plot

Similar to the traditional calibration shown above the Smooth Calibration plot shows the relationship between predicted and observed risk. the major difference is that the smooth fit allows for a more fine grained examination of this. Whereas the traditional plot will be heavily influenced by the areas with the highest density of data the smooth plot will provide the same information for this region as well as a more accurate interpretation of areas with lower density. the plot also contains information on the distribution of the outcomes relative to predicted risk.

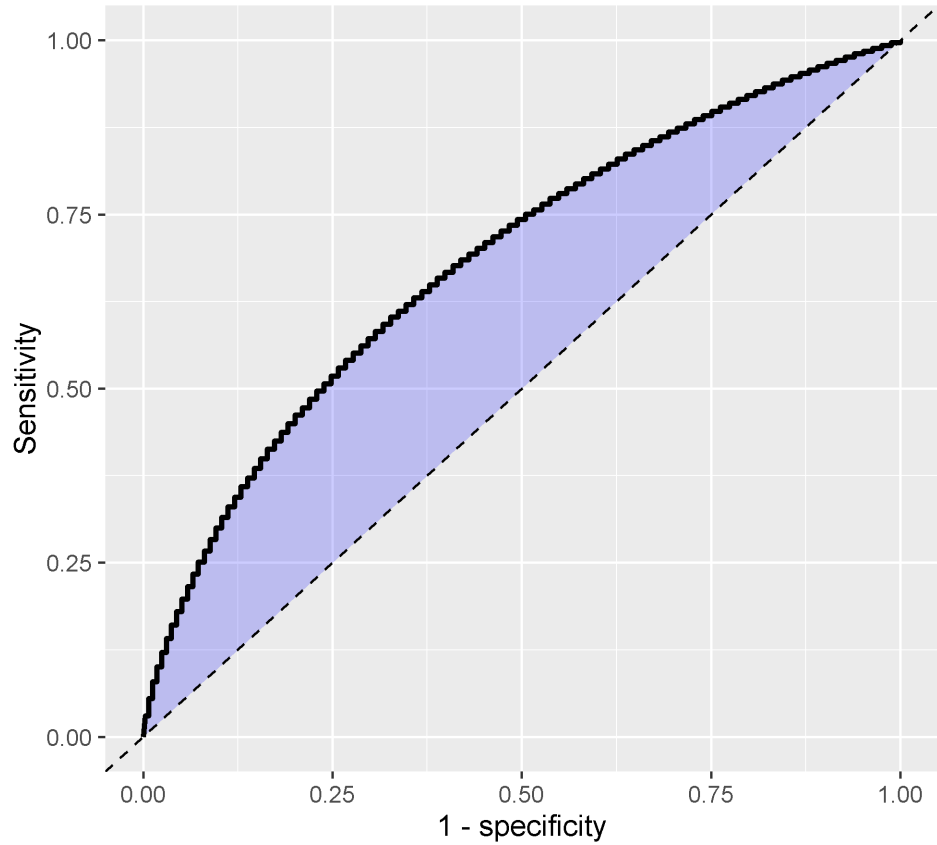


Figure 3: Receiver Operator Plot

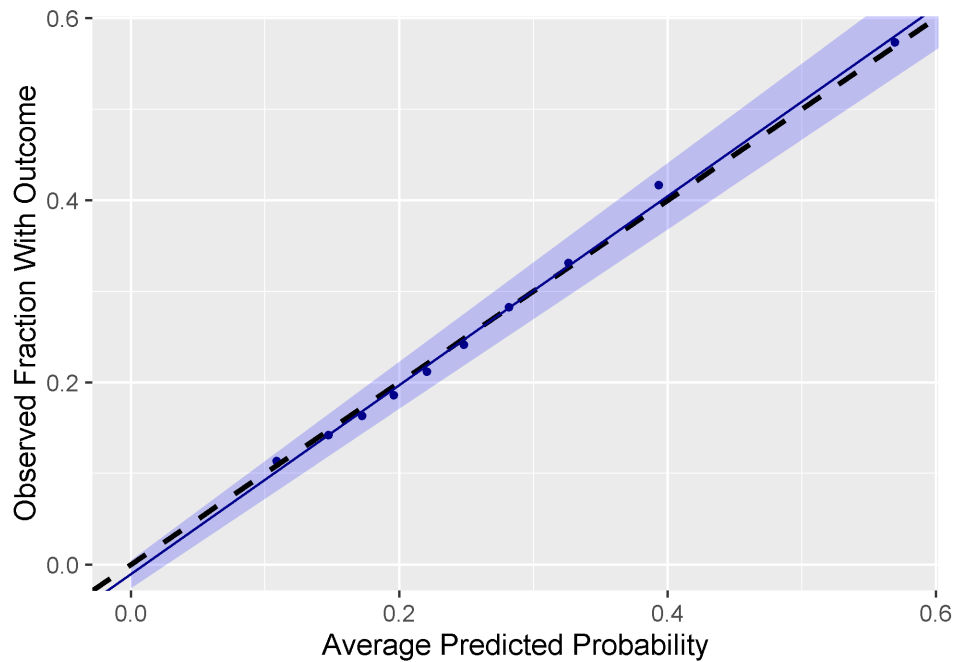


Figure 4: Calibration Plot

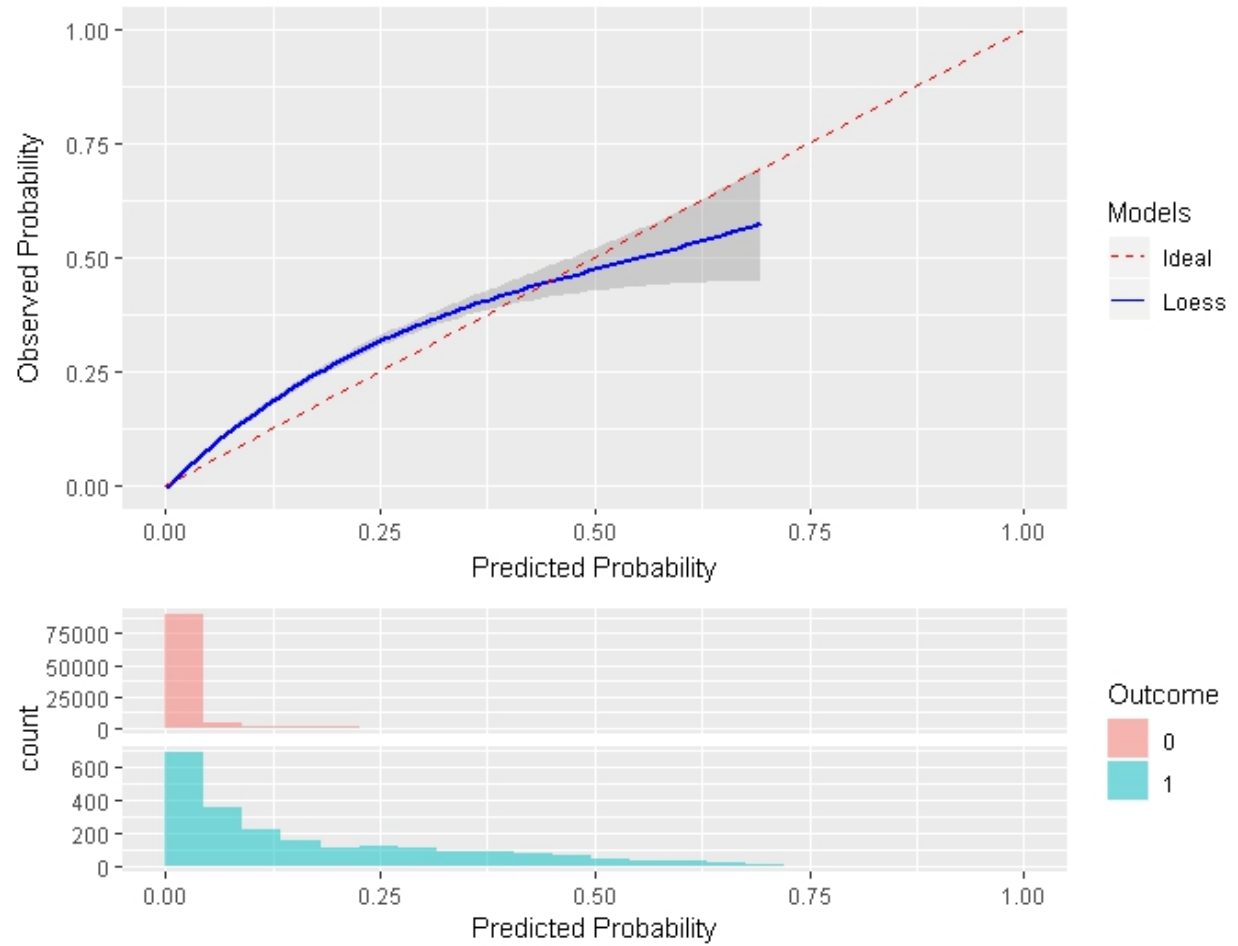


Figure 5: Smooth Calibration plot

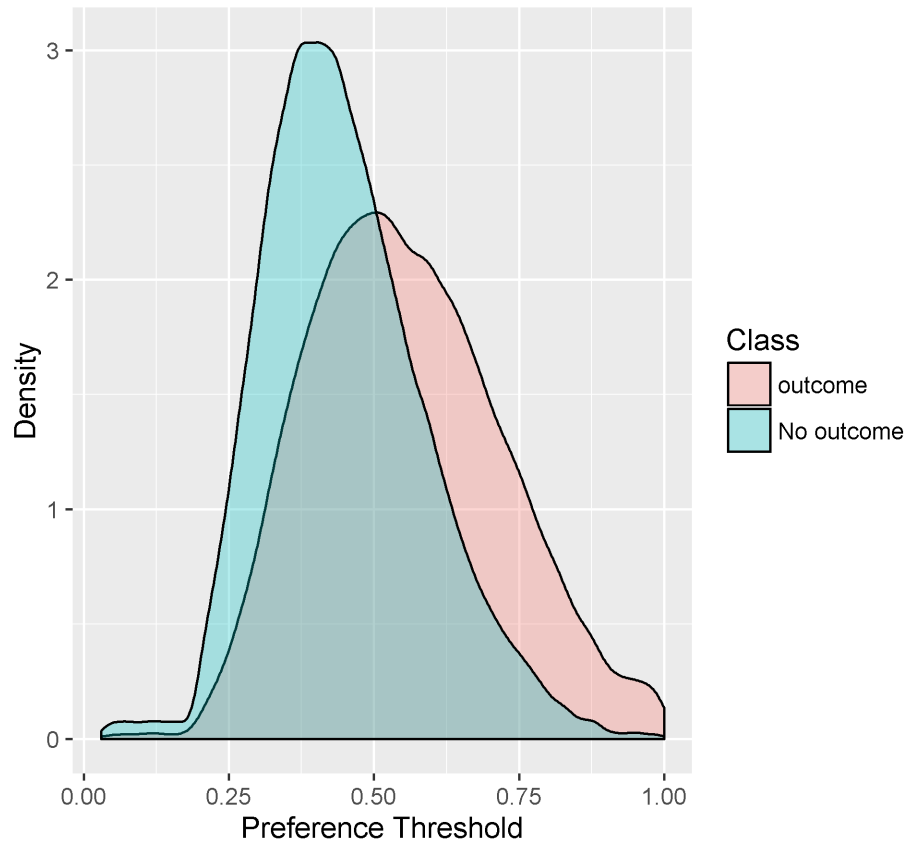


Figure 6: Preference Plot

However the increased information game comes at a computational cost. It is recommended to use the traditional plot for examination and then to produce the smooth plot for final versions.

## 6.4 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).

## 6.5 Box plots

The prediction distribution 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).

The box plots in the Figure above show that the predicted probability of the outcome is indeed higher for those with the outcome but there is also overlap between the two distribution which lead to an imperfect discrimination.

## 6.6 Test-Train similarity plot

The test-train similarity is presented by plotting the mean covariate values in the train set against those in the test set for people with and without the outcome.

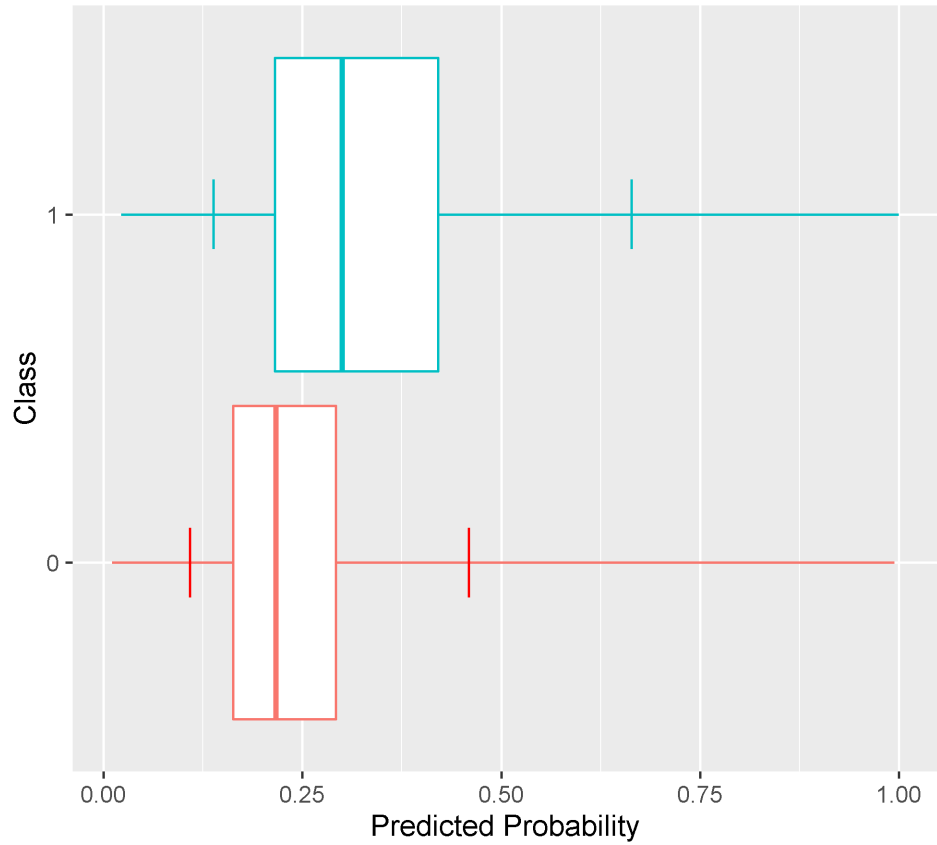


Figure 7: Prediction Distribution Box Plot

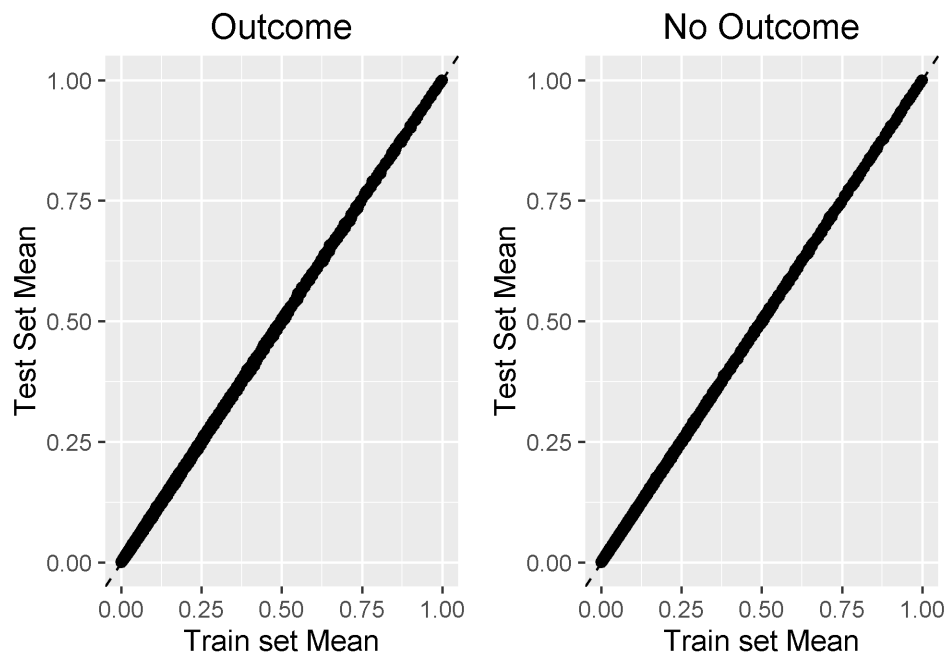


Figure 8: Similarity plots of train and test set

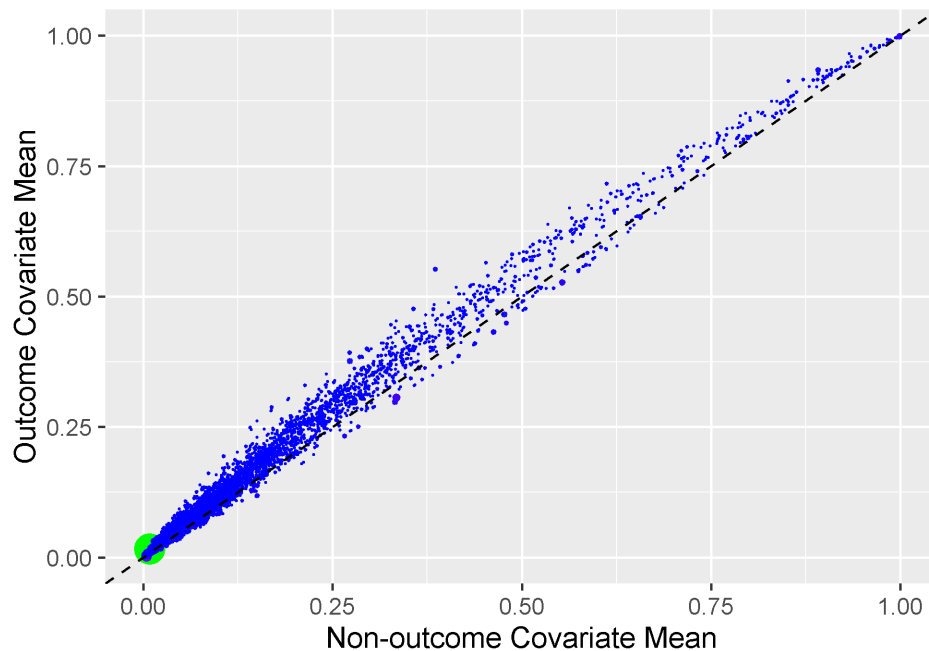


Figure 9: Variabel scatter Plot

The results for our example of re-hospitalization look very promising since the mean values of the covariates are on the diagonal.

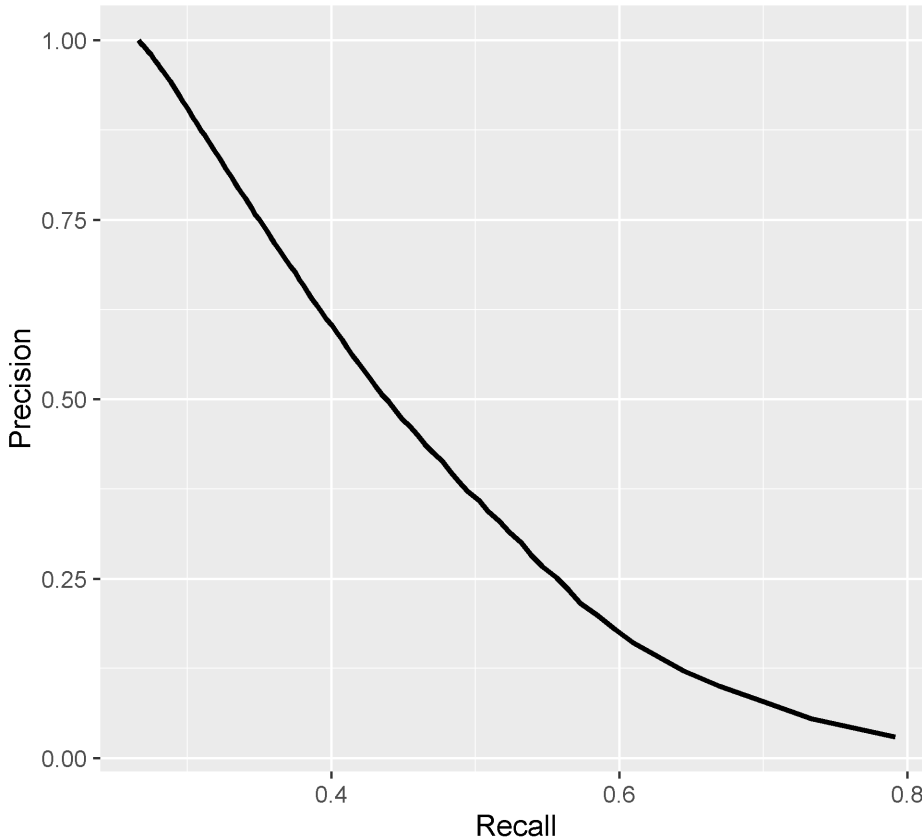
## 6.7 Variable scatter plot

The variable scatter plot shows 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 correspond to the importance of the covariates in the trained model (size of beta) and its direction (sign of beta with green meaning positive and red meaning negative), respectively.

The plot shows that the mean of most of the covariates is higher for subjects with the outcome compared to those without. Also there seem to be a very predictive, but rare covariate with a high beta.



## 6.8 Plot Precision Recall



Precision (P) is defined as the number of true positives (Tp) over the number of true positives plus the number of false positives (Fp).

```
P <- Tp / (Tp + Fp)
```

Recall (R) is defined as the number of true positives (Tp) over the number of true positives plus the number of false negatives (Fn).

```
R <- Tp / (Tp + Fn)
```

These quantities are also related to the (F1) score, which is defined as the harmonic mean of precision and recall.

```
F1 <- 2 * P * R / (P + R)
```

Note that the precision can either decrease or increase if the threshold is lowered. Lowering the threshold of a classifier may increase the denominator, by increasing the number of results returned. If the threshold was previously set too high, the new results may all be true positives, which will increase precision. If the previous threshold was about right or too low, further lowering the threshold will introduce false positives, decreasing precision.

For Recall the denominator does not depend on the classifier threshold (Tp+Fn is a constant). This means that lowering the classifier threshold may increase recall, by increasing the number of true positive results. It is also possible that lowering the threshold may leave recall unchanged, while the precision fluctuates.

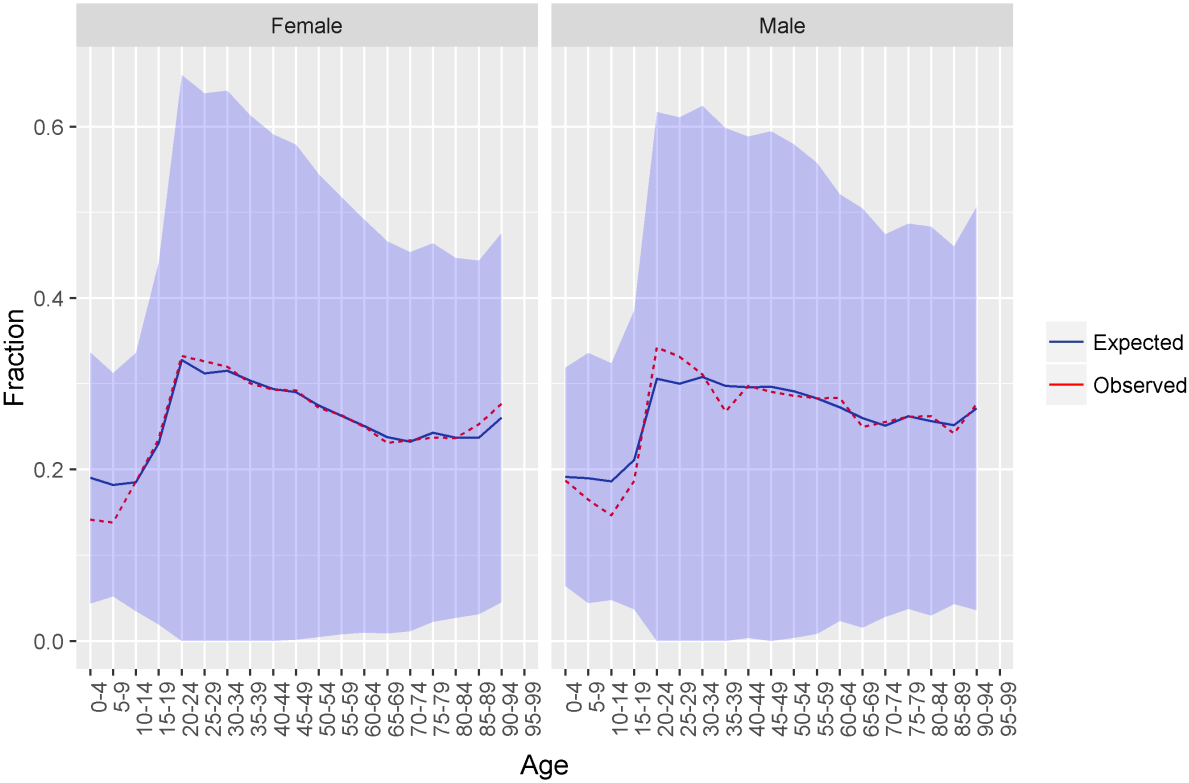


Figure 10: Demographic Summary Plot

## 6.9 Demographic Summary plot

This plot shows for females and males the expected and observed risk in different age groups together with a confidence area.

## 7 External validation

We recommend to always perform external validation, i.e. apply the final model on as much new datasets as feasible and evaluate its performance.

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

# load the new plpData and create the population
plpData <- loadPlpData(getwd(), "data")
population <- createStudyPopulation(plpData, outcomeId = 2, includeAllOutcomes = TRUE,
  firstExposureOnly = TRUE, washoutPeriod = 365, removeSubjectsWithPriorOutcome = TRUE,
  priorOutcomeLookback = 365, riskWindowStart = 1, requireTimeAtRisk = FALSE,
  riskWindowEnd = 365)

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

To make things easier we also provide a function for performing external validation of a model across one or multiple datasets:

```
# load the trained model
plpResult <- loadPlpResult(getwd(), "plpResult")

connectionDetails <- createConnectionDetails(dbms = "postgresql", server = "localhost/ohdsi",
  user = "joe", password = "supersecret")

validation <- externalValidatePlp(plpResult = plpResult, connectionDetails = connectionDetails,
  validationSchemaTarget = "new_cohort_schema", validationSchemaOutcome = "new_cohort_schema",
  validationSchemaCdm = "new_cdm_schema", validationTableTarget = "cohort_table",
  validationTableOutcome = "cohort_table", validationIdTarget = "cohort_id",
  validationIdOutcome = "outcome_id", keepPrediction = T)
```

This will run extract the new plpData from the specified schemas and cohort tables with the target cohort id of cohort id and outcome cohort id of outcome id. It will then apply the same population settings and the trained plp model. Finally, it will evaluate the performance and return the standard output as `validation$performance` and if `keepPrediction` is `TRUE` then it will also return the prediction on the population as `validation$prediction`. The can be inserted into the shiny app for viewing the model and validation by running: `viewPlp(runPlp=plpResult, validatePlp=validation )`.

If you want to validate on multiple databases available you can insert the new schemas and cohort tables as a list:

```
# load the trained model
plpResult <- loadPlpResult(getwd(), "plpResult")

connectionDetails <- createConnectionDetails(dbms = "postgresql", server = "localhost/ohdsi",
  user = "joe", password = "supersecret")

validation <- externalValidatePlp(plpResult = plpResult, connectionDetails = connectionDetails,
  validationSchemaTarget = list("new_cohort_schema1", "new_cohort_schema2"),
  validationSchemaOutcome = list("new_cohort_schema1", "new_cohort_schema2"),
  validationSchemaCdm = list("new_cdm_schema1", "new_cdm_schema2"), validationTableTarget = list("new_cohort_table1",
    "new_cohort_table2"), validationTableOutcome = list("new_cohort_table1",
    "new_cohort_table2"), validationIdTarget = "cohort_id", validationIdOutcome = "outcome_id",
  keepPrediction = T)
```

## 8 Acknowledgments

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

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

```
##
## Jenna Reys, 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
## Common Data Model},
##   author = {Jenna Reys and Martijn J. Schuemie and Marc A. Suchard and Patrick B. Ryan and Peter R.
##   year = {2018},
##   note = {R package version 2.0.5},
## }
```

Further, `PatientLevelPrediction` makes extensive use of the `Cyclops` package.

```
citation("Cyclops")
```

```
##
## To cite Cyclops in publications use:
##
## Suchard MA, Simpson SE, Zorych I, Ryan P, Madigan D (2013).
## "Massive parallelization of serial inference algorithms for
## complex generalized linear models." _ACM Transactions on Modeling
## and Computer Simulation_, *23*, 10. <URL:
## http://dl.acm.org/citation.cfm?id=2414791>.
##
## A BibTeX entry for LaTeX users is
##
## @Article{,
##   author = {M. A. Suchard and S. E. Simpson and I. Zorych and P. Ryan and D. Madigan},
##   title = {Massive parallelization of serial inference algorithms for complex generalized linear m
##   journal = {ACM Transactions on Modeling and Computer Simulation},
##   volume = {23},
##   pages = {10},
##   year = {2013},
##   url = {http://dl.acm.org/citation.cfm?id=2414791},
## }
```

**Please reference this paper if you use the PLP Package in your work:**

Reys 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.