Vignette PlasmodeSim

2022-10-19

Contents

Welcome to the vignette about the R package PlasmodeSim. This package is still under development. The goal of this package is to simulate new outcomes for patient data. This way one can obtain outcomes that follow a model you specify.

Installing PlasmodeSim using remotes

One can easily install the package using `remotes`, run:

```r
install.packages("remotes")
remotes::install_github("GidiusVanDeKamp/PlasmodeSim")
```

Binary outcomes/ Logistic Regression

We start by simulating simple binary variables. In this vignette we use a logisitic regression as model, but one could also pick other models that can be implemented as a `plpModel`.

Setting up

To start we need a plpModel and plpData. For information how to obtain these, one can look at; https://ohdsi.github.io/PatientLevelPrediction/articles/BuildingPredictiveModels.html. In this documents we load them from a saved file:

```
plpResultLogistic <- PatientLevelPrediction::loadPlpResult( "yourpathForPlpResult")
plpData <- PatientLevelPrediction::loadPlpData( "yourPathForPlpData" )
```

Simulate from a plpModel

In this example we obtain new outcomes following a fitted logistic model. We start from a plpModel, then run predictPlp. At last we generate new outcomes with the function `newOutcomes` that uses the plpPrediction.

```
plpModelLog <- plpResultLogistic$model

plpPrediction <- PatientLevelPrediction::predictPlp(
  plpModel = plpModelLog,
  plpData = plpData,
  population = plpData$cohorts
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.214 secs
## Prediction took 0.179 secs
```

When running the function predictPlp it returns some information.

```
newOut <- PlasmodeSim::newOutcomes(
  noPersons = 200,
  props = plpPrediction
)
head(newOut)
```

```
##   rowId outcomeCount
## 1    15            0
## 2    25            0
## 3    36            0
## 4    42            0
## 5    53            0
## 6    72            0
```

The column called ‘rowId’ in the output of `newOutcomes` contains the rowId’s of patients that are drawn randomly with the same probability. The patients could be drawn multiple times. If a rowId happens to be in the output twice, it can have different outcomes, but follows the same probability distribution. The function `newOutcomes` needs a data set that contains the columns ‘rowId’ and ‘value’. The column called ‘value’ contains the probability of seeing an outcome.

Simulation from an unfitted model

Here we show how to simulate outcomes from an unfitted logistic model. We use the function `makeLogisiticModel` to specify a logistic model.

```
Parameters <- plpModelLog$model$coefficients
UnfittedParameters <- Parameters
UnfittedParameters[1,1] <- -0.4
UnfittedParameters[3:5,1] <- 0.4
head(UnfittedParameters)
```

```
##   betas covariateIds
## 1  -0.4  (Intercept)
## 2   0.0         6003
## 3   0.4         8003
## 4   0.4         9003
## 5   0.4      8507001
## 6   0.0     28060210
```

For the logistic model it is necessary that the parameters are stored in a data set with a column called 'betas' and a column called 'covariateIds'. The function `makeLogisiticModel` creates a plpModel from the specified parameters. The parameters are given in a data frame with columns called 'betas' and 'covariateIds'. The column called 'betas' has the parameters of the model as numeric values. The columns called covariateIds has its elements stored as a string being '(Intercept)' or a covariateId.

```
plpModelunfitted <- PlasmodeSim::makeLogisticModel(UnfittedParameters)
newprobs <- PatientLevelPrediction::predictPlp(
  plpModel = plpModelunfitted,
  plpData = plpData,
  population = plpData$cohorts
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.17 secs
## Prediction took 0.178 secs
```

```
newOut <- PlasmodeSim::newOutcomes(
  noPersons = 2000,
  props = newprobs
)
head(newOut)
```

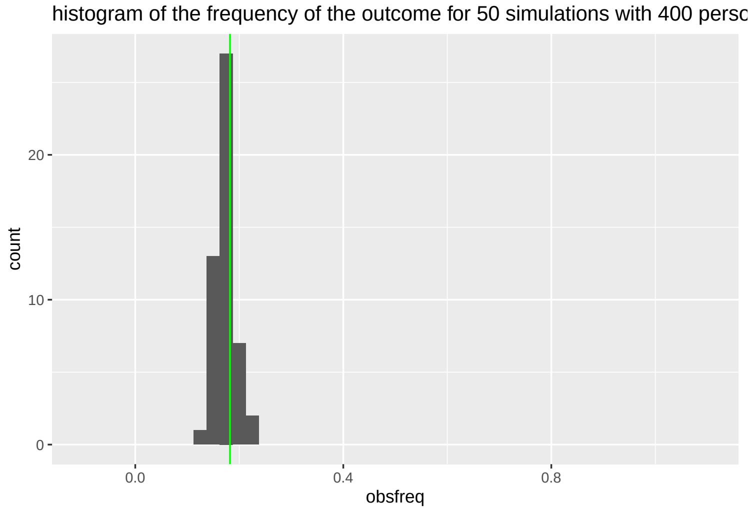
```
##   rowId outcomeCount
## 1     1            1
## 2     2            1
## 3     2            0
## 4     2            0
## 5     3            1
## 6     4            0
```

Visual simulations

The function `visualOutcome` simulates new data and then plots the frequency of the outcome. Right now the function `visualOutcome` only works for a logistic model. The green line in the plot is the average outcome in the original data set.

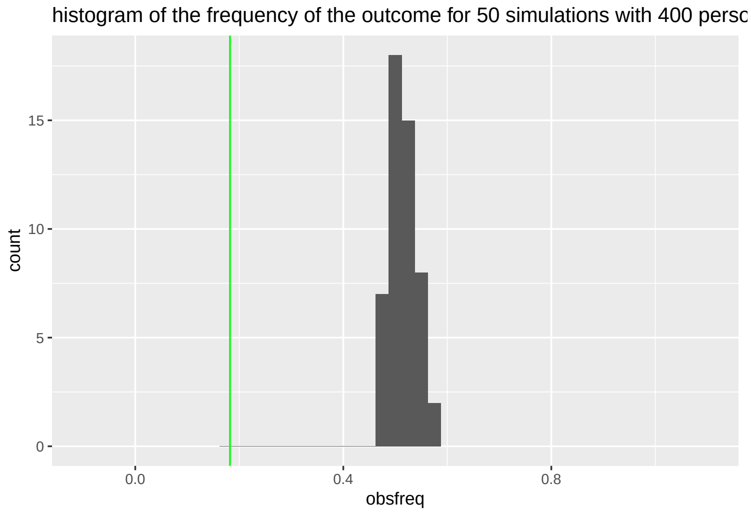
```
PlasmodeSim::visualOutcome(
  plpData = plpData,
  noSimulations = 50,
  noPersons = 400,
  parameters = Parameters
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.18 secs
## Prediction took 0.164 secs
```



```
PlasmodeSim::visualOutcome(
  plpData = plpData,
  noSimulations = 50,
  noPersons = 400,
  parameters = UnfittedParameters
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.182 secs
## Prediction took 0.175 secs
```



Above we see plotted the frequency of the outcome for a simulated data set with 200 people 50 times. We can see that the outcome count for the fitted parameters is similar to the outcome count in the original dataset. When changing the parameters the outcome count changes accordingly.

Visual of a specific covariate

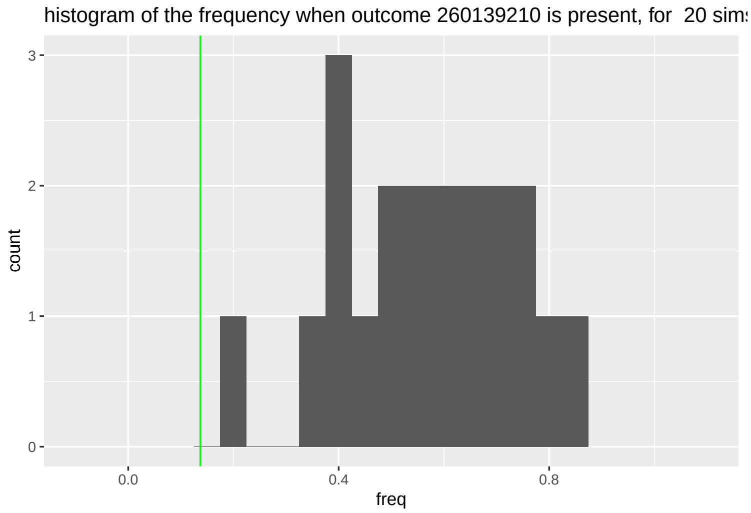
Say we are interested in the outcomes of a group with a specific covariate. To visualize this, we choose the third covariate from the model.

```
covariateIdToStudy<- plpResultLogistic$covariateSummary$covariateId[4]
UnfittedParameters[4,]
```

```
##   betas covariateIds
## 4   0.4         9003
```

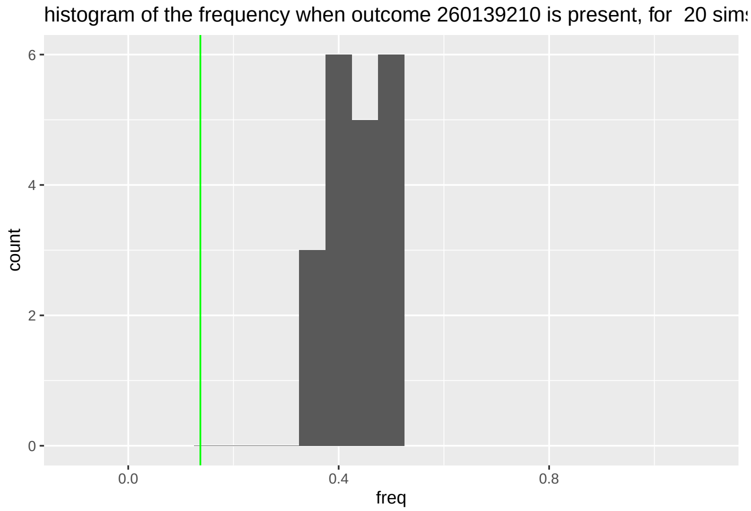
```
PlasmodeSim::visualOutcomeCovariateId(
  plpData=plpData,
  studyCovariateId= covariateIdToStudy,
  noSimulations = 20,
  noPersons = 200,
  parameters= UnfittedParameters
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.199 secs
## Prediction took 0.189 secs
```



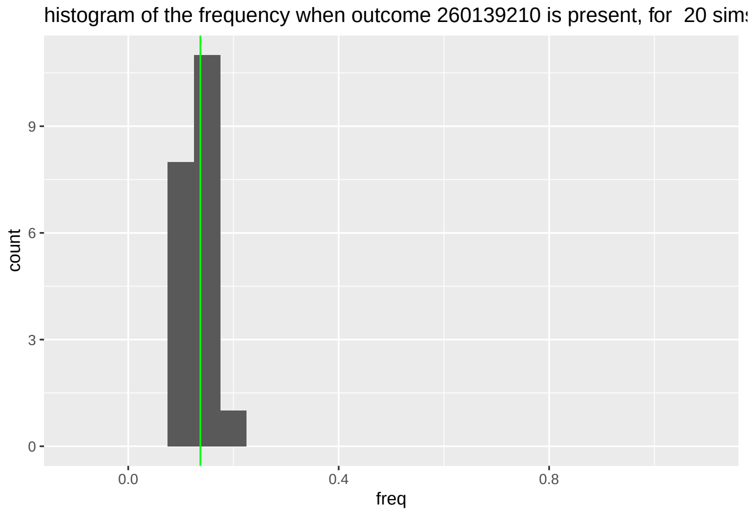
```
PlasmodeSim::visualOutcomeCovariateId2(
  plpData=plpData,
  restrictToCovariateId= covariateIdToStudy,
  noSimulations = 20,
  noPersons= 200,
  parameters= UnfittedParameters
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.177 secs
## Prediction took 0.18 secs
```



```
PlasmodeSim::visualOutcomeCovariateId2(
  plpData=plpData,
  restrictToCovariateId= covariateIdToStudy,
  noSimulations = 20,
  noPersons= 200,
  parameters= Parameters
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.186 secs
## Prediction took 0.18 secs
```



As one can see above `visualOutcomeCovariateId` and `visualOutcomeCovariateId2` are very similiar. They both calculate and plot the frequency for a group with a specific covariate present. The small difference is that `visualOutcomeCovariateId` filters a newly simulated dataset set to only keep the patients where the covariate is present, where as `visualOutcomeCovariateId2` only simulates new outcomes for patients that have the covariate present. We see they are almost identical, only `visualOutcomeCovariateId2` is spread out less because the groups for calculating the frequency with are larger. Again we see that when picking the fitted parameters, the outcome count for patients with a specific covariate, is similar to the original data set.

Survival times/ Cox model

In this part we will show how to simulate new survival times. For simulating new censored survival times, we need more than one probability, so we make use of the baseline hazard, stored in the a plpModel.

Loading the plpData

The first step is to load the data where we will simulate new outcomes for. Here we use the package Eunomia for accessing some data set.

```
connectionDetails <- Eunomia::getEunomiaConnectionDetails()

Eunomia::createCohorts(
  connectionDetails = connectionDetails,
  cdmDatabaseSchema = 'main',
  cohortDatabaseSchema = 'main',
  cohortTable = 'cohort'
)
```

```
## Creating cohort: Celecoxib
##   |                                                                      |
## Creating cohort: Diclofenac
##   |                                                                              |
## Creating cohort: GiBleed
##   |                                                                              |
## Creating cohort: NSAIDs
##   |                                                                              |
## Cohorts created in table main.cohort
```

```
##   cohortId       name
## 1        1  Celecoxib
## 2        2 Diclofenac
## 3        3    GiBleed
## 4        4     NSAIDs
##                                                                                         description
## 1    A simplified cohort definition for new users of celecoxib, designed specifically for Eunomia.
## 2    A simplified cohort definition for new users ofdiclofenac, designed specifically for Eunomia.
## 3 A simplified cohort definition for gastrointestinal bleeding, designed specifically for Eunomia.
## 4       A simplified cohort definition for new users of NSAIDs, designed specifically for Eunomia.
##   count
## 1  1844
## 2   850
## 3   479
## 4  2694
```

```r
databaseDetails <- PatientLevelPrediction::createDatabaseDetails(
  connectionDetails = connectionDetails,
  cdmDatabaseId = "eunomia",
  cdmDatabaseSchema = 'main',
  cdmDatabaseName = 'Eunomia',
  cohortDatabaseSchema = 'main',
  cohortTable = 'cohort',
  target = 4,
  outcomeDatabaseSchema = 'main',
  outcomeTable = 'cohort',
  outcomeId = 3,
  cdmVersion = 5
)

covariateSettings <- FeatureExtraction::createCovariateSettings(
  useDemographicsGender = TRUE,
  useDemographicsAgeGroup = TRUE,
  useConditionGroupEraLongTerm = TRUE,
  useDrugGroupEraLongTerm = TRUE,
  endDays = -1,
  longTermStartDays = -365
)

restrictPlpDataSettings <- PatientLevelPrediction::createRestrictPlpDataSettings(
  studyStartDate = '20000101',
  studyEndDate = '20200101',
  firstExposureOnly = TRUE,
  washoutPeriod = 30
)
```

```r
restrictPlpDataSettings <- PatientLevelPrediction::createRestrictPlpDataSettings(
  firstExposureOnly = TRUE,
  washoutPeriod = 30
)

plpData <- PatientLevelPrediction::getPlpData(
  databaseDetails = databaseDetails,
  covariateSettings = covariateSettings,
  restrictPlpDataSettings = restrictPlpDataSettings
)
```

```
##   |                                                                              |
```

```
## Warning: The 'oracleTempSchema' argument is deprecated. Use 'tempEmulationSchema' instead.
## This warning is displayed once every 8 hours.
```

```
## Constructing features on server
##   |                                                                              |
## Fetching data from server
## Fetching data took 0.176 secs
```

Defining a training set.

Most of the time we split the dataset into a training set and a testing set. In order to prepare the data for fitting the model, we have the function `MakeTraingSet`. This function copies features of the function `patientLevelPrediction::runPlp`. In order to run it, we have to create our settings: `populationSettings`, `executeSettings`, `splitSettings`, `sampleSettings`, `featureEngineeringSettings`, `preprocessSettings`. Besides all these settings, it also needs the plpData and the outcomeId.

```r
populationSettings <- PatientLevelPrediction::createStudyPopulationSettings(
  binary = TRUE,
  includeAllOutcomes = FALSE,
  firstExposureOnly = FALSE,
  washoutPeriod = 180,
  removeSubjectsWithPriorOutcome = FALSE,
  priorOutcomeLookback = 99999,
  requireTimeAtRisk = TRUE,
  minTimeAtRisk = 1,
  riskWindowStart = 1,
  startAnchor = 'cohort start',
  riskWindowEnd = 7300,
  endAnchor = 'cohort start'
)
executeSettings <- PatientLevelPrediction::createExecuteSettings(
  runSplitData = TRUE,
  runSampleData = FALSE,
  runfeatureEngineering = FALSE,
  runPreprocessData = TRUE,
  runModelDevelopment = TRUE,
  runCovariateSummary = TRUE
)
```

```
splitSettings <- PatientLevelPrediction::createDefaultSplitSetting(
  testFraction = 0.25,
  trainFraction = 0.75,
  splitSeed = 123,
  nfold = 3,
  type = 'stratified'
)
sampleSettings <- PatientLevelPrediction::createSampleSettings(
  type = 'none'
)
featureEngineeringSettings <-
  PatientLevelPrediction::createFeatureEngineeringSettings(
  type = 'none'
)
preprocessSettings <- PatientLevelPrediction::createPreprocessSettings(
  minFraction = 0,
  normalize = TRUE,
  removeRedundancy = TRUE
)

TrainingSet <- PlasmodeSim::MakeTraingSet(
  plpData = plpData,
  executeSettings = executeSettings,
  populationSettings = populationSettings,
  splitSettings = splitSettings,
  sampleSettings = sampleSettings,
  preprocessSettings = preprocessSettings,
  featureEngineeringSettings = featureEngineeringSettings,
  outcomeId = 3
)
```

```
## Outcome is 0 or 1
## seed: 123
## Creating a 25% test and 75% train (into 3 folds) random stratified split by class
## Data split into 656 test cases and 1974 train cases (658, 658, 658)
## Train Set:
## Fold 1 658 patients with 120 outcomes - Fold 2 658 patients with 120 outcomes - Fold 3 658 patients w
## 103 covariates in train data
## Test Set:
## 656 patients with 119 outcomes
## Removing 2 redundant covariates
## Normalizing covariates
## Tidying covariates took 0.493 secs
## Train Set:
## Fold 1 658 patients with 120 outcomes - Fold 2 658 patients with 120 outcomes - Fold 3 658 patients w
## 101 covariates in train data
## Test Set:
## 656 patients with 119 outcomes
```

Fitting the model with censoring

We pick the desired model by setting the `modelsettings`. Then we can run the function `fitModelWithCensoring`. This function fits two plpModels: one for the censoring and one for outcomes. They both are of the type specified with the modelsettings. It stores these plpModels as a list.

```
modelSettings <- PatientLevelPrediction::setCoxModel()

fitCensor <- PlasmodeSim::fitModelWithCensoring(
  Trainingset = TrainingSet$Train,
  modelSettings = modelSettings
)
```

```
## Running Cyclops
## Done.
## GLM fit status:  OK
## Creating variable importance data frame
## Prediction took 0.157 secs
## Running Cyclops
## Done.
## GLM fit status:  OK
## Creating variable importance data frame
## Prediction took 0.137 secs
```

Generating new outcome times

Now that we have our model with the censoring specified, we can simulate new outcomes. We call the function `simulateSurvivaltimesWithCensoring`. It uses the populationSettings for finding the last time that can be included in the outcome times.

```
NewOutcomes <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = fitCensor,
  plpData = plpData,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 10
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.172 secs
## Prediction took 0.253 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.174 secs
## Prediction took 0.248 secs
```

```
head(NewOutcomes)
```

```
##   rowId survivalTime outcomeCount
## 1   425         6096            0
## 2  1557         7293            0
## 3  2066         2024            0
## 4   664         1329            0
## 5    48         5593            0
## 6   299           18            1
```

Since the censoring model stores two models as a list, one can easily generate uncensored outcomes. This can be done by using the function `simulateSurvivaltimes`. One could also use this function for generating censoring times.

```
newdata <- PlasmodeSim::simulateSurvivaltimes(
  plpModel = fitCensor$outcomesModel,
  plpData = plpData,
  numberToSimulate = 10,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.174 secs
## Prediction took 0.254 secs
```

```
head(newdata)
```

```
##   rowId outcome
## 1  2536    7300
## 2  1882    7300
## 3  1494    7300
## 4  1609    7300
## 5   911    7300
## 6  2610    7300
```

Defining an unfitted model without censoring

Just as before, we can define a model that has not been fitted to the data. We specify a Cox model by specifying the two sets of coefficients/parameters and two baseline survival functions.

```
plpModel <- fitCensor$outcomesModel
coeff <- plpModel$model$coefficients
survival <- plpModel$model$baselineSurvival$surv
times <- plpModel$model$baselineSurvival$time

unfittedmodel <- PlasmodeSim::defineCoxModel(
  coefficients = coeff,
  baselinehazard = survival,
  timesofbaselinhazard = times,
  featureEngineering = NULL #  = NULL is the standard setting.
)

newdata <- PlasmodeSim::simulateSurvivaltimes(
  plpModel = unfittedmodel,
  plpData = plpData,
  numberToSimulate = 10,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings
)
```

```
## Prediction took 0.178 secs
```

```
head(newdata)
```

```
##   rowId outcome
## 1    17      18
## 2  1783    7300
## 3  2600    7300
## 4   964    7300
## 5    30      81
## 6  1182    7300
```

Defining an unfitted model with censoring

There is no function to define an unfitted model with censoring. However, this can be done easily by making two Cox models and storing them in a list. The elements in this list should have the names 'censorModel' and 'outcomeModel'. In this example we use the unfitted model, specified in the code above, for the outcomes and use the fitted censoring model.

```
#we can swap outcomes with censoring.
unfittedcensor<- list(censorModel = unfittedmodel,
                      outcomesModel = fitCensor$outcomesModel)

NewOutcomes <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = unfittedcensor,
  plpData = plpData,
  population =  TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 200
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.175 secs
## Prediction took 0.243 secs
## Prediction took 0.178 secs
```

```
head(NewOutcomes)
```

```
##   rowId survivalTime outcomeCount
## 1  1393           71            0
## 2  1363           36            0
## 3   485         7300            0
## 4   769         7300            0
## 5   244           30            0
## 6   614         7300            0
```

Adjusting the BaselineSurvival

If one wants to get a grip on the outcome count on a specific time, one can call the function `adjustBaselineSurvival`. This can be useful in cases that one wants to obtain multiple data sets, that have different parameters, but with the same frequency of outcomes. The function `adjustBaselineSurvival` changes the base line function of a model in such a way that for the training data at the specified time the outcome rate is a specified probability. Since this function solves an equation it needs an interval to find this solution specified.

```r
adjustedModel <- PlasmodeSim::adjustBaselineSurvival(
  plpModel = plpModel,
  TrainingSet = TrainingSet$Train,
  plpData = plpData,
  populationSettings = populationSettings,
  timeToFixAt = 3592,
  propToFixWith = 0.87,
  intervalSolution= c(-100,100)
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.179 secs
## Prediction took 0.244 secs
```

```r
NewOutcomes <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = list(censorModel = fitCensor$outcomesModel,
                     outcomesModel = adjustedModel),
  plpData = plpData,
  population =  TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 200
)
```

```
## Prediction took 0.182 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.174 secs
## Prediction took 0.248 secs
```

```r
head(NewOutcomes)
```

```
##   rowId survivalTime outcomeCount
## 1  2367           10            1
## 2   404         7300            0
## 3  1501            5            1
## 4  1545            9            1
## 5  1942           39            0
## 6  1862           10            1
```

Plotting Kaplan Meier estimates

The function `kaplanMeierPlot` visualizes the Kaplan Meier estimate of a given data set. It works with ggplot. We can easily compare the simulated data sets with the original data by putting them in one plot. For the true data set we set the colour to red.

```r
NewOutcomes <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = fitCensor,
  plpData = plpData,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 1974
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.173 secs
## Prediction took 0.245 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.171 secs
## Prediction took 0.253 secs
```

```
NewOutcomes2 <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = fitCensor,
  plpData = plpData,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 1974
)
```

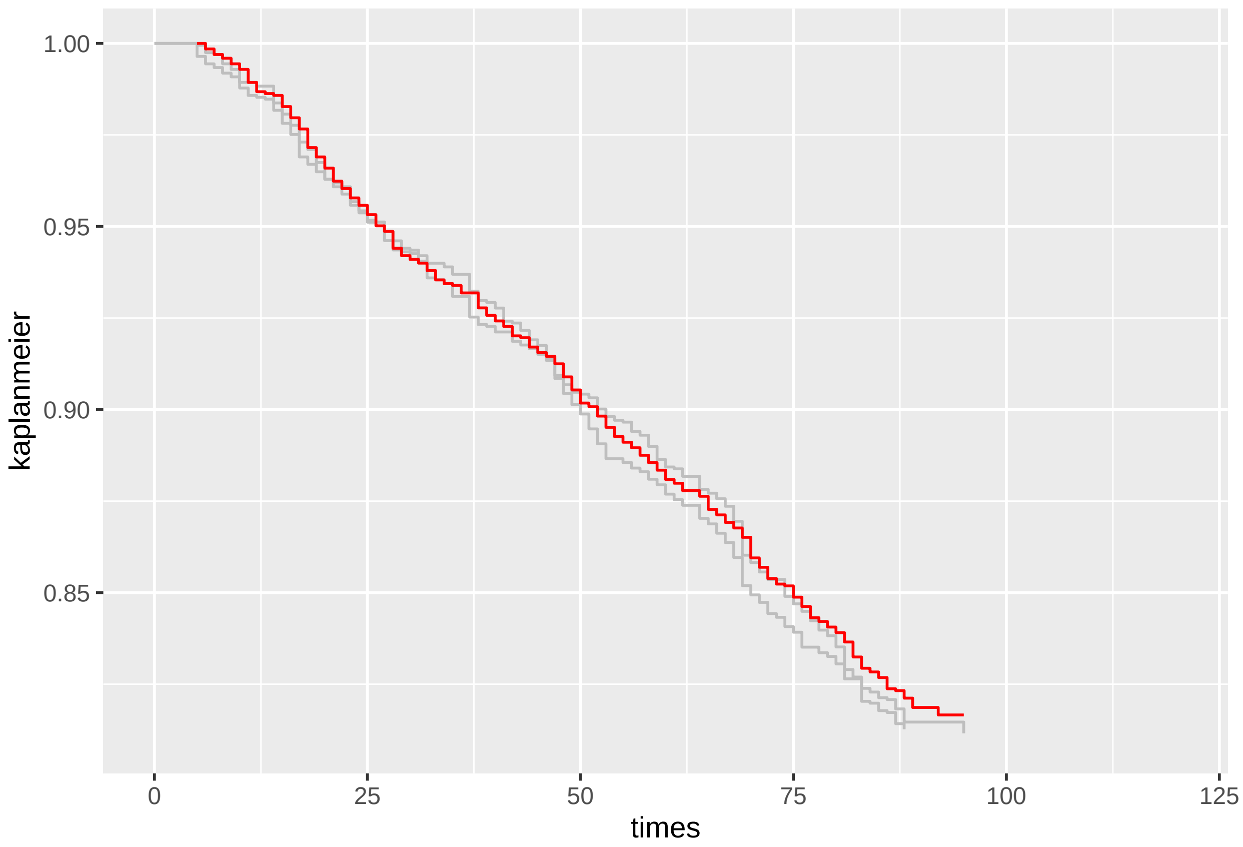
```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.206 secs
## Prediction took 0.298 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.203 secs
## Prediction took 0.271 secs
```

```
ggplot2::ggplot()+
  PlasmodeSim::KaplanMeierPlot( NewOutcomes )+
  PlasmodeSim::KaplanMeierPlot( NewOutcomes2 )+
  PlasmodeSim::KaplanMeierPlot( TrainingSet$Train$labels, colour = 'red' )+
  ggplot2::xlim(c(0,120))
```

```
## Warning: Removed 502 rows containing missing values (‘geom_step()‘).
```

```
## Warning: Removed 510 rows containing missing values (‘geom_step()‘).
```

```
## Warning: Removed 790 rows containing missing values (‘geom_step()‘).
```



Above we see that the newly generated data follows the original distribution. However, it seems that the outcomes are more frequent in the original dataset. We can also generate datasets where all the patients have one specific covariate present. We do this with the function `simulateSurvivaltimesWithCensoringCovariate`. This function works in a similar way as `simulateSurvivaltimesWithCensoring` but filters the population to make sure the covariate specified is present.

```
fitCensor$outcomesModel$model$coefficients[2,1] <- 0.5
fitCensor$outcomesModel$model$coefficients[2,2] # red
```

```
## [1] "8003"
```

```
fitCensor$outcomesModel$model$coefficients[3,1] <- 0
fitCensor$outcomesModel$model$coefficients[3,2] # orange
```

```
## [1] "9003"
```

```
fitCensor$outcomesModel$model$coefficients[4,1] <- -0.75
fitCensor$outcomesModel$model$coefficients[4,2] # yellow
```

```
## [1] "8507001"
```

```
numberToSimulate  <- 2000
newOut1 <- PlasmodeSim::simulateSurvivaltimesWithCensoringCovariate(
  censorModel = fitCensor,
  plpData = plpData,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = numberToSimulate,
  covariateToStudy = 8003
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.207 secs
## Prediction took 0.245 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.213 secs
## Prediction took 0.243 secs
```

```
newOut2 <- PlasmodeSim::simulateSurvivaltimesWithCensoringCovariate(
  censorModel = fitCensor,
  plpData= plpData,
  population=  TrainingSet$Train$labels,
  populationSettings=populationSettings,
  numberToSimulate=numberToSimulate,
  covariateToStudy = 9003
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.191 secs
## Prediction took 0.202 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.189 secs
## Prediction took 0.205 secs
```

```
newOut3 <- PlasmodeSim::simulateSurvivaltimesWithCensoringCovariate(
  censorModel = fitCensor,
  plpData= plpData,
  population=TrainingSet$Train$labels,
  populationSettings=populationSettings,
  numberToSimulate=numberToSimulate,
  covariateToStudy = 8507001
)
```

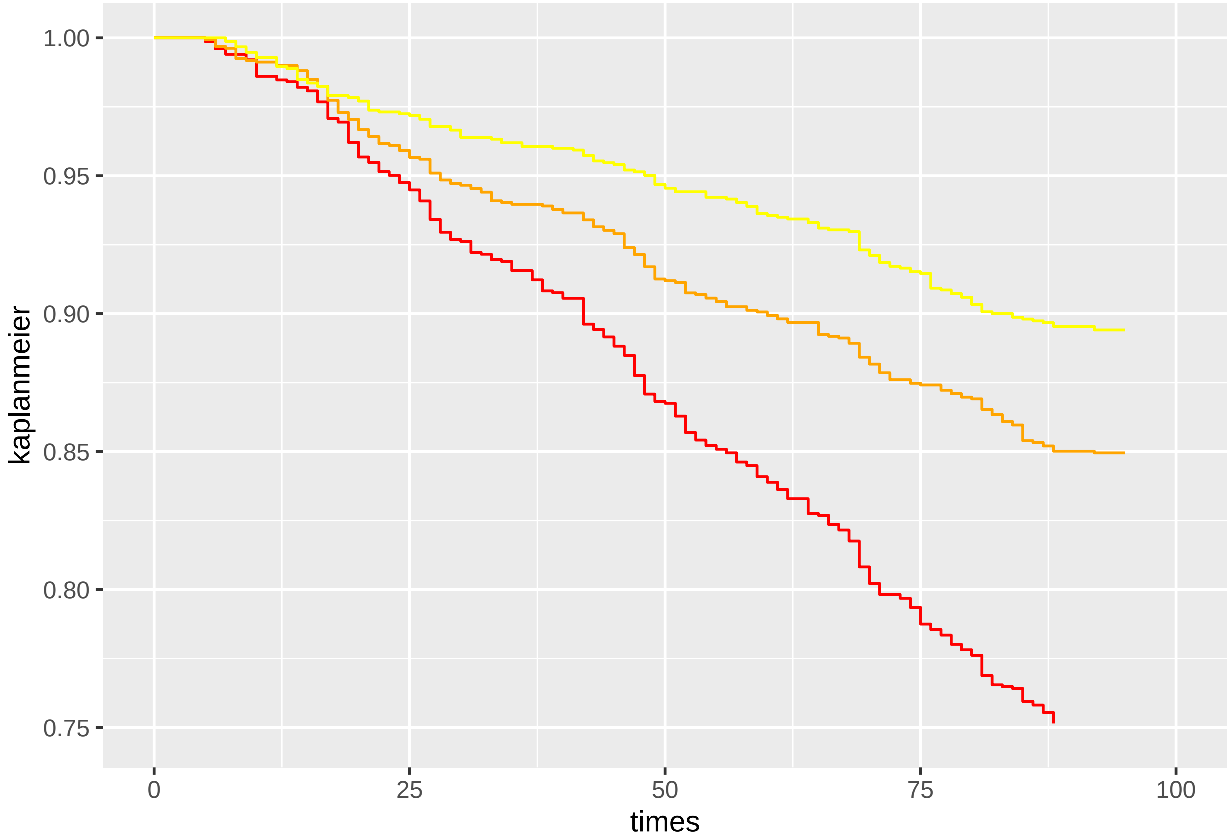
```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.184 secs
## Prediction took 0.225 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.186 secs
## Prediction took 0.23 secs
```

```
ggplot2::ggplot()+
  PlasmodeSim::KaplanMeierPlot( newOut1, colour = 'red')+
  PlasmodeSim::KaplanMeierPlot( newOut2, colour = 'orange')+
  PlasmodeSim::KaplanMeierPlot( newOut3, colour = 'yellow')+
  ggplot2::xlim( c(0,100))
```

```
## Warning: Removed 421 rows containing missing values (‘geom_step()‘).

## Warning: Removed 498 rows containing missing values (‘geom_step()‘).

## Warning: Removed 457 rows containing missing values (‘geom_step()‘).
```



runPlasmode

The function runPlasmode returns some newly simulated survivaltimes, from a model it fits.

```
runPlas <- PlasmodeSim::runPlasmode(
  plpData = plpData,
  outcomeId = 3,
  populationSettings = populationSettings,
  splitSettings = splitSettings,
  sampleSettings = sampleSettings,
  featureEngineeringSettings = featureEngineeringSettings,
  preprocessSettings = preprocessSettings,
  modelSettings = modelSettings,
  executeSettings = executeSettings,
  numberToSimulate = 5
)
```

```
## Outcome is 0 or 1
## seed: 123
## Creating a 25% test and 75% train (into 3 folds) random stratified split by class
## Data split into 656 test cases and 1974 train cases (658, 658, 658)
## Train Set:
## Fold 1 658 patients with 120 outcomes - Fold 2 658 patients with 120 outcomes - Fold 3 658 patients w
## 103 covariates in train data
## Test Set:
## 656 patients with 119 outcomes
## Removing 2 redundant covariates
## Normalizing covariates
## Tidying covariates took 0.496 secs
## Train Set:
## Fold 1 658 patients with 120 outcomes - Fold 2 658 patients with 120 outcomes - Fold 3 658 patients w
## 101 covariates in train data
## Test Set:
## 656 patients with 119 outcomes
## Running Cyclops
## Done.
## GLM fit status:  OK
## Creating variable importance data frame
## Prediction took 0.137 secs
## Running Cyclops
## Done.
## GLM fit status:  OK
## Creating variable importance data frame
## Prediction took 0.136 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.187 secs
## Prediction took 0.257 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.181 secs
## Prediction took 0.246 secs
```

```
runPlas
```

```
##   rowId survivalTime outcomeCount
## 1   572         5484            0
## 2  1726         6611            0
## 3   419         7111            0
## 4   522           27            1
## 5   425         7293            0
```

Possible extensions

Following below is a list of suggestions for possible extensions to make the package more useful:

- The runPlasmode should have a working analysisId, analysisName and logsettings, like runPlp has.
- One could extend the fitmodel by adding an option for different models for the censoring.
- Take a look at the feature engineering in the `definecoxmodel` function.
- Add more functions that define unfitted models.

- Make the functions run faster by filtering the population on the rowids drawn, before making their outcomes.