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")
```

Problem

This package is designed to simulated data sets, with the goal that one can test different statistical methods on these data sets. Beside simulating these new data sets, we implemented some ways to visualize data sets.

We want to obtain a data set that follows a certain model, so that one can test different models on this data set. We have an outcome Y and some data $X_1, X_2, \cdots, X_n$ for each patient. We want to generate new Y while leaving the X's as they are. This way we keep some characteristics of the data set, but we do have a true model. So it is a good set to test a statistical method.
In the first part of this vignette we show how to generate data in the case that Y is binary. We will show how to generate Y's that follow a logistic model. So we have

$$P(Y=1|X_1\cdots,X_n) = \frac{1}{1+\exp(-(\beta_0+\beta_1X_1+\beta_2X_2\cdots\beta_3X_3))},$$

for some β's we specify. However the ways of generating new data that follow a different model is also possible, if it is implemented in the package `PatientLevelPrediction`. The plpData stores the information about each patient $Y, X_1, \cdots, X_n$, and the plpModel describes the chance of an outcome from the data from a patient.

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. After that 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 0.21 secs
## Prediction took 0.185 secs
```

When running the function predictPlp it returns some information.

```
newOutcomesFittedModel <- PlasmodeSim::newOutcomes(
  noPersons = 2000,
  props = plpPrediction
)
head(newOutcomesFittedModel)
```

```
##   rowId outcomeCount
## 1     1            0
## 2     2            0
## 3     3            0
## 4     3            0
## 5     4            1
## 6     7            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.

```r
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 column called 'covariateIds' has its elements stored as a string being '(Intercept)' or a covariateId.

```r
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 0.188 secs
## Prediction took 0.183 secs
```

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

```
##   rowId outcomeCount
## 1     4            1
## 2     7            0
## 3     9            1
## 4     9            1
## 5    11            1
## 6    15            1
```

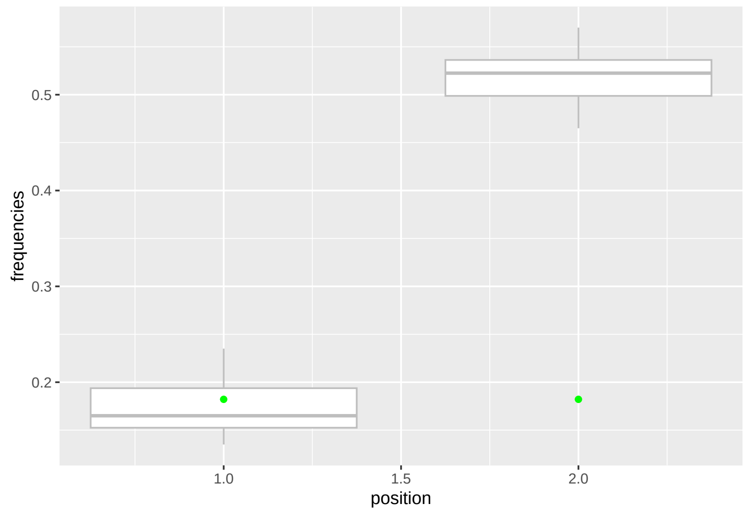
Visual simulations

As mentioned before it is important to test on data that is similar to the data that a model will be used for (or trained on). That is why we want to see if the properties from the original set are also present in the test set. We will do this with plots.

We can use the functions `frequencyOutcomePlot` and `frequencyCovariatePlot` to plot the frequencies in the new data sets. These functions works together with `ggplot`. We have generated 2000 points. The function calculates multiple frequencies by chopping the data in smaller sets. It also shows the frequency of the outcomes in the plpData with a green dot.

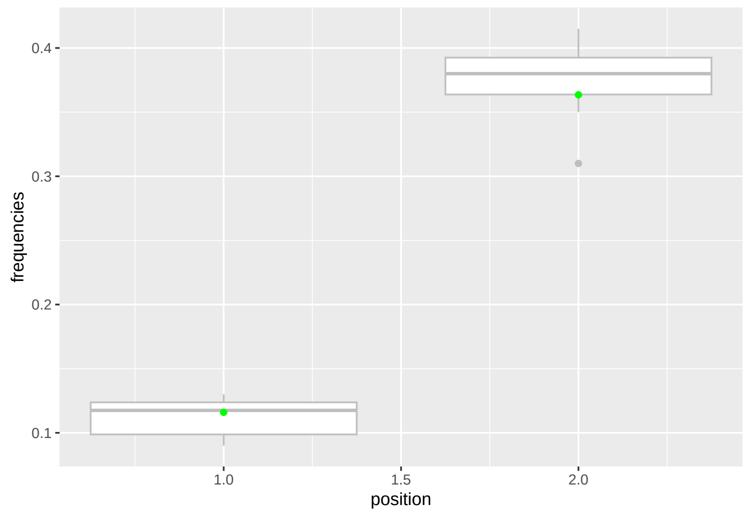
In the plot below we see on the the outcome frequency of the original dataset with a green dot. On the left we see the frequencies of the data set generated with the fitted model. There are 10 frequencies calculated with groups of 200 patients. On the right we see the same but for the generated data set that comes from an unfitted model. This function makes a random division to calculate the groups. So one might want to use `set.seed()` if a fixed outcome is desired. We see that for the fitted model the outcome frequency is similar to the original data set.

```r
ggplot2::ggplot()+
PlasmodeSim::frequencyOutcomePlot(
  OutcomeData = newOutcomesFittedModel,
  noSyms = 10,
  noPatientsInSym = 200,
  plpData = plpData,
  placeInPlot = 1)+
PlasmodeSim::frequencyOutcomePlot(
  OutcomeData = newOutcomesUnfitted,
  noSyms = 10,
  noPatientsInSym = 200,
  plpData = plpData,
  placeInPlot = 2)
```



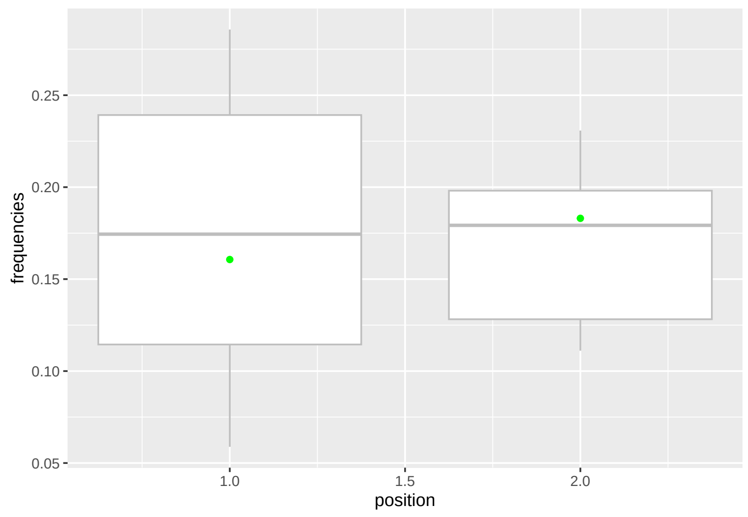
We are also interested in seeing the frequencies of a specific covariate in the new data set. We do this with the function `frequencyCovariatePlot`. This function works quite similar, it also chops up the data set. Again we have frequency of the original data set present as a green dot.

```
ggplot2::ggplot()+
PlasmodeSim::frequencyCovariatePlot(
  OutcomeData = newOutcomesFittedModel,
  noSyms = 10,
  noPatientsInSym = 200,
  covariateToStudy = 6003,
  plpData = plpData,
  placeInPlot = 1)+
PlasmodeSim::frequencyCovariatePlot(
  OutcomeData = newOutcomesFittedModel,
  noSyms = 10,
  noPatientsInSym = 200,
  covariateToStudy = 8003,
  plpData = plpData,
  placeInPlot = 2)
```



At last we also want to know the frequency of an outcome for when an covariate is present. For this purpose we have the function `frequencyOutcomeCovariatePlot`. This time the function also chops the data into 10 sets with 200 patients, but then limits those 200 persons even further, by selection only the patients with the specified covariate.

```
ggplot2::ggplot()+
PlasmodeSim::frequencyOutcomeCovariatePlot(
  OutcomeData = newOutcomesFittedModel,
  noSyms = 10,
  noPatientsInSym = 200,
  covariateToStudy = 6003,
  plpData = plpData,
  placeInPlot = 1)+
PlasmodeSim::frequencyOutcomeCovariatePlot(
  OutcomeData = newOutcomesFittedModel,
  noSyms = 10,
  noPatientsInSym = 200,
  covariateToStudy = 8003,
  plpData = plpData,
  placeInPlot = 2)
```



Survival times/ Cox model

In this part we will show how to simulate new survival times. So now the outcome will not be binary. The outcomes are some positive number, for when the outcome happens. The data can also be right censored. We will show how to generate uncensored and censored data set.

For simulating new censored survival times, we need more than one probability, we need a survival function ($S(t) = P(Y > t)$). In this vignette we will show how to simulate from an Cox model. The Cox model assumes that the baseline hazard changes when an covariate is present. The baseline hazard is the survival function ($S_0(t)$) when all the covariates are equal to zero. One could think of the baseline hazard as an intercept that changes over time. To be more precise: for any patient the survival function in a cox model is given by:

$$S(t) = S_0(t)^{\exp(\beta_1 X_1 + \cdots + \beta_n X_n)}.$$

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.

```r
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
)

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 0.197 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 `MakeTrainingSet`. 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.

```
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::makeTrainingSet(
  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 0.531 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 0.139 secs
## Running Cyclops
## Done.
## GLM fit status:  OK
## Creating variable importance data frame
## Prediction took 0.153 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. The function draws an outcome time and a censoring time and returns the minimum of these as the new value.

```
NewOutcomesFittedCensorModel <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = fitCensor,
  plpData = plpData,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 2000
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.355 secs
## Prediction took 0.258 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.185 secs
## Prediction took 0.269 secs
```

```
head(NewOutcomesFittedCensorModel)
```

```
##   rowId survivalTime outcomeCount
## 1   425         7300            0
## 2  1557           48            1
## 3  2066         7293            0
## 4   664         7300            0
## 5    48         7293            0
## 6   299         2683            0
```

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.

```
NewUnfilteredSurvivaltimes <- PlasmodeSim::simulateSurvivaltimes(
  plpModel = fitCensor$outcomesModel,
  plpData = plpData,
  numberToSimulate = 2000,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.191 secs
## Prediction took 0.259 secs
```

```
head(NewUnfilteredSurvivaltimes)
```

```
##   rowId outcome
## 1   640    7300
## 2   797    7300
## 3  2626    7300
## 4   770      18
## 5   162    7300
## 6   756      54
```

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.
)

NewOutcomesUnfittedModel <- PlasmodeSim::simulateSurvivaltimes(
  plpModel = unfittedmodel,
  plpData = plpData,
  numberToSimulate = 2000,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings
)
```

```
## Prediction took 0.183 secs
```

```
head(NewOutcomesUnfittedModel)
```

```
##   rowId outcome
## 1  1067    7300
## 2     6    7300
## 3  2068    7300
## 4  1832    7300
## 5  1362      65
## 6   252    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)

NewOutcomesUnfittedCensorModel <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = unfittedcensor,
  plpData = plpData,
  population =  TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 2000
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.2 secs
## Prediction took 0.262 secs
## Prediction took 0.181 secs
```

```r
head(NewOutcomesUnfittedCensorModel)
```

```
##   rowId survivalTime outcomeCount
## 1   399         7300            0
## 2  1195         7300            0
## 3  1631         7300            0
## 4  1588         7300            0
## 5   492         7300            0
## 6   869         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 (S_0) in such a way that for the training data at the specified time (t) the outcome rate is a specified probability (p). It finds a δ so that one can set the new baseline to $(S_0)^\delta$. It finds this δ by solving:

$$\frac{1}{n}\sum_{i=1}^{n}\Big(S_Y(t|X_i)\Big)^\delta = 1 - p.$$

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 0.182 secs
## Prediction took 0.261 secs
```

```r
NewOutcomes <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = list(censorModel = fitCensor$outcomesModel,
                     outcomesModel = adjustedModel),
  plpData = plpData,
  population =  TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 2000
)
```

```
## Prediction took 0.183 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.191 secs
## Prediction took 0.262 secs
```

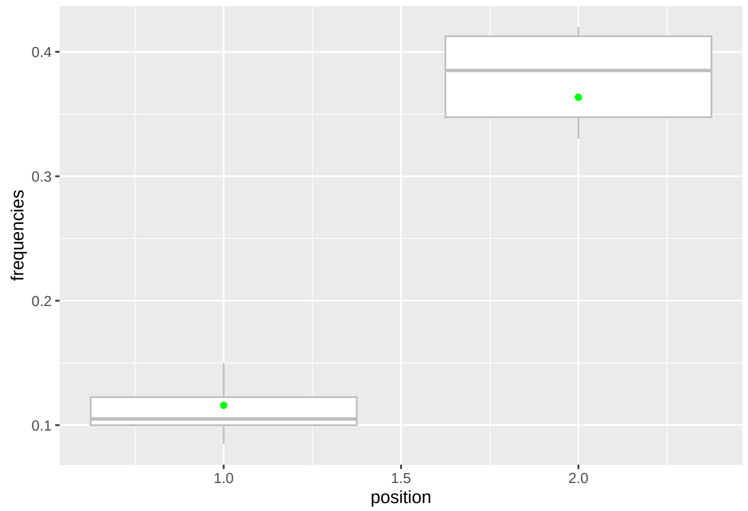
```
head(NewOutcomes)
```

```
##   rowId survivalTime outcomeCount
## 1  1530           16            1
## 2  1041           25            0
## 3  1610           23            1
## 4  1157           23            1
## 5   576           26            1
## 6   913           19            1
```

Plotting Kaplan Meier estimates

Again we are interested if the features of the original data set, are also present in the simulated data set. We first look at the frequency of the covariates in the simulated data.

```
ggplot2::ggplot()+
PlasmodeSim::frequencyCovariatePlot(
  OutcomeData = NewOutcomesFittedCensorModel,
  noSyms = 10,
  noPatientsInSym = 200,
  covariateToStudy = 6003,
  plpData = plpData,
  placeInPlot = 1)+
PlasmodeSim::frequencyCovariatePlot(
  OutcomeData = NewOutcomesFittedCensorModel,
  noSyms = 10,
  noPatientsInSym = 200,
  covariateToStudy = 8003,
  plpData = plpData,
  placeInPlot = 2)
```



However for the outcome we are also interested in the time of the outcome. That is why we do not use a boxplot but a Kaplan Meier plot. This type of plot plots the Kaplan Meier estimator that is given by:

$$\hat{S}(t) = \prod_{i:t_i \le t} \left(1 - \frac{d_i}{n_i}\right).$$

Where we have that d_i is the number of outcomes that happend at time t_i and n_i is the number of patients known to have survived or not yet have been censored upto time t_i. It is an estimator for the survival function.

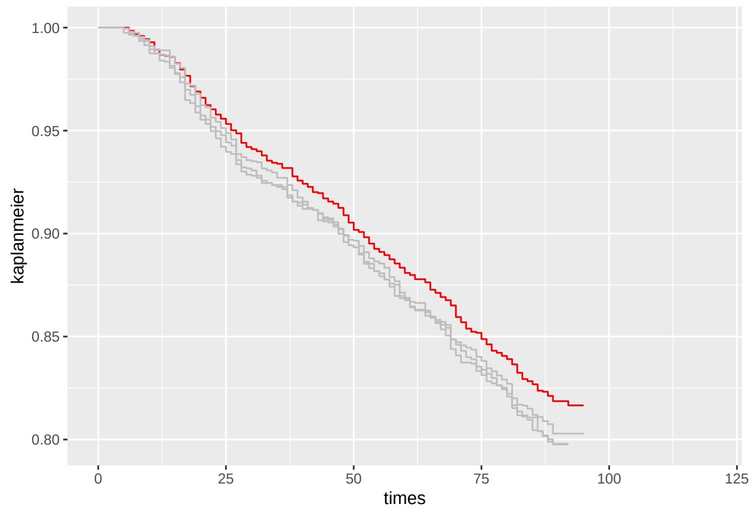
The function `kaplanMeierPlot` visualizes the Kaplan Meier estimate of a given data set. It also works with ggplot. We can easily compare the simulated data sets with the original data by putting them in one plot. For the true data set we set the colour to red.

```
toPlotFittedCensorModel <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  censorModel = fitCensor,
  plpData = plpData,
  population = TrainingSet$Train$labels,
  populationSettings = populationSettings,
  numberToSimulate = 4000
)
```

```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.222 secs
## Prediction took 0.285 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.19 secs
## Prediction took 0.26 secs
```

```
FittedCensorModel1 <- toPlotFittedCensorModel[1:2000,]
FittedCensorModel2 <- toPlotFittedCensorModel[-(1:2000),]
FittedCensorModel3 <- NewOutcomesFittedCensorModel

ggplot2::ggplot()+
  PlasmodeSim::KaplanMeierPlot( Data = TrainingSet$Train$labels,
                                colour = 'red' )+
  PlasmodeSim::KaplanMeierPlot( Data = FittedCensorModel1 )+
  PlasmodeSim::KaplanMeierPlot( Data = FittedCensorModel2 )+
  PlasmodeSim::KaplanMeierPlot( Data = FittedCensorModel3 )+
  ggplot2::xlim(c(0,120))
```

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

## Warning: Removed 482 rows containing missing values (‘geom_step()‘).

## Warning: Removed 511 rows containing missing values (‘geom_step()‘).

## Warning: Removed 496 rows containing missing values (‘geom_step()‘).
```

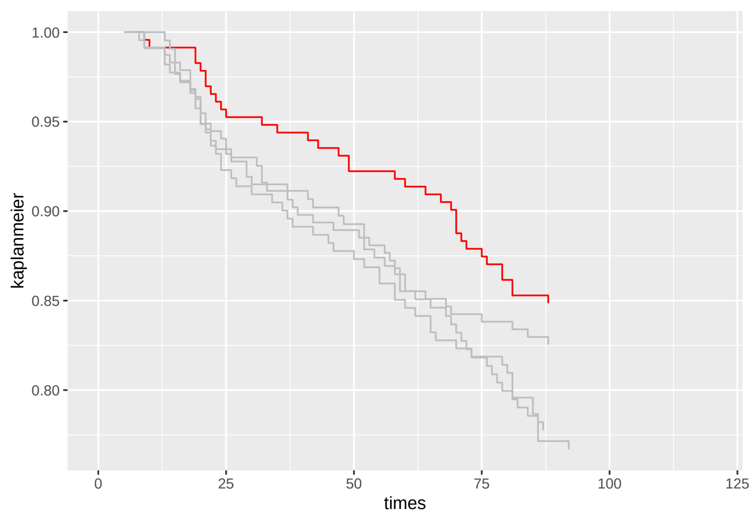


Above we see that the newly generated data follows the original distribution. However, it seems that the outcomes are more frequent in the original dataset.

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.

```
ggplot2::ggplot()+
  PlasmodeSim::KaplanMeierPlotFilterCovariate( Data = TrainingSet$Train$labels,
                                               covariateToStudy = 6003,
                                               plpData = plpData,
                                               colour = 'red' )+
  PlasmodeSim::KaplanMeierPlotFilterCovariate( Data = FittedCensorModel1,
                                               covariateToStudy = 6003,
                                               plpData = plpData )+
  PlasmodeSim::KaplanMeierPlotFilterCovariate( Data = FittedCensorModel2,
                                               covariateToStudy = 6003,
                                               plpData = plpData )+
  PlasmodeSim::KaplanMeierPlotFilterCovariate( Data = FittedCensorModel3,
                                               covariateToStudy = 6003,
                                               plpData = plpData )+
  ggplot2::xlim(c(0,120))
```

```
## Warning: Removed 87 rows containing missing values (`geom_step()`).

## Warning: Removed 70 rows containing missing values (`geom_step()`).

## Warning: Removed 95 rows containing missing values (`geom_step()`).

## Warning: Removed 85 rows containing missing values (`geom_step()`).
```



runPlasmode

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

```r
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 0.51 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 0.141 secs
## Running Cyclops
## Done.
## GLM fit status:  OK
## Creating variable importance data frame
## Prediction took 0.146 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.189 secs
## Prediction took 0.254 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.193 secs
## Prediction took 0.258 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.