

# VignetteSurvivalTimes

2022-11-10

## Contents

<b>Survival times</b>	<b>2</b>
Loading the plpData . . . . .	2
Defining a training set. . . . .	4
Fitting the model with censoring . . . . .	6
Generating new outcomes times . . . . .	6
Defining an unfitted model . . . . .	7
Defining an unfitted model with censoring . . . . .	8
Adjusting the BaselineSurvival . . . . .	9
Plotting Kaplan Meier estimates . . . . .	10
Possible extencions . . . . .	11

first we make sure we have the uptodate package

```
remotes::install_github("GidiusVanDeKamp/PlasmodeSim")
```

```
## cli      (3.3.0 -> 3.4.1) [CRAN]
## vctrs    (0.5.0 -> 0.5.1) [CRAN]
##
##   There is a binary version available but the source version is later:
##       binary source needs_compilation
## vctrs  0.5.0  0.5.1                TRUE
##
## package 'cli' successfully unpacked and MD5 sums checked

## Warning: cannot remove prior installation of package 'cli'

## Warning in file.copy(savedcopy, lib, recursive = TRUE): problem copying C:
## \Users\gidiu\AppData\Local\R\win-library\4.2\OOLOCK\cli\libs\x64\cli.dll to C:
## \Users\gidiu\AppData\Local\R\win-library\4.2\cli\libs\x64\cli.dll: Permission
## denied

## Warning: restored 'cli'

##
## The downloaded binary packages are in
## C:\Users\gidiu\AppData\Local\Temp\RtmpsfCVw7\downloaded_packages

## Warning in i.p(...): installation of package 'vctrs' had non-zero exit status

## * checking for file 'C:\Users\gidiu\AppData\Local\Temp\RtmpsfCVw7\remotes2d804d94782b\GidiusVanDeKamp
## * preparing 'PlasmodeSim':
## * checking DESCRIPTION meta-information ... OK
## * checking for LF line-endings in source and make files and shell scripts
## * checking for empty or unneeded directories
## Omitted 'LazyData' from DESCRIPTION
## * building 'PlasmodeSim_0.1.0.tar.gz'
##
```

## Survival times

In this part we will show how to simulate new survival times. For simulating new censored survival times we need more than one probability, we use the baselinehazard, 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 a accessing some data set.

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

```
# create the database
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 of diclofenac, 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
```

```
# -----
# Points PatientLevelPredictionPackage to the Eunomia database
# Tells Eunomia to extract the cohort stored with id = 4 as the target cohort
# and cohort with id = 3 as the outcome cohort. The other settings (...Schema)
# tell the database where to look for the target and the outcome cohorts
# -----
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
  )

  # Use ?FeatureExtraction::createCovariateSettings to see what the options are
  # There are a lot...
  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
  )

  # issue with studyStartDate/studyEndDate
  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 0.375 secs

```

## Defining a training set.

Most of the time we split the dataset into training and a test set. In order to prepare the data for fitting the model we have the function `MakeTraingSet`. What copies features of the function `patientLevelPrediction::runPlp`. In order to run it we have to create our settings: `populationSettings`, `executeSettings`, `splitSettings`, `sampleSettings`, `featureEngineeringSettings`, `preprocessSe` 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::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
## 103 covariates in train data
## Test Set:
## 656 patients with 119 outcomes
## Removing 2 redundant covariates
## Normalizing covariates
## Tidying covariates took 0.529 secs
## Train Set:
## Fold 1 658 patients with 120 outcomes - Fold 2 658 patients with 120 outcomes - Fold 3 658 patients
## 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, both of the type specified in the `modelSettings`. It stores these `plpModels` in 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 0.184 secs
## Running Cyclops
## Done.
## GLM fit status: OK
## Creating variable importance data frame
## Prediction took 0.137 secs
```

## Generating new outcomes times

now that we have our model with the censoring, we can simulate new data. We call the function `simulateSurvivaltimesWithCensoring`

```
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 0.176 secs
## Prediction took 0.253 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.176 secs
## Prediction took 0.26 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 to models as a list one can easily generate unsensored outcomes by doing:

```
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 0.185 secs
## Prediction took 0.259 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

just as before we can define a model that hasnt been fitted to the data. we specif u a cox by specifying the coefficients/parameters and the baseline.

```

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 0.182 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. That is because this can be done easily by making to cox models and storing them in a list. this elements in this list should have the names `sensorModel` and `outcomeModel`

```

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

NewOutcomes <- PlasmodeSim::simulateSurvivaltimesWithCensoring(
  sensorModel = 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 0.182 secs
## Prediction took 0.254 secs
## Prediction took 0.179 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 want to get a grip on the outcome count on a specific time one can call the function `adjustBaselineSurvival`. this changes the base line function of a model such 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 specified interval to find this solution.

```
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 0.193 secs
## Prediction took 0.258 secs
```

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

```
## Prediction took 0.179 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.173 secs
## Prediction took 0.249 secs
```

```
head(NewOutcomes)
```

```
##   rowId survivalTime outcomeCount
## 1  2367           13             1
## 2   404           33             1
## 3  1501           14             1
## 4  1545           19             1
## 5  1942            0             1
## 6  1862           18             1
```

## Plotting Kaplan Meier estimates

the function `kaplanMeierPlot` visualised the kamplanmeier estimate of a given dataset. It works with `ggplot`. We can easily compare the simulated data sets with the real dataset by putting them in one plot. For the true data set we set the colour to red.

```
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 0.189 secs
## Prediction took 0.243 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.183 secs
## Prediction took 0.249 secs
```

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

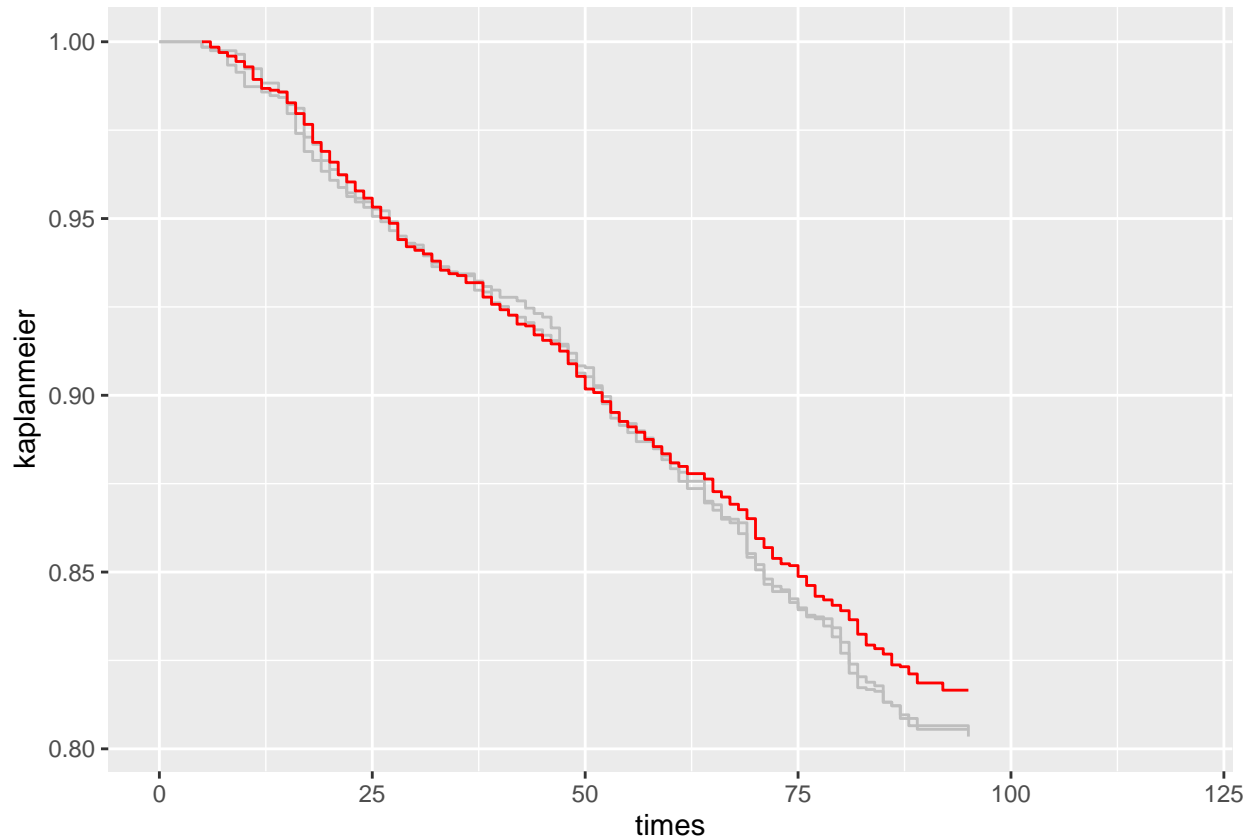
```
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.172 secs
## Prediction took 0.254 secs
## Removing infrequent and redundant covariates and normalizing
## Removing infrequent and redundant covariates covariates and normalizing took 0.178 secs
## Prediction took 0.249 secs
```

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

```
## Warning: Removed 499 rows containing missing values ('geom_step()').
```

```
## Warning: Removed 497 rows containing missing values ('geom_step()').
```

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



## Possible extensions

Here is a list of future 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 a different model for the censoring.
- take a look at the feature engineering in the `definecoxmodel` function
- add more functions that define unfitted models.
- make it faster by filtering the population on the row ids drawn, before making their outcomes.