Package ‘PatientLevelPrediction’

February 11, 2025

Type Package

Title Develop Clinical Prediction Models Using the Common Data Model

Version 6.4.0

Date 2025-02-05

Description A user friendly way to create patient level prediction models using the Observational Medical Outcomes Partnership Common Data Model. Given a cohort of interest and an outcome of interest, the package can use data in the Common Data Model to build a large set of features. These features can then be used to fit a predictive model with a number of machine learning algorithms. This is further described in Reps (2017) <doi:10.1093/jamia/ocy032>.

License Apache License 2.0

URL `https://ohdsi.github.io/PatientLevelPrediction/`, `https://github.com/OHDSI/PatientLevelPrediction`

BugReports `https://github.com/OHDSI/PatientLevelPrediction/issues`

VignetteBuilder knitr

Depends R (>= 4.0.0)

Imports Andromeda, Cyclops (>= 3.0.0), DatabaseConnector (>= 6.0.0), digest, dplyr, FeatureExtraction (>= 3.0.0), Matrix, memuse, ParallelLogger (>= 2.0.0), pROC, PRROC, rlang, SqlRender (>= 1.1.3), tidyr, utils

Suggests curl, Eunomia (>= 2.0.0), glmnet, ggplot2, gridExtra, IterativeHardThresholding, knitr, lightgbm, Metrics, mgcv, OhdsiShinyAppBuilder (>= 1.0.0), parallel, polspline, readr, ResourceSelection, ResultModelManager (>= 0.2.0), reticulate (>= 1.30), rmarkdown, RSQLite, scoring, survival, survminer, testthat, withr, xgboost (> 1.3.2.1)

RoxygenNote 7.3.2

Encoding UTF-8

Config/testthat/edition 3

NeedsCompilation no

Author Egill Fridgeirsson [aut, cre],
Jenna Reps [aut],
Martijn Schuemie [aut],
Marc Suchard [aut],
Patrick Ryan [aut],
Peter Rijnbeek [aut],
Observational Health Data Science and Informatics [cph]

Maintainer Egill Fridgeirsson <e.fridgeirsson@erasmusmc.nl>

Repository CRAN

Date/Publication 2025-02-11 09:30:14 UTC

Contents

`averagePrecision` *Calculate the average precision*

Description

Calculate the average precision

Usage

```
averagePrecision(prediction)
```

Arguments

`prediction`	A prediction object

Details

Calculates the average precision from a predition object

Value

The average precision value

Examples

```
prediction <- data.frame(
  value = c(0.1, 0.2, 0.3, 0.4, 0.5),
  outcomeCount = c(0, 1, 0, 1, 1)
)
averagePrecision(prediction)
```

`brierScore` *brierScore*

Description

brierScore

Usage

```
brierScore(prediction)
```

Arguments

`prediction`	A prediction dataframe

Details

Calculates the brierScore from prediction object

Value

A list containing the brier score and the scaled brier score

Examples

```
prediction <- data.frame(
  value = c(0.1, 0.2, 0.3, 0.4, 0.5),
  outcomeCount = c(0, 1, 0, 1, 1))
brierScore(prediction)
```

calibrationLine *calibrationLine*

Description

calibrationLine

Usage

```
calibrationLine(prediction, numberOfStrata = 10)
```

Arguments

`prediction`	A prediction object
`numberOfStrata`	The number of groups to split the prediction into

Value

A list containing the calibrationLine coefficients, the aggregate data used to fit the line and the Hosmer-Lemeshow goodness of fit test

Examples

```
prediction <- data.frame(
  value = c(0.1, 0.2, 0.3, 0.4, 0.5),
  outcomeCount = c(0, 1, 0, 1, 1))
calibrationLine(prediction, numberOfStrata = 1)
```

`computeAuc` *Compute the area under the ROC curve*

Description

Compute the area under the ROC curve

Usage

```
computeAuc(prediction, confidenceInterval = FALSE)
```

Arguments

`prediction`	A prediction object as generated using the `predict` functions.
`confidenceInterval`	Should 95 percebt confidence intervals be computed?

Details

Computes the area under the ROC curve for the predicted probabilities, given the true observed outcomes.

Value

A data.frame containing the AUC and optionally the 95% confidence interval

Examples

```
prediction <- data.frame(
  value = c(0.1, 0.2, 0.3, 0.4, 0.5),
  outcomeCount = c(0, 1, 0, 1, 1))
computeAuc(prediction)
```

`computeGridPerformance`
Computes grid performance with a specified performance function

---

Description

Computes grid performance with a specified performance function

Usage

```
computeGridPerformance(prediction, param, performanceFunct = "computeAuc")
```

Arguments

`prediction`	a dataframe with predictions and outcomeCount per rowId
`param`	a list of hyperparameters
`performanceFunct`	a string specifying which performance function to use . Default `'compute_AUC'`

Value

A list with overview of the performance

Examples

```
prediction <- data.frame(rowId = c(1, 2, 3, 4, 5),
                         outcomeCount = c(0, 1, 0, 1, 0),
                         value = c(0.1, 0.9, 0.2, 0.8, 0.3),
                         index = c(1, 1, 1, 1, 1))
param <- list(hyperParam1 = 5, hyperParam2 = 100)
computeGridPerformance(prediction, param, performanceFunct = "computeAuc")
```

`configurePython` *Sets up a python environment to use for PLP (can be conda or venv)*

Description

Sets up a python environment to use for PLP (can be conda or venv)

Usage

```
configurePython(envname = "PLP", envtype = NULL, condaPythonVersion = "3.11")
```

Arguments

`envname`	A string for the name of the virtual environment (default is 'PLP')
`envtype`	An option for specifying the environment as'conda' or 'python'. If NULL then the default is 'conda' for windows users and 'python' for non-windows users
`condaPythonVersion`	String, Python version to use when creating a conda environment

Details

This function creates a python environment that can be used by PatientLevelPrediction and installs all the required package dependancies.

Value

location of the created conda or virtual python environment

Examples

```
## Not run:
 configurePython(envname="PLP", envtype="conda")

## End(Not run)
```

`covariateSummary` *covariateSummary*

Description

Summarises the covariateData to calculate the mean and standard deviation per covariate if the labels are given it also stratifies this by class label and if the trainRowIds and testRowIds specifying the patients in the train/test sets respectively are input, these values are also stratified by train and test set

Usage

```
covariateSummary(
  covariateData,
  cohort,
  labels = NULL,
  strata = NULL,
  variableImportance = NULL,
  featureEngineering = NULL
)
```

Arguments

`covariateData`	The covariateData part of the plpData that is extracted using `getPlpData`
`cohort`	The patient cohort to calculate the summary
`labels`	A data.frame with the columns rowId and outcomeCount
`strata`	A data.frame containing the columns rowId, strataName
`variableImportance`	A data.frame with the columns covariateId and value (the variable importance value)
`featureEngineering`	(currently not used) A function or list of functions specifying any feature engineering to create covariates before summarising

Details

The function calculates various metrics to measure the performance of the model

Value

A data.frame containing: CovariateCount, CovariateMean and CovariateStDev for any specified stratification

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=100)
covariateSummary <- covariateSummary(plpData$covariateData, plpData$cohorts)
head(covariateSummary)
```

---

createCohortCovariateSettings

Extracts covariates based on cohorts

Description

Extracts covariates based on cohorts

Usage

```
createCohortCovariateSettings(
  cohortName,
  settingId,
  cohortDatabaseSchema = NULL,
  cohortTable = NULL,
  cohortId,
  startDay = -30,
  endDay = 0,
  count = FALSE,
  ageInteraction = FALSE,
  lnAgeInteraction = FALSE,
  analysisId = 456
)
```

Arguments

`cohortName`	Name for the cohort
`settingId`	A unique id for the covariate time and
`cohortDatabaseSchema`	The schema of the database with the cohort. If nothing is specified then the cohortDatabaseSchema from databaseDetails at runtime is used.
`cohortTable`	the table name that contains the covariate cohort. If nothing is specified then the cohortTable from databaseDetails at runtime is used.
`cohortId`	cohort id for the covariate cohort
`startDay`	The number of days prior to index to start observing the cohort

`endDay`	The number of days prior to index to stop observing the cohort
`count`	If FALSE the covariate value is binary (1 means cohort occurred between index+startDay and index+endDay, 0 means it did not) If TRUE then the covariate value is the number of unique cohort_start_dates between index+startDay and index+endDay
`ageInteraction`	If TRUE multiple covariate value by the patient's age in years
`lnAgeInteraction`	If TRUE multiple covariate value by the log of the patient's age in years
`analysisId`	The analysisId for the covariate

Details

The user specifies a cohort and time period and then a covariate is constructed whether they are in the cohort during the time periods relative to target population cohort index

Value

An object of class `covariateSettings` specifying how to create the cohort covariate with the covariateId cohortId x 100000 + settingId x 1000 + analysisId

Examples

```
createCohortCovariateSettings(cohortName="testCohort",
                              settingId=1,
                              cohortId=1,
                              cohortDatabaseSchema="cohorts",
                              cohortTable="cohort_table")
```

`createDatabaseDetails` *Create a setting that holds the details about the cdmDatabase connection for data extraction*

Description

Create a setting that holds the details about the cdmDatabase connection for data extraction

Usage

```
createDatabaseDetails(
  connectionDetails,
  cdmDatabaseSchema,
  cdmDatabaseName,
  cdmDatabaseId,
  tempEmulationSchema = cdmDatabaseSchema,
  cohortDatabaseSchema = cdmDatabaseSchema,
  cohortTable = "cohort",
  outcomeDatabaseSchema = cdmDatabaseSchema,
  outcomeTable = cohortTable,
  targetId = NULL,
  outcomeIds = NULL,
  cdmVersion = 5,
  cohortId = NULL
)
```

Arguments

`connectionDetails`	An R object of type `connectionDetails` created using the function `createConnectionDetails` in the `DatabaseConnector` package.
`cdmDatabaseSchema`	The name of the database schema that contains the OMOP CDM instance. Requires read permissions to this database. On SQL Server, this should specifiy both the database and the schema, so for example 'cdm_instance.dbo'.
`cdmDatabaseName`	A string with the name of the database - this is used in the shiny app and when externally validating models to name the result list and to specify the folder name when saving validation results (defaults to cdmDatabaseSchema if not specified)
`cdmDatabaseId`	A string with a unique identifier for the database and version - this is stored in the plp object for future reference and used by the shiny app (defaults to cdmDatabaseSchema if not specified)
`tempEmulationSchema`	For dmbs like Oracle only: the name of the database schema where you want all temporary tables to be managed. Requires create/insert permissions to this database.
`cohortDatabaseSchema`	The name of the database schema that is the location where the target cohorts are available. Requires read permissions to this database.
`cohortTable`	The tablename that contains the target cohorts. Expectation is cohortTable has format of COHORT table: COHORT_DEFINITION_ID, SUBJECT_ID, COHORT_START_DATE, COHORT_END_DATE.
`outcomeDatabaseSchema`	The name of the database schema that is the location where the data used to define the outcome cohorts is available. Requires read permissions to this database.
`outcomeTable`	The tablename that contains the outcome cohorts. Expectation is outcomeTable has format of COHORT table: COHORT_DEFINITION_ID, SUBJECT_ID, COHORT_START_DATE, COHORT_END_DATE.
`targetId`	An integer specifying the cohort id for the target cohort
`outcomeIds`	A single integer or vector of integers specifying the cohort ids for the outcome cohorts
`cdmVersion`	Define the OMOP CDM version used: currently support "4" and "5".
`cohortId`	(depreciated: use targetId) old input for the target cohort id

Details

This function simply stores the settings for communicating with the cdmDatabase when extracting the target cohort and outcomes

Value

A list with the the database specific settings:

- `connectionDetails`: An R object of type `connectionDetails` created using the function `createConnectionDetails` in the `DatabaseConnector` package.
- `cdmDatabaseSchema`: The name of the database schema that contains the OMOP CDM instance.
- `cdmDatabaseName`: A string with the name of the database - this is used in the shiny app and when externally validating models to name the result list and to specify the folder name when saving validation results (defaults to cdmDatabaseSchema if not specified).
- `cdmDatabaseId`: A string with a unique identifier for the database and version - this is stored in the plp object for future reference and used by the shiny app (defaults to cdmDatabaseSchema if not specified).
- `tempEmulationSchema`: The name of a databae schema where you want all temporary tables to be managed. Requires create/insert permissions to this database.
- `cohortDatabaseSchema`: The name of the database schema that is the location where the target cohorts are available. Requires read permissions to this schema.
- `cohortTable`: The tablename that contains the target cohorts. Expectation is cohortTable has format of COHORT table: COHORT_DEFINITION_ID, SUBJECT_ID, COHORT_START_DATE, COHORT_END_DATE.
- `outcomeDatabaseSchema`: The name of the database schema that is the location where the data used to define the outcome cohorts is available. Requires read permissions to this database.
- `outcomeTable`: The tablename that contains the outcome cohorts. Expectation is outcomeTable has format of COHORT table: COHORT_DEFINITION_ID, SUBJECT_ID, COHORT_START_DATE, COHORT_END_DATE.
- `targetId`: An integer specifying the cohort id for the target cohort
- `outcomeIds`: A single integer or vector of integers specifying the cohort ids for the outcome cohorts
- `cdmVersion`: Define the OMOP CDM version used: currently support "4" and "5".

Examples

```
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
# create the database details for Eunomia example database
createDatabaseDetails(
  connectionDetails = connectionDetails,
  cdmDatabaseSchema = "main",
  cdmDatabaseName = "main",
  cohortDatabaseSchema = "main",
  cohortTable = "cohort",
  outcomeDatabaseSchema = "main",
  outcomeTable = "cohort",
  targetId = 1, # users of celecoxib
  outcomeIds = 3, # GIbleed
  cdmVersion = 5)
```

createDatabaseSchemaSettings

Create the PatientLevelPrediction database result schema settings

Description

This function specifies where the results schema is and lets you pick a different schema for the cohorts and databases

Usage

```
createDatabaseSchemaSettings(
  resultSchema = "main",
  tablePrefix = "",
  targetDialect = "sqlite",
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  cohortDefinitionSchema = resultSchema,
  tablePrefixCohortDefinitionTables = tablePrefix,
  databaseDefinitionSchema = resultSchema,
  tablePrefixDatabaseDefinitionTables = tablePrefix
)
```

Arguments

`resultSchema`	(string) The name of the database schema with the result tables.
`tablePrefix`	(string) A string that appends to the PatientLevelPrediction result tables
`targetDialect`	(string) The database management system being used
`tempEmulationSchema`	(string) The temp schema used when the database management system is oracle
`cohortDefinitionSchema`	(string) The name of the database schema with the cohort definition tables (defaults to resultSchema).
`tablePrefixCohortDefinitionTables`	(string) A string that appends to the cohort definition tables
`databaseDefinitionSchema`	(string) The name of the database schema with the database definition tables (defaults to resultSchema).
`tablePrefixDatabaseDefinitionTables`	(string) A string that appends to the database definition tables

Details

This function can be used to specify the database settings used to upload PatientLevelPrediction results into a database

Value

Returns a list of class 'plpDatabaseResultSchema' with all the database settings

Examples

```
createDatabaseSchemaSettings(resultSchema = "cdm",
                             tablePrefix = "plp_")
```

`createDefaultExecuteSettings`

Creates default list of settings specifying what parts of runPlp to execute

---

Description

Creates default list of settings specifying what parts of runPlp to execute

Usage

```
createDefaultExecuteSettings()
```

Details

runs split, preprocess, model development and covariate summary

Value

list with TRUE for split, preprocess, model development and covariate summary

Examples

```
createDefaultExecuteSettings()
```

createDefaultSplitSetting

Create the settings for defining how the plpData are split into test/validation/train sets using default splitting functions (either random stratified by outcome, time or subject splitting)

Description

Create the settings for defining how the plpData are split into test/validation/train sets using default splitting functions (either random stratified by outcome, time or subject splitting)

Usage

```
createDefaultSplitSetting(
  testFraction = 0.25,
  trainFraction = 0.75,
  splitSeed = sample(1e+05, 1),
  nfold = 3,
  type = "stratified"
)
```

Arguments

`testFraction`	(numeric) A real number between 0 and 1 indicating the test set fraction of the data
`trainFraction`	(numeric) A real number between 0 and 1 indicating the train set fraction of the data. If not set train is equal to 1 - test
`splitSeed`	(numeric) A seed to use when splitting the data for reproducibility (if not set a random number will be generated)
`nfold`	(numeric) An integer > 1 specifying the number of folds used in cross validation
`type`	(character) Choice of: • 'stratified' Each data point is randomly assigned into the test or a train fold set but this is done stratified such that the outcome rate is consistent in each partition • 'time' Older data are assigned into the training set and newer data are assigned into the test set • 'subject' Data are partitioned by subject, if a subject is in the data more than once, all the data points for the subject are assigned either into the test data or into the train data (not both).

Details

Returns an object of class `splitSettings` that specifies the splitting function that will be called and the settings

Value

An object of class `splitSettings`

Examples

```
createDefaultSplitSetting(testFraction=0.25, trainFraction=0.75, nfold=3,
                          splitSeed=42)
```

`createExecuteSettings` *Creates list of settings specifying what parts of runPlp to execute*

Description

Creates list of settings specifying what parts of runPlp to execute

Usage

```
createExecuteSettings(
  runSplitData = FALSE,
  runSampleData = FALSE,
  runFeatureEngineering = FALSE,
  runPreprocessData = FALSE,
  runModelDevelopment = FALSE,
  runCovariateSummary = FALSE
)
```

Arguments

`runSplitData`	TRUE or FALSE whether to split data into train/test
`runSampleData`	TRUE or FALSE whether to over or under sample
`runFeatureEngineering`	TRUE or FALSE whether to do feature engineering
`runPreprocessData`	TRUE or FALSE whether to do preprocessing
`runModelDevelopment`	TRUE or FALSE whether to develop the model
`runCovariateSummary`	TRUE or FALSE whether to create covariate summary

Details

define what parts of runPlp to execute

Value

list with TRUE/FALSE for each part of runPlp

Examples

```
# create settings with only split and model development
createExecuteSettings(runSplitData = TRUE, runModelDevelopment = TRUE)
```

---

createExistingSplitSettings

Create the settings for defining how the plpData are split into test/validation/train sets using an existing split - good to use for reproducing results from a different run

---

Description

Create the settings for defining how the plpData are split into test/validation/train sets using an existing split - good to use for reproducing results from a different run

Usage

```
createExistingSplitSettings(splitIds)
```

Arguments

`splitIds`	(data.frame) A data frame with rowId and index columns of type integer/numeric. Index is -1 for test set, positive integer for train set folds

Value

An object of class `splitSettings`

Examples

```
# rowId 1 is in fold 1, rowId 2 is in fold 2, rowId 3 is in the test set
# rowId 4 is in fold 1, rowId 5 is in fold 2
createExistingSplitSettings(splitIds = data.frame(rowId = c(1, 2, 3, 4, 5),
                                                  index = c(1, 2, -1, 1, 2)))
```

---

createFeatureEngineeringSettings

Create the settings for defining any feature engineering that will be done

---

Description

Create the settings for defining any feature engineering that will be done

Usage

```
createFeatureEngineeringSettings(type = "none")
```

Arguments

`type`	(character) Choice of: • 'none' No feature engineering - this is the default

Details

Returns an object of class `featureEngineeringSettings` that specifies the sampling function that will be called and the settings

Value

An object of class `featureEngineeringSettings`

Examples

```
createFeatureEngineeringSettings(type = "none")
```

`createGlmModel` *createGlmModel*

Description

Create a generalized linear model that can be used in the PatientLevelPrediction package.

Usage

```
createGlmModel(coefficients, intercept = 0, mapping = "logistic")
```

Arguments

`coefficients`	A dataframe containing two columns, coefficients and covariateId, both of type numeric. The covariateId column must contain valid covariateIds that match those used in the `FeatureExtraction` package.
`intercept`	A numeric value representing the intercept of the model.
`mapping`	A string representing the mapping from the linear predictors to outcome probabilities. For generalized linear models this is the inverse of the link function. Supported values is only "logistic" for logistic regression model at the moment.

Value

A model object containing the model (Coefficients and intercept) and the prediction function.

Examples

```
coefficients <- data.frame(
  covariateId = c(1002),
  coefficient = c(0.05))
model <- createGlmModel(coefficients, intercept = -2.5)
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=50)
prediction <- predictPlp(model, plpData, plpData$cohorts)
# see the predicted risk values
prediction$value
```

`createIterativeImputer`

Create Iterative Imputer settings

Description

This function creates the settings for an iterative imputer which first removes features with more than `missingThreshold` missing values and then imputes the missing values iteratively using chained equations

Usage

```
createIterativeImputer(
  missingThreshold = 0.3,
  method = "pmm",
  methodSettings = list(pmm = list(k = 5, iterations = 5))
)
```

Arguments

`missingThreshold`	The threshold for missing values to remove a feature
`method`	The method to use for imputation, currently only "pmm" is supported
`methodSettings`	A list of settings for the imputation method to use. Currently only "pmm" is supported with the following settings: • k: The number of donors to use for matching • iterations: The number of iterations to use for imputation

Value

The settings for the iterative imputer of class `featureEngineeringSettings`

Examples

```
# create imputer to impute values with missingness less than 30% using
# predictive mean matching in 5 iterations with 5 donors
createIterativeImputer(missingThreshold = 0.3, method = "pmm",
                       methodSettings = list(pmm = list(k = 5, iterations = 5)))
```

createLearningCurve *createLearningCurve*

Description

Creates a learning curve object, which can be plotted using the `plotLearningCurve()` function.

Usage

```
createLearningCurve(
  plpData,
  outcomeId,
  parallel = TRUE,
  cores = 4,
  modelSettings,
  saveDirectory = NULL,
  analysisId = "learningCurve",
  populationSettings = createStudyPopulationSettings(),
  splitSettings = createDefaultSplitSetting(),
  trainFractions = c(0.25, 0.5, 0.75),
  trainEvents = NULL,
  sampleSettings = createSampleSettings(),
  featureEngineeringSettings = createFeatureEngineeringSettings(),
 preprocessSettings = createPreprocessSettings(minFraction = 0.001, normalize = TRUE),
  logSettings = createLogSettings(),
 executeSettings = createExecuteSettings(runSplitData = TRUE, runSampleData = FALSE,
   runFeatureEngineering = FALSE, runPreprocessData = TRUE, runModelDevelopment = TRUE,
    runCovariateSummary = FALSE)
)
```

Arguments

`plpData`	An object of type `plpData` - the patient level prediction data extracted from the CDM.
`outcomeId`	(integer) The ID of the outcome.
`parallel`	Whether to run the code in parallel
`cores`	The number of computer cores to use if running in parallel
`modelSettings`	An object of class `modelSettings` created using one of the function:

- `setLassoLogisticRegression()` A lasso logistic regression model
- `setGradientBoostingMachine()` A gradient boosting machine
- `setAdaBoost()` An ada boost model
- `setRandomForest()` A random forest model
- `setDecisionTree()` A decision tree model
- `setKNN()` A KNN model

`saveDirectory`	The path to the directory where the results will be saved (if NULL uses working directory)
`analysisId`	(integer) Identifier for the analysis. It is used to create, e.g., the result folder. Default is a timestamp.
`populationSettings`	An object of type `populationSettings` created using `createStudyPopulationSettings` that specifies how the data class labels are defined and addition any exclusions to apply to the plpData cohort
`splitSettings`	An object of type `splitSettings` that specifies how to split the data into train/validation/test. The default settings can be created using `createDefaultSplitSetting`.
`trainFractions`	A list of training fractions to create models for. Note, providing `trainEvents` will override your input to `trainFractions`.
`trainEvents`	Events have shown to be determinant of model performance. Therefore, it is recommended to provide `trainEvents` rather than `trainFractions`. Note, providing `trainEvents` will override your input to `trainFractions`. The format should be as follows: • `c(500, 1000, 1500)` - a list of training events
`sampleSettings`	An object of type `sampleSettings` that specifies any under/over sampling to be done. The default is none.
`featureEngineeringSettings`	An object of `featureEngineeringSettings` specifying any feature engineering to be learned (using the train data)
`preprocessSettings`	An object of `preprocessSettings`. This setting specifies the minimum fraction of target population who must have a covariate for it to be included in the model training and whether to normalise the covariates before training
`logSettings`	An object of `logSettings` created using `createLogSettings` specifying how the logging is done
`executeSettings`	An object of `executeSettings` specifying which parts of the analysis to run

Value

A learning curve object containing the various performance measures obtained by the model for each training set fraction. It can be plotted using `plotLearningCurve`.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
outcomeId <- 3
modelSettings <- setLassoLogisticRegression(seed=42)
learningCurve <- createLearningCurve(plpData, outcomeId, modelSettings = modelSettings,
saveDirectory = file.path(tempdir(), "learningCurve"), cores = 2)
# clean up
unlink(file.path(tempdir(), "learningCurve"), recursive = TRUE)
```

`createLogSettings` *Create the settings for logging the progression of the analysis*

Description

Create the settings for logging the progression of the analysis

Usage

```
createLogSettings(
  verbosity = "DEBUG",
  timeStamp = TRUE,
  logName = "runPlp Log"
)
```

Arguments

`verbosity`	Sets the level of the verbosity. If the log level is at or higher in priority than the logger threshold, a message will print. The levels are: • DEBUG Highest verbosity showing all debug statements • TRACE Showing information about start and end of steps • INFO Show informative information (Default) • WARN Show warning messages • ERROR Show error messages • FATAL Be silent except for fatal errors
`timeStamp`	If TRUE a timestamp will be added to each logging statement. Automatically switched on for TRACE level.
`logName`	A string reference for the logger

Details

Returns an object of class `logSettings` that specifies the logger settings

Value

An object of class `logSettings` containing the settings for the logger

Examples

```
# create a log settings object with DENUG verbosity, timestamp and log name
# "runPlp Log". This needs to be passed to `runPlp`.
createLogSettings(verbosity = "DEBUG", timeStamp = TRUE, logName = "runPlp Log")
```

`createModelDesign` *Specify settings for developing a single model*

Description

Specify settings for developing a single model

Usage

```
createModelDesign(
  targetId = NULL,
  outcomeId = NULL,
  restrictPlpDataSettings = createRestrictPlpDataSettings(),
  populationSettings = createStudyPopulationSettings(),
  covariateSettings = FeatureExtraction::createDefaultCovariateSettings(),
  featureEngineeringSettings = NULL,
  sampleSettings = NULL,
  preprocessSettings = NULL,
  modelSettings = NULL,
  splitSettings = createDefaultSplitSetting(),
  runCovariateSummary = TRUE
)
```

Arguments

`targetId`	The id of the target cohort that will be used for data extraction (e.g., the ATLAS id)
`outcomeId`	The id of the outcome that will be used for data extraction (e.g., the ATLAS id)
`restrictPlpDataSettings`	The settings specifying the extra restriction settings when extracting the data created using `createRestrictPlpDataSettings()`.
`populationSettings`	The population settings specified by `createStudyPopulationSettings()`
`covariateSettings`	The covariate settings, this can be a list or a single `'covariateSetting'` object.
`featureEngineeringSettings`	Either NULL or an object of class `featureEngineeringSettings` specifying any feature engineering used during model development
`sampleSettings`	Either NULL or an object of class `sampleSettings` with the over/under sampling settings used for model development
`preprocessSettings`	Either NULL or an object of class `preprocessSettings` created using `createPreprocessingSettings(`
`modelSettings`	The model settings such as `setLassoLogisticRegression()`
`splitSettings`	The train/validation/test splitting used by all analyses created using `createDefaultSplitSetting()`
`runCovariateSummary`	Whether to run the covariateSummary

Details

This specifies a single analysis for developing as single model

Value

A list with analysis settings used to develop a single prediction model

Examples

```
# L1 logistic regression model to predict the outcomeId 2 using the targetId 2
# with with default population, restrictPlp, split, and covariate settings
createModelDesign(
  targetId = 1,
  outcomeId = 2,
  modelSettings = setLassoLogisticRegression(seed=42),
  populationSettings = createStudyPopulationSettings(),
  restrictPlpDataSettings = createRestrictPlpDataSettings(),
  covariateSettings = FeatureExtraction::createDefaultCovariateSettings(),
  splitSettings = createDefaultSplitSetting(splitSeed = 42),
  runCovariateSummary = TRUE
)
```

`createNormalizer` *Create the settings for normalizing the data @param type The type of normalization to use, either "minmax" or "robust"*

Description

Create the settings for normalizing the data @param type The type of normalization to use, either "minmax" or "robust"

Usage

```
createNormalizer(type = "minmax", settings = list())
```

Arguments

`type`	The type of normalization to use, either "minmax" or "robust"
`settings`	A list of settings for the normalization. For robust normalization, the settings list can contain a boolean value for clip, which clips the values to be between -3 and 3 after normalization. See https://arxiv.org/abs/2407.04491

Value

An object of class `featureEngineeringSettings`

An object of class `featureEngineeringSettings`’

Examples

```
# create a minmax normalizer that normalizes the data between 0 and 1
normalizer <- createNormalizer(type = "minmax")
# create a robust normalizer that normalizes the data by the interquartile range
# and squeezes the values to be between -3 and 3
normalizer <- createNormalizer(type = "robust", settings = list(clip = TRUE))
```

`createPlpResultTables` *Create the results tables to store PatientLevelPrediction models and results into a database*

Description

This function executes a large set of SQL statements to create tables that can store models and results

Usage

```
createPlpResultTables(
  connectionDetails,
  targetDialect = "postgresql",
  resultSchema,
  deleteTables = TRUE,
  createTables = TRUE,
  tablePrefix = "",
  tempEmulationSchema = getOption("sqlRenderTempEmulationSchema"),
  testFile = NULL
)
```

Arguments

connectionDetails	The database connection details
targetDialect	The database management system being used
resultSchema	The name of the database schema that the result tables will be created.
deleteTables	If true any existing tables matching the PatientLevelPrediction result tables names will be deleted
createTables	If true the PatientLevelPrediction result tables will be created
tablePrefix	A string that appends to the PatientLevelPrediction result tables
tempEmulationSchema	The temp schema used when the database management system is oracle
testFile	(used for testing) The location of an sql file with the table creation code

Details

This function can be used to create (or delete) PatientLevelPrediction result tables

Value

Returns NULL but creates or deletes the required tables in the specified database schema(s).

Examples

```
# create a sqlite database with the PatientLevelPrediction result tables
connectionDetails <- DatabaseConnector::createConnectionDetails(
  dbms = "sqlite",
  server = file.path(tempdir(), "test.sqlite"))
createPlpResultTables(connectionDetails = connectionDetails,
                      targetDialect = "sqlite",
                      resultSchema = "main",
                      tablePrefix = "plp_")
# delete the tables
createPlpResultTables(connectionDetails = connectionDetails,
                      targetDialect = "sqlite",
                      resultSchema = "main",
                      deleteTables = TRUE,
                      createTables = FALSE,
                      tablePrefix = "plp_")
# clean up the database file
unlink(file.path(tempdir(), "test.sqlite"))
```

createPreprocessSettings

Create the settings for preprocessing the trainData.

Description

Create the settings for preprocessing the trainData.

Usage

```
createPreprocessSettings(
  minFraction = 0.001,
  normalize = TRUE,
  removeRedundancy = TRUE
)
```

Arguments

`minFraction`	The minimum fraction of target population who must have a covariate for it to be included in the model training
`normalize`	Whether to normalise the covariates before training (Default: TRUE)

`removeRedundancy`	Whether to remove redundant features (Default: TRUE) Redundant features are features that within an analysisId together cover all observations. For example with ageGroups, if you have ageGroup 0-18 and 18-100 and all patients are in one of these groups, then one of these groups is redundant.

Details

Returns an object of class `preprocessingSettings` that specifies how to preprocess the training data

Value

An object of class `preprocessingSettings`

Examples

```
# Create the settings for preprocessing, remove no features, normalise the data
createPreprocessSettings(minFraction = 0.0, normalize = TRUE, removeRedundancy = FALSE)
```

`createRandomForestFeatureSelection`
Create the settings for random foreat based feature selection

---

Description

Create the settings for random foreat based feature selection

Usage

```
createRandomForestFeatureSelection(ntrees = 2000, maxDepth = 17)
```

Arguments

`ntrees`	number of tree in forest
`maxDepth`	MAx depth of each tree

Details

Returns an object of class `featureEngineeringSettings` that specifies the sampling function that will be called and the settings

Value

An object of class `featureEngineeringSettings`

Examples

```
## Not run: #' featureSelector <- createRandomForestFeatureSelection(ntrees = 2000, maxDepth = 10)
```

createRareFeatureRemover

Create the settings for removing rare features

Description

Create the settings for removing rare features

Usage

```
createRareFeatureRemover(threshold = 0.001)
```

Arguments

threshold	The minimum fraction of the training data that must have a feature for it to be included

Value

An object of class `featureEngineeringSettings`

Examples

```
# create a rare feature remover that removes features that are present in less
# than 1% of the population
rareFeatureRemover <- createRareFeatureRemover(threshold = 0.01)
plpData <- getEunomiaPlpData()
analysisId <- "rareFeatureRemover"
saveLocation <- file.path(tempdir(), analysisId)
results <- runPlp(
  plpData = plpData,
  featureEngineeringSettings = rareFeatureRemover,
  outcomeId = 3,
 executeSettings = createExecuteSettings(
   runModelDevelopment = TRUE,
   runSplitData = TRUE,
   runFeatureEngineering = TRUE),
 saveDirectory = saveLocation,
 analysisId = analysisId)
# clean up
unlink(saveLocation, recursive = TRUE)
```

createRestrictPlpDataSettings

createRestrictPlpDataSettings define extra restriction settings when calling getPlpData

Description

This function creates the settings used to restrict the target cohort when calling getPlpData

Usage

```
createRestrictPlpDataSettings(
  studyStartDate = "",
  studyEndDate = "",
  firstExposureOnly = FALSE,
  washoutPeriod = 0,
  sampleSize = NULL
)
```

Arguments

`studyStartDate`	A calendar date specifying the minimum date that a cohort index date can appear. Date format is 'yyyymmdd'.
`studyEndDate`	A calendar date specifying the maximum date that a cohort index date can appear. Date format is 'yyyymmdd'. Important: the study end data is also used to truncate risk windows, meaning no outcomes beyond the study end date will be considered.
`firstExposureOnly`	Should only the first exposure per subject be included? Note that this is typically done in the `createStudyPopulation` function, but can already be done here for efficiency reasons.
`washoutPeriod`	The mininum required continuous observation time prior to index date for a person to be included in the at risk cohort. Note that this is typically done in the `createStudyPopulation` function, but can already be done here for efficiency reasons.
`sampleSize`	If not NULL, the number of people to sample from the target cohort

Details

Users need to specify the extra restrictions to apply when downloading the target cohort

Value

A setting object of class `restrictPlpDataSettings` containing a list of the settings:

- `studyStartDate`: A calendar date specifying the minimum date that a cohort index date can appear

- `studyEndDate`: A calendar date specifying the maximum date that a cohort index date can appear
- `firstExposureOnly`: Should only the first exposure per subject be included
- `washoutPeriod`: The mininum required continuous observation time prior to index date for a person to be included in the at risk cohort
- `sampleSize`: If not NULL, the number of people to sample from the target cohort

Examples

```
# restrict to 2010, first exposure only, require washout period of 365 day
# and sample 1000 people
createRestrictPlpDataSettings(studyStartDate = "20100101", studyEndDate = "20101231",
firstExposureOnly = TRUE, washoutPeriod = 365, sampleSize = 1000)
```

---

createSampleSettings *Create the settings for defining how the trainData from* `splitData` *are sampled using default sample functions.*

Description

Create the settings for defining how the trainData from `splitData` are sampled using default sample functions.

Usage

```
createSampleSettings(
  type = "none",
  numberOutcomestoNonOutcomes = 1,
  sampleSeed = sample(10000, 1)
)
```

Arguments

`type`	(character) Choice of: • 'none' No sampling is applied - this is the default • 'underSample' Undersample the non-outcome class to make the data more balanced • 'overSample' Oversample the outcome class by adding in each outcome multiple times
`numberOutcomestoNonOutcomes`	(numeric) A numeric specifying the required number of outcomes per non-outcomes
`sampleSeed`	(numeric) A seed to use when splitting the data for reproducibility (if not set a random number will be generated)

Details

Returns an object of class `sampleSettings` that specifies the sampling function that will be called and the settings

Value

An object of class `sampleSettings`

Examples

```
# sample even rate of outcomes to non-outcomes
sampleSetting <- createSampleSettings(type = "underSample",
                                      numberOutcomestoNonOutcomes = 1,
                                      sampleSeed = 42)
```

`createSimpleImputer` *Create Simple Imputer settings*

Description

This function creates the settings for a simple imputer which imputes missing values with the mean or median

Usage

```
createSimpleImputer(method = "mean", missingThreshold = 0.3)
```

Arguments

`method`	The method to use for imputation, either "mean" or "median"
`missingThreshold`	The threshold for missing values to be imputed vs removed

Value

The settings for the single imputer of class `featureEngineeringSettings`

Examples

```
# create imputer to impute values with missingness less than 10% using the median
# of observed values
createSimpleImputer(method = "median", missingThreshold = 0.10)
```

createSklearnModel *Plug an existing scikit learn python model into the PLP framework*

Description

Plug an existing scikit learn python model into the PLP framework

Usage

```
createSklearnModel(
  modelLocation = "/model",
  covariateMap = data.frame(columnId = 1:2, covariateId = c(1, 2), ),
  covariateSettings,
  populationSettings,
  isPickle = TRUE
)
```

Arguments

`modelLocation`	The location of the folder that contains the model as model.pkl
`covariateMap`	A data.frame with the columns: columnId and covariateId. `covariateId` from FeatureExtraction is the standard OHDSI covariateId. `columnId` is the column location the model expects that covariate to be in. For example, if you had a column called 'age' in your model and this was the 3rd column when fitting the model, then the values for columnId would be 3, covariateId would be 1002 (the covariateId for age in years) and
`covariateSettings`	The settings for the standardized covariates
`populationSettings`	The settings for the population, this includes the time-at-risk settings and inclusion criteria.
`isPickle`	If the model should be saved as a pickle set this to TRUE if it should be saved as json set this to FALSE.

Details

This function lets users add an existing scikit learn model that is saved as model.pkl into PLP format. covariateMap is a mapping between standard covariateIds and the model columns. The user also needs to specify the covariate settings and population settings as these are used to determine the standard PLP model design.

Value

An object of class plpModel, this is a list that contains: model (the location of the model.pkl), preprocessing (settings for mapping the covariateIds to the model column mames), modelDesign (specification of the model design), trainDetails (information about the model fitting) and covariateImportance.

You can use the output as an input in PatientLevelPrediction::predictPlp to apply the model and calculate the risk for patients.

`createSplineSettings` *Create the settings for adding a spline for continuous variables*

Description

Create the settings for adding a spline for continuous variables

Usage

```
createSplineSettings(continousCovariateId, knots, analysisId = 683)
```

Arguments

`continousCovariateId`	The covariateId to apply splines to
`knots`	Either number of knots of vector of split values
`analysisId`	The analysisId to use for the spline covariates

Details

Returns an object of class `featureEngineeringSettings` that specifies the sampling function that will be called and the settings

Value

An object of class `featureEngineeringSettings`

Examples

```
# create splines for age (1002) with 5 knots
createSplineSettings(continousCovariateId = 1002, knots = 5, analysisId = 683)
```

`createStratifiedImputationSettings` *Create the settings for using stratified imputation.*

Description

Create the settings for using stratified imputation.

Usage

```
createStratifiedImputationSettings(covariateId, ageSplits = NULL)
```

Arguments

`covariateId`	The covariateId that needs imputed values
`ageSplits`	A vector of age splits in years to create age groups

Details

Returns an object of class `featureEngineeringSettings` that specifies how to do stratified imputation. This function splits the covariate into age groups and fits splines to the covariate within each age group. The spline values are then used to impute missing values.

Value

An object of class `featureEngineeringSettings`

Examples

```
# create a stratified imputation settings for covariate 1050 with age splits
# at 50 and 70
stratifiedImputationSettings <-
  createStratifiedImputationSettings(covariateId = 1050, ageSplits = c(50, 70))
```

`createStudyPopulation` *Create a study population*

Description

Create a study population

Usage

```
createStudyPopulation(
  plpData,
  outcomeId = plpData$metaData$databaseDetails$outcomeIds[1],
  populationSettings = createStudyPopulationSettings(),
  population = NULL
)
```

Arguments

`plpData`	An object of type `plpData` as generated using `getplpData`.
`outcomeId`	The ID of the outcome.
`populationSettings`	An object of class populationSettings created using `createPopulationSettings`
`population`	If specified, this population will be used as the starting point instead of the cohorts in the `plpData` object.

Details

Create a study population by enforcing certain inclusion and exclusion criteria, defining a risk window, and determining which outcomes fall inside the risk window.

Value

A data frame specifying the study population. This data frame will have the following columns:

rowId A unique identifier for an exposure

subjectId The person ID of the subject

cohortStartdate The index date

outcomeCount The number of outcomes observed during the risk window

timeAtRisk The number of days in the risk window

survivalTime The number of days until either the outcome or the end of the risk window

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 100)
# Create study population, require time at risk of 30 days. The risk window is 1 to 90 days.
populationSettings <- createStudyPopulationSettings(requireTimeAtRisk = TRUE,
                                                    minTimeAtRisk = 30,
                                                    riskWindowStart = 1,
                                                    riskWindowEnd = 90)
population <- createStudyPopulation(plpData, outcomeId = 3, populationSettings)
```

`createStudyPopulationSettings` *create the study population settings*

Description

create the study population settings

Usage

```
createStudyPopulationSettings(
  binary = TRUE,
  includeAllOutcomes = TRUE,
  firstExposureOnly = FALSE,
  washoutPeriod = 0,
  removeSubjectsWithPriorOutcome = TRUE,
  priorOutcomeLookback = 99999,
  requireTimeAtRisk = TRUE,
  minTimeAtRisk = 364,
  riskWindowStart = 1,
  startAnchor = "cohort start",
  riskWindowEnd = 365,
  endAnchor = "cohort start",
  restrictTarToCohortEnd = FALSE
)
```

Arguments

`binary`	Forces the outcomeCount to be 0 or 1 (use for binary prediction problems)
`includeAllOutcomes`	(binary) indicating whether to include people with outcomes who are not observed for the whole at risk period
`firstExposureOnly`	Should only the first exposure per subject be included? Note that this is typically done in the `createStudyPopulation` function,
`washoutPeriod`	The mininum required continuous observation time prior to index date for a person to be included in the cohort.
`removeSubjectsWithPriorOutcome`	Remove subjects that have the outcome prior to the risk window start?
`priorOutcomeLookback`	How many days should we look back when identifying prior outcomes?
`requireTimeAtRisk`	Should subject without time at risk be removed?
`minTimeAtRisk`	The minimum number of days at risk required to be included

`riskWindowStart`	The start of the risk window (in days) relative to the index date (+ days of exposure if the `addExposureDaysToStart` parameter is specified).
`startAnchor`	The anchor point for the start of the risk window. Can be "cohort start" or "cohort end".
`riskWindowEnd`	The end of the risk window (in days) relative to the index data (+ days of exposure if the `addExposureDaysToEnd` parameter is specified).
`endAnchor`	The anchor point for the end of the risk window. Can be "cohort start" or "cohort end".
`restrictTarToCohortEnd`	If using a survival model and you want the time-at-risk to end at the cohort end date set this to T

Value

An object of type populationSettings containing all the settings required for creating the study population

Examples

```
# Create study population settings with a washout period of 30 days and a
# risk window of 1 to 90 days
populationSettings <- createStudyPopulationSettings(washoutPeriod = 30,
                                                    riskWindowStart = 1,
                                                    riskWindowEnd = 90)
```

`createTempModelLoc` *Create a temporary model location*

Description

Create a temporary model location

Usage

```
createTempModelLoc()
```

Value

A string for the location of the temporary model location

Examples

```
modelLoc <- createTempModelLoc()
dir.exists(modelLoc)
# clean up
unlink(modelLoc, recursive = TRUE)
```

createUnivariateFeatureSelection

Create the settings for defining any feature selection that will be done

Description

Create the settings for defining any feature selection that will be done

Usage

```
createUnivariateFeatureSelection(k = 100)
```

Arguments

k	This function returns the K features most associated (univariately) to the outcome

Details

Returns an object of class `featureEngineeringSettings` that specifies the function that will be called and the settings. Uses the scikit-learn SelectKBest function with chi2 for univariate feature selection.

Value

An object of class `featureEngineeringSettings`

Examples

```
## Not run:  #' # create a feature selection that selects the 100 most associated features
featureSelector <- createUnivariateFeatureSelection(k = 100)

## End(Not run)
```

createValidationDesign

createValidationDesign - Define the validation design for external validation

Description

createValidationDesign - Define the validation design for external validation

Usage

```
createValidationDesign(
  targetId,
  outcomeId,
  populationSettings = NULL,
  restrictPlpDataSettings = NULL,
  plpModelList,
  recalibrate = NULL,
  runCovariateSummary = TRUE
)
```

Arguments

`targetId`	The targetId of the target cohort to validate on
`outcomeId`	The outcomeId of the outcome cohort to validate on
`populationSettings`	A list of population restriction settings created by `createPopulationSettings`. Default is NULL and then this is taken from the model
`restrictPlpDataSettings`	A list of plpData restriction settings created by `createRestrictPlpDataSettings`. Default is NULL and then this is taken from the model.
`plpModelList`	A list of plpModels objects created by `runPlp` or a path to such objects
`recalibrate`	A vector of characters specifying the recalibration method to apply,
`runCovariateSummary`	whether to run the covariate summary for the validation data

Value

A validation design object of class `validationDesign` or a list of such objects

Examples

```
# create a validation design for targetId 1 and outcomeId 2 one l1 model and
# one gradient boosting model
createValidationDesign(1, 2, plpModelList = list(
"pathToL1model", "PathToGBMModel"))
```

`createValidationSettings` *createValidationSettings define optional settings for performing external validation*

Description

This function creates the settings required by externalValidatePlp

Usage

```
createValidationSettings(recalibrate = NULL, runCovariateSummary = TRUE)
```

Arguments

`recalibrate`	A vector of characters specifying the recalibration method to apply
`runCovariateSummary`	Whether to run the covariate summary for the validation data

Details

Users need to specify whether they want to sample or recalibate when performing external validation

Value

A setting object of class `validationSettings` containing a list of settings for externalValidatePlp

Examples

```
# do weak recalibration and don't run covariate summary
createValidationSettings(recalibrate = "weakRecalibration",
                         runCovariateSummary = FALSE)
```

`diagnoseMultiplePlp` *Run a list of predictions diagnoses*

Description

Run a list of predictions diagnoses

Usage

```
diagnoseMultiplePlp(
  databaseDetails = createDatabaseDetails(),
 modelDesignList = list(createModelDesign(targetId = 1, outcomeId = 2, modelSettings =
    setLassoLogisticRegression()), createModelDesign(targetId = 1, outcomeId = 3,
    modelSettings = setLassoLogisticRegression())),
  cohortDefinitions = NULL,
 logSettings = createLogSettings(verbosity = "DEBUG", timeStamp = TRUE, logName =
    "diagnosePlp Log"),
  saveDirectory = NULL
)
```

Arguments

databaseDetails	The database settings created using `createDatabaseDetails()`
modelDesignList	A list of model designs created using `createModelDesign()`
cohortDefinitions	A list of cohort definitions for the target and outcome cohorts
logSettings	The setting spexcifying the logging for the analyses created using `createLogSettings()`
saveDirectory	Name of the folder where all the outputs will written to.

Details

This function will run all specified prediction design diagnoses.

Value

A data frame with the following columns:

analysisId	The unique identifier for a set of analysis choices.
targetId	The ID of the target cohort populations.
outcomeId	The ID of the outcomeId.
dataLocation	The location where the plpData was saved
the settings ids	The ids for all other settings used for model development.

---

diagnosePlp *diagnostic - Investigates the prediction problem settings - use before training a model*

Description

This function runs a set of prediction diagnoses to help pick a suitable T, O, TAR and determine whether the prediction problem is worth executing.

Usage

```
diagnosePlp(
  plpData = NULL,
  outcomeId,
  analysisId,
  populationSettings,
  splitSettings = createDefaultSplitSetting(),
  sampleSettings = createSampleSettings(),
  saveDirectory = NULL,
  featureEngineeringSettings = createFeatureEngineeringSettings(),
  modelSettings = setLassoLogisticRegression(),
 logSettings = createLogSettings(verbosity = "DEBUG", timeStamp = TRUE, logName =
    "diagnosePlp Log"),
  preprocessSettings = createPreprocessSettings()
)
```

Arguments

`plpData`	An object of type `plpData` - the patient level prediction data extracted from the CDM. Can also include an initial population as plpData$popualtion.
`outcomeId`	(integer) The ID of the outcome.
`analysisId`	(integer) Identifier for the analysis. It is used to create, e.g., the result folder. Default is a timestamp.
`populationSettings`	An object of type `populationSettings` created using `createStudyPopulationSettings` that specifies how the data class labels are defined and addition any exclusions to apply to the plpData cohort
`splitSettings`	An object of type `splitSettings` that specifies how to split the data into train/validation/test. The default settings can be created using `createDefaultSplitSetting`.
`sampleSettings`	An object of type `sampleSettings` that specifies any under/over sampling to be done. The default is none.
`saveDirectory`	The path to the directory where the results will be saved (if NULL uses working directory)
`featureEngineeringSettings`	An object of `featureEngineeringSettings` specifying any feature engineering to be learned (using the train data)
`modelSettings`	An object of class `modelSettings` created using one of the function: • setLassoLogisticRegression() A lasso logistic regression model • setGradientBoostingMachine() A gradient boosting machine • setAdaBoost() An ada boost model • setRandomForest() A random forest model • setDecisionTree() A decision tree model
`logSettings`	An object of `logSettings` created using `createLogSettings` specifying how the logging is done
`preprocessSettings`	An object of `preprocessSettings`. This setting specifies the minimum fraction of target population who must have a covariate for it to be included in the model training and whether to normalise the covariates before training

Details

Users can define set of Ts, Os, databases and population settings. A list of data.frames containing details such as follow-up time distribution, time-to-event information, characteriszation details, time from last prior event, observation time distribution.

Value

An object containing the model or location where the model is saved, the data selection settings, the preprocessing and training settings as well as various performance measures obtained by the model.

- `distribution`: List for each O of a data.frame containing: i) Time to observation end distribution, ii) Time from observation start distribution, iii) Time to event distribution and iv) Time from last prior event to index distribution (only for patients in T who have O before index)
- `incident`: List for each O of incidence of O in T during TAR
- `characterization`: List for each O of Characterization of T, TnO, Tn~O

Examples

```
# load the data
plpData <- getEunomiaPlpData()
populationSettings <- createStudyPopulationSettings(minTimeAtRisk = 1)
saveDirectory <- file.path(tempdir(), "diagnosePlp")
diagnosis <- diagnosePlp(plpData = plpData, outcomeId = 3, analysisId = 1,
     populationSettings = populationSettings, saveDirectory = saveDirectory)
# clean up
unlink(saveDirectory, recursive = TRUE)
```

`evaluatePlp` *evaluatePlp*

Description

Evaluates the performance of the patient level prediction model

Usage

```
evaluatePlp(prediction, typeColumn = "evaluationType")
```

Arguments

`prediction`	The patient level prediction model's prediction
`typeColumn`	The column name in the prediction object that is used to stratify the evaluation

Details

The function calculates various metrics to measure the performance of the model

Value

An object of class plpEvaluation containing the following components

- evaluationStatistics: A data frame containing the evaluation statistics'
- thresholdSummary: A data frame containing the threshold summary'
- demographicSummary: A data frame containing the demographic summary'
- calibrationSummary: A data frame containing the calibration summary'
- predictionDistribution: A data frame containing the prediction distribution'

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n= 1500)
population <- createStudyPopulation(plpData, outcomeId = 3,
                                   populationSettings = createStudyPopulationSettings())
data <- splitData(plpData, population, splitSettings=createDefaultSplitSetting(splitSeed=42))
data$Train$covariateData <- preprocessData(data$Train$covariateData,
                                           createPreprocessSettings())
path <- file.path(tempdir(), "plp")
model <- fitPlp(data$Train, modelSettings=setLassoLogisticRegression(seed=42),
                analysisId=1, analysisPath = path)
evaluatePlp(model$prediction) # Train and CV metrics
```

`externalValidateDbPlp` *externalValidateDbPlp - Validate a model on new databases*

Description

This function extracts data using a user specified connection and cdm_schema, applied the model and then calcualtes the performance

Usage

```
externalValidateDbPlp(
  plpModel,
  validationDatabaseDetails = createDatabaseDetails(),
  validationRestrictPlpDataSettings = createRestrictPlpDataSettings(),
  settings = createValidationSettings(recalibrate = "weakRecalibration"),
  logSettings = createLogSettings(verbosity = "INFO", logName = "validatePLP"),
  outputFolder = NULL
)
```

Arguments

`plpModel`	The model object returned by runPlp() containing the trained model
`validationDatabaseDetails`	A list of objects of class `databaseDetails` created using `createDatabaseDetails`
`validationRestrictPlpDataSettings`	A list of population restriction settings created by `createRestrictPlpDataSettings()`
`settings`	A settings object of class `validationSettings` created using `createValidationSettings`
`logSettings`	An object of `logSettings` created using `createLogSettings` specifying how the logging is done
`outputFolder`	The directory to save the validation results to (subfolders are created per database in validationDatabaseDetails)

Details

Users need to input a trained model (the output of runPlp()) and new database connections. The function will return a list of length equal to the number of cdm_schemas input with the performance on the new data

Value

An externalValidatePlp object containing the following components

- model: The model object
- executionSummary: A list of execution details
- prediction: A dataframe containing the predictions
- performanceEvaluation: A dataframe containing the performance metrics
- covariateSummary: A dataframe containing the covariate summary

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
# first fit a model on some data, default is a L1 logistic regression
saveLoc <- file.path(tempdir(), "development")
results <- runPlp(plpData,
                  outcomeId = 3,
                  saveDirectory = saveLoc,
                  populationSettings =
                   createStudyPopulationSettings(requireTimeAtRisk=FALSE)
                  )
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
Eunomia::createCohorts(connectionDetails)
# now validate the model on Eunomia
validationDatabaseDetails <- createDatabaseDetails(
  connectionDetails = connectionDetails,
  cdmDatabaseSchema = "main",
  cdmDatabaseName = "main",
  cohortDatabaseSchema = "main",
  cohortTable = "cohort",
  outcomeDatabaseSchema = "main",
  outcomeTable = "cohort",
  targetId = 1, # users of celecoxib
  outcomeIds = 3, # GIbleed
  cdmVersion = 5)
path <- file.path(tempdir(), "validation")
externalValidateDbPlp(results$model, validationDatabaseDetails, outputFolder = path)
# clean up
unlink(saveLoc, recursive = TRUE)
unlink(path, recursive = TRUE)
```

`extractDatabaseToCsv` *Exports all the results from a database into csv files*

Description

Exports all the results from a database into csv files

Usage

```
extractDatabaseToCsv(
  conn = NULL,
  connectionDetails,
  databaseSchemaSettings = createDatabaseSchemaSettings(resultSchema = "main"),
  csvFolder,
  minCellCount = 5,
  sensitiveColumns = getPlpSensitiveColumns(),
  fileAppend = NULL
)
```

Arguments

`conn`	The connection to the database with the results
`connectionDetails`	The connectionDetails for the result database
`databaseSchemaSettings`	The result database schema settings
`csvFolder`	Location to save the csv files
`minCellCount`	The min value to show in cells that are sensitive (values less than this value will be replaced with -1)
`sensitiveColumns`	A named list (name of table columns belong to) with a list of columns to apply the minCellCount to.
`fileAppend`	If set to a string this will be appended to the start of the csv file names

Details

Extracts the results from a database into a set of csv files

Value

The directory path where the results were saved

Examples

```
# develop a simple model on simulated data
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 500)
saveLoc <- file.path(tempdir(), "extractDatabaseToCsv")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
# now upload the results to a sqlite database
databasePath <- insertResultsToSqlite(saveLoc)
# now extract the results to csv
connectionDetails <-
  DatabaseConnector::createConnectionDetails(dbms = "sqlite",
                                             server = databasePath)
extractDatabaseToCsv(
  connectionDetails = connectionDetails,
  csvFolder = file.path(saveLoc, "csv")
)
# show csv file
list.files(file.path(saveLoc, "csv"))
# clean up
unlink(saveLoc, recursive = TRUE)
```

fitPlp *fitPlp*

Description

Train various models using a default parameter grid search or user specified parameters

Usage

```
fitPlp(trainData, modelSettings, search = "grid", analysisId, analysisPath)
```

Arguments

`trainData`	An object of type `trainData` created using `splitData` data extracted from the CDM.
`modelSettings`	An object of class `modelSettings` created using one of the `createModelSettings` functions
`search`	The search strategy for the hyper-parameter selection (currently not used)

`analysisId`	The id of the analysis
`analysisPath`	The path of the analysis

Details

The user can define the machine learning model to train

Value

An object of class `plpModel` containing:

`model`	The trained prediction model
`preprocessing`	The preprocessing required when applying the model
`prediction`	The cohort data.frame with the predicted risk column added
`modelDesign`	A list specifiying the modelDesign settings used to fit the model
`trainDetails`	The model meta data
`covariateImportance`	The covariate importance for the model

Examples

```
# simulate data
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
# create study population, split into train/test and preprocess with default settings
population <- createStudyPopulation(plpData, outcomeId = 3)
data <- splitData(plpData, population, createDefaultSplitSetting())
data$Train$covariateData <- preprocessData(data$Train$covariateData)
saveLoc <- file.path(tempdir(), "fitPlp")
# fit a lasso logistic regression model using the training data
plpModel <- fitPlp(data$Train, modelSettings=setLassoLogisticRegression(seed=42),
                   analysisId=1, analysisPath=saveLoc)
# show evaluationSummary for model
evaluatePlp(plpModel$prediction)$evaluationSummary
# clean up
unlink(saveLoc, recursive = TRUE)
```

`getCalibrationSummary` *Get a sparse summary of the calibration*

Description

Get a sparse summary of the calibration

Usage

```
getCalibrationSummary(
  prediction,
  predictionType,
  typeColumn = "evaluation",
  numberOfStrata = 10,
  truncateFraction = 0.05
)
```

Arguments

`prediction`	A prediction object as generated using the `predict` functions.
`predictionType`	The type of prediction (binary or survival)
`typeColumn`	A column that is used to stratify the results
`numberOfStrata`	The number of strata in the plot.
`truncateFraction`	This fraction of probability values will be ignored when plotting, to avoid the x-axis scale being dominated by a few outliers.

Details

Generates a sparse summary showing the predicted probabilities and the observed fractions. Predictions are stratified into equally sized bins of predicted probabilities.

Value

A dataframe with the calibration summary

Examples

```
# simulate data
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=500)
# create study population, split into train/test and preprocess with default settings
population <- createStudyPopulation(plpData, outcomeId = 3)
data <- splitData(plpData, population, createDefaultSplitSetting())
data$Train$covariateData <- preprocessData(data$Train$covariateData)
saveLoc <- file.path(tempdir(), "calibrationSummary")
# fit a lasso logistic regression model using the training data
plpModel <- fitPlp(data$Train, modelSettings=setLassoLogisticRegression(seed=42),
                   analysisId=1, analysisPath=saveLoc)
calibrationSummary <- getCalibrationSummary(plpModel$prediction,
                                            "binary",
                                            numberOfStrata = 10,
                                            typeColumn = "evaluationType")
calibrationSummary
# clean up
unlink(saveLoc, recursive = TRUE)
```

getCohortCovariateData

Extracts covariates based on cohorts

Description

Extracts covariates based on cohorts

Usage

```
getCohortCovariateData(
  connection,
  tempEmulationSchema = NULL,
  oracleTempSchema = NULL,
  cdmDatabaseSchema,
  cdmVersion = "5",
  cohortTable = "#cohort_person",
  rowIdField = "row_id",
  aggregated,
  cohortIds,
  covariateSettings,
  ...
)
```

Arguments

`connection`	The database connection
`tempEmulationSchema`	The schema to use for temp tables
`oracleTempSchema`	DEPRECATED The temp schema if using oracle
`cdmDatabaseSchema`	The schema of the OMOP CDM data
`cdmVersion`	version of the OMOP CDM data
`cohortTable`	the table name that contains the target population cohort
`rowIdField`	string representing the unique identifier in the target population cohort
`aggregated`	whether the covariate should be aggregated
`cohortIds`	cohort id for the target cohort
`covariateSettings`	settings for the covariate cohorts and time periods
`...`	additional arguments from FeatureExtraction

Details

The user specifies a cohort and time period and then a covariate is constructed whether they are in the cohort during the time periods relative to target population cohort index

Value

CovariateData object with covariates, covariateRef, and analysisRef tables

Examples

```
library(DatabaseConnector)
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
# create some cohort of people born in 1969, index date is their date of birth
con <- connect(connectionDetails)
executeSql(con, "INSERT INTO main.cohort
                 SELECT 1969 as COHORT_DEFINITION_ID, PERSON_ID as SUBJECT_ID,
                 BIRTH_DATETIME as COHORT_START_DATE, BIRTH_DATETIME as COHORT_END_DATE
                 FROM main.person WHERE YEAR_OF_BIRTH = 1969")
covariateData <- getCohortCovariateData(connection = con,
                                       cdmDatabaseSchema = "main",
                                       aggregated = FALSE,
                                       rowIdField = "SUBJECT_ID",
                                      cohortTable = "cohort",
                                     covariateSettings = createCohortCovariateSettings(
                                                         cohortName="summerOfLove",
                                                          cohortId=1969,
                                                          settingId=1,
                                                       cohortDatabaseSchema="main",
                                                          cohortTable="cohort"))
covariateData$covariateRef
covariateData$covariates
```

getDemographicSummary *Get a demographic summary*

Description

Get a demographic summary

Usage

```
getDemographicSummary(prediction, predictionType, typeColumn = "evaluation")
```

Arguments

prediction	A prediction object
predictionType	The type of prediction (binary or survival)
typeColumn	A column that is used to stratify the results

Details

Generates a data.frame with a prediction summary per each 5 year age group and gender group

Value

A dataframe with the demographic summary

Examples

```
# simulate data
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=500)
# create study population, split into train/test and preprocess with default settings
population <- createStudyPopulation(plpData, outcomeId = 3)
data <- splitData(plpData, population, createDefaultSplitSetting())
data$Train$covariateData <- preprocessData(data$Train$covariateData)
saveLoc <- file.path(tempdir(), "demographicSummary")
# fit a lasso logistic regression model using the training data
plpModel <- fitPlp(data$Train, modelSettings=setLassoLogisticRegression(seed=42),
                   analysisId=1, analysisPath=saveLoc)
demographicSummary <- getDemographicSummary(plpModel$prediction,
                                            "binary",
                                            typeColumn = "evaluationType")
# show the demographic summary dataframe
str(demographicSummary)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`getEunomiaPlpData` *Create a plpData object from the Eunomia database'*

Description

This function creates a plpData object from the Eunomia database. It gets the connection details, creates the cohorts, and extracts the data. The cohort is predicting GIbleed in new users of celecoxib.

Usage

```
getEunomiaPlpData(covariateSettings = NULL)
```

Arguments

`covariateSettings`	A list of covariateSettings objects created using the `createCovariateSettings` function in the `FeatureExtraction` package. If nothing is specified covariates with age, gender, conditions and drug era are used.

Value

An object of type `plpData`, containing information on the cohorts, their outcomes, and baseline covariates. Information about multiple outcomes can be captured at once for efficiency reasons. This object is a list with the following components:

outcomes A data frame listing the outcomes per person, including the time to event, and the outcome id

cohorts A data frame listing the persons in each cohort, listing their exposure status as well as the time to the end of the observation period and time to the end of the cohort

covariateData An Andromeda object created with the `FeatureExtraction` package. This object contains the following items:

- **covariates** An Andromeda table listing the covariates per person in the two cohorts. This is done using a sparse representation: covariates with a value of 0 are omitted to save space. Usually has three columns, rowId, covariateId and covariateValue'.
- **covariateRef** An Andromeda table describing the covariates that have been extracted.
- **AnalysisRef** An Andromeda table with information about which analysisIds from 'FeatureExtraction' were used.

Examples

```
covariateSettings <- FeatureExtraction::createCovariateSettings(
  useDemographicsAge = TRUE,
  useDemographicsGender = TRUE,
  useConditionOccurrenceAnyTimePrior = TRUE
)
plpData <- getEunomiaPlpData(covariateSettings = covariateSettings)
```

`getPlpData` *Extract the patient level prediction data from the server*

Description

This function executes a large set of SQL statements against the database in OMOP CDM format to extract the data needed to perform the analysis.

Usage

```
getPlpData(databaseDetails, covariateSettings, restrictPlpDataSettings = NULL)
```

Arguments

`databaseDetails`	The cdm database details created using `createDatabaseDetails()`
`covariateSettings`	An object of type `covariateSettings` or a list of such objects as created using the `createCovariateSettings` function in the `FeatureExtraction` package.
`restrictPlpDataSettings`	Extra settings to apply to the target population while extracting data. Created using `createRestrictPlpDataSettings()`. This is optional.

Details

Based on the arguments, the at risk cohort data is retrieved, as well as outcomes occurring in these subjects. The at risk cohort is identified through user-defined cohorts in a cohort table either inside the CDM instance or in a separate schema. Similarly, outcomes are identified through user-defined cohorts in a cohort table either inside the CDM instance or in a separate schema. Covariates are automatically extracted from the appropriate tables within the CDM. If you wish to exclude concepts from covariates you will need to manually add the concept_ids and descendants to the `excludedCovariateConceptIds` of the `covariateSettings` argument.

Value

'r plpDataObjectDoc()'

Examples

```
# use Eunomia database
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
Eunomia::createCohorts(connectionDetails)
outcomeId <- 3 # GIbleed
databaseDetails <- createDatabaseDetails(
  connectionDetails = connectionDetails,
  cdmDatabaseSchema = "main",
  cdmDatabaseName = "main",
  cohortDatabaseSchema = "main",
  cohortTable = "cohort",
  outcomeDatabaseSchema = "main",
  outcomeTable = "cohort",
  targetId = 1,
  outcomeIds = outcomeId,
  cdmVersion = 5
)

covariateSettings <- FeatureExtraction::createCovariateSettings(
  useDemographicsAge = TRUE,
  useDemographicsGender = TRUE,
  useConditionOccurrenceAnyTimePrior = TRUE
)
```

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

getPredictionDistribution

Calculates the prediction distribution

Description

Calculates the prediction distribution

Usage

```
getPredictionDistribution(
  prediction,
  predictionType = "binary",
  typeColumn = "evaluation"
)
```

Arguments

`prediction`	A prediction object
`predictionType`	The type of prediction (binary or survival)
`typeColumn`	A column that is used to stratify the results

Details

Calculates the quantiles from a predition object

Value

The 0.00, 0.1, 0.25, 0.5, 0.75, 0.9, 1.00 quantile pf the prediction, the mean and standard deviation per class

Examples

```
prediction <- data.frame(rowId = 1:100,
                         outcomeCount = stats::rbinom(1:100, 1, prob=0.5),
                         value = runif(100),
                         evaluation = rep("Train", 100))
getPredictionDistribution(prediction)
```

`getThresholdSummary` *Calculate all measures for sparse ROC*

Description

Calculate all measures for sparse ROC

Usage

```
getThresholdSummary(
  prediction,
  predictionType = "binary",
  typeColumn = "evaluation"
)
```

Arguments

`prediction`	A prediction object
`predictionType`	The type of prediction (binary or survival)
`typeColumn`	A column that is used to stratify the results

Details

Calculates the TP, FP, TN, FN, TPR, FPR, accuracy, PPF, FOR and Fmeasure from a prediction object

Value

A data.frame with TP, FP, TN, FN, TPR, FPR, accuracy, PPF, FOR and Fmeasure

Examples

```
prediction <- data.frame(rowId = 1:100,
                         outcomeCount = stats::rbinom(1:100, 1, prob=0.5),
                         value = runif(100),
                         evaluation = rep("Train", 100))
summary <- getThresholdSummary(prediction)
str(summary)
```

ici — *Calculate the Integrated Calibration Index from Austin and Steyerberg https://onlinelibrary.wiley.com/doi/full/10.1002/sim.8281*

Description

Calculate the Integrated Calibration Index from Austin and Steyerberg https://onlinelibrary.wiley.com/doi/full/10.1002/sim.82

Usage

```
ici(prediction)
```

Arguments

`prediction`	the prediction object found in the plpResult object

Details

Calculate the Integrated Calibration Index

Value

Integrated Calibration Index value or NULL if the calculation fails

Examples

```
prediction <- data.frame(rowId = 1:100,
                         outcomeCount = stats::rbinom(1:100, 1, prob=0.5),
                         value = runif(100),
                         evaluation = rep("Train", 100))
ici(prediction)
```

insertCsvToDatabase — *Function to insert results into a database from csvs*

Description

This function converts a folder with csv results into plp objects and loads them into a plp result database

Usage

```
insertCsvToDatabase(
  csvFolder,
  connectionDetails,
  databaseSchemaSettings,
  modelSaveLocation,
  csvTableAppend = ""
)
```

Arguments

`csvFolder`	The location to the csv folder with the plp results
`connectionDetails`	A connection details for the plp results database that the csv results will be inserted into
`databaseSchemaSettings`	A object created by `createDatabaseSchemaSettings` with all the settings specifying the result tables to insert the csv results into
`modelSaveLocation`	The location to save any models from the csv folder - this should be the same location you picked when inserting other models into the database
`csvTableAppend`	A string that appends the csv file names

Details

The user needs to have plp csv results in a single folder and an existing plp result database

Value

Returns a data.frame indicating whether the results were inported into the database

Examples

```
# develop a simple model on simulated data
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "extractDatabaseToCsv")
results <- runPlp(plpData, outcomeId=3, saveDirectory=saveLoc)
# now upload the results to a sqlite database
databasePath <- insertResultsToSqlite(saveLoc)
# now extract the results to csv
connectionDetails <-
 DatabaseConnector::createConnectionDetails(dbms = "sqlite",
                                            server = databasePath)
extractDatabaseToCsv(connectionDetails = connectionDetails,
                     csvFolder = file.path(saveLoc, "csv"))
# show csv file
list.files(file.path(saveLoc, "csv"))
# now insert the csv results into a database
newDatabasePath <- file.path(tempdir(), "newDatabase.sqlite")
connectionDetails <-
 DatabaseConnector::createConnectionDetails(dbms = "sqlite",
                                            server = newDatabasePath)
insertCsvToDatabase(csvFolder = file.path(saveLoc, "csv"),
                     connectionDetails = connectionDetails,
                     databaseSchemaSettings = createDatabaseSchemaSettings(),
                     modelSaveLocation = file.path(saveLoc, "models"))
# clean up
unlink(saveLoc, recursive = TRUE)
```

insertResultsToSqlite *Create sqlite database with the results*

Description

This function create an sqlite database with the PLP result schema and inserts all results

Usage

```
insertResultsToSqlite(
  resultLocation,
  cohortDefinitions = NULL,
  databaseList = NULL,
  sqliteLocation = file.path(resultLocation, "sqlite")
)
```

Arguments

`resultLocation`	(string) location of directory where the main package results were saved
`cohortDefinitions`	A set of one or more cohorts extracted using ROhdsiWebApi::exportCohortDefinitionSet()
`databaseList`	A list created by `createDatabaseList` to specify the databases
`sqliteLocation`	(string) location of directory where the sqlite database will be saved

Details

This function can be used upload PatientLevelPrediction results into an sqlite database

Value

Returns the location of the sqlite database file

Examples

```
plpData <- getEunomiaPlpData()
saveLoc <- file.path(tempdir(), "insertResultsToSqlite")
results <- runPlp(plpData, outcomeId = 3, analysisId = 1, saveDirectory = saveLoc)
databaseFile <- insertResultsToSqlite(saveLoc, cohortDefinitions = NULL,
                                      sqliteLocation = file.path(saveLoc, "sqlite"))
# check there is some data in the database
library(DatabaseConnector)
connectionDetails <- createConnectionDetails(
  dbms = "sqlite",
  server = databaseFile)
conn <- connect(connectionDetails)
# All tables should be created
getTableNames(conn, databaseSchema = "main")
```

```
# There is data in the tables
querySql(conn, "SELECT * FROM main.model_designs limit 10")
# clean up
unlink(saveLoc, recursive = TRUE)
```

`listAppend` *join two lists*

Description

join two lists

Usage

```
listAppend(a, b)
```

Arguments

a	A list
b	Another list

Details

This function joins two lists

Value

the joined list

Examples

```
a <- list(a = 1, b = 2)
b <- list(c = 3, d = 4)
listAppend(a, b)
```

listCartesian *Cartesian product*

Description

Computes the Cartesian product of all the combinations of elements in a list

Usage

```
listCartesian(allList)
```

Arguments

allList	a list of lists

Value

A list with all possible combinations from the input list of lists

Examples

```
listCartesian(list(list(1, 2), list(3, 4)))
```

loadPlpAnalysesJson *Load the multiple prediction json settings from a file*

Description

Load the multiple prediction json settings from a file

Usage

```
loadPlpAnalysesJson(jsonFileLocation)
```

Arguments

jsonFileLocation	The location of the file 'predictionAnalysisList.json' with the modelDesignList

Details

This function interprets a json with the multiple prediction settings and creates a list that can be combined with connection settings to run a multiple prediction study

Value

A list with the modelDesignList and cohortDefinitions

Examples

```
modelDesign <- createModelDesign(targetId = 1, outcomeId = 2,
                                 modelSettings = setLassoLogisticRegression())
saveLoc <- file.path(tempdir(), "loadPlpAnalysesJson")
savePlpAnalysesJson(modelDesignList = modelDesign, saveDirectory = saveLoc)
loadPlpAnalysesJson(file.path(saveLoc, "predictionAnalysisList.json"))
# clean use
unlink(saveLoc, recursive = TRUE)
```

`loadPlpData` *Load the plpData from a folder*

Description

`loadPlpData` loads an object of type plpData from a folder in the file system.

Usage

```
loadPlpData(file, readOnly = TRUE)
```

Arguments

`file`	The name of the folder containing the data.
`readOnly`	If true, the data is opened read only.

Details

The data will be written to a set of files in the folder specified by the user.

Value

An object of class plpData.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 500)
saveLoc <- file.path(tempdir(), "loadPlpData")
savePlpData(plpData, saveLoc)
dir(saveLoc)
# clean up
unlink(saveLoc, recursive = TRUE)
```

loadPlpModel *loads the plp model*

Description

loads the plp model

Usage

```
loadPlpModel(dirPath)
```

Arguments

dirPath	The location of the model

Details

Loads a plp model that was saved using `savePlpModel()`

Value

`The plpModel object`

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "loadPlpModel")
plpResult <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
savePlpModel(plpResult$model, file.path(saveLoc, "savedModel"))
loadedModel <- loadPlpModel(file.path(saveLoc, "savedModel"))
# show design of loaded model
str(loadedModel$modelDesign)

# clean up
unlink(saveLoc, recursive = TRUE)
```

loadPlpResult *Loads the evalaution dataframe*

Description

Loads the evalaution dataframe

Usage

```
loadPlpResult(dirPath)
```

Arguments

`dirPath`	The directory where the evaluation was saved

Details

Loads the evaluation

Value

```
The runPlp object
```

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "loadPlpResult")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
savePlpResult(results, saveLoc)
loadedResults <- loadPlpResult(saveLoc)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`loadPlpShareable` *Loads the plp result saved as json/csv files for transparent sharing*

Description

Loads the plp result saved as json/csv files for transparent sharing

Usage

```
loadPlpShareable(loadDirectory)
```

Arguments

`loadDirectory`	The directory with the results as json/csv files

Details

Load the main results from json/csv files into a runPlp object

Value

```
The runPlp object
```

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "loadPlpShareable")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
savePlpShareable(results, saveLoc)
dir(saveLoc)
loadedResults <- loadPlpShareable(saveLoc)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`loadPrediction` *Loads the prediction dataframe to json*

Description

Loads the prediction dataframe to json

Usage

```
loadPrediction(fileLocation)
```

Arguments

`fileLocation`	The location with the saved prediction

Details

Loads the prediciton json file

Value

`The prediction data.frame`

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "loadPrediction")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
savePrediction(results$prediction, saveLoc)
dir(saveLoc)
loadedPrediction <- loadPrediction(file.path(saveLoc, "prediction.json"))
```

MapIds *Map covariate and row Ids so they start from 1*

Description

this functions takes covariate data and a cohort/population and remaps the covariate and row ids, restricts to pop and saves/creates mapping

Usage

```
MapIds(covariateData, cohort = NULL, mapping = NULL)
```

Arguments

covariateData	a covariateData object
cohort	if specified rowIds restricted to the ones in cohort
mapping	A pre defined mapping to use

Value

a new `covariateData` object with remapped covariate and row ids

Examples

```
covariateData <- Andromeda::andromeda(
  covariates = data.frame(rowId = c(1, 3, 5, 7, 9),
                          covariateId = c(10, 20, 10, 10, 20),
                          covariateValue = c(1, 1, 1, 1, 1)),
  covariateRef = data.frame(covariateId = c(10, 20),
                            covariateNames = c("covariateA",
                                               "covariateB"),
                            analysisId = c(1, 1)))
mappedData <- MapIds(covariateData)
# columnId and rowId are now starting from 1 and are consecutive
mappedData$covariates
```

migrateDataModel *Migrate Data model*

Description

Migrate data from current state to next state

It is strongly advised that you have a backup of all data (either sqlite files, a backup database (in the case you are using a postgres backend) or have kept the csv/zip files from your data generation.

Usage

```
migrateDataModel(connectionDetails, databaseSchema, tablePrefix = "")
```

Arguments

`connectionDetails`	DatabaseConnector connection details object
`databaseSchema`	String schema where database schema lives
`tablePrefix`	(Optional) Use if a table prefix is used before table names (e.g. "cd_")

Value

Nothing. Is called for side effects of migrating data model in the database

---

`modelBasedConcordance` *Calculate the model-based concordance, which is a calculation of the expected discrimination performance of a model under the assumption the model predicts the "TRUE" outcome as detailed in van Klaveren et al. https://pubmed.ncbi.nlm.nih.gov/27251001/*

Description

Calculate the model-based concordance, which is a calculation of the expected discrimination performance of a model under the assumption the model predicts the "TRUE" outcome as detailed in van Klaveren et al. https://pubmed.ncbi.nlm.nih.gov/27251001/

Usage

```
modelBasedConcordance(prediction)
```

Arguments

`prediction`	the prediction object found in the plpResult object

Details

Calculate the model-based concordance

Value

The model-based concordance value

Examples

```
prediction <- data.frame(value = runif(100))
modelBasedConcordance(prediction)
```

`outcomeSurvivalPlot` *Plot the outcome incidence over time*

Description

Plot the outcome incidence over time

Usage

```
outcomeSurvivalPlot(
  plpData,
  outcomeId,
 populationSettings = createStudyPopulationSettings(binary = TRUE, includeAllOutcomes =
   TRUE, firstExposureOnly = FALSE, washoutPeriod = 0, removeSubjectsWithPriorOutcome =
   TRUE, priorOutcomeLookback = 99999, requireTimeAtRisk = FALSE, riskWindowStart = 1,
   startAnchor = "cohort start", riskWindowEnd = 3650, endAnchor = "cohort start"),
  riskTable = TRUE,
  confInt = TRUE,
  yLabel = "Fraction of those who are outcome free in target population"
)
```

Arguments

`plpData`	The plpData object returned by running getPlpData()
`outcomeId`	The cohort id corresponding to the outcome
`populationSettings`	The population settings created using `createStudyPopulationSettings`
`riskTable`	(binary) Whether to include a table at the bottom of the plot showing the number of people at risk over time
`confInt`	(binary) Whether to include a confidence interval
`yLabel`	(string) The label for the y-axis

Details

This creates a survival plot that can be used to pick a suitable time-at-risk period

Value

A `ggsurvplot` object

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
plotObject <- outcomeSurvivalPlot(plpData, outcomeId = 3)
print(plotObject)
```

pfi *Permutation Feature Importance*

Description

Calculate the permutation feature importance (pfi) for a PLP model.

Usage

```
pfi(
  plpResult,
  population,
  plpData,
  repeats = 1,
  covariates = NULL,
  cores = NULL,
  log = NULL,
  logthreshold = "INFO"
)
```

Arguments

`plpResult`	An object of type `runPlp`
`population`	The population created using createStudyPopulation() who will have their risks predicted
`plpData`	An object of type `plpData` - the patient level prediction data extracted from the CDM.
`repeats`	The number of times to permute each covariate
`covariates`	A vector of covariates to calculate the pfi for. If NULL it uses all covariates included in the model.
`cores`	Number of cores to use when running this (it runs in parallel)
`log`	A location to save the log for running pfi
`logthreshold`	The log threshold (e.g., INFO, TRACE, ...)

Details

The function permutes the each covariate/features `repeats` times and calculates the mean AUC change caused by the permutation.

Value

A dataframe with the covariateIds and the pfi (change in AUC caused by permuting the covariate) value

Examples

```
library(dplyr)
# simulate some data
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
# now fit a model
saveLoc <- file.path(tempdir(), "pfi")
plpResult <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
population <- createStudyPopulation(plpData, outcomeId = 3)
pfi(plpResult, population, plpData, repeats = 1, cores = 1)
# compare to model coefficients
plpResult$model$covariateImportance %>% dplyr::filter(.data$covariateValue != 0)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`plotDemographicSummary` *Plot the Observed vs. expected incidence, by age and gender*

Description

Plot the Observed vs. expected incidence, by age and gender

Usage

```
plotDemographicSummary(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "roc.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the Observed vs. expected incidence, by age and gender #'

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotDemographicSummary")
plpResult <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotDemographicSummary(plpResult)
# clean up
unlink(saveLoc, recursive = TRUE)
```

---

plotF1Measure — *Plot the F1 measure efficiency frontier using the sparse thresholdSummary data frame*

Description

Plot the F1 measure efficiency frontier using the sparse thresholdSummary data frame

Usage

```
plotF1Measure(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "roc.png"
)
```

Arguments

plpResult	A plp result object as generated using the runPlp function.
typeColumn	The name of the column specifying the evaluation type
saveLocation	Directory to save plot (if NULL plot is not saved)
fileName	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the F1 measure efficiency frontier using the sparse thresholdSummary data frame

Value

A ggplot object. Use the ggsave function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotF1Measure")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotF1Measure(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`plotGeneralizability` *Plot the train/test generalizability diagnostic*

Description

Plot the train/test generalizability diagnostic

Usage

```
plotGeneralizability(
  covariateSummary,
  saveLocation = NULL,
  fileName = "Generalizability.png"
)
```

Arguments

`covariateSummary`	A prediction object as generated using the `runPlp` function.
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the train/test generalizability diagnostic #'

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
population <- createStudyPopulation(plpData, outcomeId = 3)
data <- splitData(plpData, population = population)
strata <- data.frame(
 rowId = c(data$Train$labels$rowId, data$Test$labels$rowId),
 strataName = c(rep("Train", nrow(data$Train$labels)),
                rep("Test", nrow(data$Test$labels))))
covariateSummary <- covariateSummary(plpData$covariateData,
                                     cohort = dplyr::select(population, "rowId"),
 strata = strata, labels = population)
plotGeneralizability(covariateSummary)
```

---

plotLearningCurve *plotLearningCurve*

Description

Create a plot of the learning curve using the object returned from `createLearningCurve`.

Usage

```
plotLearningCurve(
  learningCurve,
  metric = "AUROC",
  abscissa = "events",
  plotTitle = "Learning Curve",
  plotSubtitle = NULL,
  fileName = NULL
)
```

Arguments

`learningCurve`	An object returned by `createLearningCurve` function.
`metric`	Specifies the metric to be plotted: • `'AUROC'` - use the area under the Receiver Operating Characteristic curve • `'AUPRC'` - use the area under the Precision-Recall curve • `'sBrier'` - use the scaled Brier score
`abscissa`	Specify the abscissa metric to be plotted: • `'events'` - use number of events • `'observations'` - use number of observations
`plotTitle`	Title of the learning curve plot.

`plotSubtitle`	Subtitle of the learning curve plot.
`fileName`	Filename of plot to be saved, for example `'plot.png'`. See the function ggsave in the ggplot2 package for supported file formats.

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
outcomeId <- 3
modelSettings <- setLassoLogisticRegression(seed=42)
learningCurve <- createLearningCurve(plpData, outcomeId, modelSettings = modelSettings,
saveDirectory = file.path(tempdir(), "learningCurve"), cores = 2)
plotLearningCurve(learningCurve)
```

`plotNetBenefit` *Plot the net benefit*

Description

Plot the net benefit

Usage

```
plotNetBenefit(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "netBenefit.png",
  evalType = NULL,
  ylim = NULL,
  xlim = NULL
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

evalType	Which evaluation type to plot for. For example `Test`, `Train`. If NULL everything is plotted
ylim	The y limits for the plot, if NULL the limits are calculated from the data
xlim	The x limits for the plot, if NULL the limits are calculated from the data

Value

A list of ggplot objects or a single ggplot object if only one evaluation type is plotted

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotNetBenefit")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotNetBenefit(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

plotPlp *Plot all the PatientLevelPrediction plots*

Description

Plot all the PatientLevelPrediction plots

Usage

```
plotPlp(plpResult, saveLocation = NULL, typeColumn = "evaluation")
```

Arguments

plpResult	Object returned by the runPlp() function
saveLocation	Name of the directory where the plots should be saved (NULL means no saving)
typeColumn	The name of the column specifying the evaluation type (to stratify the plots)

Details

Create a directory with all the plots

Value

TRUE if it ran, plots are saved in the specified directory

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotPlp")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotPlp(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

---

plotPrecisionRecall — *Plot the precision-recall curve using the sparse thresholdSummary data frame*

Description

Plot the precision-recall curve using the sparse thresholdSummary data frame

Usage

```
plotPrecisionRecall(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "roc.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the precision-recall curve using the sparse thresholdSummary data frame

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotPrecisionRecall")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotPrecisionRecall(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`plotPredictedPDF` *Plot the Predicted probability density function, showing prediction overlap between true and false cases*

Description

Plot the Predicted probability density function, showing prediction overlap between true and false cases

Usage

```
plotPredictedPDF(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "PredictedPDF.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the predicted probability density function, showing prediction overlap between true and false cases

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotPredictedPDF")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotPredictedPDF(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`plotPredictionDistribution`
Plot the side-by-side boxplots of prediction distribution, by class

---

Description

Plot the side-by-side boxplots of prediction distribution, by class

Usage

```
plotPredictionDistribution(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "PredictionDistribution.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the side-by-side boxplots of prediction distribution, by class #'

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotPredictionDistribution")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotPredictionDistribution(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`plotPreferencePDF` *Plot the preference score probability density function, showing prediction overlap between true and false cases #'*

Description

Plot the preference score probability density function, showing prediction overlap between true and false cases #'

Usage

```
plotPreferencePDF(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "plotPreferencePDF.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function `ggsave` in the ggplot2 package for supported file formats.

Details

Create a plot showing the preference score probability density function, showing prediction overlap between true and false cases #'

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotPreferencePDF")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotPreferencePDF(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`plotSmoothCalibration` *Plot the smooth calibration as detailed in Calster et al. "A calibration heirarchy for risk models was defined: from utopia to empirical data" (2016)*

Description

Plot the smooth calibration as detailed in Calster et al. "A calibration heirarchy for risk models was defined: from utopia to empirical data" (2016)

Usage

```
plotSmoothCalibration(
  plpResult,
  smooth = "loess",
  span = 0.75,
  nKnots = 5,
  scatter = FALSE,
  bins = 20,
  sample = TRUE,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "smoothCalibration.pdf"
)
```

Arguments

`plpResult`	The result of running `runPlp` function. An object containing the model or location where the model is save, the data selection settings, the preprocessing and training settings as well as various performance measures obtained by the model.
`smooth`	options: 'loess' or 'rcs'
`span`	This specifies the width of span used for loess. This will allow for faster computing and lower memory usage.
`nKnots`	The number of knots to be used by the rcs evaluation. Default is 5

`scatter`	plot the decile calibrations as points on the graph. Default is False
`bins`	The number of bins for the histogram. Default is 20.
`sample`	If using loess then by default 20,000 patients will be sampled to save time
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function `ggsave` in the ggplot2 package for supported file formats.

Details

Create a plot showing the smoothed calibration

Value

A ggplot object.

Examples

```
# generate prediction dataaframe with 1000 patients
predictedRisk <- stats::runif(1000)
# overconfident for high risk patients
actualRisk <- ifelse(predictedRisk < 0.5, predictedRisk, 0.5 + 0.5 * (predictedRisk - 0.5))
outcomeCount <- stats::rbinom(1000, 1, actualRisk)
# mock data frame
prediction <- data.frame(rowId = 1:1000,
                         value = predictedRisk,
                         outcomeCount = outcomeCount,
                         evaluationType = "Test")
attr(prediction, "modelType") <- "binary"
calibrationSummary <- getCalibrationSummary(prediction, "binary",
                                            numberOfStrata = 10,
                                            typeColumn = "evaluationType")
plpResults <- list()
plpResults$performanceEvaluation$calibrationSummary <- calibrationSummary
plpResults$prediction <- prediction
plotSmoothCalibration(plpResults)
```

`plotSparseCalibration` *Plot the calibration*

Description

Plot the calibration

Usage

```
plotSparseCalibration(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "roc.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the calibration #'

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotSparseCalibration")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotSparseCalibration(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

---

plotSparseCalibration2 *Plot the conventional calibration*

Description

Plot the conventional calibration

Usage

```
plotSparseCalibration2(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "roc.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the calibration #'

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotSparseCalibration2")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotSparseCalibration2(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`plotSparseRoc` *Plot the ROC curve using the sparse thresholdSummary data frame*

Description

Plot the ROC curve using the sparse thresholdSummary data frame

Usage

```
plotSparseRoc(
  plpResult,
  typeColumn = "evaluation",
  saveLocation = NULL,
  fileName = "roc.png"
)
```

Arguments

`plpResult`	A plp result object as generated using the `runPlp` function.
`typeColumn`	The name of the column specifying the evaluation type
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the Receiver Operator Characteristics (ROC) curve.

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotSparseRoc")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotSparseRoc(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

---

plotVariableScatterplot

Plot the variable importance scatterplot

Description

Plot the variable importance scatterplot

Usage

```
plotVariableScatterplot(
  covariateSummary,
  saveLocation = NULL,
  fileName = "VariableScatterplot.png"
)
```

Arguments

`covariateSummary`	A prediction object as generated using the `runPlp` function.
`saveLocation`	Directory to save plot (if NULL plot is not saved)
`fileName`	Name of the file to save to plot, for example 'plot.png'. See the function ggsave in the ggplot2 package for supported file formats.

Details

Create a plot showing the variable importance scatterplot #'

Value

A ggplot object. Use the `ggsave` function to save to file in a different format.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
saveLoc <- file.path(tempdir(), "plotVariableScatterplot")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
plotVariableScatterplot(results$covariateSummary)
# clean up
```

`predictCyclops` *Create predictive probabilities*

Description

Create predictive probabilities

Usage

```
predictCyclops(plpModel, data, cohort)
```

Arguments

`plpModel`	An object of type `predictiveModel` as generated using `fitPlp`.
`data`	The new plpData containing the covariateData for the new population
`cohort`	The cohort to calculate the prediction for

Details

Generates predictions for the population specified in plpData given the model.

Value

The value column in the result data.frame is: logistic: probabilities of the outcome, poisson: Poisson rate (per day) of the outome, survival: hazard rate (per day) of the outcome.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
population <- createStudyPopulation(plpData, outcomeId = 3)
data <- splitData(plpData, population)
plpModel <- fitPlp(data$Train, modelSettings = setLassoLogisticRegression(),
                   analysisId = "test", analysisPath = NULL)
prediction <- predictCyclops(plpModel, data$Test, data$Test$labels)
# view prediction dataframe
head(prediction)
```

`predictGlm` *predict using a logistic regression model*

Description

Predict risk with a given plpModel containing a generalized linear model.

Usage

```
predictGlm(plpModel, data, cohort)
```

Arguments

`plpModel`	An object of type `plpModel` - a patient level prediction model
`data`	An object of type `plpData` - the patient level prediction data extracted from the CDM.
`cohort`	The population dataframe created using `createStudyPopulation` who will have their risks predicted or a cohort without the outcome known

Value

A dataframe containing the prediction for each person in the population

Examples

```
coefficients <- data.frame(
  covariateId = c(1002),
  coefficient = c(0.05))
model <- createGlmModel(coefficients, intercept = -2.5)
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=50)
prediction <- predictGlm(model, plpData, plpData$cohorts)
# see the predicted risk values
head(prediction)
```

`predictPlp` *predictPlp*

Description

Predict the risk of the outcome using the input plpModel for the input plpData

Usage

```
predictPlp(plpModel, plpData, population, timepoint)
```

Arguments

`plpModel`	An object of type `plpModel` - a patient level prediction model
`plpData`	An object of type `plpData` - the patient level prediction data extracted from the CDM.
`population`	The population created using createStudyPopulation() who will have their risks predicted or a cohort without the outcome known
`timepoint`	The timepoint to predict risk (survival models only)

Details

The function applied the trained model on the plpData to make predictions

Value

A data frame containing the predicted risk values

Examples

```
coefficients <- data.frame(
  covariateId = c(1002),
  coefficient = c(0.05)
)
model <- createGlmModel(coefficients, intercept = -2.5)
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 50)
prediction <- predictPlp(model, plpData, plpData$cohorts)
# see the predicted risk values
head(prediction)
```

`preprocessData` *A function that wraps around FeatureExtraction::tidyCovariateData to normalise the data and remove rare or redundant features*

Description

A function that wraps around FeatureExtraction::tidyCovariateData to normalise the data and remove rare or redundant features

Usage

```
preprocessData(covariateData, preprocessSettings = createPreprocessSettings())
```

Arguments

`covariateData`	The covariate part of the training data created by `splitData` after being sampled and having any required feature engineering
`preprocessSettings`	The settings for the preprocessing created by `createPreprocessSettings` The data processed

Details

Returns an object of class `covariateData` that has been processed. This includes normalising the data and removing rare or redundant features. Redundant features are features that within an analysisId together cover all obervations.

Value

The covariateData object with the processed covariates

Examples

```
library(dplyr)
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
preProcessedData <- preprocessData(plpData$covariateData, createPreprocessSettings())
# check age is normalized by max value
preProcessedData$covariates %>% dplyr::filter(.data$covariateId == 1002)
```

`print.plpData` *Print a plpData object*

Description

Print a plpData object

Usage

```
## S3 method for class 'plpData'
print(x, ...)
```

Arguments

`x`	The plpData object to print
`...`	Additional arguments

Value

A message describing the object

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=10)
print(plpData)
```

`print.summary.plpData` *Print a summary.plpData object*

Description

Print a summary.plpData object

Usage

```
## S3 method for class 'summary.plpData'
print(x, ...)
```

Arguments

`x`	The summary.plpData object to print
`...`	Additional arguments

Value

A message describing the object

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=10)
summary <- summary(plpData)
print(summary)
```

`recalibratePlp` *recalibratePlp*

Description

Recalibrating a model using the recalibrationInTheLarge or weakRecalibration methods

Usage

```
recalibratePlp(
  prediction,
  analysisId,
  typeColumn = "evaluationType",
  method = c("recalibrationInTheLarge", "weakRecalibration")
)
```

Arguments

`prediction`	A prediction dataframe
`analysisId`	The model analysisId
`typeColumn`	The column name where the strata types are specified
`method`	Method used to recalibrate ('recalibrationInTheLarge' or 'weakRecalibration')

Details

'recalibrationInTheLarge' calculates a single correction factor for the average predicted risks to match the average observed risks. 'weakRecalibration' fits a glm model to the logit of the predicted risks, also known as Platt scaling/logistic recalibration.

Value

A prediction dataframe with the recalibrated predictions added

Examples

```
prediction <- data.frame(rowId = 1:100,
                         value = runif(100),
                         outcomeCount = stats::rbinom(100, 1, 0.1),
                         evaluationType = rep("validation", 100))
attr(prediction, "metaData") <- list(modelType = "binary")
# since value is unformally distributed but outcomeCount is not (prob <- 0.1)
# the predictions are mis-calibrated
outcomeRate <- mean(prediction$outcomeCount)
observedRisk <- mean(prediction$value)
message("outcome rate is: ", outcomeRate)
message("observed risk is: ", observedRisk)
# lets recalibrate the predictions
prediction <- recalibratePlp(prediction,
                             analysisId = "recalibration",
                             method = "recalibrationInTheLarge")
recalibratedRisk <- mean(prediction$value)
message("recalibrated risk with recalibration in the large is: ", recalibratedRisk)
prediction <- recalibratePlp(prediction,
                             analysisId = "recalibration",
                             method = "weakRecalibration")
recalibratedRisk <- mean(prediction$value)
message("recalibrated risk with weak recalibration is: ", recalibratedRisk)
```

`recalibratePlpRefit` *recalibratePlpRefit*

Description

Recalibrating a model by refitting it

Usage

```
recalibratePlpRefit(plpModel, newPopulation, newData, returnModel = FALSE)
```

Arguments

`plpModel`	The trained plpModel (runPlp$model)
`newPopulation`	The population created using createStudyPopulation() who will have their risks predicted
`newData`	An object of type `plpData` - the patient level prediction data extracted from the CDM.
`returnModel`	Logical: return the refitted model

Value

An prediction dataframe with the predictions of the recalibrated model added

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "recalibratePlpRefit")
plpResults <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
newData <- simulatePlpData(simulationProfile, n = 1000)
newPopulation <- createStudyPopulation(newData, outcomeId = 3)
predictions <- recalibratePlpRefit(plpModel = plpResults$model,
                                   newPopulation = newPopulation,
                                   newData = newData)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`runMultiplePlp` *Run a list of predictions analyses*

Description

Run a list of predictions analyses

Usage

```
runMultiplePlp(
  databaseDetails = createDatabaseDetails(),
  modelDesignList = list(createModelDesign(targetId = 1, outcomeId = 2, modelSettings =
    setLassoLogisticRegression()), createModelDesign(targetId = 1, outcomeId = 3,
    modelSettings = setLassoLogisticRegression())),
  onlyFetchData = FALSE,
  cohortDefinitions = NULL,
  logSettings = createLogSettings(verbosity = "DEBUG", timeStamp = TRUE, logName =
    "runPlp Log"),
  saveDirectory = NULL,
  sqliteLocation = file.path(saveDirectory, "sqlite")
)
```

Arguments

`databaseDetails`	The database settings created using `createDatabaseDetails()`
`modelDesignList`	A list of model designs created using `createModelDesign()`
`onlyFetchData`	Only fetches and saves the data object to the output folder without running the analysis.
`cohortDefinitions`	A list of cohort definitions for the target and outcome cohorts
`logSettings`	The setting specifying the logging for the analyses created using `createLogSettings()`

`saveDirectory`	Name of the folder where all the outputs will written to.
`sqliteLocation`	(optional) The location of the sqlite database with the results

Details

This function will run all specified predictions as defined using .

Value

A data frame with the following columns:

`analysisId`	The unique identifier for a set of analysis choices.
`targetId`	The ID of the target cohort populations.
`outcomeId`	The ID of the outcomeId.
`dataLocation`	The location where the plpData was saved
`the settings ids`	The ids for all other settings used for model development.

Examples

```
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
databaseDetails <- createDatabaseDetails(connectionDetails = connectionDetails,
                                         cdmDatabaseSchema = "main",
                                         cohortDatabaseSchema = "main",
                                         cohortTable = "cohort",
                                         outcomeDatabaseSchema = "main",
                                         outcomeTable = "cohort",
                                         targetId = 1,
                                         outcomeIds = 2)
Eunomia::createCohorts(connectionDetails = connectionDetails)
covariateSettings <-
 FeatureExtraction::createCovariateSettings(useDemographicsGender = TRUE,
                                            useDemographicsAge = TRUE,
                                            useConditionOccurrenceLongTerm = TRUE)
# GI Bleed in users of celecoxib
modelDesign <- createModelDesign(targetId = 1,
                                 outcomeId = 3,
                                 modelSettings = setLassoLogisticRegression(seed = 42),
                                 populationSettings = createStudyPopulationSettings(),
                              restrictPlpDataSettings = createRestrictPlpDataSettings(),
                                 covariateSettings = covariateSettings,
                              splitSettings = createDefaultSplitSetting(splitSeed = 42),
                                 preprocessSettings = createPreprocessSettings())
# GI Bleed in users of NSAIDs
modelDesign2 <- createModelDesign(targetId = 4,
                                  outcomeId = 3,
                                  modelSettings = setLassoLogisticRegression(seed = 42),
                                  populationSettings = createStudyPopulationSettings(),
                              restrictPlpDataSettings = createRestrictPlpDataSettings(),
                                  covariateSettings = covariateSettings,
                              splitSettings = createDefaultSplitSetting(splitSeed = 42),
                                  preprocessSettings = createPreprocessSettings())
saveLoc <- file.path(tempdir(), "runMultiplePlp")
multipleResults <- runMultiplePlp(databaseDetails = databaseDetails,
                                  modelDesignList = list(modelDesign, modelDesign2),
                                  saveDirectory = saveLoc)
# You should see results for two developed models in the ouuput. The output is as well
# uploaded to a sqlite database in the saveLoc/sqlite folder,
dir(saveLoc)
# The dir output should show two Analysis_ folders with the results,
# two targetId_ folders with th extracted data, and a sqlite folder with the database
# The results can be explored in the shiny app by calling viewMultiplePlp(saveLoc)

# clean up (viewing the results in the shiny app is won't work after this)
unlink(saveLoc, recursive = TRUE)
```

runPlp — *runPlp - Develop and internally evaluate a model using specified settings*

Description

This provides a general framework for training patient level prediction models. The user can select various default feature selection methods or incorporate their own, The user can also select from a range of default classifiers or incorporate their own. There are three types of evaluations for the model patient (randomly splits people into train/validation sets) or year (randomly splits data into train/validation sets based on index year - older in training, newer in validation) or both (same as year spliting but checks there are no overlaps in patients within training set and validaiton set - any overlaps are removed from validation set)

Usage

```
runPlp(
  plpData,
  outcomeId = plpData$metaData$databaseDetails$outcomeIds[1],
  analysisId = paste(Sys.Date(), outcomeId, sep = "-"),
  analysisName = "Study details",
  populationSettings = createStudyPopulationSettings(),
  splitSettings = createDefaultSplitSetting(type = "stratified", testFraction = 0.25,
    trainFraction = 0.75, splitSeed = 123, nfold = 3),
  sampleSettings = createSampleSettings(type = "none"),
  featureEngineeringSettings = createFeatureEngineeringSettings(type = "none"),
  preprocessSettings = createPreprocessSettings(minFraction = 0.001, normalize = TRUE),
  modelSettings = setLassoLogisticRegression(),
  logSettings = createLogSettings(verbosity = "DEBUG", timeStamp = TRUE, logName =
    "runPlp Log"),
  executeSettings = createDefaultExecuteSettings(),
  saveDirectory = NULL
)
```

Arguments

`plpData`	An object of type `plpData` - the patient level prediction data extracted from the CDM. Can also include an initial population as plpData$popualtion.
`outcomeId`	(integer) The ID of the outcome.
`analysisId`	(integer) Identifier for the analysis. It is used to create, e.g., the result folder. Default is a timestamp.
`analysisName`	(character) Name for the analysis
`populationSettings`	An object of type `populationSettings` created using `createStudyPopulationSettings` that specifies how the data class labels are defined and addition any exclusions to apply to the plpData cohort
`splitSettings`	An object of type `splitSettings` that specifies how to split the data into train/validation/test. The default settings can be created using `createDefaultSplitSetting`.
`sampleSettings`	An object of type `sampleSettings` that specifies any under/over sampling to be done. The default is none.
`featureEngineeringSettings`	An object of `featureEngineeringSettings` specifying any feature engineering to be learned (using the train data)
`preprocessSettings`	An object of `preprocessSettings`. This setting specifies the minimum fraction of target population who must have a covariate for it to be included in the model training and whether to normalise the covariates before training
`modelSettings`	An object of class `modelSettings` created using one of the function: • setLassoLogisticRegression() A lasso logistic regression model • setGradientBoostingMachine() A gradient boosting machine • setAdaBoost() An ada boost model • setRandomForest() A random forest model • setDecisionTree() A decision tree model • setKNN() A KNN model
`logSettings`	An object of `logSettings` created using `createLogSettings` specifying how the logging is done
`executeSettings`	An object of `executeSettings` specifying which parts of the analysis to run
`saveDirectory`	The path to the directory where the results will be saved (if NULL uses working directory)

Details

This function takes as input the plpData extracted from an OMOP CDM database and follows the specified settings to develop and internally validate a model for the specified outcomeId.

Value

An plpResults object containing the following:

- model The developed model of class `plpModel`
- executionSummary A list containing the hardward details, R package details and execution time
- performanceEvaluation Various internal performance metrics in sparse format
- prediction The plpData cohort table with the predicted risks added as a column (named value)
- covariateSummary A characterization of the features for patients with and without the outcome during the time at risk
- analysisRef A list with details about the analysis

Examples

```
# simulate some data
data('simulationProfile')
plpData <- simulatePlpData(simulationProfile, n = 1000)
# develop a model with the default settings
saveLoc <- file.path(tempdir(), "runPlp")
results <- runPlp(plpData = plpData, outcomeId = 3, analysisId = 1,
                  saveDirectory = saveLoc)
# to check the results you can view the log file at saveLoc/1/plpLog.txt
# or view with shiny app using viewPlp(results)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`savePlpAnalysesJson` *Save the modelDesignList to a json file*

Description

Save the modelDesignList to a json file

Usage

```
savePlpAnalysesJson(
  modelDesignList = list(createModelDesign(targetId = 1, outcomeId = 2, modelSettings =
    setLassoLogisticRegression()), createModelDesign(targetId = 1, outcomeId = 3,
     modelSettings = setLassoLogisticRegression())),
   cohortDefinitions = NULL,
   saveDirectory = NULL
)
```

Arguments

`modelDesignList`	A list of modelDesigns created using `createModelDesign()`
`cohortDefinitions`	A list of the cohortDefinitions (generally extracted from ATLAS)
`saveDirectory`	The directory to save the modelDesignList settings

Details

This function creates a json file with the modelDesignList saved

Value

The json string of the ModelDesignList

Examples

```
modelDesign <- createModelDesign(targetId = 1,
                                 outcomeId = 2,
                                 modelSettings = setLassoLogisticRegression())
saveLoc <- file.path(tempdir(), "loadPlpAnalysesJson")
jsonFile <- savePlpAnalysesJson(modelDesignList = modelDesign, saveDirectory = saveLoc)
# clean up
unlink(saveLoc, recursive = TRUE)
```

`savePlpData` *Save the plpData to folder*

Description

`savePlpData` saves an object of type plpData to folder.

Usage

```
savePlpData(plpData, file, envir = NULL, overwrite = FALSE)
```

Arguments

`plpData`	An object of type `plpData` as generated using `getPlpData`.
`file`	The name of the folder where the data will be written. The folder should not yet exist.
`envir`	The environment for to evaluate variables when saving
`overwrite`	Whether to force overwrite an existing file

Value

Called for its side effect, the data will be written to a set of files in the folder specified by the user.

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 500)
saveLoc <- file.path(tempdir(), "savePlpData")
savePlpData(plpData, saveLoc)
dir(saveLoc, full.names = TRUE)

# clean up
unlink(saveLoc, recursive = TRUE)
```

`savePlpModel` *Saves the plp model*

Description

Saves the plp model

Usage

```
savePlpModel(plpModel, dirPath)
```

Arguments

`plpModel`	A trained classifier returned by running `runPlp()$model`
`dirPath`	A location to save the model to

Details

Saves the plp model to a user specificed folder

Value

```
The directory path where the model was saved
```

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "savePlpModel")
plpResult <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
path <- savePlpModel(plpResult$model, file.path(saveLoc, "savedModel"))
# show the saved model
dir(path, full.names = TRUE)

# clean up
unlink(saveLoc, recursive = TRUE)
```

savePlpResult — *Saves the result from runPlp into the location directory*

Description

Saves the result from runPlp into the location directory

Usage

```
savePlpResult(result, dirPath)
```

Arguments

`result`	The result of running runPlp()
`dirPath`	The directory to save the csv

Details

Saves the result from runPlp into the location directory

Value

The directory path where the results were saved

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "savePlpResult")
results <- runPlp(plpData, outcomeId = 3, saveDirectory = saveLoc)
# save the results
newSaveLoc <- file.path(tempdir(), "savePlpResult", "saved")
savePlpResult(results, newSaveLoc)
# show the saved results
dir(newSaveLoc, recursive = TRUE, full.names = TRUE)

# clean up
unlink(saveLoc, recursive = TRUE)
unlink(newSaveLoc, recursive = TRUE)
```

savePlpShareable *Save the plp result as json files and csv files for transparent sharing*

Description

Save the plp result as json files and csv files for transparent sharing

Usage

```
savePlpShareable(result, saveDirectory, minCellCount = 10)
```

Arguments

`result`	An object of class runPlp with development or validation results
`saveDirectory`	The directory the save the results as csv files
`minCellCount`	Minimum cell count for the covariateSummary and certain evaluation results

Details

Saves the main results json/csv files (these files can be read by the shiny app)

Value

```
The directory path where the results were saved
```

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
saveLoc <- file.path(tempdir(), "savePlpShareable")
results <- runPlp(plpData, saveDirectory = saveLoc)
newSaveLoc <- file.path(tempdir(), "savePlpShareable", "saved")
path <- savePlpShareable(results, newSaveLoc)
# show the saved result
dir(newSaveLoc, full.names = TRUE, recursive = TRUE)

# clean up
unlink(saveLoc, recursive = TRUE)
unlink(newSaveLoc, recursive = TRUE)
```

savePrediction — *Saves the prediction dataframe to a json file*

Description

Saves the prediction dataframe to a json file

Usage

```
savePrediction(prediction, dirPath, fileName = "prediction.json")
```

Arguments

`prediction`	The prediciton data.frame
`dirPath`	The directory to save the prediction json
`fileName`	The name of the json file that will be saved

Details

Saves the prediction data frame returned by predict.R to an json file and returns the fileLocation where the prediction is saved

Value

```
The file location where the prediction was saved
```

Examples

```
prediction <- data.frame(
  rowIds = c(1, 2, 3),
  outcomeCount = c(0, 1, 0),
  value = c(0.1, 0.9, 0.2)
)
saveLoc <- file.path(tempdir())
savePrediction(prediction, saveLoc)
dir(saveLoc)

# clean up
unlink(file.path(saveLoc, "prediction.json"))
```

`setAdaBoost` *Create setting for AdaBoost with python DecisionTreeClassifier base estimator*

Description

Create setting for AdaBoost with python DecisionTreeClassifier base estimator

Usage

```
setAdaBoost(
  nEstimators = list(10, 50, 200),
  learningRate = list(1, 0.5, 0.1),
  algorithm = list("SAMME"),
  seed = sample(1e+06, 1)
)
```

Arguments

`nEstimators`	(list) The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.
`learningRate`	(list) Weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the learningRate and nEstimators parameters There is a trade-off between learningRate and nEstimators.
`algorithm`	Only ‘SAMME’ can be provided. The ’algorithm’ argument will be deprecated in scikit-learn 1.8.
`seed`	A seed for the model

Value

a modelSettings object

Examples

```
## Not run:
model <- setAdaBoost(nEstimators = list(10),
                     learningRate = list(0.1),
                     seed = 42)

## End(Not run)
```

setCoxModel *Create setting for lasso Cox model*

Description

Create setting for lasso Cox model

Usage

```
setCoxModel(
  variance = 0.01,
  seed = NULL,
  includeCovariateIds = c(),
  noShrinkage = c(),
  threads = -1,
  upperLimit = 20,
  lowerLimit = 0.01,
  tolerance = 2e-07,
  maxIterations = 3000
)
```

Arguments

`variance`	Numeric: prior distribution starting variance
`seed`	An option to add a seed when training the model
`includeCovariateIds`	a set of covariate IDS to limit the analysis to
`noShrinkage`	a set of covariates whcih are to be forced to be included in the final model. default is the intercept
`threads`	An option to set number of threads when training model
`upperLimit`	Numeric: Upper prior variance limit for grid-search
`lowerLimit`	Numeric: Lower prior variance limit for grid-search
`tolerance`	Numeric: maximum relative change in convergence criterion from successive iterations to achieve convergence
`maxIterations`	Integer: maximum iterations of Cyclops to attempt before returning a failed-to-converge error

Value

`modelSettings` object

Examples

```
coxL1 <- setCoxModel()
```

setDecisionTree — *Create setting for the scikit-learn DecisionTree with python*

Description

Create setting for the scikit-learn DecisionTree with python

Usage

```
setDecisionTree(
  criterion = list("gini"),
  splitter = list("best"),
  maxDepth = list(as.integer(4), as.integer(10), NULL),
  minSamplesSplit = list(2, 10),
  minSamplesLeaf = list(10, 50),
  minWeightFractionLeaf = list(0),
  maxFeatures = list(100, "sqrt", NULL),
  maxLeafNodes = list(NULL),
  minImpurityDecrease = list(10^-7),
  classWeight = list(NULL),
  seed = sample(1e+06, 1)
)
```

Arguments

`criterion`	The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.
`splitter`	The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.
`maxDepth`	(list) The maximum depth of the tree. If NULL, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
`minSamplesSplit`	The minimum number of samples required to split an internal node
`minSamplesLeaf`	The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least minSamplesLeaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.
`minWeightFractionLeaf`	The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sampleWeight is not provided.
`maxFeatures`	(list) The number of features to consider when looking for the best split (int/’sqrt’/NULL)
`maxLeafNodes`	(list) Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. (int/NULL)

`minImpurityDecrease`	Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.
`classWeight`	(list) Weights associated with classes 'balance' or NULL
`seed`	The random state seed

Value

a modelSettings object

Examples

```
## Not run:
model <- setDecisionTree(criterion = list("gini"),
                         maxDepth = list(4),
                         minSamplesSplit = list(2),
                         minSamplesLeaf = list(10),
                         seed = 42)

## End(Not run)
```

---

setGradientBoostingMachine

Create setting for gradient boosting machine model using gbm_xgboost implementation

---

Description

Create setting for gradient boosting machine model using gbm_xgboost implementation

Usage

```
setGradientBoostingMachine(
  ntrees = c(100, 300),
  nthread = 20,
  earlyStopRound = 25,
  maxDepth = c(4, 6, 8),
  minChildWeight = 1,
  learnRate = c(0.05, 0.1, 0.3),
  scalePosWeight = 1,
  lambda = 1,
  alpha = 0,
  seed = sample(1e+07, 1)
)
```

Arguments

`ntrees`	The number of trees to build
`nthread`	The number of computer threads to use (how many cores do you have?)
`earlyStopRound`	If the performance does not increase over earlyStopRound number of trees then training stops (this prevents overfitting)
`maxDepth`	Maximum depth of each tree - a large value will lead to slow model training
`minChildWeight`	Minimum sum of of instance weight in a child node - larger values are more conservative
`learnRate`	The boosting learn rate
`scalePosWeight`	Controls weight of positive class in loss - useful for imbalanced classes
`lambda`	L2 regularization on weights - larger is more conservative
`alpha`	L1 regularization on weights - larger is more conservative
`seed`	An option to add a seed when training the final model

Value

A modelSettings object that can be used to fit the model

Examples

```
modelGbm <- setGradientBoostingMachine(
  ntrees = c(10, 100), nthread = 20,
  maxDepth = c(4, 6), learnRate = c(0.1, 0.3)
)
```

---

setIterativeHardThresholding
Create setting for Iterative Hard Thresholding model

Description

Create setting for Iterative Hard Thresholding model

Usage

```
setIterativeHardThresholding(
  K = 10,
  penalty = "bic",
  seed = sample(1e+05, 1),
  exclude = c(),
  forceIntercept = FALSE,
  fitBestSubset = FALSE,
  initialRidgeVariance = 0.1,
  tolerance = 1e-08,
  maxIterations = 10000,
  threshold = 1e-06,
  delta = 0
)
```

Arguments

`K`	The maximum number of non-zero predictors
`penalty`	Specifies the IHT penalty; possible values are `BIC` or `AIC` or a numeric value
`seed`	An option to add a seed when training the model
`exclude`	A vector of numbers or covariateId names to exclude from prior
`forceIntercept`	Logical: Force intercept coefficient into regularization
`fitBestSubset`	Logical: Fit final subset with no regularization
`initialRidgeVariance`	integer
`tolerance`	numeric
`maxIterations`	integer
`threshold`	numeric
`delta`	numeric

Value

`modelSettings` object

Examples

```
modelIht <- setIterativeHardThresholding(K = 5, seed = 42)
```

`setLassoLogisticRegression` *Create modelSettings for lasso logistic regression*

Description

Create modelSettings for lasso logistic regression

Usage

```
setLassoLogisticRegression(
  variance = 0.01,
  seed = NULL,
  includeCovariateIds = c(),
  noShrinkage = c(0),
  threads = -1,
  forceIntercept = FALSE,
  upperLimit = 20,
  lowerLimit = 0.01,
  tolerance = 2e-06,
  maxIterations = 3000,
  priorCoefs = NULL
)
```

Arguments

`variance`	Numeric: prior distribution starting variance
`seed`	An option to add a seed when training the model
`includeCovariateIds`	a set of covariateIds to limit the analysis to
`noShrinkage`	a set of covariates whcih are to be forced to be included in in the final model. Default is the intercept
`threads`	An option to set number of threads when training model.
`forceIntercept`	Logical: Force intercept coefficient into prior
`upperLimit`	Numeric: Upper prior variance limit for grid-search
`lowerLimit`	Numeric: Lower prior variance limit for grid-search
`tolerance`	Numeric: maximum relative change in convergence criterion from from successive iterations to achieve convergence
`maxIterations`	Integer: maximum iterations of Cyclops to attempt before returning a failed-to-converge error
`priorCoefs`	Use coefficients from a previous model as starting points for model fit (transfer learning)

Value

`modelSettings` object

Examples

```
modelLasso <- setLassoLogisticRegression(seed=42)
```

setLightGBM

Create setting for gradient boosting machine model using lightGBM (https://github.com/microsoft/LightGBM/tree/master/R-package).

Description

Create setting for gradient boosting machine model using lightGBM (https://github.com/microsoft/LightGBM/tree/master/R-package).

Usage

```
setLightGBM(
  nthread = 20,
  earlyStopRound = 25,
  numIterations = c(100),
  numLeaves = c(31),
  maxDepth = c(5, 10),
  minDataInLeaf = c(20),
  learningRate = c(0.05, 0.1, 0.3),
  lambdaL1 = c(0),
  lambdaL2 = c(0),
  scalePosWeight = 1,
  isUnbalance = FALSE,
  seed = sample(1e+07, 1)
)
```

Arguments

`nthread`	The number of computer threads to use (how many cores do you have?)
`earlyStopRound`	If the performance does not increase over earlyStopRound number of trees then training stops (this prevents overfitting)
`numIterations`	Number of boosting iterations.
`numLeaves`	This hyperparameter sets the maximum number of leaves. Increasing this parameter can lead to higher model complexity and potential overfitting.
`maxDepth`	This hyperparameter sets the maximum depth . Increasing this parameter can also lead to higher model complexity and potential overfitting.
`minDataInLeaf`	This hyperparameter sets the minimum number of data points that must be present in a leaf node. Increasing this parameter can help to reduce overfitting
`learningRate`	This hyperparameter controls the step size at each iteration of the gradient descent algorithm. Lower values can lead to slower convergence but may result in better performance.
`lambdaL1`	This hyperparameter controls L1 regularization, which can help to reduce overfitting by encouraging sparse models.
`lambdaL2`	This hyperparameter controls L2 regularization, which can also help to reduce overfitting by discouraging large weights in the model.

`scalePosWeight`	Controls weight of positive class in loss - useful for imbalanced classes
`isUnbalance`	This parameter cannot be used at the same time with scalePosWeight, choose only one of them. While enabling this should increase the overall performance metric of your model, it will also result in poor estimates of the individual class probabilities.
`seed`	An option to add a seed when training the final model

Value

A list of settings that can be used to train a model with `runPlp`

Examples

```
modelLightGbm <- setLightGBM(
  numLeaves = c(20, 31, 50), maxDepth = c(-1, 5, 10),
  minDataInLeaf = c(10, 20, 30), learningRate = c(0.05, 0.1, 0.3)
)
```

`setMLP` *Create setting for neural network model with python's scikit-learn. For bigger models, consider using* `DeepPatientLevelPrediction` *package.*

Description

Create setting for neural network model with python's scikit-learn. For bigger models, consider using `DeepPatientLevelPrediction` package.

Usage

```
setMLP(
  hiddenLayerSizes = list(c(100), c(20)),
  activation = list("relu"),
  solver = list("adam"),
  alpha = list(0.3, 0.01, 1e-04, 1e-06),
  batchSize = list("auto"),
  learningRate = list("constant"),
  learningRateInit = list(0.001),
  powerT = list(0.5),
  maxIter = list(200, 100),
  shuffle = list(TRUE),
  tol = list(1e-04),
  warmStart = list(TRUE),
  momentum = list(0.9),
  nesterovsMomentum = list(TRUE),
  earlyStopping = list(FALSE),
  validationFraction = list(0.1),
  beta1 = list(0.9),
  beta2 = list(0.999),
  epsilon = list(1e-08),
  nIterNoChange = list(10),
  seed = sample(1e+05, 1)
)
```

Arguments

`hiddenLayerSizes`	(list of vectors) The ith element represents the number of neurons in the ith hidden layer.
`activation`	(list) Activation function for the hidden layer. • "identity": no-op activation, useful to implement linear bottleneck, returns f(x) = x • "logistic": the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x)). • "tanh": the hyperbolic tan function, returns f(x) = tanh(x). • "relu": the rectified linear unit function, returns f(x) = max(0, x)
`solver`	(list) The solver for weight optimization. (‘lbfgs’, ‘sgd’, ‘adam’)
`alpha`	(list) L2 penalty (regularization term) parameter.
`batchSize`	(list) Size of minibatches for stochastic optimizers. If the solver is ‘lbfgs’, the classifier will not use minibatch. When set to “auto”, batchSize=min(200, n_samples).
`learningRate`	(list) Only used when solver=’sgd’ Learning rate schedule for weight updates. ‘constant’, ‘invscaling’, ‘adaptive’, default=’constant’
`learningRateInit`	(list) Only used when solver=’sgd’ or ‘adam’. The initial learning rate used. It controls the step-size in updating the weights.
`powerT`	(list) Only used when solver=’sgd’. The exponent for inverse scaling learning rate. It is used in updating effective learning rate when the learning_rate is set to ‘invscaling’.
`maxIter`	(list) Maximum number of iterations. The solver iterates until convergence (determined by ‘tol’) or this number of iterations. For stochastic solvers (‘sgd’, ‘adam’), note that this determines the number of epochs (how many times each data point will be used), not the number of gradient steps.
`shuffle`	(list) boolean: Whether to shuffle samples in each iteration. Only used when solver=’sgd’ or ‘adam’.
`tol`	(list) Tolerance for the optimization. When the loss or score is not improving by at least tol for nIterNoChange consecutive iterations, unless learning_rate is set to ‘adaptive’, convergence is considered to be reached and training stops.
`warmStart`	(list) When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.
`momentum`	(list) Momentum for gradient descent update. Should be between 0 and 1. Only used when solver=’sgd’.

`nesterovsMomentum`	(list) Whether to use Nesterov's momentum. Only used when solver='sgd' and momentum > 0.
`earlyStopping`	(list) boolean Whether to use early stopping to terminate training when validation score is not improving. If set to true, it will automatically set aside 10 percent of training data as validation and terminate training when validation score is not improving by at least tol for n_iter_no_change consecutive epochs.
`validationFraction`	(list) The proportion of training data to set aside as validation set for early stopping. Must be between 0 and 1. Only used if earlyStopping is True.
`beta1`	(list) Exponential decay rate for estimates of first moment vector in adam, should be in 0 to 1.
`beta2`	(list) Exponential decay rate for estimates of second moment vector in adam, should be in 0 to 1.
`epsilon`	(list) Value for numerical stability in adam.
`nIterNoChange`	(list) Maximum number of epochs to not meet tol improvement. Only effective when solver='sgd' or 'adam'.
`seed`	A seed for the model

Value

a modelSettings object

Examples

```
## Not run:
model <- setMLP(hiddenLayerSizes = list(c(20)), alpha=list(3e-4), seed = 42)

## End(Not run)
```

---

setNaiveBayes *Create setting for naive bayes model with python*

Description

Create setting for naive bayes model with python

Usage

```
setNaiveBayes()
```

Value

a modelSettings object

Examples

```
## Not run:
plpData <- getEunomiaPlpData()
model <- setNaiveBayes()
analysisId <- "naiveBayes"
saveLocation <- file.path(tempdir(), analysisId)
results <- runPlp(plpData, modelSettings = model,
                  saveDirectory = saveLocation,
                  analysisId = analysisId)
# clean up
unlink(saveLocation, recursive = TRUE)

## End(Not run)
```

`setPythonEnvironment` *Use the python environment created using configurePython()*

Description

Use the python environment created using configurePython()

Usage

```
setPythonEnvironment(envname = "PLP", envtype = NULL)
```

Arguments

`envname`	A string for the name of the virtual environment (default is 'PLP')
`envtype`	An option for specifying the environment as'conda' or 'python'. If NULL then the default is 'conda' for windows users and 'python' for non-windows users

Details

This function sets PatientLevelPrediction to use a python environment

Value

A string indicating the which python environment will be used

Examples

```
## Not run:  #' # create a conda environment named PLP
configurePython(envname="PLP", envtype="conda")

## End(Not run)
```

setRandomForest — *Create setting for random forest model using sklearn*

Description

Create setting for random forest model using sklearn

Usage

```
setRandomForest(
  ntrees = list(100, 500),
  criterion = list("gini"),
  maxDepth = list(4, 10, 17),
  minSamplesSplit = list(2, 5),
  minSamplesLeaf = list(1, 10),
  minWeightFractionLeaf = list(0),
  mtries = list("sqrt", "log2"),
  maxLeafNodes = list(NULL),
  minImpurityDecrease = list(0),
  bootstrap = list(TRUE),
  maxSamples = list(NULL, 0.9),
  oobScore = list(FALSE),
  nJobs = list(NULL),
  classWeight = list(NULL),
  seed = sample(1e+05, 1)
)
```

Arguments

`ntrees`	(list) The number of trees to build
`criterion`	(list) The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.
`maxDepth`	(list) The maximum depth of the tree. If NULL, then nodes are expanded until all leaves are pure or until all leaves contain less than minSamplesSplit samples.
`minSamplesSplit`	(list) The minimum number of samples required to split an internal node
`minSamplesLeaf`	(list) The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least minSamplesLeaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.
`minWeightFractionLeaf`	(list) The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sampleWeight is not provided.

`mtries`	(list) The number of features to consider when looking for the best split: • int then consider max_features features at each split. • float then max_features is a fraction and round(max_features * n_features) features are considered at each split • 'sqrt' then max_features=sqrt(n_features) • 'log2' then max_features=log2(n_features) • NULL then max_features=n_features
`maxLeafNodes`	(list) Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.
`minImpurityDecrease`	(list) A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
`bootstrap`	(list) Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.
`maxSamples`	(list) If bootstrap is True, the number of samples to draw from X to train each base estimator.
`oobScore`	(list) Whether to use out-of-bag samples to estimate the generalization score. Only available if bootstrap=True.
`nJobs`	The number of jobs to run in parallel.
`classWeight`	(list) Weights associated with classes. If not given, all classes are supposed to have weight one. NULL, “balanced”, “balanced_subsample”
`seed`	A seed when training the final model

Value

a modelSettings object

Examples

```
## Not run:
plpData <- getEunomiaPlpData()
model <- setRandomForest(ntrees = list(100),
                         maxDepth = list(4),
                         minSamplesSplit = list(2),
                         minSamplesLeaf = list(10),
                         maxSamples = list(0.9),
                         seed = 42)
saveLoc <- file.path(tempdir(), "randomForest")
results <- runPlp(plpData, modelSettings = model, saveDirectory = saveLoc)
# clean up
unlink(saveLoc, recursive = TRUE)

## End(Not run)
```

setSVM *Create setting for the python sklearn SVM (SVC function)*

Description

Create setting for the python sklearn SVM (SVC function)

Usage

```
setSVM(
  C = list(1, 0.9, 2, 0.1),
  kernel = list("rbf"),
  degree = list(1, 3, 5),
  gamma = list("scale", 1e-04, 3e-05, 0.001, 0.01, 0.25),
  coef0 = list(0),
  shrinking = list(TRUE),
  tol = list(0.001),
  classWeight = list(NULL),
  cacheSize = 500,
  seed = sample(1e+05, 1)
)
```

Arguments

`C`	(list) Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.
`kernel`	(list) Specifies the kernel type to be used in the algorithm. one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'. If none is given 'rbf' will be used.
`degree`	(list) degree of kernel function is significant only in poly, rbf, sigmoid
`gamma`	(list) kernel coefficient for rbf and poly, by default 1/n_features will be taken. 'scale', 'auto' or float, default='scale'
`coef0`	(list) independent term in kernel function. It is only significant in poly/sigmoid.
`shrinking`	(list) whether to use the shrinking heuristic.
`tol`	(list) Tolerance for stopping criterion.
`classWeight`	(list) Class weight based on imbalance either 'balanced' or NULL
`cacheSize`	Specify the size of the kernel cache (in MB).
`seed`	A seed for the model

Value

a modelSettings object

Examples

```
## Not run:
plpData <- getEunomiaPlpData()
model <- setSVM(C = list(1), gamma = list("scale"), seed = 42)
saveLoc <- file.path(tempdir(), "svm")
results <- runPlp(plpData, modelSettings = model, saveDirectory = saveLoc)
# clean up
unlink(saveLoc, recursive = TRUE)

## End(Not run)
```

`simulatePlpData` *Generate simulated data*

Description

`simulateplpData` creates a plpData object with simulated data.

Usage

```
simulatePlpData(plpDataSimulationProfile, n = 10000)
```

Arguments

`plpDataSimulationProfile`	An object of type `plpDataSimulationProfile` as generated using the `createplpDataSimulationProfile` function.
`n`	The size of the population to be generated.

Details

This function generates simulated data that is in many ways similar to the original data on which the simulation profile is based.

Value

An object of type `plpData`.

Examples

```
# first load the simulation profile to use
data("simulationProfile")
# then generate the simulated data
plpData <- simulatePlpData(simulationProfile, n = 100)
nrow(plpData$cohorts)
```

`simulationProfile` *A simulation profile for generating synthetic patient level prediction data*

Description

A simulation profile for generating synthetic patient level prediction data

Usage

```
data(simulationProfile)
```

Format

A data frame containing the following elements:

covariatePrevalence prevalence of all covariates

outcomeModels regression model parameters to simulate outcomes

metaData settings used to simulate the profile

covariateRef covariateIds and covariateNames

timePrevalence time window

exclusionPrevalence prevalence of exclusion of covariates

`sklearnFromJson` *Loads sklearn python model from json*

Description

Loads sklearn python model from json

Usage

```
sklearnFromJson(path)
```

Arguments

`path`	path to the model json file

Value

a sklearn python model object

Examples

```
## Not run:
plpData <- getEunomiaPlpData()
modelSettings <- setDecisionTree(maxDepth = list(3), minSamplesSplit = list(2),
                                  minSamplesLeaf = list(1), maxFeatures = list(100))
saveLocation <- file.path(tempdir(), "sklearnFromJson")
results <- runPlp(plpData, modelSettings = modelSettings, saveDirectory = saveLocation)
# view save model
dir(results$model$model, full.names = TRUE)
# load into a sklearn object
model <- sklearnFromJson(file.path(results$model$model, "model.json"))
# max depth is 3 as we set in beginning
model$max_depth
# clean up
unlink(saveLocation, recursive = TRUE)

## End(Not run)
```

`sklearnToJson` *Saves sklearn python model object to json in path*

Description

Saves sklearn python model object to json in path

Usage

```
sklearnToJson(model, path)
```

Arguments

`model`	a fitted sklearn python model object
`path`	path to the saved model file

Value

nothing, saves the model to the path as json

Examples

```
## Not run:
sklearn <- reticulate::import("sklearn", convert = FALSE)
model <- sklearn$tree$DecisionTreeClassifier()
model$fit(sklearn$datasets$load_iris()$data, sklearn$datasets$load_iris()$target)
saveLoc <- file.path(tempdir(), "model.json")
sklearnToJson(model, saveLoc)
# the model.json is saved in the tempdir
dir(tempdir())
# clean up
```

```
unlink(saveLoc)

## End(Not run)
```

`splitData` *Split the plpData into test/train sets using a splitting settings of class* `splitSettings`

Description

Split the plpData into test/train sets using a splitting settings of class `splitSettings`

Usage

```
splitData(
  plpData = plpData,
  population = population,
  splitSettings = createDefaultSplitSetting(splitSeed = 42)
)
```

Arguments

`plpData`	An object of type `plpData` - the patient level prediction data extracted from the CDM.
`population`	The population created using `createStudyPopulation` that define who will be used to develop the model
`splitSettings`	An object of type `splitSettings` specifying the split - the default can be created using `createDefaultSplitSetting`

Value

Returns a list containing the training data (Train) and optionally the test data (Test). Train is an Andromeda object containing

- covariates: a table (rowId, covariateId, covariateValue) containing the covariates for each data point in the train data
- covariateRef: a table with the covariate information
- labels: a table (rowId, outcomeCount, ...) for each data point in the train data (outcomeCount is the class label)
- folds: a table (rowId, index) specifying which training fold each data point is in.

Test is an Andromeda object containing

- covariates: a table (rowId, covariateId, covariateValue) containing the covariates for each data point in the test data
- covariateRef: a table with the covariate information
- labels: a table (rowId, outcomeCount, ...) for each data point in the test data (outcomeCount is the class label)

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n = 1000)
population <- createStudyPopulation(plpData)
splitSettings <- createDefaultSplitSetting(testFraction = 0.50,
                                           trainFraction = 0.50, nfold = 5)
data = splitData(plpData, population, splitSettings)
# test data should be ~500 rows (changes because of study population)
nrow(data$Test$labels)
# train data should be ~500 rows
nrow(data$Train$labels)
# should be five fold in the train data
length(unique(data$Train$folds$index))
```

`summary.plpData` *Summarize a plpData object*

Description

Summarize a plpData object

Usage

```
## S3 method for class 'plpData'
summary(object, ...)
```

Arguments

`object`	The plpData object to summarize
`...`	Additional arguments

Value

A summary of the object containing the number of people, outcomes and covariates

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=10)
summary(plpData)
```

toSparseM *Convert the plpData in COO format into a sparse R matrix*

Description

Converts the standard plpData to a sparse matrix

Usage

```
toSparseM(plpData, cohort = NULL, map = NULL)
```

Arguments

plpData	An object of type plpData with covariate in coo format - the patient level prediction data extracted from the CDM.
cohort	If specified the plpData is restricted to the rowIds in the cohort (otherwise plpData$labels is used)
map	A covariate map (telling us the column number for covariates)

Details

This function converts the covariates Andromeda table in COO format into a sparse matrix from the package Matrix

Value

Returns a list, containing the data as a sparse matrix, the plpData covariateRef and a data.frame named map that tells us what covariate corresponds to each column This object is a list with the following components:

data A sparse matrix with the rows corresponding to each person in the plpData and the columns corresponding to the covariates.

covariateRef The plpData covariateRef.

map A data.frame containing the data column ids and the corresponding covariateId from covariateRef.

Examples

```
library(dplyr)
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=100)
# how many covariates are there before we convert to sparse matrix
plpData$covariateData$covariates %>%
 dplyr::group_by(.data$covariateId) %>%
 dplyr::summarise(n = n()) %>%
 dplyr::collect() %>% nrow()
sparseData <- toSparseM(plpData, cohort=plpData$cohorts)
```

```
# how many covariates are there after we convert to sparse matrix'
sparseData$dataMatrix@Dim[2]
```

validateExternal *validateExternal - Validate model performance on new data*

Description

validateExternal - Validate model performance on new data

Usage

```
validateExternal(
  validationDesignList,
  databaseDetails,
  logSettings = createLogSettings(verbosity = "INFO", logName = "validatePLP"),
  outputFolder
)
```

Arguments

validationDesignList	A list of objects created with `createValidationDesign`
databaseDetails	A list of objects of class `databaseDetails` created using `createDatabaseDetails`
logSettings	An object of `logSettings` created using `createLogSettings`
outputFolder	The directory to save the validation results to (subfolders are created per database in validationDatabaseDetails)

Value

A list of results

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n=1000)
# first fit a model on some data, default is a L1 logistic regression
saveLoc <- file.path(tempdir(), "development")
results <- runPlp(plpData, saveDirectory = saveLoc)
# then create my validation design
validationDesign <- createValidationDesign(1, 3, plpModelList = list(results$model))
# I will validate on Eunomia example database
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
Eunomia::createCohorts(connectionDetails)
databaseDetails <- createDatabaseDetails(connectionDetails = connectionDetails,
cdmDatabaseSchema = "main", cdmDatabaseName = "Eunomia", cdmDatabaseId = 1,
targetId = 1, outcomeIds = 3)
path <- file.path(tempdir(), "validation")
validateExternal(validationDesign, databaseDetails, outputFolder = path)
# see generated result files
dir(path, recursive = TRUE)
# clean up
unlink(saveLoc, recursive = TRUE)
unlink(path, recursive = TRUE)
```

`validateMultiplePlp` *externally validate the multiple plp models across new datasets*

Description

This function loads all the models in a multiple plp analysis folder and validates the models on new data

Usage

```
validateMultiplePlp(
  analysesLocation,
  validationDatabaseDetails,
  validationRestrictPlpDataSettings = createRestrictPlpDataSettings(),
  recalibrate = NULL,
  cohortDefinitions = NULL,
  saveDirectory = NULL
)
```

Arguments

`analysesLocation`	The location where the multiple plp analyses are
`validationDatabaseDetails`	A single or list of validation database settings created using `createDatabaseDetails()`
`validationRestrictPlpDataSettings`	The settings specifying the extra restriction settings when extracting the data created using `createRestrictPlpDataSettings()`.
`recalibrate`	A vector of recalibration methods (currently supports 'RecalibrationintheLarge' and/or 'weakRecalibration')
`cohortDefinitions`	A list of cohortDefinitions
`saveDirectory`	The location to save to validation results

Details

Users need to input a location where the results of the multiple plp analyses are found and the connection and database settings for the new data

Value

Nothing. The results are saved to the saveDirectory

Examples

```
# first develop a model using runMultiplePlp
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
Eunomia::createCohorts(connectionDetails = connectionDetails)
databaseDetails <- createDatabaseDetails(connectionDetails = connectionDetails,
                                         cdmDatabaseId = "1",
                                         cdmDatabaseName = "Eunomia",
                                         cdmDatabaseSchema = "main",
                                         targetId = 1,
                                         outcomeIds = 3)
covariateSettings <-
 FeatureExtraction::createCovariateSettings(useDemographicsGender = TRUE,
   useDemographicsAge = TRUE, useConditionOccurrenceLongTerm = TRUE)
modelDesign <- createModelDesign(targetId = 1,
                                 outcomeId = 3,
                                 modelSettings = setLassoLogisticRegression(seed = 42),
                                 covariateSettings = covariateSettings)
saveLoc <- file.path(tempdir(), "valiateMultiplePlp", "development")
results <- runMultiplePlp(databaseDetails = databaseDetails,
               modelDesignList = list(modelDesign),
               saveDirectory = saveLoc)
# now validate the model on a Eunomia but with a different target
analysesLocation <- saveLoc
validationDatabaseDetails <- createDatabaseDetails(connectionDetails = connectionDetails,
                                                   cdmDatabaseId = "2",
                                                   cdmDatabaseName = "EunomiaNew",
                                                   cdmDatabaseSchema = "main",
                                                   targetId = 4,
                                                   outcomeIds = 3)
newSaveLoc <- file.path(tempdir(), "valiateMultiplePlp", "validation")
validateMultiplePlp(analysesLocation = analysesLocation,
                    validationDatabaseDetails = validationDatabaseDetails,
                    saveDirectory = newSaveLoc)
# the results could now be viewed in the shiny app with viewMultiplePlp(newSaveLoc)
```

viewDatabaseResultPlp *open a local shiny app for viewing the result of a PLP analyses from a database*

Description

open a local shiny app for viewing the result of a PLP analyses from a database

Usage

```
viewDatabaseResultPlp(
  mySchema,
  myServer,
  myUser,
  myPassword,
  myDbms,
  myPort = NULL,
  myTableAppend
)
```

Arguments

`mySchema`	Database result schema containing the result tables
`myServer`	server with the result database
`myUser`	Username for the connection to the result database
`myPassword`	Password for the connection to the result database
`myDbms`	database management system for the result database
`myPort`	Port for the connection to the result database
`myTableAppend`	A string appended to the results tables (optional)

Details

Opens a shiny app for viewing the results of the models from a database

Value

Opens a shiny app for interactively viewing the results

Examples

```
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
Eunomia::createCohorts(connectionDetails)
databaseDetails <- createDatabaseDetails(connectionDetails = connectionDetails,
                                         cdmDatabaseSchema = "main",
                                         cdmDatabaseName = "Eunomia",
                                          cdmDatabaseId = "1",
                                          targetId = 1,
                                          outcomeIds = 3)
modelDesign <- createModelDesign(targetId = 1,
                                 outcomeId = 3,
                                 modelSettings = setLassoLogisticRegression())
saveLoc <- file.path(tempdir(), "viewDatabaseResultPlp", "developement")
runMultiplePlp(databaseDetails = databaseDetails, modelDesignList = list(modelDesign),
               saveDirectory = saveLoc)
# view result files
dir(saveLoc, recursive = TRUE)
viewDatabaseResultPlp(myDbms = "sqlite",
                      mySchema = "main",
                      myServer = file.path(saveLoc, "sqlite", "databaseFile.sqlite"),
                      myUser = NULL,
                      myPassword = NULL,
                      myTableAppend = "")
# clean up, shiny app can't be opened after the following has been run
unlink(saveLoc, recursive = TRUE)
```

viewMultiplePlp — *open a local shiny app for viewing the result of a multiple PLP analyses*

Description

open a local shiny app for viewing the result of a multiple PLP analyses

Usage

```
viewMultiplePlp(analysesLocation)
```

Arguments

`analysesLocation`	The directory containing the results (with the analysis_x folders)

Details

Opens a shiny app for viewing the results of the models from various T,O, Tar and settings settings.

Value

Opens a shiny app for interactively viewing the results

Examples

```
connectionDetails <- Eunomia::getEunomiaConnectionDetails()
Eunomia::createCohorts(connectionDetails)
databaseDetails <- createDatabaseDetails(connectionDetails = connectionDetails,
                                         cdmDatabaseSchema = "main",
                                         cdmDatabaseName = "Eunomia",
                                         cdmDatabaseId = "1",
                                         targetId = 1,
                                         outcomeIds = 3)
modelDesign <- createModelDesign(targetId = 1,
                                 outcomeId = 3,
                                 modelSettings = setLassoLogisticRegression())
saveLoc <- file.path(tempdir(), "viewMultiplePlp", "development")
runMultiplePlp(databaseDetails = databaseDetails, modelDesignList = list(modelDesign),
               saveDirectory = saveLoc)
# view result files
dir(saveLoc, recursive = TRUE)
# open shiny app
viewMultiplePlp(analysesLocation = saveLoc)
# clean up, shiny app can't be opened after the following has been run
unlink(saveLoc, recursive = TRUE)
```

`viewPlp` *viewPlp - Interactively view the performance and model settings*

Description

This is a shiny app for viewing interactive plots of the performance and the settings

Usage

```
viewPlp(runPlp, validatePlp = NULL, diagnosePlp = NULL)
```

Arguments

`runPlp`	The output of runPlp() (an object of class 'runPlp')
`validatePlp`	The output of externalValidatePlp (on object of class 'validatePlp')
`diagnosePlp`	The output of diagnosePlp()

Details

Either the result of runPlp and view the plots

Value

Opens a shiny app for interactively viewing the results

Examples

```
data("simulationProfile")
plpData <- simulatePlpData(simulationProfile, n= 1000)
saveLoc <- file.path(tempdir(), "viewPlp", "development")
results <- runPlp(plpData, saveDirectory = saveLoc)
# view result files
dir(saveLoc, recursive = TRUE)
# open shiny app
viewPlp(results)
# clean up, shiny app can't be opened after the following has been run
unlink(saveLoc, recursive = TRUE)
```

Index