**Package ‘pMineR’**

February 7, 2017

**Type**  Package

**Title**  Processes Mining in Medicine

**Version**  0.31

**Date**  2017-02-01

**Author**  Roberto Gatta [aut, cre],
            Jacopo Lenkowicz [aut],
            Mauro Vallati [aut],
            Alessandro Stefanini [aut],
            Nicola Dinapoli [ctb],
            Berardino De Bari [ctb],
            Lucia Sacchi [ctb]

**Maintainer**  Roberto Gatta <roberto.gatta.bs@gmail.com>

**Description**  Allows to build and train simple Process Mining (PM) models. The aim is to sup-
port PM specifically for the clinical domain from both administrative and clinical data.

**License**  GPL-3

**LazyData**  true

**Depends**  DiagrammeR (>= 0.8.2), stringr (>= 1.0.0), XML (>= 3.98-1.3),
             cluster (>= 2.0.4)

**NeedsCompilation**  yes

**RoxygenNote**  5.0.1

**Suggests**  knitr, rmarkdown

**Repository**  CRAN

**Date/Publication**  2017-02-07 17:58:40

---

**R topics documented:**

- cluster_expectationMaximization ............................................. 2
- cluster_hierarchicalTree ....................................................... 3
- cluster_partitionAroundMedoids ............................................. 5
- confCheck_easy ........................................................................ 6
- dataLoader .............................................................................. 7
A class to perform Expectation-Maximization clustering on sequential data for Process Mining issues

Description

This class performs sequence clustering on an event-log with the Expectation-Maximization (EM) algorithm. The public methods are:

- `cluster_expectationMaximization()` is the constructor of the class
- `loadDataset()` loads data taken from a `dataloader::getData()` method, into a `cluster_expectationMaximization` object
- `calculateClusters()` performs the actual clustering computation on the previously loaded dataset
- `getClusters()` returns the clusters computed by the `cluster_expectationMaximization::calculateClusters()` method
- `getClusterStats()` returns informations about the clustering result (i.e. support, between-cluster distance, within-cluster mean distance and standard deviation)
- `getClusterLog()` returns informations about the clustering computation itself (i.e. iterations needed to converge, centroids value after each iteration)

In order to better understand the use of such methods, please visit: www.pminer.info

Parameters for `cluster_expectationMaximization::calculateClusters()` method are:

- `num` the number of clusters it has to generate
- `typeofModel` the name of the Process Mining model it has to use to generate the space (up to now, only the default "firstOrderMarkovModel" is provided)

Usage

```cpp
cluster_expectationMaximization()
```
Examples

```r
## Not run:

# create a Loader
obj.L <- dataLoader();  # create a Loader

# Load a .csv using "DES" and "ID" as column names to indicate events
# and Patient's ID
obj.L$load.csv(nameFile = "./otherFiles/test_02.csv",
IDName = "ID", EVENTName = "DES", dateColumnName = "DATA")

# now create an object cluster_expectationMaximization
obj.cEM <- cluster_expectationMaximization();

# load the data into logInspector object
obj.cEM$loadDataset( obj.L$getData() );

# perform clustering computation
obj.cEM$calculateClusters( num = 5, typeOfModel = "firstOrderMarkovModel" );

# get calculated clusters
a <- obj.cEM$getClusters();

# get informations about performance of clusters
b <- obj.cEM$getClusterStats();

# get log of each iteration of the algorithm
d <- obj.cEM$getClusterLog();

## End(Not run)
```

---

**cluster_hierarchicalTree**

*A class to perform Hierarchical Tree clustering on sequential data for Process Mining issues*

---

**Description**

This class performs sequence clustering on an event-log with the Hierarchical Tree (HT) algorithm. The public methods are:

- `cluster_hierarchicalTree()` is the constructor of the class
- `loadDataset(...)` loads data taken from a `dataLoader::getData()` method, into a `cluster_hierarchicalTree()` object
- `calculateClusters()` performs the actual clustering computation on the previously loaded dataset
- `getClusters()` returns the clusters computed by the `cluster_hierarchicalTree::calculateClusters()` method
• `getClusterStats( ... )` returns informations about the clustering result (i.e. support, between-cluster distance, within-cluster mean distance and standard deviation)

• `getClusterLog( ... )` returns informations about the clustering computation itself (i.e. iterations needed to converge, centroids value after each iteration)

In order to better understand the use of such methods, please visit: www.pminer.info

Parameters for `cluster_hierarchicalTree::calculateClusters()` method are:

• `num` the number of clusters it has to generate

• `typeOfModel` the name of the Process Mining model it has to use to generate the space (up to now, only the default "firstOrdermarkovModel" is provided)

Usage

```
cluster_hierarchicalTree()
```

Examples

```r
## Not run:

# create a Loader
obj.L<-dataLoader();

# Load a .csv using "DES" and "ID" as column names to indicate events
# and Patient's ID
obj.L$load.csv(nameFile = ".\otherFiles\test_02.csv",
IDName = "ID",EVENTName = "DES",dateColumnName = "DATA")

# now create an object cluster_expectationMaximization
obj.cEM<- cluster_expectationMaximization();

# load the data into logInspector object
obj.cEM$loadDataset( obj.L$getData() );

# perform clustering computation
obj.cEM$calculateClusters( num = 2);

# get calculated clusters
a <- obj.cEM$getClusters();

# get informations about performance of clusters
b <- obj.cEM$getClusterStats();

# get log of each iteration of the algorithm
d <- obj.cEM$getClusterLog();

## End(Not run)
```
**cluster_partitionAroundMedoids**

A class to perform Partition Around Medoids clustering on sequential data for Process Mining issues

---

**Description**

This class performs sequence clustering on an event-log with the Partition Around Medoids (PAM) algorithm. The public methods are:

- `cluster_partitionAroundMedoids()` is the constructor of the class
- `loadDataset(...)` loads data taken from a `dataLoader::getData()` method, into a `cluster_partitionAroundMedoids()` object
- `calculateClusters()` performs the actual clustering computation on the previously loaded dataset
- `getClusters()` returns the clusters computed by the `cluster_partitionAroundMedoids::calculateClusters()` method
- `getClusterStats(...)` returns informations about the clustering result (i.e. support, between-cluster distance, within-cluster mean distance and standard deviation)
- `getClusterLog(...)` returns informations about the clustering computation itself (i.e. iterations needed to converge, centroids value after each iteration)

In order to better understand the use of such methods, please visit: [www.pminer.info](http://www.pminer.info)

Parameters for `cluster_partitionAroundMedoids::calculateClusters()` method are:

- `num` the number of clusters it has to generate
- `typeofModel` the name of the Process Mining model it has to use to generate the space (up to now, only the default "firstOrderMarkovModel" is provided)

**Usage**

`cluster_partitionAroundMedoids()`

**Examples**

```r
## Not run:

# create a Loader
obj.L<-dataLoader();

# Load a .csv using "DES" and "ID" as column names to indicate events
# and Patient's ID
obj.L$load.csv(nomFile = "./otherFiles/test_02.csv",
IDName = "ID",EVENTName = "DES",dateColumnName = "DATA")

# now create an object cluster_partitionAroundMedoids
obj.cIPAM<- cluster_partitionAroundMedoids();
```
confCheck_easy

A simple conformance checking class

Description

A first module for making conformance checking

Usage

confCheck_easy(verbosemode = TRUE)

Arguments

verbose.mode boolean. If TRUE some messages will appear in console, during the computation; otherwise the computation will be silent.

Examples

## Not run:

# Create a Conformance Checker obj
obj.cc <- confCheck_easy()

# Load an XML with the workflow to check
obj.cc$loadWorkFlow(WF.fileNamex=myFiles/otherFiles/import_01/rules.v2.xml)

# plot the graph related to the XML
obj.cc$plot()

# now play 20 processes, 10 correct and 10 mismatchful
# (matching and not matching can be seen thanks to the 'valido' column)
aaa <- obj.cc$play(number.of.cases = 20, min.num.of.valid.words = 10)
# Build a dataLoaderObject
objDL <- dataLoader()

# load the previously generate data.frame
objDL$load.data.frame(mydata = aaa$valid.data.frame, IDName = "patID", EVENTName = "event", dateColumnName = "date")

# now load the data into the obj
obj.cc$loadDataset(dataList = objDL$getData())
# replay the loaded data
obj.cc$replay()

# plot the result, showing the terminations in absolute values
obj.cc$plot.replay.result(whatToCount = "terminations", kindOfNumber = "absolute")
# plot the result, showing the transitions in relative values
obj.cc$plot.replay.result(whatToCount = "activations", kindOfNumber = "relative")

# get the XML result
xmlText <- obj.cc$get.XML.replay.result()
# or the same data in form of list
list.result <- obj.cc$get.list.replay.result()

# plot the timeline of the first patient
# and the timeline computed during the re-play
obj.cc$plotPatientEventTimeline(patientID = "1")
obj.cc$plotPatientReplayedTimeline(patientID = "1")

## End(Not run)

dataLoader  

Load the event-logs

Description

A loader for csv based log files. It also calculates the footprint table, transition matrix probabilities, and presents data in different shapes. The public methods are:

- `dataLoader()` the constructor
- `load.csv(...)` loads the csv file into the dataLoader object
- `load.data.frame()` loads a data.frame into the dataLoader object
- `getData()` return the processed, previously-loaded, data
- `removeEvents()` remove the events in the array passed as argument (dual of `dataLoader::keepOnlyEvents()`)
- `keepOnlyEvents()` keep only the events in the array passed as argument (dual of `dataLoader::removeEvents()`)


dataProcessor

- addDictionary() add a dictionary in order, afterward, to translate or group some event name
- getTranslation() perform a translation applying the given dictionary to the loaded csv or data.frame
- plotTimeline() plot the timeline of the events regarding a single patient

In order to better understand the use of such methods, please visit: www.pminer.info

The constructor admit the following parameters: verbose.mode are some notification wished, during the computation? The default value is true

Usage

dataloader(Verbose.mode = TRUE)

Arguments

verbose.mode boolean. If TRUE some messages will appear in console, during the computation; otherwise the computation will be silent.

Examples

## Not run:

# create a Loader
obj.L <- dataloader()

# Load a .csv
obj.L$load.csv(nameFile = "../otherFiles/mammella.csv",
IDName = "CODICE_SANITARIO_ADT",
EVENTName = "DESC_REPARTO_RICOVERO",
dateColumnName = "DATA_RICOVERO")

# return the results
obj.L$getData()

## End(Not run)

---

dataProcessor some data processing useful tools

Description

A class which provide some tools. pMineR internal use only.

Usage

dataProcessor()
A class to train First Order Markov Models

Description

This is an implementation of the First Order Markov Model (FOMM) for Process Mining issues. This class provides a minimal set of methods to handle with the FOMM model:

- `firstOrderMarkovModel()` is the constructor
- `loadDataset()` loads data taken from a `dataLoader::getData()` method, into a FOMM object
- `trainModel()` train a model using the previously loaded dataset
- `replay()` re-play a given event log on the internal FOMM model
- `play()` play the internal FOMM model a desired number of times, in order to simulate new event-logs. This methods can also, if desired, simulate event-logs which does not complies with the internal FOMM model.
- `plot()` plot the internal model
- `distanceFrom()` calculate the scalar distance to another passed FOMM model, passed as argument. The default metric returns a scalar value
- `getModel()` return the trained internal FOMM model
- `getLogObj()` return an XML containing the execution-log of a `firstOrderMarkovModel::play()` computation
- `getInstanceClass()` return the instance class Name and description (version, etc.)
- `plot.delta.graph()` plot a graph, in the desired modality, representing the difference between the internal FOMM and a passed one.
- `get.transition.Prob()` calculate the probability to go in a given number of transitions, from a state to another
- `get.time.transition.Prob()` calculate the probability to go in a given time, from a state to another
- `build.PWF()` build automatically a PWF XML definition script.
- `findReachableNodes()` and return the array containing the reacheable states, starting from the passed one.

In order to better understand the use of such methods, please visit: www.pminer.info

The constructor admit the following parameters: `parameters.list` a list containing possible parameters to tune the model.

Usage

`firstOrderMarkovModel(parameters.list = list())`
Arguments

parameters.list

a list containing the parameters. The possible ones are: 'considerAutoLoop' and 'threshold'. 'considerAutoLoop' is a boolean which indicates if the autoloops have to be admitted, while 'threshold' is the minimum value that a probability should have to do not be set to zero, in the transition matrix.

Examples

```r
## Not run:

# create a Loader
obj.L <- dataLoader();

# Load a .csv
obj.L$load.csv(nomFile = "..\otherFiles\mammella.csv",
IDName = "CODICE_SANITARIO_ADT",
EVENTName = "DESC_REPARTO_RICOVERO",
dateColumnName = "DATA_RICOVERO")

# get the loaded data
dati <- obj.L$getData()

# build a Second Order Marvov Model with a threshold of 0.2
FOMM <- firstOrderMarkovModel(parameters.list = list("threshold"=0.001))

# load the data
FOMM$loadDataset(dataList = dati)

# train a model
FOMM$trainModel()

# generate 10 new processes (nb: if the threshold is too low, it can fail...)
aaa <- FOMM$play(numberOfPlays = 10)

# get the transition matrix
TranMatrix <- FOMM$getModel(kindOfOutput = "MMatrix.perc")

# plot the model
FOMM$plot()

# generate other 20 fake-processes
ooo <- FOMM$play(numberOfPlays = 20)

## End(Not run)
```
logInspector

A class to perform a preliminary analysis on sequential data for Process Mining issues

Description

This class aims at inspecting an event-log for descriptive analysis purposes. The public methods are:

- `logInspector()` is the constructor of the class
- `loadDataset()` loads data taken from a `dataloader::getData()` method, into a `logInspector()` object
- `getEventStats()` computes and returns event-related stats, such as absolute and relative events frequency
- `getProcessStats()` computes and returns process-related stats, such as absolute and relative processes frequency
- `plotEventStats()` plots the event-related stats (input is the number of most frequent events it has to plot)
- `plotProcessStats()` plots the process-related stats (input is the number of most frequent processes it has to plot)
- `timeDistribution.stats.plot()` <not yet commented>

In order to better understand the use of such methods, please visit: www.pminer.info

Parameter for `logInspector::plotEventStats()` and `logInspector::plotProcessStats()` methods is:

- `num` the number of most frequent events/processes to plot

Usage

`logInspector()`

Examples

```r
## Not run:
# -----------------------------
# USING THE METHODS of the class
# -----------------------------
obj.l<-dataLoader();  # create a Loader

# Load a .csv using "DES" and "ID" as column names to indicate events and Patient's ID
obj.l$load.csv(nomFile = "../otherFiles/test_02.csv", IDName = "ID", EVENTName = "DES", dateColumnName = "DATA")

# now create an object logInspector
obj.logl<-logInspector();
```
# load the data into logInspector object
ombre.logi$loadDataset(obj.logi$getData());

# get event-related descriptive statistics
obj.logi$getEventStats();

# get process-related descriptive statistics
obj.logi$getProcessStats();

# plot event-related descriptive statistics
obj.logi$plotEventStats();

# plot process-related descriptive statistics
obj.logi$plotProcessStats();

## End(Not run)

---

**meta.dataLoader**

Load the event-logs serializing many dataLoaders objs

---

**Description**

This class is more abstract than dataLoader and provide some facilities, in particular to cope with
dictionaries and translations. Because it handles, internally, a set of dataLoader objects, any
dataLoader object must be referred by 'view'.

- meta.dataLoader() the constructor
- load.csv(...) loads the csv file into the dataLoader object
- load.data.frame() loads a data.frame into the dataLoader object
- getData() return the processed, previously-loaded, data
- removeEvents() remove the events in the array passed as argument (dual of dataLoader::removeEvents() )
- keepOnlyEvents() keep only the events in the array passed as argument (dual of dataLoader::removeEvents() )
- addDictionary() add a dictionary in order, afterward, to translate or group some event name
- getTranslation() perform a translation applying the given dictionary to the loaded csv or data.frame
- plot.Timeline() plot the timeline of the events regarding a single patient

In order to better understand the use of such methods, please visit: www.pminer.info

The constructor admit the following parameters: verbose.mode are some notification wished, during
the computation? The default value is true


```r
## Not run:

# create a Loader
obj.L<-meta.dataLoader();

# create a view
obj.L$createView(view.name = "mammella")
obj.L$createView(view.name = "retto")

# Load a .csv into the view 'mammella'
obj.L$load.csv(nomeFile = "/otherFiles/mammella.csv",
IDName = "CODICE_SANITARIO_ADT",
EVENTName = "DESC_REPARTO_RICOVERO",
dateColumnName = "DATA_RICOVERO", view="mammella")

# Load a .csv into the view 'retto'
obj.L$load.csv(nomeFile = "/otherFiles/mammella.csv",
IDName = "CODICE_SANITARIO_ADT",
EVENTName = "DESC_REPARTO_RICOVERO",
dateColumnName = "DATA_RICOVERO", view="retto")

# get the data from the view 'retto'
aaa <- obj.L$getData(view = "retto")
```

```
## End(Not run)
```

---

**plotTimeline**

*Plot a patient’s timeline*

**Description**

It plot a patient’s timeline given an event log well formatted in the input `eventTable`

**Usage**

```r
plotTimeline(eventTable, output.format.date = "%d/%m/%Y", cex.axis = 0.6,
             cex.text = 0.7)
```
Argument

eventTable: a table containing the event logs. The table has to have as columnName c('DATA', 'EVENT').

output.format.date: the format of the passed date. The default value is 'd / m / y'.
cex.axis: cex for timeline-text

cex.text: cex for event-text

---

secondOrderMarkovModel

A class to train Second Order Markov Models.

Description

This is an implementation of the Second Order Markov Model (SOMM) for Process Mining issues. This class provides a minimal set of methods to handle with the FOMM model:

- secondOrderMarkovModel( ) is the constructor
- loadDataset( ) loads data taken from a dataLoader::getData() method, into a SOMM object
- trainModel( ) train a model using the previously loaded dataset
- replay( ) re-play a given event log on the internal SOMM model
- play( ) play the internal FOMM model a desired number of times, in order to simulate new event-logs. This methods can also, if desired, simulate event-logs which does not complies with the internal SOMM model.
- getModel( ) return the trained internal SOMM model

In order to better understand the use of such methods, please visit: www.pminer.info

The constructor admit the following parameters: parameters.list a list containing possible parameters to tune the model.

Usage

secondOrderMarkovModel(parameters.list = list())

Arguments

parameters.list: a list containing the parameters. The possible ones are: ‘considerAutoLoop’ and ‘threshold’. ‘considerAutoLoop’ is a boolean which indicates if the autoloops have to be admitted, while ‘threshold’ is the minimum value that a probability should have to do not be set to zero, in the transition matrix.
Examples

```r
## Not run:

# create a Loader
obj.L <- dataLoader();

# Load a .csv
obj.L$load.csv(nameFile = "../otherFiles/mammella.csv",
               IDName = "CODICE_SANITARIO_ADT",
               EVENTName = "DESC_REPARTO_RICOVERO",
               dateColumnName = "DATA_RICOVERO")

# get the loaded data
dati <- obj.L$getData()

# build a Second Order Markov Model with a threshold of 0.2
SOMM <- secondOrderMarkovModel(
         parameters.list = list("threshold"=0.002))

# load the data
SOMM$loadDataset(dataList = dati)

# train a model
SOMM$trainModel()

# generate 10 new processes (nb: if the
# threshold is too low, it can fail...)
aaa <- SOMM$play(numberOfPlays = 10)

# get the transition matrix
TranMatrix <- SOMM$getModel( kindOfOutput = "MM.2.Matrix.perc"
)

## End(Not run)
```

<table>
<thead>
<tr>
<th>utilities</th>
<th>Some useful tools</th>
</tr>
</thead>
</table>

Description

A class which provide some tools. pMineR internal use only.

Usage

```r
utils()
```
Index

cluster_expectationMaximization, 2
cluster_hierarchicalTree, 3
cluster_partitionAroundMedoids, 5
confCheck_easy, 6

dataLoader, 7
dataProcessor, 8

firstOrderMarkovModel, 9
logInspector, 11
meta.dataLoader, 12
plotTimeline, 13
secondOrderMarkovModel, 14
utils, 15