

# Package: EABN (via r-universe)

September 22, 2024

**Version** 0.6-2

**Date** 2023/06/21

**Title** Evidence Accumulation Bayes Net Engine

**Author** Russell Almond

**Maintainer** Russell Almond <ralmond@fsu.edu>

**Depends** R (>= 3.0), methods, Peanut (>= 0.8), mongo, Proc4 (>= 0.8),  
withr, RNetica

**Imports** futile.logger, mongolite, jsonlite

**Suggests** PNetica, knitr, rmarkdown, tidyR

**Description** Extracts observables from a sequence of events.

**License** Artistic-2.0

**URL** <http://pluto.coe.fsu.edu/Proc4>

**Collate** Evidence.R StudentRec.R EAEngine.R EAEngineMongo.R  
EAEngineNDB.R EngineGears.R Runners.R

**VignetteBuilder** knitr

**Support** c('Bill & Melinda Gates Foundation grant `` Games as  
Learning/Assessment: Stealth Assessment' (#0PP1035331, Val  
Shute, PI)', 'National Science Foundation grant `` DIP:  
Game-based Assessment and Support of STEM-related Competencies'  
(#1628937, Val Shute, PI)', 'National Science Foundation grant  
`` Mathematical Learning via Architectural Design and Modeling  
Using E-Rebuild.' (#1720533, Fengfeng Ke, PI)', 'Institute of  
Educational Statistics Grant: `` Exploring adaptive cognitive and  
affective learning support for next-generation STEM learning  
games.' (#R305A170376-20, Val Shute and Russell Almond, PIs')

**Repository** <https://ralmond.r-universe.dev>

**RemoteUrl** <https://github.com/ralmond/EABN>

**RemoteRef** HEAD

**RemoteSha** 45a946fc8d78045944befbae1bb9397d788c161f

## Contents

EABN-package . . . . .	2
accumulateEvidence . . . . .	5
BNEngine-class . . . . .	8
BNEngineMongo . . . . .	11
BNEngineMongo-class . . . . .	14
BNEngineNDB . . . . .	17
BNEngineNDB-class . . . . .	19
configStats . . . . .	22
doBuild . . . . .	23
doRunrun . . . . .	27
EvidenceSet . . . . .	32
EvidenceSet-class . . . . .	33
fetchSM . . . . .	35
getRecordForUser . . . . .	37
getSR . . . . .	39
history . . . . .	43
loadManifest . . . . .	45
logEvidence . . . . .	47
logIssue . . . . .	50
mainLoop . . . . .	51
observables . . . . .	53
parseEvidence . . . . .	55
parseStats . . . . .	56
parseStudentRecord . . . . .	57
setupDefaultSR . . . . .	60
sm . . . . .	62
stat . . . . .	63
StudentRecord . . . . .	64
StudentRecord-class . . . . .	66
StudentRecordSet . . . . .	68
StudentRecordSet-class . . . . .	70
trimTable . . . . .	72
updateHist . . . . .	73
updateSM . . . . .	75
updateStats . . . . .	78
<b>Index</b>	<b>81</b>

---

EABN-package

*Evidence Accumulation Bayes Net Engine*


---

## Description

Extracts observables from a sequence of events.

## Details

The DESCRIPTION file: This package was not yet installed at build time.

Index: This package was not yet installed at build time.

The most important object in the package is the [BNEngine](#) which does most of the work of scoring. In particular, it takes a [P4Message](#) object containing observables, and a [StudentRecord](#) object and updates the student record.

It comes in two variants, [BNEngineMongo](#) which links to a Mongo database, and [BNEngineNDB](#) which processes raw messages without the database.

The functions [doBuild](#) builds the [BNEngine](#), and the function [doRunrun](#) runs the engine on a queue of messages in the database. The function [handleEvidence](#) processes a single evidence message.

## Author(s)

Russell Almond

Maintainer: Russell Almond <ralmond@fsu.edu>

## References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapters 5 and 13.

## See Also

[Proc4](#) – Low level support for messaging. [EIEvent](#) – Evidence Accumulation which produces the input messages for EABN.

## Examples

```
cat("This sample file is available in", system.file("conf/RunEABN.R",
  package="EABN"), "\n")
```

```
## Not run:
library(R.utils)
library(EABN)
library(PNetica)
library(futile.logger)
library(jsonlite)
```

```
if (interactive()) {
  ## Edit these for the local application
  appStem <- "P4test"
  loglevel <- ""
  noprep <- FALSE
  override <- FALSE
} else {
  appStem <- cmdArg("app", NULL)
  if (is.null(app) || !grepl("^ecd://", app))
```

```

    stop("No app specified, use '--args app=ecd://...')
    loglevel <- cmdArg("level","")
    noprep <- as.logical(cmdArg("noprep",FALSE))
    override <- as.logical(cmdArg("override",FALSE))
  }

## This is the default location for INI files for Proc4 tools.
source("/usr/local/share/Proc4/EAini.R")

## Assumes the path config.dir (set in the INI file) contains a file
## config.json giving the location of the necessary configuration files
## and network files.
EA.config <- jsonlite::fromJSON(file.path(config.dir,"config.json"),FALSE)

app <- as.character(Proc4.config$app$appStem)
if (length(app)==0L || any(app=="NULL")) {
  stop("Could not find app for ",appStem)
}
if (!(isTRUE(match(appStem,EA.config$appStem)))) {
  stop("Configuration not set for app ",appStem)
}

## Start Netica
sess <- NeticaSession(LicenseKey=NeticaLicenseKey)
startSession(sess)

logfile <- (file.path(logpath, sub("<app>",appStem,EA.config$logname)))
if (interactive()) {
  flog.appender(appender.tee(logfile))
} else {
  flog.appender(appender.file(logfile))
}
flog.threshold(EA.config$logLevel)

## Load extensions.
for (ext in EA.config$extensions) {
  if (is.character(ext) && nchar(ext) > 0L) {
    if (file.exists(file.path(config.dir,ext))) {
      source(file.path(config.dir,ext))
    } else {
      flog.error("Can't find extension file
    }
  }
}

## This will build the engine and run all messages in the QUEUE.

eng <- doRunrun(app,sess,EA.config,EAeng.local,config.dir,outdir,
  logfile=logfile,override=override,noprep=noprep)

## The engine object can now be used to process further messages or

```

```
## access the EABN database.

## End(Not run)
```

---

accumulateEvidence	<i>Merge evidence from an evidence set with the student record.</i>
--------------------	---

---

## Description

The function `accumulateEvidence` combines the evidence in the [EvidenceSet](#) with the exiting beliefs in the [StudentRecord](#), updating the student record. The function `handleEvidence` is a wrapper around this which takes care of finding and updating the evidence sets.

## Usage

```
accumulateEvidence(eng, rec, evidMess, debug = 0)
handleEvidence(eng, evidMess, srser = NULL, debug = 0)
```

## Arguments

<code>eng</code>	The <a href="#">BNEngine</a> which controls the process.
<code>rec</code>	The <a href="#">StudentRecord</a> which will be updated.
<code>srser</code>	A serialized version of the student record for the no-database version of the model.
<code>evidMess</code>	An <a href="#">EvidenceSet</a> which has the evidence to be incorporated.
<code>debug</code>	An integer flag. If greater than 1, then <a href="#">recover()</a> will be called at strategic places during the processing to allow inspection of the process.

## Details

The function `accumulateEvidence` performs the following steps:

1. Update the student record to associate it with the new evidence ([updateRecord](#)).
2. Update the student model with the new evidence ([updateSM](#)).
3. Update the statistics for the new student model ([updateStats](#)).
4. Update the history for the new evidence ([updateHist](#)).
5. Announce the availability of new statistics ([announceStats](#)).
6. Save the updated student record ([saveSR](#)).

The function `handleEvidence` is a wrapper around `accumulateEvidence` which finds the student record. Note for [BNEngineNDB](#), it is expected that the student record will be passed in as a serialized object (see [getRecordForUser](#)). It performs the following steps:

1. Fetch the student record for the `uid` associated with the evidence set ([getRecordForUser](#)).

2. Mark the evidence as belonging to this student record ([logEvidence](#)).
3. Update the record by calling `accumulateEvidence`.
4. Mark the evidence as processed ([markAsProcessed](#)).

If an error is encountered, then the error message is added to the evidence set.

### Value

The modified [StudentRecord](#) which was just processed. If an error occurs during the call to `accumulateEvidence` both function will return an object of class `try-error` instead of the student record.

### Logging, Error Handling and Debugging

The functions `handleEvidence`, `accumulateEvidence` and many of the functions they call use the [flog.logger](#) protocol. The default logging level of INFO will give messages in response to the announcements and warnings when an error occur. The DEBUG and TRACE levels will provide more information about the details of the update algorithm.

The body of `accumulateEvidence` is wrapped in [withFlogging](#) which captures and logs errors. This function returns an object of class `try-error` when an error occurs. Although `handleEvidence` does not use the flogging error handler, it will still pass on the `try-error` if one is generated.

The debug argument can be used to pause execution. Basically, [recover\(\)](#) will be called between every step. This only happens in interactive mode as it just does not make sense in batch model.

### Known Bugs

There is a bug in version 5.04 of Netica which causes the `absorbNodes` function when called with a node that does not have display information to generate an internal Netica error. This has been fixed with version 6.07, which is currently in beta release (Linux only).

To work around, make sure that either all nodes do not have display information, or that all do.

### Author(s)

Russell Almond

### References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapters 5 and 13.

### See Also

Classes: [BNEngine](#) [BNEngineMongo](#), [BNEngineNDB](#) [StudentRecord](#), [EvidenceSet](#)

Main Loop Functions: [mainLoop](#), [getRecordForUser](#), [logEvidence](#), [updateRecord](#), [updateSM](#), [updateStats](#), [updateHist](#), [announceStats](#), [saveSR](#)

## Examples

```
## Requires database setup, also PNetica
library(RNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman,session=sess,
  address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[",app,"]"),
  db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app,warehouse=Nethouse,
  listenerSet=ls,manifest=netman,
  profModel="miniPP_CM",
  histNodes="Physics",
  statmat=stattab,
  activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng,netman)
eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

sr0 <- getRecordForUser(eng,"S1")

eap0 <- stat(sr0,"Physics_EAP")

e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
  obs=list("CompensatoryObs"="Right"))

e1 <- logEvidence(eng,sr0,e1)
sr1 <- accumulateEvidence(eng,sr0,e1)
stopifnot(!is(sr1,'try-error'))
stopifnot(m_id(sr1)!=m_id(sr0),sr1@prev_id==m_id(sr0))
stopifnot(seqno(sr1)==1L, seqno(e1)==1L)

eap1 <- stat(sr1,"Physics_EAP")
stopifnot(abs(eap1-eap0) > .001)
```

```

stopifnot(nrow(history(sr1,"Phycis"))==2L)

## handle Evidence.
sr1.ser <- as.json(sr1)
e2 <- EvidenceSet(uid="S2",app="Test",context="PPconjEM",
                  obs=list("ConjunctiveObs"="Wrong"))

sr2 <- handleEvidence(eng,e2,jsonlite::fromJSON(sr1.ser))
stopifnot(!is(sr2,'try-error'))
eap2 <- stat(sr2,"Physics_EAP")
stopifnot(uid(sr2)==uid(sr1),
          m_id(sr1)==sr2@prev_id,
          nrow(history(sr2,"Physics"))==3L,
          abs(eap1-eap2) > .001)

## <<HERE>> Need test with Mongo engine.

```

---

BNEngine-class

Class "BNEngine"

---

## Description

A generic engine for handling evidence messages ([EvidenceSet](#) objects).

## Details

This is the basic class for running the evidence accumulation process. This is actually an abstract class, there are two subclasses: [BNEngineMongo](#), which uses the Mongo database to store student records and as a message queue, and [BNEngineNDB](#), which operates without a database. Note that the BNEngine constructor generates an error.

The following functions form the core of the Engine Protocol:

[loadManifest](#) This loads the network manifest for the [PnetWarehouse](#).

[setupDefaultSR](#) Sets up the default Student Record (used for creating new student records)

[configStats](#) Configures the statistics that are reported in the main loop.

[baselineHist](#) Sets up the baselines for histories.

[mainLoop](#) This runs through a queue of messages, handling the evidence.

[handleEvidence](#) Handles evidence from one scoring context and one user.

[accumulateEvidence](#) Does the actual work of processing the evidence.

[getRecordForUser](#) Fetches the student record for a user, essentially a call to [getSR](#).

[logEvidence](#) Logs the evidence as part of the student record.

[updateSM](#) Updates the student model for the new evidence.

[updateStats](#) Calculates new statistics for the revised student model.

[updateHist](#) Updates the history for the revised student model.

[announceStats](#) Updates other processes about the existance of updated statistics.



**Extends**

All reference classes extend and inherit methods from "[envRefClass](#)".

**Methods**

**app** signature(x = "BNEngine"): Returns the guid identifying the application that this engine is handling.

**notifyListeners** signature(sender = "BNEngine"): Notifies other processes that student records have been updated.

**fetchNextEvidence** signature(eng = "BNEngine"): Returns the next unprocessed [EvidenceSet](#) in the queue.

**markProcessed** signature(eng = "BNEngine", eve = "EvidenceSet"): marks the eve argument as processed.

**Fields**

**app**: Object of class character giving an globally unique identifier for the application

**srs**: Object of class [StudentRecordSet](#) of NULL giving the student record set for the application.

**profModel**: Object of class character giving the name of the proficiency model (for the default student record) in the warehouse manifest.

**listenerSet**: Object of class [ListenerSet](#) giving a set of listeners who will listen for new statistics.

**statistics**: Object of class list containing [Statistic](#) objects to be run on every update cycle.

**histNodes**: Object of class character giving the names of the nodes in the proficiency model whose history will be recorded.

**warehouseObj**: Object of class [PnetWarehouse](#) which stores the Bayes nets, both evidence models and student models are stored here.

**waittime**: Object of class numeric giving the time in seconds the main event loop should wait before checking again for messages.

**processN**: Object of class numeric giving the number of times that the main loop should run before stopping. If Inf, then the main loop will run without stopping.

**Class-Based Methods**

**activate()**: Sets the flag to indicate that the process is running.

**deactivate()**: Clears the flag to indicate that the process is no longer running.

**shouldHalt()**: This function checks the database to see whether or not the flag is set to cause the process to halt after processing the current record..

**stopWhenFinished()**: This function checks the database to see whether or not the flag is set to cause the process to stop when the event queue is empty.

**setHistNodes(nodenames)**: Sets the names of the history nodes. Note this should be called before the call to [baselineHist](#) or the history nodes will not be set properly in the default student record.

**fetchNextEvidence()**: Fetches the next evidence set to be handled.

setError(mess, e): Adds an error flag to an evidence set that generated an error.

getHistNodes(): Retrieves the history nodes.

saveStats(statmat): Updates the set of statistics associated with this engine.

studentRecords(): Fetches the [StudentRecordSet](#) associated with the engine. Note: This method should be called instead of the raw field as it will initialize the field if it is not set up yet.

fetchStats(): Fetches statistic objects from the database.

stats(): Returns the set of [Statistic](#) objects associated with the engine.

fetchManifest(): Fetches the network manifest from the database.

setManifest(manifest): Sets the manifest for the [PnetWarehouse](#).

saveManifest(manifest): Saves the network manifest to the database.

show(): Provides a printed representation of the database.

setProcessed(mess): Sets an evidence set message as processed.

warehouse(): Returns the [PnetWarehouse](#) associated with this engine. Again, this function should be called in preference to directly accessing the field as it forces initialization when necessary.

evidenceSets(): A reference to the collection of evidence sets.

### Author(s)

Russell Almond

### References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.

### See Also

Subclasses: [BNEngineMongo](#), [BNEngineNDB](#)

Constituent parts: [StudentRecordSet](#), [PnetWarehouse](#)

Setup Functions: [loadManifest](#), [setupDefaultSR](#), [configStats](#), [baselineHist](#),

Main Loop Functions: [mainLoop](#), [handleEvidence](#), [getRecordForUser](#), [logEvidence](#), [accumulateEvidence](#), [updateRecord](#), [updateSM](#), [updateStats](#), [updateHist](#), [announceStats](#),

### Examples

```
showClass("BNEngine")
```

---

BNEngineMongo

Creates a Bayes Net Engine attached to a Mongo database.

---

## Description

The [BNEngineMongo](#) is a [BNEngine](#) which is attached to a [MongoDB](#) database, which hold both the queue and the [StudentRecordSet](#).

## Usage

```
newBNEngineMongo(app = "default", warehouse, listenerSet = NULL, processN = Inf,
  statistics = list(),
  dburi = "mongodb://localhost", sslops = mongolite::ssl_options(),
  eadbname = "EARecords", admindbname = "Proc4", waittime = 0.25,
  profModel = character(), histNodes = character(),
  errorRestart = c("checkNoScore", "stopProcessing", "scoreAvailable"),
  srcol = "StudentRecords",
  mongoverbose = FALSE,
  srs = StudentRecordSet(app = app, warehouse = warehouse,
    db = MongoDB(srcol, eadbname, dburi, verbose = mongoverbose,
      options = sslops)),
  manifestCol = "Manifest", manifestDB = MongoDB(manifestCol,
    eadbname, dburi, verbose = mongoverbose, options = sslops),
  evidenceCol = "EvidenceSets", evidenceQueue = new("MongoQueue",
    app = app, messDB = MongoDB(evidenceCol, eadbname, dburi,
      verbose = mongoverbose, options = sslops), builder = Proc4::buildMessage),
  histcol = "histNodes", histNodesDB = MongoDB(histcol, eadbname,
    dburi, verbose = mongoverbose, options = sslops),
  statcol = "Statistics",
  statDB = MongoDB(statcol, eadbname, dburi, verbose = mongoverbose,
    options = sslops),
  admincol = "AuthorizedApps", adminDB = MongoDB(admincol,
    admindbname, dburi, verbose = mongoverbose, options = sslops),
  ...)
```

## Arguments

app	A character scalar giving the globally unique identifier for the application.
warehouse	A <a href="#">PnetWarehouse</a> which stores the default student model and evidence models. (It will also store the student models.
listenerSet	A <a href="#">ListenerSet</a> which contains the listeners for clients of the engine's messages.
statistics	Object of class list containing <a href="#">Statistic</a> objects to be run on every update cycle.
dburi	A character scalar giving the login information for the mongo database. See <a href="#">makeDBuri</a> .

sslops	Options for SSL connections to database. See <a href="#">ssl_options</a> .
eadbname	The name for the EA database.
admindbname	The name of the admin database used to check for shutdown requests.
processN	The number of records to process before stopping. The default value Inf runs the process until the active flag is cleared.
waittime	The amount of time (in seconds) to wait before checking again for new evidence sets when the evidence set queue is empty.
profModel	The name of the proficiency model (its ID in the warehouse manifest).
histNodes	A character vector giving the names of the nodes for which history will automatically be recorded.
errorRestart	A character scalar describing how to handle errors. The default, "checkNoScore" will continue scoring to try to find additional errors, but will not report statistics; the "scoreAvailable" option reports the scores based on the evidence sets which do not produce errors. The "stopProcessing" option immediately stops processing.
srcol	A character scalar giving the name of the database backing the student record set. Ignored if srs is specified.
mongoverbose	A flag. If true, extra debugging information from database calls is generated.
srs	A <a href="#">StudentRecordSet</a> object for storing the student records.
manifestCol	The name of the column containing the manifest data, ignored if manifestDB is supplied.
manifestDB	A <a href="#">JSONDB</a> the database where manifest information is cached.
evidenceCol	The name of the column containing the evidence sets, ignored if evidenceQueue is supplied.
evidenceQueue	A <a href="#">MessageQueue</a> where the evidence sets exist.
histcol	The name of the column into which history data should be stored, ignored if histNodesDB is supplied.
histNodesDB	A <a href="#">JSONDB</a> database where history information is stored.
statcol	The name of the column into which statistics should be stored, ignored if statDB is supplied.
statDB	A <a href="#">JSONDB</a> database where statistics are stored.
admincol	The name of the column in the administrative database where engine status information is stored, ignored if adminDB is supplied.
adminDB	A <a href="#">JSONDB</a> where status information about the engine is stored.
...	Extra arguments are ignored. This allows arguments for other engine versions to be set in the parameters and ignored.

## Details

This creates an uninitialized [BNEngine](#), specifically a [BNEngineMongo](#).

The app, warehouse, and listenerSet arguments need to be supplied, for most of the rest, the default arguments work.

In particular, most of the “db” arguments are built using the default arguments. The [makeDBuri](#) function provides a useful shorthand for calculating the dburi field.

**Value**

An object of calls [BNEngineMongo](#) which is capable of scoring student models.

**Note**

Much of this information comes from the “config.json” file, with the `dburi`, `eadbname`, `admindbname`, and `sslops` arguments come from the “EA.ini” file.

**Author(s)**

Russell Almond

**References**

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.

**See Also**

Classes: [BNEngine](#), [BNEngineNDB](#)

Constituent parts: [StudentRecordSet](#), [PnetWarehouse](#) [ListenerSet](#)

Setup Functions: [loadManifest](#), [setupDefaultSR](#), [configStats](#), [baselineHist](#),

Main Loop Functions: [mainLoop](#), [accumulateEvidence](#), [handleEvidence](#), [getRecordForUser](#), [logEvidence](#), [updateSM](#), [updateStats](#), [updateHist](#), [announceStats](#),

**Examples**

```
## Not run:
## Requires database setup, also PNetica
library(RNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EATest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman, session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)
ls <- ListenerSet(sender= paste("EAEngine[", basename(app), "]", ),
                 dbname="EARecords", dburi=makeDBuri(host="localhost"),
                 listeners=listeners,
```

```

        colname="Messages")

eng <- newBNEngineMongo(app=app,warehouse=Nethouse,
                        listenerSet=ls,
                        dburi=makeDBuri(host="localhost"),
                        dbname="EARecords",profModel="miniPP_CM",
                        histNodes="Physics")

## Standard initialization methods.
loadManifest(eng,netman)
eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

## End(Not run)

```

---

BNEngineMongo-class	Class "BNEngineMongo"
---------------------	-----------------------

---

## Description

A Bayes net engine hooked to a Mongo database.

## Extends

Class "[BNEngine](#)", directly.

All reference classes extend and inherit methods from "[envRefClass](#)".

## Activation

At the start of each iteration of the [mainLoop](#), it checks `eng$shouldHalt()` method. If this returns TRUE, then execution is immediately halted. When the queue is empty, it checks the `eng$stopWhenFinished()` method. If this returns true, then the main loop also terminates.

The BNEngineMongo this checks the "AuthorizedApps" collection in the database to see if the current app is active and the value of the field EAsignal. The `eng$activate()` method sets this field to "Running". If the field is set to "Halt", then `eng$shouldHalt()` will return TRUE and the [mainLoop](#) will stop after processing the current evidence set. If the field is set to "Halt" (actually, anything other than "Running"), then `eng$stopWhenFinished()` will return TRUE and the mainLoop will stop when the queue is empty.

The methods `eng$activate()` and `eng$deactivate()` set and clear the EAactive flag in the "AuthorizedApps" database.

## Fields

**app:** Object of class character giving an globally unique identifier for the application  
**srs:** Object of class [StudentRecordSet](#) of NULL giving the student record set for the application.  
**profModel:** Object of class character giving the name of the proficiency model (for the default student record) in the warehouse manifest.  
**listenerSet:** Object of class [ListenerSet](#) giving a set of listeners who will listen for new statistics.  
**statistics:** Object of class list containing [Statistic](#) objects to be run on every update cycle.  
**histNodes:** Object of class character giving the names of the nodes in the proficiency model whose history will be recorded.  
**warehouseObj:** Object of class [PnetWarehouse](#) which stores the Bayes nets, both evidence models and student models are stored here.  
**waittime:** Object of class numeric giving the time in seconds the main event loop should wait before checking again for messages.  
**processN:** Object of class numeric giving the number of times that the main loop should run before stopping. If Inf, then the main loop will run without stopping.  
**dburi:** Object of class character giving the URI for the mongo database.  
**dbname:** Object of class character giving the name of the database to be used.  
**manifestDB:** Object of class [MongoDB](#) giving the collection used to store the manifest. This object may not be initialized so it should be accessed through the class-based function `manifestdb()`.  
**evidenceDB:** Object of class [MongoDB](#) accessing the evidence set collection. This object may not be initialized so it should be accessed through the class-based function `evidenceSets()`.  
**statDB:** Object of class [MongoDB](#) giving the statistics to use. This object may not be initialized so it should be accessed through the class-based function `statdb()`.  
**histNodesDB:** Object of class [MongoDB](#) giving the history nodes. This object may not be initialized so it should be accessed through the class-based function `histNodesdb()`.  
**admindbname:** Object of class character giving name admin (mongo) database, used for various listeners and the `is.active()` method.  
**adminDB:** Object of class [MongoDB](#) giving the link to the admin database. This object may not be initialized so it should be accessed through the class-based function `admindb()`.

## Methods

**activate():** Sets the flag in the admin database to indicate that the process is running.  
**deactivate():** Clears the flag in the admin database to indicate that the process is no longer running.  
**shouldHalt():** This function checks the admin database to see whether or not the flag is set to cause the process to halt after processing the current record..  
**stopWhenFinished():** This function checks the admin database to see whether or not the flag is set to cause the process to stop when the event queue is empty.  
**statdb():** Returns the database containing the statistic objects.  
**studentRecords():** Returns the [StudentRecordSet](#) associated with this engine.

`fetchStats()`: Fetches the statistics marked in the database configuration.

`initialize(app, warehouse, listeners, username, password, host, port, dbname, P4dbname, profModel, waitt`  
initializes the class. Note that some initialization is done in the various `XXXdb()` functions, so these should be called instead of directly accessing the fields.

`manifestdb()`: Returns the [MongoDB-class](#) handle to the manifest information collection.

`admindb()`: Returns the [MongoDB-class](#) handle to the “AuthorizedApps” collection.

`histNodesdb()`: Returns the [MongoDB-class](#) handle to the hist nodes collection.

`saveManifest(manifest)`: Saves the current [PnetWarehouse](#) manifest to the `manifestdb()` collection

`fetchManifest()`: Retrieves the saved manifest from the `manifestdb()` collection.

`fetchNextEvidence()`: Retrieves the next [EvidenceSet](#) from the `evidenceSets()` collection.  
Returns NULL if there are not unprocessed evidence sets.

`saveStats(statmat)`: Saves the update statistic definitions to the `statdb()` collection.

`setHistNodes(nodenames)`: Saves the history nodes to the `histNodesdb()` collection.

`isActive()`: Checks to see if the active flag is set.

`setError(mess, e)`: Added an error message to an evidence set.

`evidenceSets()`: Returns a [MongoDB-class](#) handle to the collection/queue of evidence sets.

`getHistNodes()`: Fetches the history nodes from the `histNodesdb()` collection.

`show()`: Provides a printed representation of the engine.

The following methods are inherited (from the corresponding class): `evidenceSets ("BNEngine")`, `getHistNodes ("BNEngine")`, `stats ("BNEngine")`, `setProcessed ("BNEngine")`, `setManifest ("BNEngine")`, `activate ("BNEngine")`, `isActive ("BNEngine")`, `saveManifest ("BNEngine")`, `studentRecords ("BNEngine")`, `saveStats ("BNEngine")`, `fetchNextEvidence ("BNEngine")`, `warehouse ("BNEngine")`, `show ("BNEngine")`, `setHistNodes ("BNEngine")`, `setError ("BNEngine")`, `fetchManifest ("BNEngine")`, `fetchStats ("BNEngine")`

## Note

The database connections are not created right away, so it is important to use the class-based functions, `manifestdb()`, `statdb()`, `evidenceSets()`, `histNodesdb()`, `studentRecords()`, and `admindb()` rather than accessing the fields directly.

## Author(s)

Russell Almond

## References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.



**See Also**

Classes: [BNEngine](#), [BNEngineNDB](#)

Constituent parts: [StudentRecordSet](#), [PnetWarehouse](#)

Setup Functions: [loadManifest](#), [setupDefaultSR](#), [configStats](#), [baselineHist](#),

Main Loop Functions: [mainLoop](#), [accumulateEvidence](#), [handleEvidence](#), [getRecordForUser](#), [logEvidence](#), [updateSM](#), [updateStats](#), [updateHist](#), [announceStats](#),

**Examples**

```
showClass("BNEngineMongo")
```

---

BNEngineNDB

*Creates a Bayes net engine not attached to a database.*

---

**Description**

The [BNEngineNDB](#) is a [BNEngine](#) which is not attached to the database. In particular, it cannot store student records, so it cannot maintain state between scoring sessions without external help.

**Usage**

```
newBNEngineNDB(app = "default", warehouse, listenerSet = NULL,
  manifest = data.frame(), processN = Inf, waittime = 0.25,
  profModel = character(), statmat = data.frame(),
  evidenceQueue = new("ListQueue",app, list()),
  activeTest = "EAActive",
  errorRestart=c("checkNoScore","stopProcessing","scoreAvailable"),
  srs =StudentRecordSet(app=app,warehouse=warehouse, db=MongoDB(noMongo=TRUE)),
  ...)
```

**Arguments**

app	A character scalar giving the globally unique identifier for the application.
warehouse	A <a href="#">PnetWarehouse</a> which stores the default student model and evidence models. (It will also store the student models.
listenerSet	A <a href="#">ListenerSet</a> which contains the listeners for clients of the engine's messages.
manifest	A data frame providing a manifest for the <a href="#">PnetWarehouse</a> .
processN	The number of records to process before stopping. The default value Inf runs the process until the active flag is cleared.
waittime	The amount of time (in seconds) to wait before checking again for new evidence sets when the evidence set queue is empty.
profModel	The name of the proficiency model (its ID in the warehouse manifest).

statmat	A data.frame describing the statistics. See <a href="#">configStats</a> .
evidenceQueue	A object of class <a href="#">MessageQueue-class</a> containing evidence sets to be processed.
activeTest	The pathname for the file whose existence will be used to determine when the engine should shut down.
errorRestart	A character scalar describing how to handle errors. The default, "checkNoScore" will continue scoring to try to find additional errors, but will not report statistics; the "scoreAvailable" option reports the scores based on the evidence sets which do not produce errors. The "stopProcessing" option immediately stops processing.
srs	A <a href="#">StudentRecordSet</a> object used to manage student records.
...	Extra arguments are ignored. This allows arguments for other engine versions to be set in the parameters and ignored.

### Details

This creates an uninitialized [BNEngine](#), specifically a [BNEngineNDB](#).

### Value

An object of calls [BNEngineNDB](#) which is capable of scoring student models.

### Author(s)

Russell Almond

### References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.

### See Also

Classes: [BNEngine](#), [BNEngineMongo](#)

Constituent parts: [StudentRecordSet](#), [PnetWarehouse](#) [ListenerSet](#)

Setup Functions: [loadManifest](#), [setupDefaultSR](#), [configStats](#), [baselineHist](#),

Main Loop Functions: [mainLoop](#), [accumulateEvidence](#), [handleEvidence](#), [getRecordForUser](#), [logEvidence](#), [updateSM](#), [updateStats](#), [updateHist](#), [announceStats](#),

### Examples

```
## Requires database setup, also PNetica
library(RNetica) ## Must load to setup Netica DLL
appid <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
```

```

net.dir <- file.path(library(help="PNetica")$path,"testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman,session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[",appid,"]"),
                 listeners=listeners)

eng <- newBNEngineNDB(app=appid,warehouse=Nethouse,
                    listenerSet=ls,manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    statmat=stattab,
                    activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng,netman)
eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

```

---

BNEngineNDB-class	Class "BNEngineNDB"
-------------------	---------------------

---

## Description

A [BNEngine](#) instance which is *not* connected to a database.

## Extends

Class "[BNEngine](#)", directly.

All reference classes extend and inherit methods from "[envRefClass](#)".

## Methods

**evidence** signature(x = "BNEngineNDB"): Returns list of [EvidenceSets](#) in the queue.

**evidence** signature(x = "BNEngineNDB", value="list"): Sets the list of [EvidenceSets](#) in the queue.

### Activation

At the start of each iteration of the `mainLoop`, it checks `eng$shouldHalt()` method. If this returns TRUE, then execution is immediately halted. When the queue is empty, it checks the `eng$stopWhenFinished()` method. If this returns true, then the main loop also terminates.

In the no database version, the process communicates with the rest of the system by checking the file referenced in the `activeTest` field. The `eng$activate()` creates this file with the extension `‘.running’`. Renaming the file to have the extension `.finish` will cause `eng$stopWhenFinished()` to return true, that is the `mainLoop` will finish when the queue is empty. Renaming the file to have the extension `.halt` will cause `eng$shouldHalt()` to return true, and `mainLoop` will stop when it finishes processing the current event.

### Fields

`app`: Object of class character giving an globally unique identifier for the application

`srs`: Object of class `StudentRecordSet` of NULL giving the student record set for the application.

`profModel`: Object of class character giving the name of the proficiency model (for the default student record) in the warehouse manifest.

`listenerSet`: Object of class `ListenerSet` giving a set of listeners who will listen for new statistics.

`statistics`: Object of class list containing `Statistic` objects to be run on every update cycle.

`histNodes`: Object of class character giving the names of the nodes in the proficiency model whose history will be recorded.

`warehouseObj`: Object of class `PnetWarehouse` which stores the Bayes nets, both evidence models and student models are stored here.

`waittime`: Object of class numeric giving the time in seconds the main event loop should wait before checking again for messages.

`processN`: Object of class numeric giving the number of times that the main loop should run before stopping. If Inf, then the main loop will run without stopping.

`manifest`: Object of class `data.frame` which provides the manifest for the `PnetWarehouse`

`histnodes`: Object of class character which gives the names of the nodes for whom history will be recorded.

`evidenceQueue`: A list of `EvidenceSet` events to be processed.

`statmat`: Object of class `data.frame` which gives the descriptions of the `Statistic` objects to be used with the net.

`activeTest`: A pathname to the file whose existence will be checked to determine whether or not the engine should be considered active.

### Class-Based Methods

`activate()`: Creates the `activeTest` to indicate that the process is running.

`deactivate()`: Deletes the `activeTest` file to indicate that the process is no longer running.

`shouldHalt()`: This function checks the `activeTest` file to see whether or not the flag is set to cause the process to halt after processing the current record..

`stopWhenFinished()`: This function checks the `activeTest` file database to see whether or not the flag is set to cause the process to stop when the event queue is empty.

`studentRecords()`: Returns the [StudentRecordSet](#) associated with this engine.

`fetchStats()`: Fetches the statistics marked in the database configuration.

`fetchStats()`: Fetches the statistics or information in the `statmat` field.

`initialize(app, warehouse, listeners, profModel, waittime, statistics, histNodes, evidenceQueue, process)`  
Initializes this class

`saveManifest(manifest)`: This sets the internal manifest field.

`fetchManifest()`: This returns the internal manifest field.

`fetchNextEvidence()`: This returns the first evidence set from the `evidenceQueue` field, and removes that element from the queue.

`saveStats(statmat)`: This saves the statistic table to the internal field.

`evidenceSets()`: This returns NULL

`show()`: This produces a printable summary.

The following methods are inherited (from the corresponding class): `evidenceSets ("BNEngine")`, `stats ("BNEngine")`, `setProcessed ("BNEngine")`, `setManifest ("BNEngine")`, `activate ("BNEngine")`, `isActivated ("BNEngine")`, `saveManifest ("BNEngine")`, `setHistNodes ("BNEngine")`, `studentRecords ("BNEngine")`, `saveStats ("BNEngine")`, `fetchNextEvidence ("BNEngine")`, `setError ("BNEngine")`, `getHistNodes ("BNEngine")`, `warehouse ("BNEngine")`, `show ("BNEngine")`, `fetchManifest ("BNEngine")`, `fetchStats ("BNEngine")`

## Note

The assumption of this engine is that the serialized student model will be passed in along with the evidence and will be returned along with the updated statistics.

## Author(s)

Russell Almond

## References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.

## See Also

Classes: [BNEngine](#), [BNEngineMongo](#)

Constituent parts: [StudentRecordSet](#), [PnetWarehouse](#)

Setup Functions: [loadManifest](#), [setupDefaultSR](#), [configStats](#), [baselineHist](#),

Main Loop Functions: [mainLoop](#), [accumulateEvidence](#), [handleEvidence](#), [getRecordForUser](#), [logEvidence](#), [updateSM](#), [updateStats](#), [updateHist](#), [announceStats](#),

## Examples

```
showClass("BNEngineNDB")
```

---

configStats

*Configures the Statistic Objects for the BNEngine*


---

## Description

As part of the scoring cycle, the [BNEngine](#) calculates the values of certain statistics of the student model. This function sets up those statistics.

## Usage

```
configStats(eng, statmat = data.frame())
```

## Arguments

eng	The <a href="#">BNEngine</a> to be configured.
statmat	A data frame containing the statistic descriptions, see details.

## Details

A [Statistic](#) is a functional that is applied to the student model ([sm](#)) of a [StudentRecord](#). At the end of the evidence processing cycle, the function [updateStats](#) is called to calculate new values for the specified statistics.

The statmat argument should be a `data.frame` with three columns (all of mode character):

**Name** This column gives an identifier for the statistic used in the output message.

**Fun** This column gives the name of a function (see [Statistic](#) for a list of possible values) which calculates the statistic value.

**Node** This gives the name of a node in the competency model which is the focus of the statistic.

If the statmat argument is not supplied, then a default value based on the engine type is used. For the [BNEngineMongo](#) this data frame is taken from a table in the database. For the [BNEngineNDB](#) the default statmat is stored in a field in the engine.

## Value

The modified engine argument is returned.

## Author(s)

Russell Almond

## References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.

**See Also**

Classes: [BNEngine](#), [Statistic updateStats](#), [announceStats](#)

**Examples**

```
## Requires PNetica
library(RNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman, session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[", app, "]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app, warehouse=Nethouse,
                    listenerSet=ls, manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    activeTest="EAActive.txt")

## Standard initialization methods.
configStats(eng, stattab)
stats <- eng$stats()
stopifnot(all(sapply(stats, StatName)==stattab$Name),
          all(sapply(stats, function(s) s@fun)==stattab$Fun),
          all(sapply(stats, function(s) s@node)==stattab$Node))
```

---

doBuild

---

*Build or rebuild the Bayes nets for a scoring engine.*


---

**Description**

This function downloads the table specifications from the internet and rebuilds the Bayesian networks for a particular scoring application. It takes the information from the “tables” subdirectory (under config.dir) and builds the nets in the “nets” subdirectory.

## Usage

```
doBuild(sess, EA.tables, config.dir, override = FALSE)
```

## Arguments

<code>sess</code>	A <a href="#">NeticaSession</a> object used to build the Bayes nets.
<code>EA.tables</code>	A list containing configuration details. See the ‘Configuration’ section below.
<code>config.dir</code>	The pathname of the directory that contains the tables and the nets subdirectories.
<code>override</code>	A logical flag. If true, the code will ignore locks and rebuild the nets anyway.

## Details

This program applies the scripts from the [Peanut-package](#) to rebuild the nets. It assumes the existence of five tables which describe the scoring model:

**Nets.csv** Manifest of all networks. See [Warehouse](#) and [BNWarehouse](#).

**Nodes.csv** Manifest of all nodes in all networks. See [Warehouse](#) and [NNWarehouse](#).

**Omega.csv** Description of the competency model. See [Omega2Pnet](#).

**Q.csv** Description of the evidence model. See [Qmat2Pnet](#).

**Statistics.csv** A description of the statistics being used. See [configStats](#).

These are expected to reside in the “tables” subdirectory of the `config.dir` and have the names described above (although these details can be overridden by the configuration, see ‘Configuration’ below).

The following steps are followed in the rebuilding.

1. The tables (CSV files) are downloaded from internet sources (see Downloading Tables below) into the “tables” directory.
2. The tables are loaded into R and a [PnetWarehouse](#) and [PnodeWarehouse](#) are built for the models.
3. The [Omega2Pnet](#) script is run to build the proficiency model.
4. The [Qmat2Pnet](#) script is run to build the evidence models.
5. The nets are written out to the “nets” subdirectory of `config.dir`. The net manifest is written to the subdirectory in the file “NetManifest.csv” and the statistic list is written in the file “StatisticList.csv”. These values can be overridden with the configuration.

## Value

This function is invoked for its side effects, which are stored in the “nets” subdirectory of the `config.dir` directory.



## Configuration

There are a large number of parameters which can be configured. These are passed in through the `EA.tables` argument, which is a list of parameters. The intention is that this can be read in from a JSON file (using `fromJSON`). In the current implementation, the `EA.tables` parameter set is a sub-object of the larger `EA.config` parameter set.

The following fields are available:

**netdir** This is the name of the subdirectory of `config.dir` in which the constructed nets will be saved. Default value is “nets”.

**tabdir** This is the name of the subdirectory of `config.dir` in which the network specification tables are found. The default value is “tables”.

**TableID** This is a parameter passed to the download script to identify the place from which the tables should be downloaded. The intent is for this to be a Google Sheets ID such as, “16LcEuCspZjiBoZ3-Y1R3jxi1COXmh9vuTa9GwH1A\_7Q”.

**downloadScript** This is the name of the script which is run to download the tables. The default value is “download.sh”. See the Downloading Tables section below.

**NetsName** This is the name (less the .csv extension) of the file containing the network manifest. The default value is “Nets”.

**NodesName** This is the name (less the .csv extension) of the file containing the node manifest. The default value is “Nodes”.

**OmegaName** This is the name (less the .csv extension) of the file containing the Omega matrix (Proficiency model specification). The default value is “Omega”.

**QName** This is the name (less the .csv extension) of the file containing the Q matrix (Evidence model specification). The default value is “Q”.

**StatName** This is the name (less the .csv extension) of the file containing the statistic list. The default value is “Statistics”.

**profModel** This is the name of the proficiency model. If no value is supplied, the value is inferred from the first non-missing value of the “Hub” column in the network manifest.

**manifestFile** The name of the file in which the list of available networks is output. The default value is “PPManifest.csv”.

**statFile** The name of the file (in the “nets” directory) in which statistics list is output. The default value is “StatisticList.csv”.

## Downloading Tables

The complete specification is given in five different tables. This can be represented a five different sheets (pages) on a typical spreadsheet program. In various projects it has been useful to create a Google Sheets document with these five pages which can be accessed by the project team. Thus, one team member can make changes and the other download it. (This would probably work with a different document collaboration system, but this has not been tested.)

Google Sheets are identified by a long string in the URL. This is the “TableID” field in the `EA.tables` configuration list. (In theory, this could be replaced by an appropriate identifier if something other than Google Sheets was used.) The script “download.sh” (the name can be overridden in the configuration) is called using `system2` with the “table” directory path and the “TableID” as arguments. It then downloads the tables.

The bash implementation for use with Google sheets is to first define a BASEURL variable: `BASEURL="https://docs.google."` and then to call `curl` to download the sheets, e.g., `curl "${BASEURL}/gviz/tq?tx=out:csv&sheet={Nets}"` `>Nets.csv`.

In theory, the sheets could be downloaded directly from the URLs using `read.csv`, however, there were issues with that solution. This solution also allows the `download.sh` script to take care of any authentication which needs to be done (as the Google APIs here are a moving target).

## Locking

It is probably a bad idea to rebuild the nets which a different incarnation is using the net directory to score. It is almost certainly a bad idea for two different programs to rebuild the nets in the same directory at the same time.

To prevent such clashes, the `doRunrun` function adds a file with the extension `.lock` to the directory when it is scoring. The `doBuild` function adds the file `netbuilder.lock` while it is rebuilding the nets.

If when `doBuild` starts, if a `.lock` file is found in the "nets" directory, it issues a warning, and unless the `override` parameter is set to `TRUE` it stops. Use the `override` only with extreme caution.

## Logging

Logging is done through the `futile.logger{flog.logger}` mechanism. This allows logs to be save to a file.

## Author(s)

Russell Almond

## References

Almond, R. G. (2010). 'I can name that Bayesian network in two matrixes.' *International Journal of Approximate Reasoning*. **51**, 167-178.

Almond, R. G. (presented 2017, August). Tabular views of Bayesian networks. In John-Mark Agosta and Tomas Singlair (Chair), *Bayesian Modeling Application Workshop 2017*. Symposium conducted at the meeting of Association for Uncertainty in Artificial Intelligence, Sydney, Australia. (International) Retrieved from <http://bmaw2017.azurewebsites.net/>

## See Also

`doRunrun`, `configStats`

`Warehouse`, `BNWarehouse`, `NNWarehouse`, `Omega2Pnet`, `Qmat2Pnet`,

## Examples

```
## This example is in:
file.path(help(package="EABN")$path,"conf","EABuild.R")
## Not run:
## Set up config.dir, logpath and NeticaLicenseKey
source("/usr/local/share/Proc4/EAini.R")
```

```
EA.config <- jsonlite::fromJSON(file.path(config.dir,"config.json"),FALSE)

EA.tables <- EA.config$Tables
EA.tables$netdir <- EA.config$netdir

sess <- RNetica::NeticaSession(LicenseKey=NeticalLicenseKey)
startSession(sess)

futile.logger::flog.appender(appender.file(file.path(logpath,
                                                    sub("<app>", "builder",EA.config$logname))))
futile.logger::flog.threshold(EA.config$loglevel)

doBuild(sess,EA.tables,config.dir)

## End(Not run)
```

---

doRunrun	<i>This runs the Evidence Accumulation Bayes net engine to scor or rescore an assessment.</i>
----------	---

---

## Description

This is a system to run the Bayes net scoring engine, taking most of the details from a configuration file. It creates the [BNEngine](#) instance then and then runs it in either scoring or rescoring mode. Configuration information is taken from the EA.config and EAeng.local parameters.

## Usage

```
doRunrun(appid, sess, EA.config, EAeng.local, config.dir,
outdir=config.dir, override = FALSE, logfile="", noprep=FALSE)
```

## Arguments

appid	A character string giving the global unique identifier for the application being run. This is normally formatted like a URL, and basename(app) is used as a short name.
sess	A <a href="#">NeticaSession</a> object to use for the Netica link.
EA.config	A named list containing the configuration details. See the ‘Configuration’ section below.
EAeng.local	A named list containing additional parameters for the engine constructor. The intention that these are local configuration parameters (e.g., database names and passwords) as opposed to more global information. Note this must have an element named “dburi” which gives the URI for the database, or which should be blank if the no database engine is to be used.
config.dir	The pathname of the directory that contains the the nets subdirectories.
outdir	The pathname of the directory to which output files will be written.
override	A logical flag. If true, the code will ignore locks and restart the run anyway.

logfile	Name for the file in which to do logging.
noprep	Logical flag. If true, then the database and listener preparation steps will be skipped. This is for forcing a continuation without resetting the configuration.

## Details

The goal is to start a run for scoring (evidence accumulation step) an assessment using the [BNEngine](#) class. This function takes care of many of the configuration details and preparatory steps and then calls [mainLoop](#) to do the major work. In particular, the steps done by this system are as follows:

1. Configure the listeners.
2. Configure the engine, including loading manifest and scoring list.
3. Clean old scores from the database (optional depending on configuration.)
4. Remove selected evidence sets from the collection. Import new evidence sets into the database and mark selected evidence as unprocessed.
5. Launch engine using [mainLoop](#).
6. Build and register the statistics and history file.

Note that this will run in either rerun mode, where it will score an selection of existing records and stop, or in server mode where it will continue waiting for new messages until it gets a shut down signal.

## Value

This returns the engine invisibly, in case the calling program wants to do something with it.

## Configuration

There are a large number of parameters which can be configured. These are passed in through the `EA.config` argument, which is a list of parameters. The intention is that this can be read in from a JSON file (using [fromJSON](#)). The `RunEABN.R` script loads these from a file called `config.json`. A sample of this file is available on github <https://github.com/ralmond/PP-EA>.

The following fields are available:

**ConfigName** An identifier for the configuration. Default value "PP-main". Documentation only, not used by doRunrun.

**Branch** The branch name for the git branch for this configuraiton. Default value "PP-main". Documentation only, not used by doRunrun.

**Version** A version number for the configuration. Documentation only, not used by doRunrun.

**Date** A edit date for the configuration. Documentation only, not used by doRunrun.

**appStem** A list of app stems that will be affected. Sample value ["P4Test"].

**rebuildNets** A logical flag, should the nets be rebuilt. Example value true.

**logLevel** This controls the [flog.threshold](#). Default value "INFO". Note that doRunrun does not set the log value, that should be done in the calling script.

**logname** This is the name of the file to which logs should be sent. Example value "EA\_<app>0.log". Note that doRunrun does not set the log file, that should be done in the calling script.

- Tables** This is a whole object describing the EA. tables field see [doBuild](#).
- sender** The sender field on output messages. Example value "EA\_<app>".
- lscolname** The name of the column to which the listener set should log messages. Example value "Messages".
- listeners** This is a list of listener descriptions. See the section ‘Listner Configuration’ below.
- SRreset** Logical value, should the student records be reset before running. Example value true.
- listenerReset** Which listeners should be reset before running. This should be a character scalar or vector. The values should be names of listeners. The special value “Self” refers to the ListenerSet object, and the special value “ALL” resets all listeners. See [resetListeners](#). Example value "ALL".
- netdir** The name of the subdirectory of config.dir which contains the nets. Default value "nets".
- EAEngine** A complex object describing engine parameters. See the section ‘Engine Configuration’ below.
- filter** A complex object describing how to prefilter the database. See the section ‘Database Filters’ below.
- extensions** This should be a list of paths (relative to config.dir) containing additional R code to load. This is not used by doRunrun, but is supplied for use in scripts that might use doRunrun.
- limitNN** An integer: how many events should be processed. Two special string values are also accepted. “ALL” will process all records currently in the database and stop. “Inf” will cause the process to run in server mode until it is shut down.
- listenerExports** Information about data tables which should be exported at the end of the run. See [generateListenerExports](#).
- A number of these values do “<app>” substitution, that is they will substitute the string “<app>” for the short name of the application.

## Listener Configuration

The listeners consist of a [ListenerSet](#) and a collection of [Listener](#) objects. The listener objects are made by using the information from the “listeners” element of the EA.config argument. This should be a list of specifications (each specification itself is a list). These are passed to [buildListener](#), which provides some examples. The “listenerExports” part of the configuration is used to call [generateListenerExports](#) when the engine stops.

The listener set is controlled by the EAeng.local\$dburi value and the “lscolname” field. If dburi is a name of a database, then the [ListenerSet](#) is logged into the “lscolname” collection. If dburi is null or an empty string, then the listener set will not do logging.

## Engine Configuration

The type of engine used is controlled by the EAeng.local\$dburi value. If this is a URI, then the [BNEngineMongo](#) class is used. If it is null or the empty string, then the [BNEngineNDB](#) class is used instead.

The arguments to the appropriate constructor are found between the EAeng.local and EA.config\$EAEngine collections. The intent is for the former to include details (e.g., database user names and passwords)

which are local to the server on which EABN is running, and for `EA.config$EAEngine` to include more public details which are local to a particular run.

See [BNEngineMongo](#) or [BNEngineNDB](#) for the expected fields. Note that the “processN” field is taken care of separately after the database operations (next section).

## Database Filtering

The `EA.config$filter` field controls the database filtering process. There are four steps:

**Remove** old records from the database.

**Import** new records into the database.

**Purge** unused records from the database.

**Reprocess** Reset the processed flag to ensure records get reprocessed.

These are controlled by the following elements in the `EA.config$filter` list:

**doRemove** Logical, should records be removed before import.

**remove** Filter to use for removal. The value `{}` will remove all records for the given app.

**importFile** A list of filenames (in the `config.dir`) which contain evidence sets to be imported before scoring.

**doPurge** Logical, should records be removed after import.

**purge** Filter for the purging (after import removal). Leaving this empty will probably not be satisfactory.

**doReprocess** Logical, should existing records have the processed flag cleared? Typically TRUE for rerun mode and FALSE for server mode.

**reprocess** Filter for the selected records to be marked for reprocessing. The value `{}` will mark all records (for this app) for reprocessing.

## Locking

It is probably a bad idea to rebuild the nets which a different incarnation is using the net directory to score. It is almost certainly a bad idea for two different programs to rebuild the nets in the same directory at the same time.

To prevent such clashes, the [doRunrun](#) function adds a file with the extension `.lock` to the directory when it is scoring. The `doBuild` function adds the file `netbuilder.lock` while it is rebuilding the nets.

If when `doBuild` starts, if a `.lock` file is found in the “nets” directory, it issues an warning, and unless the `override` parameter is set to TRUE it stops. Use the `override` only with extreme caution.

The [BNEngineMongo](#) version also checks the database for a running flag. If it is found, then again the engine will not start unless the `override` flag is true.

## Data Files

If the value of `EA.config$statListener` is not null, then the final statistic values for all users are put into a table which is exported (to the file `EA.config$statfile`).

If the value of `EA.config$histListener` is not null, then the history of all statistic values for all users are put into a table which is exported (to the file `EA.config$histfile`).

Both the `statfile` and `histfile` are registered using the [ListenerSet\\$registerOutput](#) method.

## Logging

Logging is done through the `futile.logger{flog.logger}` mechanism. This allows logs to be save to a file.

The “logLevel” and “logname” fields are put in the configuration specification to assist scripts in configuring the logging system.

Both the log file is registered using the `ListenerSet$registerOutput` method.

## Note

This function is meant to be called by the `RunEABN.R` script found in the config directory. (`file.path(help(package="EABN")$path, "conf", "RunEABN.R")`)

The shell script `EABN` found in the same directory will run this script.

## Author(s)

Russell Almond

## References

The Bobs (1983) Psychokiller. *My I'm Large*. Rhino Records. <https://www.youtube.com/watch?v=-Gu4PKnCLDg>. (Reference is about 2:30 minutes into song.)

## See Also

`BNEngine`, `mainLoop`, `doBuild`  
`resetProcessedMessages`, `cleanMessageQueue`, `importMessages`  
`ListenerSet`, `buildListenerSet`, `generateListenerExports`, `resetListeners`

## Examples

```
## This example is in:
file.path(help(package="EABN")$path, "conf", "RunEABN.R")
## Not run:
library(R.utils)
library(EABN)
library(PNetica)

appStem <- cmdArg("app", NULL)
if (FALSE) {
  appStem <- "userControl"
}

source("/usr/local/share/Proc4/EAini.R")

EA.config <- jsonlite::fromJSON(file.path(config.dir, "config.json"), FALSE)

app <- as.character(Proc4.config$appStems[appStem])
if (length(app)==0L || any(app=="NULL")) {
  stop("Could not find app for ", appStem)
}
```

```
## Start Netica
sess <- NeticaSession(LicenseKey=NeticalLicenseKey)
startSession(sess)

logfile <- (file.path(logpath, sub("<app>",appStem,EA.config$logname)))
if (interactive()) {
  futile.logger::flog.appender(appender.tee(logfile))
} else {
  futile.logger::flog.appender(appender.file(logfile))
}
futile.logger::flog.threshold(EA.config$logLevel)

eng <- doRunrun(app,sess,EA.config,EAeng.local,config.dir,outdir,
               logfile=logfile)

## End(Not run)
```

EvidenceSet

*Creates an Evidence Set Message***Description**

An [EvidenceSet](#) is a [P4Message](#) which contains observable variables for the Bayes net engine. It provides the observavbles associated with a single scoring context.

**Usage**

```
EvidenceSet(uid, context, timestamp = Sys.time(), obs = list(), app =
"default", mess = "Accumulate", sender = "EI", processed = FALSE)
```

**Arguments**

uid	A character scalar giving unique identifier for the student/player.
context	A character scalar giving a unique identifier for the scoring context (often game level or task).
timestamp	The time at which the evidence was recorded (POSIXt format).
obs	A named list giving the observable variables. The names and legal values correspond to the context and app values.
app	A character scalar giving the globally unique identifier of the application.
mess	A character scalar giving the message associated with the observables. (Part of the Proc 4 proctol).
sender	A character scalar giving the identity of the process which created the message. This will usually be an evidence identification process.
processed	A flag that is set when the evidence set has been processed.



**Details**

Aside from the [seqno](#) field, this is pretty much a generic [P4Message](#). The data of the P4Message is the [observables](#) value for the [EvidenceSet](#).

**Value**

An object of class [EvidenceSet](#).

**Author(s)**

Russell Almond

**See Also**

Class: [EvidenceSet](#) Methods: [observables](#), [seqno](#), [parseEvidence](#)

Using classes: [StudentRecord](#)

**Examples**

```
e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
  obs=list("CompensatoryObs"="Right"))

e2 <- EvidenceSet(uid="S1",app="Test",context="PPdurAttEM",
  obs=list("Attempts"=2,"Duration"=38.3))
```

---

EvidenceSet-class	<i>Class "EvidenceSet"</i>
-------------------	----------------------------

---

**Description**

An [EvidenceSet](#) is a collection of observables that comes from a particular context (scoring window, task). It also has information about where it appears in the sequence of evidence that is recorded about a student. It is an extension of the [P4Message](#) class.

**Objects from the Class**

Objects can be created calls to the function [EvidenceSet](#)(uid, context, timestamp, obs, app, mess, sender).

**Slots**

**seqno:** Object of class "integer" which contains the order in which this object was processed.  
**\_id:** Object of class "character" which contains the database ID.  
**app:** Object of class "character" which gives a guid for the application.  
**uid:** Object of class "character" which gives an id for the student.  
**context:** Object of class "character" which gives an id for the scoring context.  
**sender:** Object of class "character" which gives an ID for the source of the evidence.  
**mess:** Object of class "character" which gives a message about what is contained in the message.  
**timestamp:** Object of class "POSIXt" which tells when the evidence was collected.  
**processed:** Object of class "logical" which is a flag to tell of the evidence has been incorporated into the [StudentRecord](#).  
**pError:** Object of class "ANY" which contains processing error.  
**data:** Named list which contains the evidence.

**Extends**

Class "[P4Message](#)", directly.

**Methods**

**as.jlist** signature(obj = "EvidenceSet", ml = "list"): This is a helper function used in serialization. See [as.json](#).  
**observables** signature(x = "EvidenceSet"): returns a named list of observables (the data) field.  
**seqno** signature(x = "EvidenceSet"): returns the sequence number.  
**seqno<-** signature(x = "EvidenceSet"): sets the sequence number.  
**show** signature(object = "EvidenceSet"): prints a summary of the evidence set.  
**toString** signature(x = "EvidenceSet"): provides a summary string for the evidence set.

**Author(s)**

Russell Almond

**See Also**

[StudentRecord](#), [accumulateEvidence](#), [handleEvidence](#), [logEvidence](#),  
[parseEvidence](#), [seqno](#), [observables](#)

**Examples**

```
showClass("EvidenceSet")
```

---

`fetchSM`*Fetches student model from database or JSON*

---

### Description

The function `fetchSM` retrieves the student model from a [PnetWarehouse](#) or if not there, attempts to recreate it from a serialized version. The function `unpackSM` does this unpacking.

### Usage

```
fetchSM(sr, warehouse)
unpackSM(sr, warehouse)
```

### Arguments

<code>sr</code>	An object of class <a href="#">StudentRecord</a> whose student model we wish to retrieve.
<code>warehouse</code>	A <a href="#">PnetWarehouse</a> which stores the student models.

### Details

The [StudentRecord](#) object has two fields related to student models: `sm` and `smser`. The former contains the actual student model or `NULL` if it has not yet been initialized or restored from the database. The latter contains a character string which contains a serialized version of the student model. In particular, it is this serialized student model which is stored in the database, not the actual student model.

The function `fetchSM` is used to set the `sm` field. It checks the following places in order:

1. It looks in the warehouse for a student net for the given `uid` for the record.
2. It calls `unpackSM` to unpack the serialized record.

The function `unpackSM` is wrapper for the function [WarehouseUnpack](#).

### Value

The function `fetchSM` returns the modified [StudentRecord](#).

The function `unpackSM` returns the student model (a [Pnet](#)).

### Author(s)

Russell Almond

### See Also

[StudentRecord](#)  
[PnetWarehouse](#), [WarehouseUnpack](#)

## Examples

```

library(PNetica)

##Start with manifest
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
config.dir <- file.path(library(help="Peanut")$path, "auxdata")
netman1 <- read.csv(file.path(config.dir,"Mini-PP-Nets.csv"),
  row.names=1, stringsAsFactors=FALSE)
net.dir <- file.path(library(help="PNetica")$path, "testnets")
Nethouse <- PNetica::BNWarehouse(manifest=netman1,session=sess,key="Name",
  address=net.dir)

dsr <- StudentRecord("*DEFAULT*",app="ecd://epls.coe.fsu.edu/P4Test",
  context="*Baseline*")
sm(dsr) <- WarehouseSupply(Nethouse,"miniPP_CM")
PnetCompile(sm(dsr))

## dsr <- updateStats(eng,dsr)
statmat <- read.csv(file.path(config.dir,"Mini-PP-Statistics.csv"),
  stringsAsFactors=FALSE)
rownames(statmat) <- statmat$Name
statlist <- sapply(statmat$Name,function (st)
  Statistic(statmat[st,"Fun"],statmat[st,"Node"],st))
names(statlist) <- statmat$Name
dsr@stats <- lapply(statlist,
  function (stat) calcStat(stat,sm(dsr)))
names(dsr@stats) <- names(statlist)

## dsr <- baselineHist(eng,dsr)
dsr@hist <- lapply(c("Physics"),
  function (nd)
    EABN::uphist(sm(dsr),nd,NULL,"*Baseline*"))
names(dsr@hist) <- "Physics"

pnodenames <- names(PnetPnodes(sm(dsr)))

## Serialization and unserialization
dsr.ser <- as.json(dsr)

dsr1 <- parseStudentRecord(jsonlite::fromJSON(dsr.ser))
stopifnot(is.null(sm(dsr1)))
## at this point, SM has not yet been restored.

## It is there in the serial field
net1 <- unpackSM(dsr1,Nethouse)
stopifnot(all.equal(pnodenames,names(PnetPnodes(net1))))

```

```

dsr1 <- fetchSM(dsr1,Nethouse)
stopifnot(all.equal(pnodenames,names(PnetPnodes(sm(dsr1)))))

## Try this again, but first delete net from warehouse,
## So we are sure we are building it from serialized version.
WarehouseFree(Nethouse,PnetName(sm(dsr)))

dsr1 <- parseStudentRecord(jsonlite::fromJSON(dsr.ser))
stopifnot(is.null(sm(dsr1)))
## at this point, SM has not yet been restored.

## It is there in the serial field
net1 <- unpackSM(dsr1,Nethouse)
stopifnot(all.equal(pnodenames,names(PnetPnodes(net1)))))
dsr1 <- fetchSM(dsr1,Nethouse)
stopifnot(all.equal(pnodenames,names(PnetPnodes(sm(dsr1)))))

```

---

getRecordForUser	<i>Gets or makes the student record for a given student.</i>
------------------	--

---

## Description

The [BNEngine](#) contains a [StudentRecordSet](#), which is a collection of [StudentRecord](#) objects. The function `getRecordForUser` fetches one from the collection (if it exists) or creates a new one.

## Usage

```
getRecordForUser(eng, uid, srser = NULL)
```

## Arguments

<code>eng</code>	The <a href="#">BNEngine</a> in question.
<code>uid</code>	A character scalar giving the unique identifier for the student.
<code>srser</code>	A serialized version of the student record. Used to extract the student record in database-free mode. This should either be a list which is the output of <a href="#">fromJSON</a> or <code>NULL</code> .

## Details

The student record set can either be attached to a database (the `dburi` field passed to [StudentRecordSet](#) is non-empty, or not. In the database mode, records are saved in the database, so that they can be retrieved across sessions. In the database-free mode, the serialized student record (if it exists) should be passed into the `getRecordForUser` function.

If no student record is available for the `uid`, then a new one is created by cloning the default student record (see [setupDefaultSR](#)).

This function mostly just calls [getSR](#) on the [StudentRecordSet](#); however, if a new record is generated, then [announceStats](#) is called to advertise the baseline statistics for the new user.



```

        profModel="miniPP_CM",
        histNodes="Physics",
        statmat=stattab,
        activeTest="EActive.txt")

## Standard initialization methods.
loadManifest(eng,netman1)
eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

sr0a <- getRecordForUser(eng,"Student1")
sr0 <- getRecordForUser(eng,"Student1")
## This is announcing twice, so not quite working with NDB engine.

stopifnot(is.active(sm(sr0)),!is.active(sm(sr0a)))
stopifnot(all.equal(stats(sr0),stats(sr0a)))
eap0<- stat(sr0,"Physics_EAP")

e1 <- EvidenceSet(uid="Student1",app="Test",context="PPcompEM",
                 obs=list("CompensatoryObs"="Right"))

e1 <- logEvidence(eng,sr0,e1)
sr1 <- accumulateEvidence(eng,sr0,e1)
stopifnot(m_id(sr1)!=m_id(sr0),sr1@prev_id==m_id(sr0))
stopifnot(seqno(sr1)==1L, seqno(e1)==1L)

eap1 <- stat(sr1,"Physics_EAP")
stopifnot(abs(eap1-eap0) > .001)
stopifnot(nrow(history(sr1,"Phycis"))==2L)

sr1.ser <- as.json(sr1)
WarehouseFree(Nethouse,PnetName(sm(sr1))) # Delete student model to
                                           # force restore.

sr1a <- getRecordForUser(eng,"Student1",jsonlite::fromJSON(sr1.ser))
#PnetCompile(sm(sr1a))
eap1a <- stat(sr1a,"Physics_EAP")
stopifnot(abs(eap1-eap1a) < .001)
stopifnot(nrow(history(sr1a,"Phycis"))==2L)

## <<Here>> Need test with Mongo engine

```

## Description

A [StudentRecordSet](#) is a collection of [StudentRecord](#) objects. The function `getSR` fetches one from the collection if it exists. The function `newSR` creates a new one. The function `saveSR` saves the student record, and `clearSRs` clears out the saved student records.

## Usage

```
getSR(srs, uid, ser = "")
newSR(srs, uid, timestamp = Sys.time())
saveSR(srs, rec)
clearSRs(srs)
```

## Arguments

<code>srs</code>	The <a href="#">StudentRecordSet</a> in question.
<code>uid</code>	A character scalar giving the unique identifier for the student.
<code>ser</code>	A serialized version of the student record. Used to extract the student record in database-free mode. This should either be a list which is the output of <a href="#">fromJSON</a> or NULL.
<code>rec</code>	A <a href="#">StudentRecord</a> to be saved.
<code>timestamp</code>	A POSIXt datetime indicating the last modification date of the record.

## Details

The student record set can either be attached to a database (the `dburi` field passed to [StudentRecordSet](#) is non-empty, or not. In the database mode, records are saved in the database, so that they can be retrieved across sessions. In the database-free mode, the serialized student record (if it exists) should be passed into the `getSR` function.

The functions operate as follows:

`getSR` If the `ser` argument is not NULL, then the serialized student record is used to fetch the student record. Otherwise, the database (if it exists) is searched for a student record with the proper application and user ids. Then [fetchSM](#) is called to fetch the student model. If both of those methods fail, it returns NULL.

`newSR` This creates a new [StudentRecord](#) from the `defaultSR` field of the student record set (see [setupDefaultSR](#)). The function `saveSR` is called to save the new record.

`saveSR` If the database exists, the student record is saved to the database. Otherwise, if no `m_id` exists for the record one is created from the `uid` and `seqno`.

`clearSRs` In database mode, it clears the database. Otherwise, nothing is done.

## Value

The functions `getSR`, `newSR` and `saveSR` return the student record or NULL if the record was not found or created.

The function `clearSRs` returns the student record set (its argument).



**Author(s)**

Russell Almond

**See Also**

Classes: [BNEngine](#), [StudentRecordSet](#), [StudentRecord](#)

Functions: [handleEvidence](#), [setupDefaultSR](#), [fetchSM](#), [StudentRecordSet](#)

**Examples**

```
## Not run:
## Requires PNetica
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman, session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[",app,"]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app, warehouse=Nethouse,
                    listenerSet=ls, manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    statmat=stattab,
                    activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng, netman)
eng$setHistNodes("Physics")
configStats(eng, stattab)
setupDefaultSR(eng)

tr1 <- newSR(eng$studentRecords(), "Test1")
PnetCompile(sm(tr1))
stopifnot(uid(tr1)=="Test1", abs(stat(tr1, "Physics_EAP")) < .0001)
stopifnot(is.na(m_id(tr1))) # id is NA as it has not been saved yet.
```

```

tr1 <- saveSR(eng$studentRecords(),tr1)
m_id(tr1)
stopifnot(!is.na(m_id(tr1))) # Now set

sr0 <- getRecordForUser(eng,"S1")

eap0 <- stat(sr0,"Physics_EAP")

e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
  obs=list("CompensatoryObs"="Right"))

e1 <- logEvidence(eng,sr0,e1)
sr1 <- accumulateEvidence(eng,sr0,e1)
stopifnot(m_id(sr1)!=m_id(sr0),sr1@prev_id==m_id(sr0))
stopifnot(seqno(sr1)==1L, seqno(e1)==1L)

eap1 <- stat(sr1,"Physics_EAP")
stopifnot(abs(eap1-eap0) > .001)
stopifnot(nrow(history(sr1,"Phycis"))==2L)

sr1.ser <- as.json(sr1)
WarehouseFree(Nethouse,PnetName(sm(sr1))) # Delete student model to
  # force restore.

sr1a <- getSR(eng$studentRecords(),"S1",fromJSON(sr1.ser))
PnetCompile(sm(sr1a))
eap1a <- stat(sr1a,"Physics_EAP")
stopifnot(abs(eap1-eap1a) < .001)
stopifnot(nrow(history(sr1a,"Phycis"))==2L)

## End(Not run)
## Not run:
## <<Here>> Need test with Mongo implementation
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path,"testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman,session=sess,
  address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

```

```

ls <- ListenerSet(sender= paste("EAEngine[",app,"]"),
                  db=mongo::MongoDB("Messages","EARecords", makeDBuri()),
                  registryDB=mongo::MongoDB("OutputFiles","Proc4",makeDBuri()),
                  listeners=listeners)

eng <- newBNEngineMongo(app=app,warehouse=Nethouse,
                       listenerSet=ls,
                       profModel="miniPP_CM",
                       histNodes="Physics",
                       dburi=makeDBuri(),
                       dbname="EARecords",admindbname="Proc4")

## Standard initialization methods.
loadManifest(eng,netman)
eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

## End(Not run)

```

---

history

Retrieves node histories from a Student Record

---

## Description

A history is a data.frame whose rows correspond to [EvidenceSet](#) objects and whose columns correspond to the states of a [Pnode](#). Each row is a probability distribution, and they show the changes to the probabilities over time.

The function history returns the history for a single node in a given [StudentRecord](#). The function histNames returns the names of the nodes for which the record has history information.

## Usage

```

history(sr, name)
histNames(sr)

```

## Arguments

sr	A <a href="#">StudentRecord</a> whose history is to be accessed.
name	The name of the node whose history is requested.

## Details

When the student record is first initialized, the function `baselineHist` is called to setup “\*BASE-LINE\*” values for each of the history nodes identified by the `BNEngine`. These are `data.frame` objects giving the prior marginal distributions for each of the identified variables.

After the student model is updated in response to evidence (see `handleEvidence`, the `updateHist` function is called to add a new row to each of the data frames.

The `histNames` function returns the names of the history nodes being tracked by a student model. The `history` function returns the history for a node.

## Value

The function `histNames` returns a list of node names. These are suitable for the `name` argument of the `history` function.

The function `history` returns a data frame with rows corresponding to evidence sets and columns corresponding to states of the variables. Each row is a marginal probability distribution.

## Note

These are designed to work with the functions `woeHist` and `woeBal` in the `CPTtools-package`.

## Author(s)

Russell Almond

## See Also

`StudentRecord` for student records.

`baselineHist` and `updateHist` for history construction.

`BNEngine` for specifying the history nodes.

`woeHist` and `woeBal` for applications.

## Examples

```
library(PNetica)

##Start with manifest
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
config.dir <- file.path(library(help="Peanut")$path, "auxdata")
netman1 <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                    row.names=1, stringsAsFactors=FALSE)
net.dir <- file.path(library(help="PNetica")$path, "testnets")
Nethouse <- PNetica::BNWarehouse(manifest=netman1, session=sess, key="Name",
                                address=net.dir)
dsr <- StudentRecord(" *DEFAULT*", app="ecd://epls.coe.fsu.edu/P4Test",
```

```

                                context="*Baseline*")
sm(dsr) <- WarehouseSupply(Nethouse,"miniPP_CM")
PnetCompile(sm(dsr))
## dsr <- updateStats(eng,dsr)

statmat <- read.csv(file.path(config.dir,"Mini-PP-Statistics.csv"),
                    stringsAsFactors=FALSE)
rownames(statmat) <- statmat$Name
statlist <- sapply(statmat$Name,function (st)
  Statistic(statmat[st,"Fun"],statmat[st,"Node"],st))
names(statlist) <- statmat$Name
dsr@stats <- lapply(statlist,
  function (stat) calcStat(stat,sm(dsr)))
names(dsr@stats) <- names(statlist)
stat(dsr,"Physics_EAP")
stat(dsr,"Physics_Margin")

## dsr <- baselineHist(eng,dsr)

dsr@hist <- lapply(c("Physics"),
  function (nd)
    EABN::uphist(sm(dsr),nd,NULL,"*Baseline*"))
names(dsr@hist) <- "Physics"
stopifnot(histNames(dsr)=="Physics")
history(dsr,"Physics")

```

---

loadManifest	<i>Loads the manifest for the competency and evidence models in the BNEngine</i>
--------------	--

---

## Description

This sets the manifest of networks used in the scoring engine. In particular, it sets the [WarehouseManifest](#) of the [PnetWarehouse](#) associated with a [BNEngine](#).

## Usage

```
loadManifest(eng, manifest = data.frame())
```

## Arguments

eng	A <a href="#">BNEngine</a> whose manifest is to be set.
manifest	A dataframe containing a network manifest (see <a href="#">BuildNetManifest</a> ). If missing, then the manifest will be retrieved from the database or other cached source.

## Details

The [BNEngine](#) requires a proficiency or competency model (which is used to build student models) and a collection of evidence models (one for each scoring context) which are all expressed as [Pnets](#). The manifest is basically a table of which evidence model networks go with which scoring contexts. The proficiency model usually serves as the hub in the hub-and-spoke framework. (In fact, if the `profModel` argument is not supplied when the [BNEngine](#) is built, the engine will look for a network which has no hub in the manifest.

In fact, the manifest is part of the [PnetWarehouse](#) which is a field of the engine. It should have the format associate with manifests described in [WarehouseManifest](#). Note that the Bayes nets should have already been built, so the the warehouse should point to where they can be loaded from the filesystem on demand.

For the [BNEngineMongo](#), the default manifest is located in a table in the database. If no manifest is supplied, then the manifest is read from the database. For the [BNEngineNDB](#), the manifest must be specified manually when the engine is constructed (or when `loadManifest` is called).

## Value

This function returns the engine argument.

## Note

The `loadManifest` call is part of the initialization sequence for the [BNEngine](#). However, if the manifest is loaded into the [PnetWarehouse](#) as it is built, it is really redundant.

## Author(s)

Russell Almond

## References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.

## See Also

Classes: [BNEngine](#), [BNEngineMongo](#), [BNEngineNDB](#), [PnetWarehouse](#)

Functions: [WarehouseManifest](#), [BuildNetManifest](#)

## Examples

```
## Not run:
## Requires PNetica
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")
```

```

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

## Deliberately build warehouse without empty manifest.
Nethouse <- PNetica::BNWarehouse(session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[",app,"]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app,warehouse=Nethouse,
                    listenerSet=ls,manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    statmat=stattab,
                    activeTest="EAAActive.txt")

stopifnot(nrow(WarehouseManifest(eng$warehouse())) == 0L)

## Standard initialization methods.
loadManifest(eng,netman)
stopifnot(nrow(WarehouseManifest(eng$warehouse())) == 5L)

## End(Not run)

```

---

logEvidence

*Handle the relationship between evidence sets and student records.*


---

## Description

A [StudentRecord](#) differs from the baseline student record according to how many [EvidenceSet](#) objects have been incorporated into the estimate. These functions tie and student record and evidence set together.

## Usage

```

logEvidence(eng, rec, evidMess)
seqno(x)
seqno(x) <- value
evidence(x)
evidence(x) <- value

```

## Arguments

eng	A <a href="#">BNEngine</a> which is currently not used (could later be used to save the evidence to a database).
rec	A <a href="#">StudentRecord</a> into which the evidence will be incorporated.
evidMess	A <a href="#">EvidenceSet</a> which will be associated with the student record.
x	An <a href="#">EvidenceSet</a> object.
value	For seqno(x) <- value, an integer giving a new sequence number. For evidence(x) <- value, a character vector giving the sequence of evidence ID.

## Details

There are several fields in the [StudentRecord](#) class which need to be updated in the face of new evidence.

**context and timestamp** These needs to be set to the values in the new evidence message.

**seqno** This needs to be incremented.

**evidence** The new evidence needs to be prepended to this list.

**prev\_id and "\_id"** The prev\_id needs to point to the old field and the "\_id" is set to NA (it will be updated on save).

In the case of the [BNEngineMongo](#), the IDs in question are the database ids for these objects so that they can be easily found. The function [m\\_id](#) For the [BNEngineNDB](#) case presumably some external system is issuing IDs to evidence sets and student records.

The evidence field of a [StudentRecord](#) is a list of IDs ([m\\_id](#)) for the accumulated evidence.

The seqno field is an optional ordering used to track the order in which evidence sets were incorporated into the student model. The value of seqno gives the number of evidence sets incorporated into the recrod.

The logEvidence function sets the sequence number of the evidence message to one more than the last sequence number for the student record. If no [m\\_id](#) exists for the record (no database mode), then one is generated by concatenating the [uid](#) and the seqno.

## Value

The updateRecord returns a new [StudentRecord](#) object, which points back to the old one.

The logEvidence function returns the modified [EvidenceSet](#).

The function seqno returns an integer (or NA if has not been set).

The function evidence returns a character vector giving the IDs ([m\\_id](#)) of the encompated evidence sets.

## Note

This is largely untested code for future fast retraction of evidence.

The prev\_id field of the [StudentRecord](#) should leave a trace of previous student records in the database, including old serialized models. This should allow the scoring engine to quickly jump back in time.



The evidence field provides a list of the `m_ids` of all the incorporated evidence sets. This should enable one or more evidence sets to be replaced and the student model to be recalculated.

### Author(s)

Russell Almond

### See Also

[BNEngine](#), [EvidenceSet](#), [EvidenceSet StudentRecord](#), [handleEvidence P4Message](#)

### Examples

```
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)
Nethouse <- PNetica::BNWarehouse(sess=sess)

recset <- StudentRecordSet(warehouse=Nethouse,db=MongoDB(noMongo=TRUE))

sr0 <-
  StudentRecord("S1","*baseline*",as.POSIXct("2020-03-30 09:00:00"))
seqno(sr0) <- 0
sr0 <- saveSR(recset,sr0) # Sets the m_id

e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
  obs=list("CompensatoryObs"="Right"))

e2 <- EvidenceSet(uid="S1",app="Test",context="PPdurAttEM",
  obs=list("Attempts"=2,"Duration"=38.3))

stopifnot(is.na(seqno(e1)), seqno(sr0)==0L)
stopifnot(length(evidence(sr0))==0L)

e1 <- logEvidence(NULL,sr0,e1)
stopifnot(seqno(e1)==1L,!is.na(m_id(e1)))

sr1 <- updateRecord(sr0,e1)
stopifnot(is.na(m_id(sr1)),sr1@prev_id==m_id(sr0))
sr1 <- saveSR(recset,sr1) # Sets the m_id

stopifnot(length(evidence(sr1))==1L,any(m_id(e1)==evidence(sr1)))
stopifnot(context(sr1)==context(e1),timestamp(sr1)==timestamp(e1))

e2 <- logEvidence(NULL,sr1,e2)
stopifnot(seqno(e2)==2L,!is.na(m_id(e2)))

sr2 <- updateRecord(sr1,e2)
stopifnot(is.na(m_id(sr2)),sr2@prev_id==m_id(sr1))
sr2 <- saveSR(recset,sr2) # Sets the m_id

stopifnot(length(evidence(sr2))==2L,any(m_id(e2)==evidence(sr2)))
stopifnot(context(sr2)==context(e2),timestamp(sr2)==timestamp(e2))
```

---

`logIssue`*Manage error messages associated with a `StudentRecord`.*

---

### Description

The function `logIssue()` adds an issue to a `StudentRecord`. The function `getIssues()` returns a list of issues.

### Usage

```
logIssue(sr, issue)
## S4 method for signature 'StudentRecord,ANY'
logIssue(sr,issue)
## S4 method for signature 'StudentRecord,character'
logIssue(sr,issue)
getIssues(sr)
## S4 method for signature 'StudentRecord'
getIssues(sr)
```

### Arguments

<code>sr</code>	A <code>StudentRecord</code> object to be examined or modified.
<code>issue</code>	An issue to be logged. This should be a character object or something which can be coerced to a character object.

### Details

The idea is to be able to log error messages and warning which occur when processing evidence for this person. These are converted to strings, so they can be saved

### Value

The function `getIssues()` returns a character vector containing the encountered issues.

The function `logIssue()` returns the modified student record.

### Author(s)

Russell Almond

### See Also

`StudentRecord`, `markAsError`

## Examples

```
sr0 <-
  StudentRecord("S1", "*baseline*", as.POSIXct("2020-03-30 09:00:00"))

sr0 <- logIssue(sr0, "Test Issue")
err <- simpleError("Another test error.")
sr0 <- logIssue(sr0, err)
getIssues(sr0)
```

---

mainLoop

*This function loops through the processing of evidence sets.*


---

## Description

The mainLoop is used when the [BNEngine](#) is used as a server. It checks the queue (database or internal list), for unprocessed [EvidenceSet](#) objects, and calls [handleEvidence](#) on them in the order of their [timestamps](#). As a server, this is potentially an infinite loop, see details for ways of gracefully terminating the loop.

## Usage

```
mainLoop(eng, N=NULL)
```

## Arguments

eng	An <a href="#">BNEngine</a> which will handle the evidence sets.
N	If supplied, this should be an integer. The loop will then handle that many cycles before quitting.

## Details

The evidenceQueue field of the [BNEngine](#) class is an object of type [MessageQueue](#). All events have a [processed](#) field which is set to true when the evidence set is processed. The function [fetchNextMessage](#) fetches the oldest unprocessed evidence set, while [markAsProcessed](#) sets the processed flag.

The mainLoop function iterates over the following steps.

1. Fetch the oldest unprocessed Event: eve <- [fetchNextMessage](#)(eng).
2. Process the evidence set: out <- [handleEvidence](#)(eng, eve). (Note: this expression will always return. If it generates an error, the error will be logged and an object of class try-error will be returned.)
3. Mark the event as processed: [markAsProcessed](#)(eng, eve).

At its simplest level, the function produces an infinite loop over these three statements, with some additional steps related to logging and control.

First, if the event queue is empty, the process sleeps for a time given by `eng$waittime` and then checks the queue again. At the same time, it checks status of the active flag for the process using the `eng$stopWhenFinished()` call. If this returns true and the queue is empty, processing will terminate.

To facilitate testing, the field `eng$processN` can be set to a finite value. This number is decremented at every cycle, and when it reaches 0, the `mainLoop` is terminated, whether or not there are any remaining events to be processed. Setting `eng$processN` to an infinite value, will result in an infinite loop that can only be stopped by using the active flag (or interrupting the process).

### Value

There is no return value. The function is used entirely for its side effects.

### Activation

When the loop begins, it calls the `eng$activate()` method to mark the engine as active. When the loop finishes (outside of the main try/catch block, so it should always return), it calls the `eng$deactivate()` method to signal that the engine has terminated.

External processes can signal the engine through the `eng$shouldHalt()` and `eng$stopWhenFinished()`. The former is checked every iteration, and the main loop halts when it becomes true. This allows for an immediate stop when needed. The latter is checked only when the queue is empty and details whether or not the process should continue to wait for more messages in the queue.

**Database Engine.** For the Mongo engine ([BNEngineMongo](#)) the communication channel is the `AuthorizedApps` collection in the administrative database. In particular, the `EAsignal` field is read by both methods. The `eng$activate()` method changes the value of that field to “Running”. Changing the value of the field to “Halt” will cause the `eng$shouldHalt()` to be true triggering a halt before processing the next evidence set. Changing the value of that field to “Finish” will cause `eng$stopWhenFinished()` to be true, causing the loop to stop then the queue is empty.

The following command issues from the Mongo shell will shut down the server for an application containing the string “appName” as part of its name (note “Halt” could be replaced with “finish”).

```
db.AuthorizedApps.update({app:{$regex:"appName"}}, {$set:{"EAsignal":"Halt"}});
```

**No Database Engine.** For the Mongo engine ([BNEngineMongo](#)) the communication channel is a file named `activeTest`. The name (extension) of this file is changed to produce the signals. The `eng$activate()` method creates it with the extension `.running`. Changing the extension to `.finish` or `.halt` will send the appropriate signal. The `eng$deactivate()` method removes the file.

### Note

Currently, when running in server model (i.e., with `eng$processN` set to infinity), there are two ways of stopping the engine: a clean stop after all events are processed using the active flag, and an immediate stop, possibly mid cycle, by killing the server process. It became apparent during testing that there was a need for a graceful but immediate stop, i.e., a stop after processing the current event. This should appear in later versions.

**Author(s)**

Russell Almond

**See Also**

[BNEngine](#), [BNEngineMongo](#), [BNEngineNDB](#), [MessageQueue](#)  
[fetchNextMessage](#), [handleEvidence](#), [markAsProcessed](#)

**Examples**

```
## Not run:
## From EABN.R script
app <- "ecd://epls.coe.fsu.edu/P4test"
loglevel <- "DEBUG"

source("/usr/local/share/Proc4/EAini.R")
futile.logger::flog.appender(appender.file(logfile))
futile.logger::flog.threshold(loglevel)

sess <- NeticaSession(LicenseKey=NeticalLicenseKey)
startSession(sess)
listeners <- lapply(names(EA.listenerSpecs),
                    function (ll) do.call(ll,EA.listenerSpecs[[ll]]))
names(listeners) <- names(EA.listenerSpecs)

eng <- do.call(BNEngineMongo,
              c(EAeng.params,list(session=sess,listeners=listeners),
                EAeng.common))
loadManifest(eng)
configStats(eng)
setupDefaultSR(eng)

## Activate engine (if not already activated.)
eng$activate()
mainLoop(eng)
## Wait for cows to come home.

## End(Not run)
```

---

observables

---

*Access parts of an evidence set message.*


---

**Description**

The function `observables` access the list of observables contained in this [EvidenceSet](#). The function `seqno` access the order in which the evidence sets were incorporated into the student record.

**Usage**

```
observables(x)
```

**Arguments**

x                    An [EvidenceSet](#) object.

**Details**

The `observables` function access the data field of the underlying [P4Message](#). This should be a named list of values that the [BNEngine](#) knows how to process.

**Value**

The function `observables` returns a named list of observable values.

**Author(s)**

Russell Almond

**See Also**

[EvidenceSet](#), [EvidenceSet StudentRecord](#), [handleEvidence P4Message](#)

**Examples**

```
e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
  obs=list("CompensatoryObs"="Right"))

e2 <- EvidenceSet(uid="S1",app="Test",context="PPdurAttEM",
  obs=list("Attempts"=2,"Duration"=38.3))

stopifnot(all.equal(observables(e1),
  list("CompensatoryObs"="Right")))

stopifnot(all.equal(observables(e2)$Attempts,2))

stopifnot(is.na(seqno(e1)))
seqno(e1) <- 1
stopifnot(seqno(e1)==1L)
```

---

parseEvidence	<i>Convert EvidenceSet objects to and from JSON</i>
---------------	---

---

### Description

The `as.json` function takes an [EvidenceSet](#) (among other objects) and turns it into JSON. The function `parseEvidence` takes the list produced as the output to `fromJSON` and turns it back into an [EvidenceSet](#) object.

### Usage

```
parseEvidence(rec)
## S4 method for signature 'EvidenceSet,list'
as.jlist(obj, ml, serialize=TRUE)
```

### Arguments

<code>rec</code>	A list which comes from running <code>fromJSON</code> on a JSON string, or database extraction method.
<code>obj</code>	The object being serialized; usually <code>attributes(obj)</code> .
<code>ml</code>	A list of fields of the object.
<code>serialize</code>	A logical flag. If true, <code>serializeJSON</code> is used to protect the data field (and other objects which might contain complex R code).

### Details

See the description for `as.json` for more description of the JSON conversion protocol.

The `parseEvidence` method is designed to be used with the `getOneRec` and `getManyRecs` functions for fetching information from the database.

### Value

The function `parseEvidence` returns an object of class [EvidenceSet](#).

The `as.jlist` method returns a list which can be passed to `toJSON` to produce legible JSON from the R object.

### Author(s)

Russell Almond

### See Also

[EvidenceSet](#), [as.json](#), [getOneRec](#), [getManyRecs](#)

## Examples

```
e1 <- EvidenceSet(uid="S1", app="Test", context="PPcompEM",
  obs=list("CompensatoryObs"="Right"))

e2 <- EvidenceSet(uid="S1", app="Test", context="PPdurAttEM",
  obs=list("Attempts"=2, "Duration"=38.3))

e1.ser <- as.json(e1)
e1a <- parseEvidence(jsonlite::fromJSON(e1.ser))
e2.ser <- as.json(e2)
e2a <- parseEvidence(jsonlite::fromJSON(e2.ser))

stopifnot(all.equal(e1,e1a), all.equal(e2,e2a))
```

---

parseStats

*Functions for (un)serializing stats from student records.*

---

## Description

The functions `unparseStats` and `stats2json` serialize the statistics as a JSON record. The function `parseStats` reverses the process.

## Usage

```
parseStats(slist)
unparseStats(slist, flatten=FALSE)
stats2json(slist, flatten=FALSE)
```

## Arguments

<code>slist</code>	A list of statistics. For <code>parseStats</code> this should be the output of <code>fromJSON</code> . For the others, this is just a list of statistic values.
<code>flatten</code>	If true, then vector-valued statistics (i.e., <code>PnodeMargin</code> , will have their values flattened into scalars. If not they will be left as vectors.

## Details

The function `unparseStats` massages the list of statistics so it will be output in clean JSON (in particular, using `unboxer` to make sure scalars appear as scalars and not vectors). The function `stats2json` is just `toJSON(unparseStats(slist))`.

If `flatten` is true, then vector value statistics will be flattened. For example, if the statistic “`Physics_Margin`” has three values with labels “`High`”, “`Medium`”, and “`Low`”, then it will be replaced with three statistics with the names “`Physics_Margin.High`”, “`Physics_Margin.Medium`”, and “`Physics_Margin.Low`”.

The function `parseStatistics` is designed to reverse the process.



**Value**

The function `unparseStats` returns a list which is ready to be passed to [toJSON](#). In particular, scalars are marked using [unboxer](#).

The function `stats2json` returns a string containing the JSON.

The function `parseStats` returns a list of statistics values. this is suitable for being set to the `stats` field of the [StudentRecord](#) object.

**Note**

When using `flatten=TRUE`, avoid periods, '.', in the names of statistics, as this marker is used to recreate the nested structure in `parseStats`.

**Author(s)**

Russell Almond

**See Also**

[buildObject](#) gives general information about how the parsing/unparsing protocol works.

[Statistic](#) gives a list of available statistics.

[StudentRecord](#) talks about the statistic fields of the student records.

**Examples**

```
stats <- list(Physics_EAP=0,EnergyTransfer_EAP=.15,
             Physics_Margin=c(High=1/3,Medium=1/3,
                              Low=1/3))

stats2json(stats)

stats1 <- parseStats(ununboxer(unparseStats(stats)))
stopifnot(all.equal(stats,stats1,tolerance=.0002))

stats2json(stats,flatten=TRUE)

stats2 <- parseStats(ununboxer(unparseStats(stats,flatten=TRUE)))
stopifnot(all.equal(stats,stats2,tolerance=.0002))
```

---

parseStudentRecord

*Covert Student Records to/from JSON*

---

**Description**

The [as.json](#) function takes an [StudentRecord](#) (among other objects) and turns it into JSON. The function `parseStudentRecord` takes the list produced as the output to [fromJSON](#) and turns it back into an [StudentRecord](#) object.

**Usage**

```

parseStudentRecord(rec)
## S4 method for signature 'StudentRecord,list'
as.jlist(obj, ml, serialize=TRUE)

```

**Arguments**

rec	A list which comes from running <a href="#">fromJSON</a> on a JSON string, or database extraction method.
obj	The object being serialized; usually <code>attributes(obj)</code> .
ml	A list of fields of the object.
serialize	A logical flag. If true, <a href="#">serializeJSON</a> is used to protect the data field (and other objects which might contain complex R code).

**Details**

See the description for [as.json](#) for more description of the general JSON conversion protocol.

The [StudentRecord](#) contains a [Pnet](#) field in the student model. This takes some post-processing to properly restore.

The `as.jlist` method for the [StudentRecord](#) serializes the `sm` field using the [PnetSerialize](#) method. This produces a slob (string large object) which is stored in the `smser` field of the [StudentRecord](#).

The `parseStudentRecord` function restores the `smser` field, but not the `sm` field. This must be done in the context of the [StudentRecordSet](#), or equivalently the [PnetWarehouse](#), which is currently managing the networks. To finish the process, call [fetchSM](#) to restore the student model network.

**Value**

The function `parseStudentRecord` returns a student record object with the student model not yet initialized.

The `as.jlist` method returns a list which can be passed to [toJSON](#) to produce legible JSON from the R object.

**Author(s)**

Russell Almond

**See Also**

[StudentRecord](#), [as.json](#), [getOneRec](#), [getManyRecs](#)  
[fetchSM](#), [PnetSerialize](#)

**Examples**

```

## Requires PNetica
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()

```

```

RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path,"testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman,session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[",app,"]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app,warehouse=Nethouse,
                    listenerSet=ls,manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    statmat=stattab,
                    activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng,netman)
eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

recset <- eng$studentRecords()

sr0 <- getRecordForUser(eng,"S1")
eap0 <- stat(sr0,"Physics_EAP")

sr0.ser <- as.json(sr0)
sr0a <- parseStudentRecord(jsonlite::fromJSON(sr0.ser))
sr0a <- fetchSM(sr0a,recset$warehouse())
## This should relink to the same student model
stopifnot(sm(sr0a)==sm(sr0),abs(eap0-stat(sr0a,"Physics_EAP")) <.0001)

## Next add some evidence and test again.

e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
                 obs=list("CompensatoryObs"="Right"))

e1 <- logEvidence(eng,sr0,e1)
sr1 <- accumulateEvidence(eng,sr0,e1)
eap1 <- stat(sr1,"Physics_EAP")
sr1.ser <- as.json(sr1)

```

```
## Force delete student model to make sure that it is properly
## recovered.
WarehouseFree(Nethouse,PnetName(sm(sr1)))
stopifnot(!is.active(sm(sr1))) # No longer active.

sr1a <- parseStudentRecord(jsonlite::fromJSON(sr1.ser))
sr1a <- fetchSM(sr1a,recset$warehouse())
eap1a <- stat(sr1a,"Physics_EAP")
stopifnot(all(evidence(sr1)==evidence(sr1a)),
          abs(eap1-eap1a) <.001)
```

---

setupDefaultSR

---

Set up the Default Student Record for an StudentRecordSet

---

## Description

The default student record is a field associated with a [StudentRecordSet](#) which provides a template student record for a student just starting the assessment. The `setupDefaultSR` function needs to be called at the start of every scoring session to initialize the `defaultSR` field of the student record set.

## Usage

```
setupDefaultSR(eng)
```

## Arguments

`eng`                    A [BNEngine](#) which contains the student record details.

## Details

This function creates a new [StudentRecord](#) object with the special uid “\*DEFAULT\*” and the special `context` ID “\*Baseline\*”. The student model is actually the competency or proficiency model: the baseline student model giving the population distribution of the the measured proficiencies. This is fetched by name from the [PnetWarehouse](#) attached to the engine; the name is given in the `profModel` field of the `eng`.

Setting up a default student record actually takes a number of steps:

1. The student record set (`eng$studentRecords()`) is cleared by calling [clearSRs](#).
2. A new blank student record (`uid="*DEFAULT*"`) is created.
3. The `sm` field of the new student record is initialized to the proficiency model.
4. The student model is compiled ([PnetCompile](#)).
5. The baseline statistics are calculated ([updateStats](#)).
6. The baseline history is set ([baselineHist](#)).
7. The default student record is saved in the `defaultSR` field of the [StudentRecordSet](#) and in the database ([saveSR](#)).
8. The baseline statistics are announced ([announceStats](#)).

**Value**

This function is called for its side effects.

**Author(s)**

Russell Almond

**References**

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapter 13.

**See Also**

Classes: [BNEngine](#), [StudentRecord](#), [StudentRecordSet](#), [PnetWarehouse](#)

Functions: [clearSRs](#), [PnetCompile](#), [updateStats](#), [baselineHist](#), [saveSR](#), [announceStats](#)

**Examples**

```
## Requires PNetica
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman, session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[", app, "]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app, warehouse=Nethouse,
                    listenerSet=ls, manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    statmat=stattab,
                    activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng, netman)
```

```

eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

defrec <- eng$studentRecords()$defaultSR

stopifnot(uid(defrec)=="*DEFAULT*", app(defrec)==app(eng),
          context(defrec)=="*Baseline*",
          PnetName(sm(defrec))==eng$profModel)

```

---

sm

---

*Access the student model (Pnet) associated with a studnet record*


---

## Description

A characteristic of the EABN model is that each code [StudentRecord](#) is associated with a *student model*—a [Pnet](#) which tracks our knowledge about the student’s knowledge skills and abilities. The function `sm` accesses the net.

## Usage

```

sm(x)
sm(x) <- value

```

## Arguments

x	An object of class <a href="#">StudentRecord</a> whose student model will be accessed.
value	A <a href="#">Pnet</a> object which will be the new student model.

## Value

The function `sm` returns an object which implements the [Pnet](#) protocol, or none is the student model has not been generated.

The setter version returns the student record.

## Author(s)

Russell Almond

## See Also

[fetchSM](#), [unpackSM](#), [setupDefaultSR](#)

## Examples

```
library(PNetica)

##Start with manifest
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
config.dir <- file.path(library(help="Peanut")$path, "auxdata")
netman1 <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                    row.names=1, stringsAsFactors=FALSE)
net.dir <- file.path(library(help="PNetica")$path, "testnets")
Nethouse <- PNetica::BNWarehouse(manifest=netman1, session=sess, key="Name",
                                address=net.dir)

dsr <- StudentRecord("*DEFAULT*", app="ecd://epls.coe.fsu.edu/P4Test",
                    context="*Baseline*")
sm(dsr) <- WarehouseSupply(Nethouse, "miniPP_CM")
PnetCompile(sm(dsr))
```

---

stat

---

*Access statistics from a Student Record*


---

## Description

These functions access the stats field of a [StudentRecord](#) object. The function `stat` accesses a single statistics and `stats` returns all of the statistics. The function `statNames` returns the names of the available statistics.

## Usage

```
stat(sr, name)
stats(x)
statNames(sr)
```

## Arguments

<code>sr, x</code>	A <a href="#">StudentRecord</a> object whose statistics are to be accessed.
<code>name</code>	A character object giving the name of the specific statistic to access.

## Value

The function `stat` returns the value of a single statistic, which could be numeric, character or something else.

The function `stats` returns a named list of statistics.

The function `statNames` returns a character vector.

**Author(s)**

Russell Almond

**See Also**

[StudentRecord](#) for the student record class.

[Statistic](#) for statistic objects which return the statistics.

**Examples**

```
stats <- list(Physics_EAP=0,EnergyTransfer_EAP=.15,
             Physics_Margin=c(High=1/3,Medium=1/3,
                              Low=1/3))

dsr <- StudentRecord("*DEFAULT*",app="ecd://epls.coe.fsu.edu/P4Test",
                    context="*Baseline*",stats=stats)

stats(dsr)
stopifnot(all.equal(stats,stats,tolerance=.0002))

statNames(dsr)
stopifnot(all(statNames(dsr)==names(stats)))

stat(dsr,"Physics_Margin")
stopifnot(all.equal(stat(dsr,"Physics_Margin"),stats[[3]],tolerance=.0002))
```

---

StudentRecord

*Constructor for StudentRecord object*

---

**Description**

This is the constructor for a [StudentRecord](#) object. Basically, this is a wrapper around the studnet model for the appropriate user, with meta-data about the evidence that has been absorbed.

**Usage**

```
StudentRecord(uid, context = "", timestamp = Sys.time(), smser = list(),
             sm = NULL, stats = list(), hist = list(), evidence = character(),
             app = "default", seqno = -1L, prev_id = NA_character_)
```



**Arguments**

uid	A user identifier for the student/player.
context	An identifier for the scoring context/window.
timestamp	Timestamp of the last evidence set absorbed for this user.
smser	A serialized Bayesian network (see <a href="#">WarehouseUnpack</a> ).
sm	A <a href="#">Pnet</a> containing the student model (or NULL if it has not been initialized).
stats	A list of statistics calculated for the model.
hist	A list of node histories for the measured nodes.
evidence	A character vector of ids for the absorbed evidence sets.
app	A guid (string) identifying the application.
seqno	A sequence number, basically a count of absorbed evidence sets.
prev_id	The database ID of the previous student model.

**Value**

An object of class [StudentRecord](#).

**Author(s)**

Russell Almond

**See Also**

[StudentRecord](#)

**Examples**

```
library(PNetica)

##Start with manifest
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
config.dir <- file.path(library(help="Peanut")$path, "auxdata")
netman1 <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                    row.names=1, stringsAsFactors=FALSE)
net.dir <- file.path(library(help="PNetica")$path, "testnets")
Nethouse <- PNetica::BNWarehouse(manifest=netman1, session=sess, key="Name",
                                address=net.dir)

dsr <- StudentRecord("*DEFAULT*", app="ecd://epls.coe.fsu.edu/P4Test",
                    context="*Baseline*")
sm(dsr) <- WarehouseSupply(Nethouse, "miniPP_CM")
PnetCompile(sm(dsr))
```

```

## dsr <- updateStats(eng,dsr)
statmat <- read.csv(file.path(config.dir,"Mini-PP-Statistics.csv"),
                    stringsAsFactors=FALSE)
rownames(statmat) <- statmat$Name
statlist <- sapply(statmat$Name,function (st)
  Statistic(statmat[st,"Fun"],statmat[st,"Node"],st))
names(statlist) <- statmat$Name
dsr@stats <- lapply(statlist,
  function (stat) calcStat(stat,sm(dsr)))
names(dsr@stats) <- names(statlist)
stat(dsr,"Physics_EAP")
stat(dsr,"Physics_Margin")

## dsr <- baselineHist(eng,dsr)

dsr@hist <- lapply(c("Physics"),
  function (nd)
    EABN::uphist(sm(dsr),nd,NULL,"*Baseline*"))
names(dsr@hist) <- "Physics"
history(dsr,"Physics")

## Serialization and unserialization
dsr.ser <- as.json(dsr)

dsr1 <- parseStudentRecord(jsonlite::fromJSON(dsr.ser))
dsr1 <- fetchSM(dsr1,Nethouse)

### dsr and dsr1 should be the same.
stopifnot(
  app(dsr)==app(dsr1),
  uid(dsr)==uid(dsr1),
  context(dsr)==context(dsr1),
  # problems with timezones
  # all.equal(timestamp(dsr),timestamp(dsr1)),
  all.equal(seqno(dsr),seqno(dsr1)),
  all.equal(stats(dsr),stats(dsr1),tolerance=.0002),
  all.equal(history(dsr,"Physics"),history(dsr1,"Physics")),
  PnetName(sm(dsr)) == PnetName(sm(dsr1))
)

```

---

StudentRecord-class	<i>Class "StudentRecord"</i>
---------------------	------------------------------

---

## Description

This is a wrapper for the Bayesian network information for a particular student. It contains a local copy of the Bayesian network.

## Objects from the Class

Objects can be created by calls to the function `StudentRecord`, `uid`, `context`, `timestamp`, `smser`, `sm`, `stats`, `hist`, `evidence`).

## Slots

`_id`: Object of class "character" The [mongo](#) ID of the object, empty character if it has not been saved in the database. If Mongo is not being used, this field can be used for other kinds of IDs.

`app`: Object of class "character" that gives the identifier for the application this record is used with.

`uid`: Object of class "character" which is the unique identifier for the user (student, player).

`context`: Object of class "character" which identifies the scoring context (scoring window).

`evidence`: Object of class "character" giving the IDs of the evidence sets applied to this student model.

`timestamp`: Object of class "POSIXt" giving the timestamp of the last evidence set applied to this model.

`sm`: Object of class "[Pnet](#)", the actual student model (or NULL if it is not yet built).

`smser`: Object of class "list" the serialized student model.

`seqno`: Object of class "integer" a sequence number, that is the number of evidence sets applied.

`stats`: Object of class "list" the most recent statistics generated from this model.

`hist`: Object of class "list" list of history lists for the designed history variables. There is one element for each history variable.

`issues`: A character vector giving errors and warnings from processing evidence for this record.

`prev_id`: Object of class "character" the Mongo ID of the previous student model.

## Methods

**app** signature(`x` = "StudentRecord"): returns the application id associated with this record.

**as.jlist** signature(`obj` = "StudentRecord", `ml` = "list"): serialized the record as JSON

**context** signature(`x` = "StudentRecord"): return the context (scoring window) identifier associated with the last processed evidence set.

**evidence** signature(`x` = "StudentRecord"): returns the ids of the absorbed evidence sets.

**evidence<-** signature(`x` = "StudentRecord"): sets the ids of the absorbed evidence sets.

**histNames** signature(`sr` = "StudentRecord"): returns the names of the history variables.

**history** signature(`sr` = "StudentRecord", `name` = "character"): returns the history list for the variable.

**seqno** signature(`x` = "StudentRecord"): returns the sequence number for this record.

**seqno<-** signature(`x` = "StudentRecord"): sets the sequence number for this record.

**show** signature(`object` = "StudentRecord"): prints the record.

**sm** signature(`x` = "StudentRecord"): returns the Bayes net ([Pnet](#)) associated with this record.

**sm<-** signature(`x` = "StudentRecord", `value`="ANY"): sets the Bayes net ([Pnet](#)) associated with this record.

**stat** signature(sr = "StudentRecord", name = "character"): returns the current value of the named statistics.

**statNames** signature(sr = "StudentRecord"): returns the names of the statistics.

**stats** signature(x = "StudentRecord"): returns all of the statistics.

**timestamp** signature(x = "StudentRecord"): returns the timestamp of the last absorbed evidence set.

**toString** signature(x = "StudentRecord"): creates a printed representation.

**uid** signature(x = "StudentRecord"): returns the ID for the student/player.

### Author(s)

Russell Almond

### References

Almond, R.G., Mislevy, R.J., Steinberg, L.S., Williamson, D.M. and Yan, D. (2015) *Bayesian Networks in Educational Assessment*. Springer. Chapter 13.

### See Also

[StudentRecord](#), [EvidenceSet](#), [StudentRecordSet](#)

### Examples

```
showClass("StudentRecord")
```

---

StudentRecordSet	<i>Constructor for "StudentRecordSet" class</i>
------------------	---

---

### Description

A [StudentRecordSet](#) is a collection of collection of [StudentRecord](#) objects. It is always connected to a [PnetWarehouse](#) and could be connected to a database as well.

### Usage

```
StudentRecordSet(app = "default", warehouse = NULL,
  db=MongoDB("StudentRecords", "EARecords"), ...)
```

### Arguments

app	A character scalar providing a guid for the application.
warehouse	An object of type <a href="#">PnetWarehouse</a> that contains already built student models.
db	A <a href="#">JSONDB</a> object which store the student records.
...	Other arguments for future extensions.

## Details

A StudentRecordSet is a collection of student records. It contains a [PnetWarehouse](#) which contains the student models and possibly a database containing the student records.

The StudentRecordSet operates in two modes, depending on the value of db. If db references a [MongoDB-class](#) database, then the StudentRecordSet set will save student records (including serialized Bayes nets) to the database and restore them on demand. This facilitates scoring across several sessions.

If the db argument has the noMongo flag, no database connection will be created. Instead, the calls to the [getSR](#) function should pass in a serialized version of the student record function. If no serialized record is available, a new record will be created.

## Value

An object of class [StudentRecordSet](#).

## Author(s)

Russell Almond

## See Also

[StudentRecordSet](#), [StudentRecord](#), [getSR](#), [saveSR](#), [newSR](#), [clearSRs](#)

## Examples

```
library(PNetica)

##Start with manifest
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

## BNWarehouse is the PNetica Net Warehouse.
## This provides an example network manifest.
config.dir <- file.path(library(help="Peanut")$path, "auxdata")
netman1 <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                    row.names=1, stringsAsFactors=FALSE)
net.dir <- file.path(library(help="PNetica")$path, "testnets")
Nethouse <- PNetica::BNWarehouse(manifest=netman1, session=sess, key="Name",
                                address=net.dir)

## Setup to test without Mongo
SRS <- StudentRecordSet(app="Test", warehouse=Nethouse,
                        db=mongo::MongoDB(noMongo=TRUE))
stopifnot(!mdbAvailable((SRS$recorddb())))

## Setup default SR
dsr <- StudentRecord("*DEFAULT*", app="Test",
                    context="*Baseline*")
sm(dsr) <- WarehouseSupply(Nethouse, "miniPP_CM")
PnetCompile(sm(dsr))
```

```

## dsr <- updateStats(eng,dsr)
statmat <- read.csv(file.path(config.dir,"Mini-PP-Statistics.csv"),
                    stringsAsFactors=FALSE)
rownames(statmat) <- statmat$Name
statlist <- sapply(statmat$Name,function (st)
  Statistic(statmat[st,"Fun"],statmat[st,"Node"],st))
names(statlist) <- statmat$Name
dsr@stats <- lapply(statlist,
  function (stat) calcStat(stat,sm(dsr)))
names(dsr@stats) <- names(statlist)

dsr@hist <- lapply(c("Physics"),
  function (nd)
    EABN::uphist(sm(dsr),nd,NULL,"*Baseline*"))
names(dsr@hist) <- "Physics"

SRS$defaultSR <- dsr
saveSR(SRS, dsr)

## Make a new Student Record for a student.
sr1 <- newSR(SRS,"S1")
stopifnot(uid(sr1)=="S1",app(sr1)==app(dsr),
  all.equal(stats(dsr),stats(sr1),.0002))

sr1a <- getSR(SRS,"S1")

clearSRs(SRS)

```

---

StudentRecordSet-class

*Class "StudentRecordSet"*


---

## Description

This class provides a collection of student records. Optionally, it can be hitched to a database so that student can be saved and restored across scoring sessions.

## Details

The StudentRecordSet exists to hold a collection of [StudentRecord](#) objects. If, when constructed, the record set is passed information about a database, the record set is stored in the database. If not, it is merely stored in memory. The database version, in particular, allows restoring the object from memory. The primary key for the student record in the database is the app ID (which is a field in the record set) and the uid which is passed through the [getSR](#) method.

The method [getSR](#) takes different arguments based on which version is passed. In particular, the ser argument allows a serialized (JSON) version of the data to be passed in. In particular, [getSR](#) will do one of the following things (in order of priority):

1. If the ser argument is supplied, the student record will be restored from this.
2. If the StudentRecordSet is connected to a database, then the student record is restored from information in the database, based on the uid argument and the app field.
3. A new student record is created for the uid.

The record set also contains a link to a [PnetWarehouse](#) which it uses to try and find the [Pnet](#) associated with the [StudentRecord](#). If the Pnet already exists in the warehouse, it is just connected to the fetched record. If not, then it is restored from a serialized version either from the passed in serialized record, or from the serialized Pnet in the database.

### Extends

All reference classes extend and inherit methods from "[envRefClass](#)".

### Methods

- app** signature(x = "StudentRecordSet"): Returns the application this record set is associated with.
- getSR** signature(srs = "StudentRecordSet", uid="ANY", ser="character"): Returns the student record for the specified ID. If ser is supplied it should be a json list object containing the student record.
- newSR** signature(srs = "StudentRecordSet", uid="character"): Creates a new Student Record for the specified ID by cloning the default student record.
- saveSR** signature(srs = "StudentRecordSet"): If connected to a database, the SR is saved to the database.
- clearSR** signature(srs = "StudentRecordSet"): If connected to a database, the SR in the database are cleared.

### Fields

- app**: Object of class character which contains the application identifier
- dbname**: Object of class character which contains the name of the database.
- db**: Object of class [JSONDB](#) a connection to the database or NULL if the object is not connected to the database. Users should call the recorddb() function rather than access this field directly.
- warehouse**: Object of class [PnetWarehouse](#) which contains already loaded nets.
- defaultSR**: Object of class [StudentRecord](#) or NULL. This is the default student record which is cloned to create new studnet records.

### Class-Based Methods

- initialize**(app, dbname, dburi, db, warehouse, ...): Initializes the student record set.
- recorddb**(): Returns the database handle (if connected to a database) or NULL if not connected to a database. Note that this initializes the database the first time it is called, so it should be called rather than accessing the db field directly.
- clearAll**(clearDefault=FALSE): Clears all records from the database and the warehouse. If clearDefault==FALSE, then the default record is not cleared.

**Author(s)**

Russell Almond

**See Also**

[StudentRecordSet](#) for the constructor. [StudentRecord](#) for the contained objects.  
[PnetWarehouse](#) and [Pnet](#) for information about the contained Bayesian networks.  
[BNEngine](#) for the engine that holds it.

**Examples**

```
showClass("StudentRecordSet")
```

---

trimTable	<i>Trims empty columns from tables.</i>
-----------	---

---

**Description**

Downloaded spreadsheets sometimes contain empty columns at the end. This function removes all of the columns after the give column.

**Usage**

```
trimTable(tab, lastcol = "Description")
```

**Arguments**

tab	A matrix, data frame or tibble to be trimmed).
lastcol	The name of the last column to keep. Any column to the right of this one will be discarded.

**Value**

The first several columns of the table.

**Author(s)**

Russell Almond

**See Also**

[read.csv](#)

**Examples**

```
dat <- data.frame(One=1:3,Two=4:6,Three=7:9,10:12)
trimmed <- trimTable(dat,"Three")
stopifnot (ncol(trimmed)==3L)
```



---

updateHist

Update the node history in a student record

---

## Description

The [StudentRecord](#) object can track the history of zero or more [Pnode](#) in the student model ([sm](#)). The history is a data frame with columns corresponding to the states of the variables and the rows corresponding to the [EvidenceSets](#) absorbed into the student record. The function `updateHist` add a new row to each history corresponding to the evidence set. The function `baselineHist` creates the initial row.

## Usage

```
updateHist(eng, rec, evidMess, debug = 0)
baselineHist(eng, rec)
```

## Arguments

<code>eng</code>	The <a href="#">BNEngine</a> controlling the operation.
<code>rec</code>	The <a href="#">StudentRecord</a> which will be updated.
<code>evidMess</code>	The <a href="#">EvidenceSet</a> which has just been added to the student model using <code>updateSM</code> .
<code>debug</code>	An integer flag. If bigger than 1, then a call to <code>recover</code> will be made inside the function call.

## Details

A history tracks a single node in the student model as it changes in response to the incoming evidence sets. The history for a node is data frame with columns representing variable states and rows representing evidence sets (evidence from different scoring windows or tasks).

The function `baselineHist` is called as part of `setupDefaultSR`. This initializes a history data frame for each node in the `histNodes` field of the [BNEngine](#). It inserts a first row, which is always given the name “\*Baseline\*”. The values in the first row are the marginal distribution of those nodes ([PnodeMargin](#)).

The function `updateHist` adds row to each history table. The name of the row corresponds to the `context` field of the [EvidenceSet](#). The value is the current marginal distribution for the history nodes.

The function `history` retrieves the history. The functions `woeHist` and `woeBal` in the [CPTtools-package](#) describe possible applications for the history function.

## Value

Both functions return the modified [StudentRecord](#)

**Note**

With the Netica implementation, the student model needs to be compiled (`PnetCompile(sm(rec))`) before the `baselineHist` function is run.

This is probably true of `updateHist` as well, but `updateSM` recompiles the network.

**Author(s)**

Russell Almond

**References**

Madigan, Mosurski and Almond, (1997). Graphical explanation in belief networks. *Journal of Computational and Graphical Statistics*, **6**, 160–181.

Almond, Kim, Shute and Ventura (2013). Debugging the evidence chain. *Proceedings of the 2013 UAI Application Workshops (UAI2013AW)*. 1–10. CEUR workshop proceedings, vol 1024. <http://ceur-ws.org/Vol-1024/paper-01.pdf>

**See Also**

Classes: `BNEngine`, `EvidenceSet` `StudentRecord`

Functions in EABN: `accumulateEvidence`, `updateStats`, `updateSM`, `history`

Peanut Functions: `PnodeMargin`

CPTtools Functions `woeHist`, `woeBal`

**Examples**

```
## Requires PNetica
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman, session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[", app, "]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)
```

```

eng <- newBNEngineNDB(app=app,warehouse=Nethouse,
                      listenerSet=ls,manifest=netman,
                      profModel="miniPP_CM",
                      histNodes="Physics",
                      statmat=stattab,
                      activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng,netman)
eng$setHistNodes(character())
configStats(eng,stattab)
setupDefaultSR(eng)

sr1 <- getRecordForUser(eng,"S1")
history(sr1,"Physics")
stopifnot(is.null(history(sr1,"Physics")))

## Now set up history.
eng$setHistNodes("Physics")
PnetCompile(sm(sr1))
sr1 <- baselineHist(eng,sr1)
history(sr1,"Physics")
stopifnot(nrow(history(sr1,"Physics"))==1L)

e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
                  obs=list("CompensatoryObs"="Right"))
sr1 <- updateSM(eng,sr1,e1)
sr1 <- updateHist(eng,sr1,e1)

e2 <- EvidenceSet(uid="S1",app="Test",context="PPconjEM",
                  obs=list("ConjunctiveObs"="Wrong"))
sr1 <- updateSM(eng,sr1,e2)
sr1 <- updateHist(eng,sr1,e2)

history(sr1,"Physics")
stopifnot(nrow(history(sr1,"Phycis"))==3L)
woeHist(history(sr1,"Physics"),pos="High",neg=c("Medium","Low"))

```

---

updateSM

---

*Updates the Student model with additional evidence.*


---

## Description

This function is the core of the EABN algorithm. It finds and attaches the evidence model to the student model, enters the findings from the evidence message, and then detaches the evidence model, leaving the student model updated.

**Usage**

```
updateSM(eng, rec, evidMess, debug = 0)
```

**Arguments**

eng	The <a href="#">BNEngine</a> supervising the operation.
rec	The <a href="#">StudentRecord</a> for the student in question.
evidMess	The <a href="#">EvidenceSet</a> containing the new evidence.
debug	An integer describing how much debugging to do. If set to a number greater than 1, it will issue a call to <a href="#">recover</a> at various stages to aid in debugging models.

**Details**

The update algorithm performs the following step:

1. Finds the evidence model by name according to the context field of the [EvidenceSet](#). See [WarehouseSupply](#).
2. Adjoins the `sm` of the student record with the evidence model and compiles the modified network. See [PnetAdjoin](#) and [PnetCompile](#).
3. Loops over the [observables](#) in the evidence set, if they correspond to nodes in the evidence model, then instantiate their values using [PnodeEvidence](#).
4. Detatch the evidence model and recompile the network. See [PnetDetach](#).

**Value**

The updated student record is returned.

**Author(s)**

Russell Almond

**References**

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapters 5 and 13.

**See Also**

Classes: [BNEngine](#), [PnetWarehouse](#), [StudentRecord](#), [EvidenceSet](#), [Pnet](#)

Functions in EABN: [accumulateEvidence](#), [updateHist](#), [updateStats](#), [getRecordForUser](#)

Peanut Functions: [WarehouseSupply](#), [PnetAdjoin](#), [PnetCompile](#), [PnetDetach](#), [PnodeEvidence](#)

**Examples**

```

## Not run:
## Requires Netica
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path, "testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman, session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[", app, "]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app, warehouse=Nethouse,
                    listenerSet=ls, manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    statmat=stattab,
                    activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng, netman)
eng$setHistNodes("Physics")
configStats(eng, stattab)
setupDefaultSR(eng)

sr1 <- getRecordForUser(eng, "S1")
PnetCompile(sm(sr1))
eap1 <- PnodeEAP(sm(sr1), PnetFindNode(sm(sr1), "Physics"))

e1 <- EvidenceSet(uid="S1", app="Test", context="PPcompEM",
                 obs=list("CompensatoryObs"="Right"))

sr1a <- updateSM(eng, sr1, e1)
eap1a <- PnodeEAP(sm(sr1), PnetFindNode(sm(sr1), "Physics"))
## This should have changed.
stopifnot(abs(eap1-eap1a) > .001)

## End(Not run)

```

---

updateStats	<i>Recalculates statistics for changed student model.</i>
-------------	---

---

### Description

When the student model of a [StudentRecord](#) changes, because the function [updateSM](#) has been run, the statistics need to be recalculated. The function `updateStats` recalculates the statistics. The function `announceStats` lets the listeners know that new statistics are available for this user.

### Usage

```
updateStats(eng, rec, debug = 0)
announceStats(eng, rec)
```

### Arguments

<code>eng</code>	A <a href="#">BNEngine</a> controlling the operation.
<code>rec</code>	A <a href="#">StudentRecord</a> , particularly, one that has just been updated via a call to <a href="#">updateSM</a> .
<code>debug</code>	An integer flag. If the value is greater than 1, there will be a call <a href="#">recover</a> inside of the call.

### Details

The [BNEngine](#) contains a number of [Statistic](#) objects. Every time the student model (`sm`) of the [StudentRecord](#) changes, the `stats` of the record need to be updated as well.

The function `updateStats` simply loops through the statistic collection and calculates the new values. The corresponding field of the student record is then updated.

The function `announceStats` takes the new statistic values and generates a [P4Message](#) containing the new statistics. This is sent to all of the [Listener](#) objects in the [ListenerSet](#) attached to the engine.

The function `stats` returns the latest statistics from the student record.

### Value

The function `updateStats` returns the updated [StudentRecord](#) object.

The function `announceStats` is called for its side effects. Its return value should not be used.

### Author(s)

Russell Almond

## References

Almond, Mislevy, Steinberg, Yan and Williamson (2015). *Bayesian Networks in Educational Assessment*. Springer. Especially Chapters 5 and 13.

## See Also

Classes: [BNEngine](#), [ListenerSet](#) [StudentRecord](#), [Statistic](#), [P4Message](#)

Functions in EABN: [accumulateEvidence](#), [updateHist](#), [updateSM](#), [stats](#)

Peanut Functions: [calcStat](#)

Proc4Functions [notifyListeners](#)

## Examples

```
## Requires PNetica
library(PNetica) ## Must load to setup Netica DLL
app <- "ecd://epls.coe.fsu.edu/EITest"
sess <- RNetica::NeticaSession()
RNetica::startSession(sess)

config.dir <- file.path(library(help="Peanut")$path, "auxdata")
net.dir <- file.path(library(help="PNetica")$path,"testnets")

netman <- read.csv(file.path(config.dir, "Mini-PP-Nets.csv"),
                  row.names=1, stringsAsFactors=FALSE)
stattab <- read.csv(file.path(config.dir, "Mini-PP-Statistics.csv"),
                  as.is=TRUE)

Nethouse <- PNetica::BNWarehouse(netman,session=sess,
                                address=net.dir)

cl <- new("CaptureListener")
listeners <- list("cl"=cl)

ls <- ListenerSet(sender= paste("EAEngine[",app,"]"),
                 db=MongoDB(noMongo=TRUE), listeners=listeners)

eng <- newBNEngineNDB(app=app,warehouse=Nethouse,
                    listenerSet=ls,manifest=netman,
                    profModel="miniPP_CM",
                    histNodes="Physics",
                    statmat=stattab,
                    activeTest="EAActive.txt")

## Standard initialization methods.
loadManifest(eng,netman)
eng$setHistNodes("Physics")
configStats(eng,stattab)
setupDefaultSR(eng)

sr0 <- getRecordForUser(eng,"S1")
```

```
eap0 <- stat(sr0,"Physics_EAP")

e1 <- EvidenceSet(uid="S1",app="Test",context="PPcompEM",
  obs=list("CompensatoryObs"="Right"))

sr1 <- updateRecord(sr0,e1)
sr1 <- updateSM(eng,sr1,e1)
sr1 <- updateStats(eng,sr1)
eap1 <- stat(sr1,"Physics_EAP")

## This should have changed.
stopifnot(abs(eap1-eap0) > .001)

announceStats(eng,sr1)
## Look at the resulting message.
cl$lastMessage()
details(cl$lastMessage())
stopifnot(uid(cl$lastMessage())=="S1",context(cl$lastMessage())=="PPcompEM")
```



# Index

- \* **Bayesian Network**
    - EABN-package, 2
  - \* **Scoring Engine**
    - EABN-package, 2
  - \* **attribute**
    - logIssue, 50
  - \* **classes**
    - BNEngine-class, 8
    - BNEngineMongo-class, 14
    - BNEngineNDB-class, 19
    - EvidenceSet, 32
    - EvidenceSet-class, 33
    - StudentRecord-class, 66
    - StudentRecordSet-class, 70
  - \* **class**
    - StudentRecordSet, 68
  - \* **database**
    - doRunrun, 27
    - getSR, 39
    - mainLoop, 51
  - \* **error**
    - logIssue, 50
  - \* **graphs**
    - accumulateEvidence, 5
    - BNEngineMongo, 11
    - BNEngineNDB, 17
    - doRunrun, 27
    - fetchSM, 35
    - history, 43
    - sm, 62
    - updateSM, 75
  - \* **graph**
    - StudentRecord, 64
    - StudentRecord-class, 66
    - updateHist, 73
    - updateStats, 78
  - \* **interface**
    - accumulateEvidence, 5
    - BNEngineMongo, 11
    - BNEngineNDB, 17
    - doBuild, 23
    - doRunrun, 27
    - EvidenceSet, 32
    - fetchSM, 35
    - getSR, 39
    - loadManifest, 45
    - parseEvidence, 55
    - parseStats, 56
    - parseStudentRecord, 57
    - updateStats, 78
  - \* **manip**
    - accumulateEvidence, 5
    - configStats, 22
    - doBuild, 23
    - getRecordForUser, 37
    - history, 43
    - loadManifest, 45
    - logEvidence, 47
    - mainLoop, 51
    - observables, 53
    - setDefaultSR, 60
    - sm, 62
    - stat, 63
    - trimTable, 72
    - updateHist, 73
    - updateSM, 75
    - updateStats, 78
  - \* **package**
    - EABN-package, 2
- accumulateEvidence, 5, 8, 10, 13, 17, 18, 21, 34, 74, 76, 79
- announceStats, 5, 6, 8, 10, 13, 17, 18, 21, 23, 37, 60, 61
- announceStats (updateStats), 78
- app, BNEngine-method (BNEngine-class), 8
- app, StudentRecord-method (StudentRecord-class), 66

- app, StudentRecordSet-method  
(StudentRecordSet-class), 70
- as.jlist, EvidenceSet, list-method  
(parseEvidence), 55
- as.jlist, StudentRecord, list-method  
(parseStudentRecord), 57
- as.json, 34, 55, 57, 58
- baselineHist, 8–10, 13, 17, 18, 21, 44, 60, 61
- baselineHist (updateHist), 73
- BNEngine, 3, 5, 6, 11–14, 17–19, 21–23, 27,  
28, 31, 37, 38, 41, 44–46, 48, 49, 51,  
53, 54, 60, 61, 72–74, 76, 78, 79
- BNEngine (BNEngine-class), 8
- BNEngine-class, 8
- BNEngineMongo, 3, 6, 8, 10, 11, 11, 12, 13, 18,  
21, 22, 29, 30, 46, 48, 52, 53
- BNEngineMongo-class, 14
- BNEngineNDB, 3, 5, 6, 8, 10, 13, 17, 17, 18, 22,  
29, 30, 46, 48, 53
- BNEngineNDB-class, 19
- BNWarehouse, 24, 26
- buildListener, 29
- buildListenerSet, 31
- BuildNetManifest, 45, 46
- buildObject, 57
- calcStat, 79
- cleanMessageQueue, 31
- clearSRs, 60, 61, 69
- clearSRs (getSR), 39
- clearSRs, StudentRecordSet-method  
(StudentRecordSet-class), 70
- configStats, 8, 10, 13, 17, 18, 21, 22, 24, 26
- context, 60, 73
- context, StudentRecord-method  
(StudentRecord-class), 66
- doBuild, 3, 23, 29, 31
- doRunrun, 3, 26, 27, 30
- EABN (EABN-package), 2
- EABN-package, 2
- EIEvent, 3
- envRefClass, 9, 14, 19, 71
- evidence (logEvidence), 47
- evidence, BNEngineNDB-method  
(BNEngineNDB-class), 19
- evidence, StudentRecord-method  
(StudentRecord-class), 66
- evidence<- (logEvidence), 47
- evidence<-, BNEngineNDB-method  
(BNEngineNDB-class), 19
- evidence<-, StudentRecord-method  
(StudentRecord-class), 66
- EvidenceSet, 5, 6, 8, 9, 16, 19, 20, 32, 32, 33,  
43, 47–49, 51, 53–55, 68, 73, 74, 76
- EvidenceSet-class, 33
- fetchNextEvidence, BNEngine-method  
(BNEngine-class), 8
- fetchNextMessage, 51, 53
- fetchSM, 35, 38, 40, 41, 58, 62
- flog.logger, 6
- flog.threshold, 28
- fromJSON, 25, 28, 37, 40, 55–58
- futile.logger, 26, 31
- generateListenerExports, 29, 31
- getIssues (logIssue), 50
- getIssues, StudentRecord-method  
(logIssue), 50
- getManyRecs, 55, 58
- getOneRec, 55, 58
- getRecordForUser, 5, 6, 8, 10, 13, 17, 18, 21,  
37, 76
- getSR, 8, 37, 38, 39, 69, 70
- getSR, StudentRecordSet-method  
(StudentRecordSet-class), 70
- handleEvidence, 3, 8, 10, 13, 17, 18, 21, 34,  
38, 41, 44, 49, 51, 53, 54
- handleEvidence (accumulateEvidence), 5
- histNames (history), 43
- histNames, StudentRecord-method  
(StudentRecord-class), 66
- history, 43, 73, 74
- history, StudentRecord, character-method  
(StudentRecord-class), 66
- importMessages, 31
- JSONDB, 12, 68, 71
- Listener, 29, 78
- ListenerSet, 11, 13, 17, 18, 29–31, 78, 79
- loadManifest, 8, 10, 13, 17, 18, 21, 45
- logEvidence, 6, 8, 10, 13, 17, 18, 21, 34, 47
- logIssue, 50

- logIssue, StudentRecord, ANY-method  
(logIssue), 50
- logIssue, StudentRecord, character-method  
(logIssue), 50
- m\_id, 40, 48, 49
- mainLoop, 6, 8, 10, 13, 14, 17, 18, 20, 21, 28,  
31, 51
- makeDBuri, 11, 12
- markAsError, 50
- markAsProcessed, 6, 51, 53
- markProcessed, BNEngine-method  
(BNEngine-class), 8
- MessageQueue, 12, 51, 53
- mongo, 67
- MongoDB, 11, 15
- NeticaSession, 24, 27
- newBNEngineMongo (BNEngineMongo), 11
- newBNEngineNDB (BNEngineNDB), 17
- newSR, 69
- newSR (getSR), 39
- newSR, StudentRecordSet-method  
(StudentRecordSet-class), 70
- NNWarehouse, 24, 26
- notifyListeners, 79
- notifyListeners, BNEngine-method  
(BNEngine-class), 8
- observables, 33, 34, 53, 76
- observables, EvidenceSet-method  
(EvidenceSet-class), 33
- Omega2Pnet, 24, 26
- P4Message, 3, 32–34, 49, 54, 78, 79
- parseEvidence, 33, 34, 55
- parseStats, 56
- parseStudentRecord, 57
- Pnet, 35, 46, 58, 62, 65, 67, 71, 72, 76
- PnetAdjoin, 76
- PnetCompile, 60, 61, 74, 76
- PnetDetach, 76
- PnetSerialize, 58
- PnetWarehouse, 8–11, 13, 15–18, 20, 21, 24,  
35, 45, 46, 58, 60, 61, 68, 69, 71, 72,  
76
- Pnode, 43, 73
- PnodeEvidence, 76
- PnodeMargin, 56, 73, 74
- PnodeWarehouse, 24
- Proc4, 3
- processed, 51
- Qmat2Pnet, 24, 26
- read.csv, 72
- recover, 5, 6, 73, 76, 78
- resetListeners, 29, 31
- resetProcessedMessages, 31
- saveSR, 5, 6, 60, 61, 69
- saveSR (getSR), 39
- saveSR, StudentRecordSet-method  
(StudentRecordSet-class), 70
- seqno, 33, 34, 40
- seqno (logEvidence), 47
- seqno, EvidenceSet-method  
(EvidenceSet-class), 33
- seqno, StudentRecord-method  
(StudentRecord-class), 66
- seqno<- (logEvidence), 47
- seqno<-, EvidenceSet-method  
(EvidenceSet-class), 33
- seqno<-, StudentRecord-method  
(StudentRecord-class), 66
- serializeJSON, 55, 58
- setupDefaultSR, 8, 10, 13, 17, 18, 21, 37, 38,  
40, 41, 60, 62, 73
- show, EvidenceSet-method  
(EvidenceSet-class), 33
- show, StudentRecord-method  
(StudentRecord-class), 66
- sm, 22, 58, 60, 62, 73, 76, 78
- sm, StudentRecord-method  
(StudentRecord-class), 66
- sm<- (sm), 62
- sm<-, StudentRecord-method  
(StudentRecord-class), 66
- ssl\_options, 12
- stat, 63
- stat, StudentRecord, character-method  
(StudentRecord-class), 66
- Statistic, 9–11, 15, 20, 22, 23, 57, 64, 78, 79
- statNames (stat), 63
- statNames, StudentRecord-method  
(StudentRecord-class), 66
- stats, 57, 78, 79
- stats (stat), 63

- stats, StudentRecord-method
  - (StudentRecord-class), 66
- stats2json (parseStats), 56
- StudentRecord, 3, 5, 6, 22, 33–35, 37, 38, 40, 41, 43, 44, 47–50, 54, 57, 58, 60–64, 64, 65, 67–74, 76, 78, 79
- StudentRecord-class, 66
- StudentRecordSet, 9–13, 15, 17, 18, 20, 21, 37, 38, 40, 41, 58, 60, 61, 68, 68, 69, 72
- StudentRecordSet-class, 70
- system2, 25
  
- timestamp, 51
- timestamp, StudentRecord-method
  - (StudentRecord-class), 66
- toJSON, 55–58
- toString, EvidenceSet-method
  - (EvidenceSet-class), 33
- toString, StudentRecord-method
  - (StudentRecord-class), 66
- trimTable, 72
  
- uid, 5, 35, 40, 48
- uid, StudentRecord-method
  - (StudentRecord-class), 66
- unboxer, 56, 57
- unpackSM, 62
- unpackSM (fetchSM), 35
- unparseStats (parseStats), 56
- updateHist, 5, 6, 8, 10, 13, 17, 18, 21, 44, 73, 76, 79
- updateRecord, 5, 6, 10
- updateRecord (logEvidence), 47
- updateSM, 5, 6, 8, 10, 13, 17, 18, 21, 73, 74, 75, 78, 79
- updateStats, 5, 6, 8, 10, 13, 17, 18, 21–23, 60, 61, 74, 76, 78
  
- Warehouse, 24, 26
- WarehouseManifest, 45, 46
- WarehouseSupply, 76
- WarehouseUnpack, 35, 65
- withFlogging, 6
- woeBal, 44, 73, 74
- woeHist, 44, 73, 74