Package 'streamConnect'

June 14, 2024

Version 0.0-4

Date 2024-06-13

Encoding UTF-8

Title Connecting Stream Mining Components Using Sockets and Web Services

Description Adds functionality to connect stream mining components from package stream using sockets and Web services. The package can be used create distributed workflows and create plumber-based Web services which can be deployed on most common cloud services.

Depends stream (>= 2.0-0)

Imports plumber, callr, httr, readr, stringr, jsonlite

Suggests knitr, httpuv, processx, rmarkdown

BugReports https://github.com/mhahsler/streamConnect

License GPL-3

VignetteBuilder knitr

RoxygenNote 7.3.1

NeedsCompilation no

Author Michael Hahsler [aut, cre, cph] (<https://orcid.org/0000-0003-2716-1405>)

Maintainer Michael Hahsler <mhahsler@lyle.smu.edu>

Repository CRAN

Date/Publication 2024-06-13 22:30:05 UTC

Contents

`DSC_WebService` *A DSC Interface for a DSC Running as a Web Service*

Description

Provides a DSC front-end for a clusterer running as a web service. The methods `nclusters()`, `get_center()`, `get_weights()` are supported. The request is retried with `httr::RETRY()` if it fails the first time.

Usage

```
DSC_WebService(url, verbose = FALSE, ...)
```

Arguments

`url`	endpoint URI address in the format `http://host:port/<optional_path>`.
`verbose`	logical; display connection information.
`...`	further arguments are passed on to `httr::RETRY()`. Pass `httr::verbose()` as parameter `config` to get detailed connection info.

Value

A stream::DSC object.

See Also

Other WebService: `DSD_ReadWebService()`, `publish_DSC_via_WebService()`, `publish_DSD_via_WebService()`

Other dsc: `publish_DSC_via_WebService()`

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# deploy a clustering process listening for data on the port
rp1 <- publish_DSC_via_WebService("DSC_DBSTREAM(r = .05)", port = port)
rp1

# get a local DSC interface
dsc <- DSC_WebService(paste0("http://localhost", ":", port),
  verbose = TRUE, config = httr::verbose(info = TRUE))
dsc

# cluster
dsd <- DSD_Gaussians(k = 3, d = 2, noise = 0.05)
```

```
update(dsc, dsd, 500)

get_centers(dsc)
get_weights(dsc)

plot(dsc)

# kill the background clustering process.
rp1$kill()
rp1
```

DSD_ReadSocket *A DSD That Reads from a Server Port*

Description

Creates a DSD_ReadStream that reads from a port.

Usage

```
DSD_ReadSocket(host = "localhost", port, retry_args = NULL, ...)
```

Arguments

host	hostname.
port	host port.
retry_args	a list with arguments for retry().
...	further arguments are passed on to DSD_ReadStream().

Value

A stream::DSD object.

See Also

Other Socket: publish_DSD_via_Socket()

Other dsd: DSD_ReadWebService(), publish_DSD_via_Socket(), publish_DSD_via_WebService()

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- DSD_Gaussians(k = 3, d = 3) %>% publish_DSD_via_Socket(port = port)
rp1
```

```
# create a DSD that connects to the socket. Note that we need to
# specify the column names of the stream
dsd <- DSD_ReadSocket(port = port, col.names = c("x", "y", "z", ".class"))
dsd

get_points(dsd, n = 10)

plot(dsd)

close_stream(dsd)

# end the DSD process. Note: that closing the connection above
# may already kill the process.
if (rp1$is_alive()) rp1$kill()
rp1
```

DSD_ReadWebService *A DSD That Reads for a Web Service*

Description

Reads from a web service that published an operation called `get_points` which takes a parameter `n` and returns `n` data points in CSV or json format. The request is retried with `httr::RETRY()` if it fails the first time.

Usage

```
DSD_ReadWebService(url, verbose = FALSE, ...)
```

Arguments

`url`	endpoint URI address in the format `http://host:port/<optional_path>`.
`verbose`	logical; display connection information.
`...`	further arguments are passed on to `httr::RETRY()`. Pass `httr::verbose()` as parameter `config` to get detailed connection info.

Value

A stream::DSD object.

See Also

Other WebService: `DSC_WebService()`, `publish_DSC_via_WebService()`, `publish_DSD_via_WebService()`

Other dsd: `DSD_ReadSocket()`, `publish_DSD_via_Socket()`, `publish_DSD_via_WebService()`

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)", port = port)

## use json for the transport layer instead of csv
# rp1 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)",
#              port = port, serialize = "json")
rp1

# create a DSD that connects to the web service
dsd <- DSD_ReadWebService(paste0("http://localhost", ":", port))
dsd

get_points(dsd, n = 10)

plot(dsd)

# end the DSD process. Note: that closing the connection above
# may already kill the process.
rp1$kill()
rp1
```

`publish_DSC_via_WebService` *Publish a Data Stream Clustering Task via a Web Service*

Description

Uses the package plumber to publish a data stream task as a web service.

Usage

```
publish_DSC_via_WebService(
  dsc,
  port,
  task_file = NULL,
  serializer = "csv",
  serve = TRUE,
  background = TRUE,
  debug = FALSE
)
```

Arguments

dsc	A character string that creates a DSC.
port	port used to serve the task.
task_file	name of the plumber task script file.
serializer	method used to serialize the data. By default csv (comma separated values) is used. Other methods are json and rds (see plumber::serializer_csv).
serve	if TRUE, then a task file is written and a server started, otherwise, only a plumber task file is written.
background	logical; start a background process?
debug	if TRUE, then the service is started locally and a web client is started to explore the interface.

Details

The function writes a plumber task script file and starts the web server to serve the content of the stream using the endpoints

- GET /info
- POST /update requires the data to be uploaded as a file in csv format (see Examples section).
- GET /get_centers with parameter type (see get_centers()).
- GET /get_weights with parameter type (see get_weights()).

Supported serializers are csv (default), json, and rds.

APIs generated using plumber can be easily deployed. See: Hosting. By setting a task_file and serve = FALSE a plumber task script file is generated that can deployment.

Value

a processx::process object created with callr::r_bg() which runs the plumber server in the background. The process can be stopped with rp$kill() or by killing the process using the operating system with the appropriate PID. rp$get_result() can be used to check for errors in the server process (e.g., when it terminates unexpectedly).

See Also

Other WebService: DSC_WebService(), DSD_ReadWebService(), publish_DSD_via_WebService()

Other dsc: DSC_WebService()

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# Deploy a clustering process listening for data on the port
rp1 <- publish_DSC_via_WebService("DSC_DBSTREAM(r = .05)", port = port)
rp1
```

```
# look at ? DSC_WebService for a convenient interface.
# Here we we show how to connect to the port and send data manually.
library(httr)

# the info verb returns some basic information about the clusterer.
resp <- RETRY("GET", paste0("http://localhost:", port, "/info"))
d <- content(resp, show_col_types = FALSE)
d

# create a local data stream and send it to the clusterer using the update verb.
dsd <- DSD_Gaussians(k = 3, d = 2, noise = 0.05)

tmp <- tempfile()
stream::write_stream(dsd, tmp, n = 500, header = TRUE)
resp <- POST(paste0("http://localhost:", port, "/update"),
  body = list(upload = upload_file(tmp)))
unlink(tmp)
resp

# retrieve the cluster centers using the get_centers verb
resp <- GET(paste0("http://localhost:", port, "/get_centers"))
d <- content(resp, show_col_types = FALSE)
head(d)

plot(dsd, n = 100)
points(d, col = "red", pch = 3, lwd = 3)

# kill the process.
rp1$kill()
rp1

# Debug the interface (run the service and start a web interface)
if (interactive())
  publish_DSC_via_WebService("DSC_DBSTREAM(r = .05)",
         port = port, debug = TRUE)
```

`publish_DSD_via_Socket` *Write a Stream to a Socket*

Description

Use a `write_stream()` to write data to a socket connection.

Usage

```
publish_DSD_via_Socket(dsd, port, blocksize = 1024L, background = TRUE, ...)
```

Arguments

dsd	A DSD object.
port	port used to serve the DSD.
blocksize	number of data points pushed on the buffer at once.
background	logical; start a background process?
...	further arguments are passed on to socketConnection().

Details

Provide access to a data stream using a local port.

Blocking: The DSD will be blocked once the buffer is full and resume producing data when it gets unblocked.

This method does not provide a header.

Value

a processx::process object created with callr::r_bg() which runs the plumber server in the background. The process can be stopped with rp$kill() or by killing the process using the operating system with the appropriate PID. rp$get_result() can be used to check for errors in the server process (e.g., when it terminates unexpectedly).

See Also

Other Socket: DSD_ReadSocket()

Other dsd: DSD_ReadSocket(), DSD_ReadWebService(), publish_DSD_via_WebService()

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- DSD_Gaussians(k = 3, d = 3) %>% publish_DSD_via_Socket(port = port)
rp1

# connect to the port (retry waits for the socket to establish)
con <- retry(socketConnection(port = port, open = 'r'))
dsd <- retry(DSD_ReadStream(con, col.names = c("x", "y", "z", ".class")))

get_points(dsd, n = 10)

plot(dsd)

# close connection
close_stream(dsd)

# end the DSD process. Note: that closing the connection above
```

```
# may already kill the process.
rp1$kill()
rp1
```

`publish_DSD_via_WebService` *Publish a Data Stream via a Web Service*

Description

Uses the package plumber to publish a data stream as a web service.

Usage

```
publish_DSD_via_WebService(
  dsd,
  port,
  task_file = NULL,
  serializer = "csv",
  serve = TRUE,
  background = TRUE,
  debug = FALSE
)
```

Arguments

`dsd`	A character string that creates a DSD.
`port`	port used to serve the DSD.
`task_file`	name of the plumber task script file.
`serializer`	method used to serialize the data. By default `csv` (comma separated values) is used. Other methods are `json` and `rds` (see plumber::serializer_csv).
`serve`	if `TRUE`, then a task file is written and a server started, otherwise, only a plumber task file is written.
`background`	logical; start a background process?
`debug`	if `TRUE`, then the service is started locally and a web client is started to explore the interface.

Details

The function writes a plumber task script file and starts the web server to serve the content of the stream using the endpoints

- `http://localhost:port/get_points?n=100` and
- `http://localhost:port/info`.

APIs generated using plumber can be easily deployed. See: Hosting. By setting a `task_file` and `serve = FALSE` a plumber task script file is generated that can be deployment.

A convenient reader for stream data over web services is available as DSD_ReadWebService.

Value

a processx::process object created with `callr::r_bg()` which runs the plumber server in the background. The process can be stopped with `rp$kill()` or by killing the process using the operating system with the appropriate PID. `rp$get_result()` can be used to check for errors in the server process (e.g., when it terminates unexpectedly).

See Also

Other WebService: `DSC_WebService()`, `DSD_ReadWebService()`, `publish_DSC_via_WebService()`

Other dsd: `DSD_ReadSocket()`, `DSD_ReadWebService()`, `publish_DSD_via_Socket()`

Examples

```
# find a free port
port <- httpuv::randomPort()
port

# create a background DSD process sending data to the port
rp1 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)", port = port)
rp1

# connect to the port and read manually. See DSD_ReadWebService for
# a more convenient way to connect to the WebService in R.
library("httr")

# we use RETRY to give the server time to spin up
resp <- RETRY("GET", paste0("http://localhost:", port, "/info"))
d <- content(resp, show_col_types = FALSE)
d

# example: Get 100 points and plot them
resp <- GET(paste0("http://localhost:", port, "/get_points?n=100"))
d <- content(resp, show_col_types = FALSE)
head(d)

dsd <- DSD_Memory(d)
dsd
plot(dsd, n = -1)

# end the DSD process. Note: that closing the connection above
# may already kill the process.
rp1$kill()
rp1

# Publish using json

rp2 <- publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)",
           port = port, serializer = "json")
rp2

# connect to the port and read
```

```
# we use RETRY to give the server time to spin up
resp <- RETRY("GET", paste0("http://localhost:", port, "/info"))
content(resp, as = "text")

resp <- GET(paste0("http://localhost:", port, "/get_points?n=5"))
content(resp, as = "text")

# cleanup
rp2$kill()
rp2

# Debug the interface (run the service and start a web interface)
if (interactive())
  publish_DSD_via_WebService("DSD_Gaussians(k = 3, d = 3)", port = port,
         debug = TRUE)
```

`retry` *Retry an Expression that Fails*

Description

Retries and expression that fails. This is mainly used to retry establishing a connection.

Usage

```
retry(f, times = 5, wait = 1, verbose = FALSE, operation = NULL)
```

Arguments

`f`	expression
`times`	integer; number of times
`wait`	number of seconds to wait in between tries.
`verbose`	logical; show progress and errors.
`operation`	name of the operation used in the error message.

Value

the result of the expression f

Examples

```
retry(1)
```

Index