

Package ‘mergen’

December 8, 2023

Type Package

Title AI-Driven Code Generation, Explanation and Execution for Data Analysis

Version 0.1.0

Description

Employing artificial intelligence to convert data analysis questions into executable code, explanations, and algorithms. The self-correction feature ensures the generated code is optimized for performance and accuracy. 'mergen' features a user-friendly chat interface, enabling users to interact with the AI agent and extract valuable insights from their data effortlessly.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 3.5.0),

Imports openai, rmarkdown, BiocManager, assertthat (>= 0.2.1), httr, jsonlite

Suggests knitr, readxl, data.table (>= 1.9.6), testthat (>= 3.0.0), purrr (>= 0.3.4),

VignetteBuilder knitr

Collate 'setupAgent.R' 'parseBotResponse.R' 'sendPrompt.R' 'executeCode.R' 'selfcorrect.R' 'clean_code_blocks.R' 'extractInstallPkg.R' 'extractFilenames.R' 'fileHeaderPrompt.R' 'test-helper-test_argument_validation.R' 'promptContext.R'

RoxygenNote 7.2.3

Config/testthat/edition 3

NeedsCompilation no

Author Altuna Akalin [aut, cre] (<<https://orcid.org/0000-0002-0468-0117>>),
Jacqueline Anne Jansen [aut]

Maintainer Altuna Akalin <aakalin@gmail.com>

Repository CRAN

Date/Publication 2023-12-08 11:20:05 UTC

R topics documented:

check_install	2
clean_code_blocks	3
executeCode	3
extractCode	4
extractFileNames	5
extractInstallPkg	5
fileHeaderPrompt	6
promptContext	7
selfcorrect	7
sendPrompt	8
setupAgent	9

Index	11
--------------	-----------

check_install	<i>Check and Install R Package</i>
---------------	------------------------------------

Description

This function checks if an R package is installed, and if not, attempts to install it using either the standard CRAN repository or the Bioconductor repository.

Usage

```
check_install(package_name)
```

Arguments

package_name A character string specifying the name of the package to be checked and installed.

Value

No value returned. Called for installation of package.

Examples

```
# Check and install "dplyr" package
check_install("dplyr")
```

clean_code_blocks	<i>Clean code blocks returned by the agent</i>
-------------------	--

Description

This function cleans up the response returned by the agent to ensure code blocks can run. It ensures that characters such as 'R' and 'r' are cleaned from code blocks in the agents response, so that the code blocks are able to be extracted by the extractCode() function and ran as expected. It also cleans the response from any install.package calls, and recorded output, so that when code blocks are extracted, the code can run smoothly.

Usage

```
clean_code_blocks(response)
```

Arguments

response response received from the agent

Value

A string holding the response of the agent, cleaned from any unwanted characters.

Examples

```
{
  response <- "To perform PCA, do the following: ```R prcomp(data)``` This funcion will perform PCA."
  clean_code <- clean_code_blocks(response)
}
```

executeCode	<i>execute code</i>
-------------	---------------------

Description

The function executes a chunk of code either in the current working environment or saves the output as an HTML file to be rendered as a part of a web page

Usage

```
executeCode(code, output = "eval", output.file = NULL)
```

Arguments

code	code chunk as text, without any decorators or HTML-specific characters.
output	If the output is "eval", the code is executed as is. If the output is "html", the code is not executed.
output.file	If the output is "html", user can provide a file name for the html. If not provided a temporary file will be created.

Value

If the output is "eval": if running the code causes errors, errors are returned. Otherwise NA is returned. If output is "html", output file is returned.

extractCode	<i>extract the code and text from the text returned by LLM agent</i>
-------------	--

Description

This function parses the agents answer and returns the text and code as single blocks. The results can be used for code execution and might be useful for displaying purposes later on.

Usage

```
extractCode(text, delimiter = "`")
```

Arguments

text	A character string containing the text with embedded code blocks.
delimiter	A character string representing the delimiter used to enclose the code blocks (default: "`").

Value

A list with two elements: 'code' and 'text'. 'code' contains the concatenated code blocks, and 'text' contains the remaining text with code blocks removed.

Examples

```
text <- "\n\nThe following, normalize the table and do PCA.\n\n```\ndata <- read.table(\"test.txt\", header = TRUE, sep = \"\\t\")\n```\n\nresult <- extractCode(text)\nprint(result$code)\nprint(result$text)
```

extractFileNames	<i>Extract file names from user prompt</i>
------------------	--

Description

This function extracts file names from the user prompt. Current filenames that are supported by this function are *.txt, *.tsv, *.csv *.xls, *.xlsx, *.bed, *.bigWig, *.bw and *.bigBed. Other filenames will not be extracted. If no filenames are found, the function will return NA.

Usage

```
extractFileNames(text)
```

Arguments

text	user prompt
------	-------------

Value

A list holding file names from the user prompt.

Examples

```
{  
  user_prompt <- "How do I perform PCA on data in my file called test.txt?"  
  extractFileNames(text=user_prompt)  
}
```

extractInstallPkg	<i>extract package names and install them</i>
-------------------	---

Description

This function extracts all package names that are needed to run the code returned by the agent and installs them as needed.

Usage

```
extractInstallPkg(code)
```

Arguments

code	code block returned by the agent.
------	-----------------------------------

Value

No value returned. Called for installation of package.

Examples

```
# Check code for packages that need installing
code <- "library(devtools)\n x<-5"
extractInstallPkg(code)
```

fileHeaderPrompt	<i>Extract file headers from files in prompt</i>
------------------	--

Description

This function extracts file headers from files. Recommended is to use this function with the results from [extractFileNames](#)

Usage

```
fileHeaderPrompt(filenamees)
```

Arguments

filenamees list containing file names.

Value

A string containing the file headers of the files in "filenamees" list

Examples

```
## Not run:
prompt <- "how do I perform PCA on data in a file called test.txt?"
filenamees <- extractFileNames(prompt)
fileHeaderPrompt(filenamees)

## End(Not run)
```

promptContext	<i>Predefined prompt contexts for prompt engineering</i>
---------------	--

Description

This function holds various predefined prompt contexts that can be used for prompt engineering.

Usage

```
promptContext(type = "simple")
```

Arguments

type	specifies the type of context you wish to be returned. Valid options are "simple", "actAs", "CoT" and "rbionfoExp"
------	--

Value

A string holding the predefined context.

selfcorrect	<i>Self correct the code returned by the agent</i>
-------------	--

Description

The function attempts to correct the code returned by the agent by re-feeding to the agent with the error message. If there are no error messages function returns the original response.

Usage

```
selfcorrect(
  agent,
  prompt,
  context = rbionfoExp,
  attempts = 3,
  output.file = NULL,
  ...
)
```

Arguments

agent	An object containing the agent's information (e.g., type and model).
prompt	The prompt text to send to the language model.
context	Optional context to provide alongside the prompt (default is rbionfoExp).
attempts	Numeric value denoting how many times the code should be sent back for fixing.
output.file	Optional output file created holding parsed code
...	Additional arguments to be passed to the sendPrompt function.

Value

A list containing the following elements:

`init.response` A character vector representing the initial prompt response.
`init.blocks` A list of initial blocks.
`final.blocks` A list of final blocks.
`code.works` A boolean value indicating whether the code works.
`exec.result` A character string representing the execution results.
`tried.attempts` An integer representing the number of attempts.

See Also

[promptContext](#) for predefined contexts to use.

Examples

```
## Not run:

response <- selfcorrect(agent,prompt,context=rbionfoExp, max_tokens = 500)

## End(Not run)
```

sendPrompt	<i>Send a prompt to a specified language model agent and return the response.</i>
------------	---

Description

Send a prompt to a specified language model agent and return the response.

Usage

```
sendPrompt(
  agent,
  prompt,
  context = rbionfoExp,
  return.type = c("text", "object"),
  ...
)
```

Arguments

agent	An object containing the agent's information (e.g., type and model etc.).
prompt	The prompt text to send to the language model.
context	Optional context to provide alongside the prompt (default is rbionfoExp).
return.type	The type of output to return, either the text response ("text") or the entire response object ("object").
...	Additional arguments to be passed to the prompt function.

Value

The text response or the entire response object, based on the specified return type.

See Also

[promptContext](#) for predefined contexts to use.

Examples

```
## Not run:
agent <- setupAgent(name="openai", type="chat", model="gpt-4",
                   ai_api_key=Sys.getenv("OPENAI_API_KEY"))
prompt <- "tell me a joke"
response <- sendPrompt(agent, prompt)

response <- sendPrompt(agent, prompt, context=rbionfoExp, return.type="text", max_tokens = 500)

## End(Not run)
```

 setupAgent

set up an online LLM API for subsequent tasks

Description

This function sets up an large language model API for tasks.

Usage

```
setupAgent(
  name = c("openai", "replicate"),
  type = NULL,
  model = NULL,
  ai_api_key = Sys.getenv("AI_API_KEY")
)
```

Arguments

name	Name of the API you want to use. Currently supported APIs are "openai" and "replicate"
type	Specify type of model (chat or completion). This parameter only needs to be specified when using 'openai'
model	LLM model you wish to use. For openAI chat model examples are: <ul style="list-style-type: none"> • 'gtp-3-5-turbo' • 'gtp-4' For openAI completion models examples are: <ul style="list-style-type: none"> • 'text-curie-001'

- 'text-davinci-002'

For replicate models examples are:

- llama-2-70b-chat (as '02e509c789964a7ea8736978a43525956ef40397be9033abf9fd2badfe68c9e3'
- llama-2-13b-chat (as 'f4e2de70d66816a838a89eeeb621910adffb0dd0baba3976c96980970978018d'

For a full list of openAI models [click here](#). For a full list of Replicate models, [click here](#).

ai_api_key personal API key for accessing LLM

Value

A list holding agent information.

Examples

```
{
myAgent <- setupAgent(name="openai", type="chat", model="gpt-4", ai_api_key="my_key")

myAgent <- setupAgent(name="replicate", type=NULL,
                      model="02e509c789964a7ea8736978a43525956ef40397be9033abf9fd2badfe68c9e3",
                      ai_api_key="my_key")
}
```

Index

check_install, [2](#)
clean_code_blocks, [3](#)

executeCode, [3](#)
extractCode, [4](#)
extractFileNames, [5](#), [6](#)
extractInstallPkg, [5](#)

fileHeaderPrompt, [6](#)

promptContext, [7](#), [8](#), [9](#)

selfcorrect, [7](#)
sendPrompt, [7](#), [8](#)
setupAgent, [9](#)