Package ‘mergen’

June 27, 2024

Type Package

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

Version 0.2.1

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.

URL https://github.com/BIMSBbioinfo/mergen, https://bioinformatics.mdc-berlin.de/mergen/

BugReports https://github.com/BIMSBbioinfo/mergen/issues

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' 'runCodeInResponse.R'

RoxygenNote 7.3.1

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 2024-06-27 10:00:06 UTC

Contents

`check_install` *Check and Install R Package*

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

TRUE if the package is already installed or can be installed. FALSE otherwise

Examples

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

clean_code_blocks *Clean code blocks returned by the agent*

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 *execute code*

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` *extract the code and text from the text returned by LLM agent*

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```"
result <- extractCode(text)
print(result$code)
print(result$text)
````

extractFilenames *Extract file names from user prompt*

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 *extract package names and install them*

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

final status of the packages as a logical vector, TRUE if packages is already installed or could be installed. FALSE if the package can't be install.

Examples

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

---

fileHeaderPrompt *Extract file headers from files in prompt*

Description

This function extracts file headers from files. Recommended is to use this function with the results from extractFilenames

Usage

```
fileHeaderPrompt(filenames)
```

Arguments

filenames	list containing file names.

Value

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

Examples

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

## End(Not run)
```

promptContext *Predefined prompt contexts for prompt engineering*

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.

runCodeInResponse *Executes the code in the received response from the agent*

Description

The function extracts and executes the code in the response. If required it can try to correct errors in the code via different strategies.

Usage

```
runCodeInResponse(
  response,
  prompt = NULL,
  agent = NULL,
  context = NULL,
  correction = c("none", "selfcorrect", "sampleMore", "sampleThenCorrect",
    "correctThenSample"),
  attempts = 3,
  output.file = NULL,
  ...
)
```

Arguments

`response` response to be parsed for code and executed

`prompt` prompt for the response, if correction="none" it is not needed

`agent` AI agent, if correction="none" it is not needed

`context` context for the prompt, if correction="none" it is not needed

`correction` "none" no code correction is needed. "selfcorrect" feedback the errors to LLM and ask for fix. "sampleMore" sample responses for the #prompt until an executable code is returned or number of attempts reached. "correctThenSample" first try self-correct "attempt" number of times. If no executable code is returned. It will sample new responses "attempt" number of times or until executable code

`attempts` Numeric value denoting how many times the code should be sent back for fixing.

`output.file` Optional output file created holding parsed code

`...` arguments to sendPrompt()

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

`selfcorrect`,`sampleResponse`

Examples

```
## Not run:

resp.list <- runCodeInResponse(agent,prompt,context=rbionfoExp,correction="sampleMore",attempt=2)

## End(Not run)
```

`sampleResponse` *Sample more solutions when non-executable code returned by the agent*

Description

The function attempts to sample more solutions by the agent when the code returned by the agent is faulty. The function simply asks for solutions until an executable code is returned or until number of attempts is reached.

Usage

```
sampleResponse(
  agent,
  prompt,
  context = rbionfoExp,
  attempts = 3,
  output.file = NULL,
  responseWithError = 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

`responseWithError`
a list of response and errors returned from executeCode(). First element is expected to be the response and the second element is the error list returned by executeCode().

`...` 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:

resp.list <- sampleResponse(agent,prompt,context=rbionfoExp, max_tokens = 500)

## End(Not run)
```

`selfcorrect` *Self correct the code returned by the agent*

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,
  responseWithError = NULL,
  history = 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

`responseWithError`
a list of response and errors returned from executeCode().

`history` a list of previous response and prompts. Default is NULL and should be stayed as is for most use cases. First element is expected to be the response and the second element is the error list returned by executeCode().

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

Description

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

Usage

```
sendPrompt(
  agent,
  prompt,
  context = promptContext(type = ”simple”),
  return.type = c(”text”, ”object”),
  previous.msgs = NULL,
  ...
)
```

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 promptContext(type = "simple")).

`return.type` The type of output to return, either the text response ("text") or the entire response object ("object").

`previous.msgs` a list of lists for previous prompts and responses. Useful to add context of the previous messages for the API. this argument works with openai and generic models, but not with replicate API. Default: NULL

`...` Additional arguments to be passed to the LLM API. Such as maximum tokens, ("max_tokens"), to be returned. Users can also also provide other arguments in openai API-like arguments documented on [the official documentation](https://platform.openai.com/docs/reference/chat/create). Other APIs are also following similar argument naming patterns.

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,context="")

# increase tokens, it is important for getting longer responses
response <- sendPrompt(agent,prompt,context="",return.type="text", max_tokens = 500)

# get previous messages into the context
prompt="what about 2010?"
response <- sendPrompt(agent,prompt,context="",
                       return.type="text",
                       previous.msgs=list(
                                          list(
                                               "role" = "user",
                                                "content" = "Who won the world
                                                 series in 2020?"
                                                 ),
                                          list(
                                                "role" = "assistant",
                                              "content" = "The Los Angeles Dodgers"
                                              )
```

```
                              )
                    )


## 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", "generic"),
  type = NULL,
  model = NULL,
  url = NULL,
  ai_api_key = Sys.getenv("AI_API_KEY")
)
```

Arguments

`name` A string for the name of the API, one of "openai", "replicate" or "generic". Currently supported APIs are "openai" and "replicate". If the user wishes to use another API that has similar syntax to openai API this is also supported via the the "generic" option. In this case, the user should also provide a url for the API using the

`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:

- ’gtp-3-5-turbo’
- ’gtp-4’

For openAI completion models examples are:

- ’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 see https://platform.openai.com/docs/models/overview/.
For a full list of Replicate models, see https://replicate.com/collections/language-models.

`url` the url for the API in case the API "generic" is selected. (Default: NULL)

`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