Package 'llmflow'

January 31, 2026

Type Package

Title Reasoning and Acting Workflow for Automated Data Analysis

Version 3.0.2

Description Provides a framework for integrating Large Language Models (LLMs) with R programming through workflow automation. Built on the ReAct (Reasoning and Acting) architecture, enables bi-directional communication between LLMs and R environments. Features include automated code generation and execution, intelligent error handling with retry mechanisms, persistent session management, structured JSON output validation, and context-aware conversation management.

License GPL (>= 3)

Encoding UTF-8

URL https://github.com/Zaoqu-Liu/llmflow

BugReports https://github.com/Zaoqu-Liu/llmflow/issues

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Imports callr, cli, glue, jsonlite, jsonvalidate

Suggests ellmer, testthat (>= 3.0.0)

Config/testthat/edition 3

NeedsCompilation no

Author Zaoqu Liu [aut, cre] (ORCID: <https://orcid.org/0000-0002-0452-742X>)

Maintainer Zaoqu Liu <liuzaoqu@163.com>

Repository CRAN

Date/Publication 2026-01-31 18:50:12 UTC

Contents

`AutoFlow` *AutoFlow - Automated R Analysis Workflow with LLM*

Description

AutoFlow - Automated R Analysis Workflow with LLM

Usage

```
AutoFlow(
  react_llm,
  task_prompt,
  rag_llm = NULL,
  max_turns = 15,
  pkgs_to_use = c(),
  objects_to_use = list(),
  existing_session = NULL,
  verbose = TRUE,
  r_session_options = list(),
  context_window_size = 3000,
  max_observation_length = 800,
  error_escalation_threshold = 3
)
```

Arguments

`react_llm`	Chat object for ReAct task execution (required)
`task_prompt`	Task description (required)
`rag_llm`	Chat object for RAG documentation retrieval (default: NULL, uses react_llm)

`max_turns` Maximum ReAct turns (default: 15)

`pkgs_to_use` Packages to load in R session

`objects_to_use` Named list of objects to load

`existing_session`
Existing callr R session

`verbose` Verbose output (default: TRUE)

`r_session_options`
Options for callr R session

`context_window_size`
Context window size for history

`max_observation_length`
Maximum observation length

`error_escalation_threshold`
Error count threshold

Details

Dual-LLM Architecture:

AutoFlow supports using different models for different purposes: - ‘rag_llm‘: Retrieval-Augmented Generation - retrieves relevant function documentation - ‘react_llm‘: ReAct execution - performs reasoning and action loops

Why separate models? - RAG tasks are simple (extract function names) - use fast/cheap models - ReAct tasks are complex (coding, reasoning) - use powerful models - Cost savings: ~70

If ‘rag_llm‘ is NULL, both operations use ‘react_llm‘.

Value

ReAct result object

Examples

```
## Not run:
# Simple: same model for both
llm <- llm_openai(model = "gpt-4o")
result <- AutoFlow(llm, "Load mtcars and plot mpg vs hp")

# Optimized: lightweight RAG, powerful ReAct
rag <- llm_openai(model = "gpt-3.5-turbo") # Fast & cheap
react <- llm_openai(model = "gpt-4o") # Powerful
result <- AutoFlow(
  react_llm = react,
  task_prompt = "Perform PCA on iris dataset",
  rag_llm = rag
)

# Cross-provider: DeepSeek RAG + Claude ReAct
rag <- chat_deepseek(model = "deepseek-chat")
react <- chat_anthropic(model = "claude-sonnet-4-20250514")
```

```
result <- AutoFlow(react, "Complex analysis", rag_llm = rag)

# Batch evaluation with shared RAG
rag <- chat_deepseek(model = "deepseek-chat")
react <- chat_openai(model = "gpt-4o")

for (task in tasks) {
  result <- AutoFlow(react, task, rag_llm = rag, verbose = FALSE)
}

## End(Not run)
```

`extract_bash_code` *Extract Bash/Shell code from a string*

Description

This function extracts Bash/Shell code from a string by matching all content between '```bash', '```sh', '```shell' and '```'.

Usage

```
extract_bash_code(input_string)
```

Arguments

`input_string`	A string containing Bash/Shell code blocks, typically a response from an LLM

Value

A character vector containing the extracted Bash/Shell code

Examples

```
# Simple bash example
text <- "Run this:\n```bash\necho 'Hello'\n```"
extract_bash_code(text)

# Using 'sh' tag
text <- "```sh\nls -la\npwd\n```"
extract_bash_code(text)

# Using 'shell' tag
text <- "```shell\nfor i in {1..5}; do echo $i; done\n```"
extract_bash_code(text)

# Multiple blocks with different tags
response <- "
Setup script:
```bash
```
```

```
#!/bin/bash
mkdir -p /tmp/test
cd /tmp/test
```

Installation:
```sh
apt-get update
apt-get install -y git
```

Configuration:
```shell
export PATH=$PATH:/usr/local/bin
source ~/.bashrc
```
"
codes <- extract_bash_code(response)
length(codes) # Returns 3

# Complex script example
script_response <- "
Here's a backup script:
```bash
#!/bin/bash

# Set variables
BACKUP_DIR='/backup'
DATE=$(date +%Y%m%d)

# Create backup
tar -czf ${BACKUP_DIR}/backup_${DATE}.tar.gz /home/user/

# Check if successful
if [ $? -eq 0 ]; then
    echo 'Backup completed successfully'
else
    echo 'Backup failed'
    exit 1
fi
```
"
extract_bash_code(script_response)
```

---

## extract_chat_history *Extract chat history from ellmer chat object*

---

### Description

Extract chat history from ellmer chat object
```

Usage

```
extract_chat_history(
  chat_obj,
  include_tokens = TRUE,
  include_time = TRUE,
  tz = "Asia/Shanghai"
)
```

Arguments

chat_obj	An ellmer chat object
include_tokens	Whether to include token information
include_time	Whether to include timestamp information
tz	Time zone for timestamps (default "Asia/Shanghai" for CST)

Value

A data frame with chat history

---

extract_code *Generic function to extract code of any specified language*

Description

This function provides a flexible way to extract code blocks of any language from a string by specifying the language identifier(s).

Usage

```
extract_code(input_string, language, case_sensitive = FALSE)
```

Arguments

input_string	A string containing code blocks
language	Language identifier(s) to extract (e.g., "r", "python", c("bash", "sh"))
case_sensitive	Whether the language matching should be case-sensitive (default: FALSE)

Value

A character vector containing the extracted code

Examples

```
# Extract R code
text <- "```r\nx <- 1:10\n```"
extract_code(text, "r")

# Extract multiple language variants
text <- "```bash\necho 'test'\n```\n```sh\nls -la\n```"
extract_code(text, c("bash", "sh"))

# Case-sensitive extraction
text <- "```R\nplot(1:10)\n```\n```r\nprint('hello')\n```"
extract_code(text, "r", case_sensitive = TRUE) # Only matches lowercase 'r'
extract_code(text, "r", case_sensitive = FALSE) # Matches both 'R' and 'r'

# Extract custom language
text <- "```julia\nprintln(\"Julia code\")\n```"
extract_code(text, "julia")

# Extract YAML configuration
config_text <- "
Here's the configuration:
```yaml
database:
  host: localhost
  port: 5432
  name: mydb
```
"
extract_code(config_text, "yaml")

# Extract multiple TypeScript and JavaScript blocks
mixed_text <- "
TypeScript:
```typescript
interface User {
    name: string;
    age: number;
}
```

JavaScript:
```js
const user = {name: 'John', age: 30};
```
"
# Extract TypeScript
extract_code(mixed_text, "typescript")
# Extract both TypeScript and JavaScript
extract_code(mixed_text, c("typescript", "js"))
```

`extract_function_examples` *Extract Examples from a Package Function*

Description

This function extracts and cleans the examples section from a specific function's documentation in an R package. It uses the 'tools' package to access the Rd database and extracts examples using 'tools::Rd2ex()'. The output is cleaned to remove metadata headers and formatting artifacts.

Usage

```
extract_function_examples(package_name, function_name)
```

Arguments

`package_name`	A character string specifying the name of the package
`function_name`	A character string specifying the name of the function

Value

A character string containing the cleaned examples code, or 'NA' if no examples are found or an error occurs

Examples

```
## Not run:
# Extract examples from ggplot2's geom_point function
examples <- extract_function_examples("ggplot2", "geom_point")
cat(examples)

## End(Not run)
```

`extract_javascript_code` *Extract JavaScript code from a string*

Description

This function extracts JavaScript code from a string by matching all content between '"'javascript', '"'js', '"'jsx' and '"'.

Usage

```
extract_javascript_code(input_string)
```

Arguments

input_string	A string containing JavaScript code blocks, typically a response from an LLM

Value

A character vector containing the extracted JavaScript code

Examples

```
# Simple JavaScript example
text <- "Code:\n```javascript\nconsole.log('Hello');\n```"
extract_javascript_code(text)

# Using 'js' tag
text <- "```js\nconst x = 42;\n```"
extract_javascript_code(text)

# Using 'jsx' tag for React
text <- "```jsx\n<div>Hello World</div>\n```"
extract_javascript_code(text)

# Multiple blocks with different tags
response <- "
Frontend code:
```javascript
function fetchData() {
    return fetch('/api/data')
        .then(response => response.json());
}
```

React component:
```jsx
const MyComponent = () => {
    const [data, setData] = useState([]);

    useEffect(() => {
        fetchData().then(setData);
    }, []);

    return (
        <div>
            {data.map(item => <p key={item.id}>{item.name}</p>)}
        </div>
    );
};
```

Node.js backend:
```js
const express = require('express');
const app = express();
```
```

```
app.get('/api/data', (req, res) => {
    res.json([{id: 1, name: 'Item 1'}]);
});

app.listen(3000);
```
```
"
codes <- extract_javascript_code(response)
length(codes) # Returns 3
```

`extract_json` *Extract and parse JSONs from a string (LLM response)*

Description

This function extracts JSON blocks from a string and parses them using 'jsonlite::fromJSON()'. This can be used to extract all JSONs from LLM responses, immediately converting them to R objects.

Usage

```
extract_json(llm_response)
```

Arguments

`llm_response`	A character string

Details

CRITICAL FIX: Now uses simplifyVector = FALSE to preserve array structure. This ensures that JSON arrays remain as R lists, preventing single-element arrays from being simplified to character vectors. This is essential for proper schema validation when used with auto_unbox = TRUE in toJSON().

Value

A list of parsed JSON objects

`extract_python_code` *Extract Python code from a string*

Description

This function extracts Python code from a string by matching all content between ’“‘python’, ’“‘py’ and ’“‘’.

Usage

```
extract_python_code(input_string)
```

Arguments

`input_string`	A string containing Python code blocks, typically a response from an LLM

Value

A character vector containing the extracted Python code

Examples

```
# Simple example
text <- "Python code:\n```python\nprint('Hello World')\n```"
extract_python_code(text)

# Using 'py' tag
text <- "```py\nimport numpy as np\n```"
extract_python_code(text)

# Multiple blocks with different tags
response <- "
Data processing:
```python
import pandas as pd
df = pd.read_csv('data.csv')
df.head()
```

Visualization:
```py
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6])
plt.show()
```
"
codes <- extract_python_code(response)
length(codes) # Returns 2

Complex example with classes and functions
```

```
llm_response <- "
Here's a complete Python solution:
```python
class DataProcessor:
    def __init__(self, data):
        self.data = data

    def process(self):
        return [x * 2 for x in self.data]

processor = DataProcessor([1, 2, 3])
result = processor.process()
print(result)
```
"
extract_python_code(llm_response)
```

`extract_r_code` *Extract R code from a string*

Description

This function extracts R code from a string by matching all content between '"`r' or '"`R' and '"`'.

Usage

```
extract_r_code(input_string)
```

Arguments

`input_string`	A string containing R code blocks, typically a response from an LLM

Value

A character vector containing the extracted R code

Examples

```
# Simple example
text <- "Here is some R code:\n```r\nprint('Hello')\n```"
extract_r_code(text)

# Multiple code blocks
response <- "
First block:
```r
x <- 1:10
mean(x)
```
```

```
Second block:
```R
library(ggplot2)
ggplot(mtcars, aes(mpg, hp)) + geom_point()
```
"
codes <- extract_r_code(response)
length(codes) # Returns 2

# With surrounding text
llm_response <- "
To calculate the mean, use this code:
```r
data <- c(1, 2, 3, 4, 5)
result <- mean(data)
print(result)
```
The result will be 3.
"
extract_r_code(llm_response)
```

`extract_sql_code` *Extract SQL code from a string*

Description

This function extracts SQL code from a string by matching all content between "```sql" and "```" (case-insensitive).

Usage

```
extract_sql_code(input_string)
```

Arguments

`input_string`	A string containing SQL code blocks, typically a response from an LLM

Value

A character vector containing the extracted SQL code

Examples

```
# Simple SQL query
text <- "Query:\n```sql\nSELECT * FROM users;\n```"
extract_sql_code(text)

# Case-insensitive matching
```

```
text <- "```SQL\nSELECT COUNT(*) FROM orders;\n```"
extract_sql_code(text)

# Multiple SQL blocks
response <- "
Create table:
```sql
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(100),
    department VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

Insert data:
```sql
INSERT INTO employees (id, name, department, salary)
VALUES
    (1, 'John Doe', 'IT', 75000),
    (2, 'Jane Smith', 'HR', 65000);
```

Query data:
```sql
SELECT name, salary
FROM employees
WHERE department = 'IT'
ORDER BY salary DESC;
```
"
codes <- extract_sql_code(response)
length(codes) # Returns 3

# Complex query with joins
complex_query <- "
Here's the analysis query:
```sql
WITH monthly_sales AS (
    SELECT
        DATE_TRUNC('month', order_date) as month,
        SUM(total_amount) as total_sales,
        COUNT(DISTINCT customer_id) as unique_customers
    FROM orders
    WHERE order_date >= '2024-01-01'
    GROUP BY DATE_TRUNC('month', order_date)
)
SELECT
    month,
    total_sales,
    unique_customers,
    total_sales / unique_customers as avg_per_customer
FROM monthly_sales
```
```

```
ORDER BY month;
```
"
extract_sql_code(complex_query)
```

---

package_extraction_prompt *Generate Function Extraction Prompt for LLM Analysis*

---

## Description

Creates a highly refined prompt that guides LLMs to identify ONLY the most documentation-critical, domain-specific R functions from a task description. The prompt uses sophisticated filtering criteria to exclude common, well-known functions (like read.csv, mean, order) that any LLM can use correctly without explicit documentation, focusing instead on specialized functions where examples truly add value.

## Usage

```
package_extraction_prompt(
  task_description,
  include_criteria = NULL,
  exclude_criteria = NULL,
  prioritization_factors = NULL,
  emphasis = NULL
)
```

Arguments

`task_description`
: Character string. Detailed description of the R task or analysis workflow that needs to be performed. Should include: - Data types and sources involved - Analytical objectives and methods - Expected outputs or deliverables - Domain-specific context (e.g., bioinformatics, spatial analysis) The more domain-specific the description, the better the function selection.

`include_criteria`
: Character vector. Additional inclusion criteria beyond the defaults. Specify domain-specific requirements or function characteristics that should be documented. Default is NULL (use standard criteria).

`exclude_criteria`
: Character vector. Additional exclusion criteria beyond the defaults. Specify function types or patterns that should be skipped (e.g., "Basic ggplot2 themes", "Standard dplyr verbs"). Default is NULL.

`prioritization_factors`
: Character vector. Additional factors for prioritizing functions beyond the defaults. Specify what makes certain functions more important to document. Default is NULL (use standard priorities).
```

`emphasis` Character string. Additional emphasis or context to guide the extraction process. Use this to highlight specific aspects of the task or to emphasize certain types of functions. Default is NULL.

Details

This function applies a "documentation necessity test": only include functions where a proficient LLM would struggle without explicit documentation and examples. This dramatically improves output quality and reduces token waste.

The enhanced prompt applies a rigorous "documentation necessity test" with four key questions:

1. Would a proficient LLM struggle without documentation? 2. Is this function domain-specific or universally known? 3. Does it use specialized terminology or workflows? 4. Would examples significantly improve usage accuracy?

Automatic exclusions (common functions that waste tokens): - Data I/O: read.csv, write.csv, readLines - Basic operations: order, sort, subset, head, tail - Simple statistics: mean, median, sd, sum - Core structures: c, list, data.frame - Well-known tidyverse: simple dplyr::filter, dplyr::mutate - Basic control flow: if, for, while - Common utilities: paste, grep, unique

What gets included (documentation-critical functions): - Domain-specific methods (clusterProfiler::enrichGO for GO analysis) - Complex statistical procedures (DESeq2::DESeq) - Specialized transformations (sf::st_transform for spatial data) - Functions with many non-obvious parameters - Methods where wrong usage produces plausible but incorrect results

This approach ensures that "GO enrichment analysis" returns clusterProfiler functions, NOT read.csv or order.

Value

Character string containing the complete extraction prompt with:

- Clear documentation necessity principle
- Strict inclusion criteria for domain-specific functions
- Comprehensive exclusion rules with concrete examples
- Four-question decision heuristic for each function
- Concrete good/bad examples from multiple domains
- Prioritization by domain specialization and complexity
- Quality-over-quantity guidance

See Also

`retrieve_docs` for using this prompt in documentation extraction

Examples

```
# Basic usage
prompt <- package_extraction_prompt(
  "Perform GO enrichment analysis on differentially expressed genes"
)
```

```
# With domain-specific guidance
prompt <- package_extraction_prompt(
  task_description = "Single-cell RNA-seq analysis with Seurat",
  include_criteria = c(
    "Seurat-specific normalization and scaling methods"
  ),
  exclude_criteria = c(
    "Standard dplyr data manipulation"
  )
)

## Not run:
# Use with retrieve_docs (requires LLM client)
docs <- retrieve_docs(
  chat_obj = llm,
  prompt = package_extraction_prompt(
    task_description = "Perform differential expression analysis"
  )
)

## End(Not run)
```

`package_function_schema` *Create JSON Schema for Package Function Validation*

Description

Create JSON Schema for Package Function Validation

Usage

```
package_function_schema(
  min_functions = 0,
  max_functions = 10,
  description = NULL
)
```

Arguments

`min_functions`	Integer. Minimum number of functions (default: 0 to allow empty)
`max_functions`	Integer. Maximum number of functions (default: 10)
`description`	Character string. Custom description

Value

List containing JSON schema

prompt_from_history *Build prompt from chat history*

Description

Build prompt from chat history

Usage

```
prompt_from_history(
  chat_obj,
  add_text = NULL,
  add_role = "user",
  start_turn_index = 1
)
```

Arguments

chat_obj	Chat object
add_text	Additional text to append
add_role	Role for add_text ("user" or "assistant")
start_turn_index	Starting turn index (default 1)

Value

Formatted prompt string

react_r *Simplified interface - Enhanced react_r*

Description

Simplified interface - Enhanced react_r

Usage

```
react_r(chat_obj, task, ...)
```

Arguments

chat_obj	Chat object
task	Task description
...	Additional arguments passed to react_using_r

Value

Formatted result display

`react_using_r` *ReAct (Reasoning and Acting) using R code execution - Optimized Version*

Description

ReAct (Reasoning and Acting) using R code execution - Optimized Version

Usage

```
react_using_r(
  chat_obj,
  task,
  max_turns = 15,
  pkgs_to_use = c(),
  objects_to_use = list(),
  existing_session = NULL,
  verbose = TRUE,
  r_session_options = list(),
  context_window_size = 3000,
  max_observation_length = 800,
  error_escalation_threshold = 3
)
```

Arguments

`chat_obj`	Chat object from ellmer
`task`	Character string. The task description to be solved
`max_turns`	Integer. Maximum number of ReAct turns (default: 15)
`pkgs_to_use`	Character vector. R packages to load in session
`objects_to_use`	Named list. Objects to load in R session
`existing_session`	Existing callr session to continue from (optional)
`verbose`	Logical. Whether to print progress information
`r_session_options`	List. Options for callr R session
`context_window_size`	Integer. Maximum characters before history summary (default: 3000)
`max_observation_length`	Integer. Maximum observation length (default: 800)
`error_escalation_threshold`	Integer. Error count threshold for escalation (default: 3)

Value

List with complete ReAct results

`response_as_json` *Get JSON response from LLM with validation and retry*

Description

Get JSON response from LLM with validation and retry

Usage

```
response_as_json(
  chat_obj,
  prompt,
  schema = NULL,
  schema_strict = FALSE,
  max_iterations = 3
)
```

Arguments

`chat_obj`	Chat object (LLM client). Must be a properly initialized LLM client instance.
`prompt`	Character string. The user prompt to send to the LLM requesting JSON output.
`schema`	List or NULL. Optional JSON schema for response validation (as R list structure). When provided, validates the LLM response against this schema. Default is NULL.
`schema_strict`	Logical. Whether to use strict schema validation (no additional properties allowed). Only applies when schema is provided. Default is FALSE.
`max_iterations`	Integer. Maximum number of retry attempts for invalid JSON or schema validation failures. Must be positive. Default is 3.

Value

List. Parsed JSON response from the LLM.

Examples

```
## Not run:
# Basic usage without schema
result <- response_as_json(
  chat_obj = llm_client,
  prompt = "List three colors"
)

# With schema validation
schema <- list(
```

```
  type = "object",
  properties = list(
    equation = list(type = "string"),
    solution = list(type = "number")
  ),
  required = c("equation", "solution")
)
result <- response_as_json(
  chat_obj = llm_client,
  prompt = "How can I solve 8x + 7 = -23?",
  schema = schema,
  schema_strict = TRUE,
  max_iterations = 3
)

## End(Not run)
```

`response_to_r` *Response to R code generation and execution with session continuity*

Description

Response to R code generation and execution with session continuity

Usage

```
response_to_r(
  chat_obj,
  prompt,
  add_text = NULL,
  pkgs_to_use = c(),
  objects_to_use = list(),
  existing_session = NULL,
  list_packages = TRUE,
  list_objects = TRUE,
  return_session_info = TRUE,
  evaluate_code = TRUE,
  r_session_options = list(),
 return_mode = c("full", "code", "console", "object", "formatted_output", "llm_answer",
    "session"),
  max_iterations = 3
)
```

Arguments

`chat_obj`	Chat object from ellmer
`prompt`	User prompt for R code generation

`add_text`	Additional instruction text
`pkgs_to_use`	Packages to load in R session
`objects_to_use`	Named list of objects to load in R session
`existing_session`	Existing callr session to continue from (optional)
`list_packages`	Whether to list available packages in prompt
`list_objects`	Whether to list available objects in prompt
`return_session_info`	Whether to return session state information
`evaluate_code`	Whether to evaluate the generated code
`r_session_options`	Options for callr R session
`return_mode`	Return mode specification
`max_iterations`	Maximum retry attempts

Value

Result based on return_mode

`retrieve_docs` *Retrieve and Format R Function Documentation for LLM Consumption*

Description

Retrieve and Format R Function Documentation for LLM Consumption

Usage

```
retrieve_docs(
  chat_obj,
  prompt,
  schema = package_function_schema(),
  schema_strict = TRUE,
  skip_undocumented = TRUE,
  use_llm_fallback = FALSE,
  example_count = 2,
  warn_skipped = TRUE
)
```

Arguments

chat_obj	LLM chat client object
prompt	Task description
schema	JSON schema (default: package_function_schema())
schema_strict	Strict schema validation
skip_undocumented	Skip functions without docs
use_llm_fallback	Use LLM to generate examples
example_count	Number of examples per function
warn_skipped	Show warning for skipped functions

Value

Formatted documentation string

---

save_code_to_file *Save extracted code to file*

Description

This function saves extracted code to a file with appropriate extension based on the programming language.

Usage

```
save_code_to_file(code_string, filename = NULL, language = "r")
```

Arguments

code_string	String or character vector containing the code to save
filename	Output filename. If NULL, generates a timestamped filename
language	Programming language for determining file extension (default: "r")

Value

The path to the saved file

Examples

```
## Not run:
# Extract and save R code
llm_response <- "```r\nplot(1:10)\n```"
code <- extract_r_code(llm_response)
save_code_to_file(code) # Saves as "code_20240101_120000.R"

# Save with custom filename
save_code_to_file(code, "my_plot.R")

# Save Python code with auto extension
py_code <- "import pandas as pd\ndf = pd.DataFrame()"
save_code_to_file(py_code, language = "python") # Creates .py file

# Save multiple code blocks
response <- "```r\nx <- 1\n```\n```r\ny <- 2\n```"
codes <- extract_r_code(response)
save_code_to_file(codes, "combined_code.R")

## End(Not run)
```

Index