Package: decisions (via r-universe)

November 9, 2026

Version 0.1.0.9000

Title Typed Decisions from Decision Models

Description Sends texts and images to decision models: hosted or local models that return a probability for each allowed answer to a typed question (true or false, a choice, or an ordered score). Supports the web services of 'TypeSafe' <https://docs.typesafe.ai/introduction>, 'OpenAI' <https://platform.openai.com>, 'OpenRouter' <https://openrouter.ai> and 'Microsoft Foundry' <https://ai.azure.com>, which need an API (application programming interface) key and can charge for each call, and any local or hosted server that uses the 'SystemOne' request format. Returns one row for each input in the 'tidymodels' style, saves and scores a complete decision setup, and applies a calibrator fitted with 'probably'. Includes injury reports from the Severe Injury Reports of the U.S. Occupational Safety and Health Administration <https://www.osha.gov/severe-injury-reports>, food recall reports of the U.S. Food and Drug Administration <https://open.fda.gov/apis/food/enforcement/>, and simulated textile images made with 'spc4sts', Bui and Apley (2021) <doi:10.1080/00224065.2019.1707730>.

License MIT + file LICENSE

URL https://github.com/fmegahed/decisions, https://fmegahed.github.io/decisions/

BugReports https://github.com/fmegahed/decisions/issues

Depends R (>= 4.3.0)

Imports cli (>= 3.6.0), httr2 (>= 1.2.0), jsonlite (>= 1.8.5), purrr (>= 1.0.0), rlang (>= 1.3.0), S7 (>= 0.2.0), stringi, tibble

Suggests dplyr, knitr, magick, mgcv, probably, rmarkdown, rsample, testthat (>= 3.0.0), withr, yardstick

Config/testthat/edition 3

Encoding UTF-8

Language en-US

Roxygen list(markdown = TRUE)

VignetteBuilder knitr, rmarkdown

Config/roxygen2/version 8.0.0

LazyData true

Config/pak/sysreqs libicu-dev libssl-dev

Repository https://fmegahed.r-universe.dev

Date/Publication 2026-11-09 19:16:33 UTC

RemoteUrl https://github.com/fmegahed/decisions

RemoteRef HEAD

RemoteSha e614b6743b0659f9e2a2138f5ed549cb6e968c9f

Contents

decide . . . 3
decider_foundry . . . 7
decider_openai . . . 8
decider_openrouter . . . 9
decider_systemone . . . 10
decider_typesafe . . . 11
decision_setup . . . 13
evaluate_once . . . 14
ohio_injuries . . . 16
ohio_recalls . . . 17
question_choice . . . 18
question_noul . . . 19
question_score . . . 20
read_setup . . . 21
rubric_answers . . . 22
score_setup . . . 23
textiles . . . 25
write_setup . . . 26

Index **28**

`decide` *Ask a decision model typed questions*

Description

Sends each input (a text, an image or both) with the questions to one model and returns, for each input, the answer and a probability for each allowed answer.

Usage

```
decide(
  decider,
  questions,
  text = NULL,
  image = NULL,
  references = NULL,
  max_size = 768,
  upscale = FALSE,
  crop = "none",
  max_active = 10,
  rpm = NULL,
  path = NULL,
  ignore_hash = FALSE
)
```

Arguments

`decider`	A decider from `decider_typesafe()`, `decider_openai()`, `decider_foundry()`, `decider_openrouter()` or `decider_systemone()`; or a setup from `decision_setup()` or `read_setup()`. A setup gives the decider, the questions, the prepared references, the image settings and a calibrator. With a setup, give the inputs by name (`text = `, `image = `), and leave out `questions`, `references`, `max_size`, `upscale` and `crop`: they stop `decide()` with an error.
`questions`	One question from `question_noul()`, `question_choice()` or `question_score()`, or a named list of questions. The names are the question names of the request, kept exactly. With two or more questions, the names, cleaned as the answer names are, are the prefixes of the output columns; two names that give the same prefix are refused. One question without a name is sent as `question`.
`text`	A character vector with one text per input, or `NULL`.
`image`	A character vector of file paths or URLs, a list of magick images, or a magick vector of several images (for example `magick::image_read(paths)`), with one image per input; or `NULL`. A file with several frames (GIF, TIFF, PDF) gives its first frame. With `text`, element i of `text` and element i of `image` make input i.
`references`	Reference images for the answers, or `NULL`: a named list in which each name is an allowed answer of the question and each element holds the images of that answer (paths, URLs or magick images). This release takes references with one question only.

`max_size` The longest side of each image in pixels after the resize (one number of 1 or more), or `NULL` to keep the size. The default is `768`.

`upscale` `TRUE` enlarges an image that is smaller than `max_size`. `FALSE` (the default) never enlarges.

`crop` `"none"` (the default), `"center"`, a box `c(x, y, width, height)` in pixels, or a function that takes a magick image and returns a box. `max_size`, `upscale` and `crop` apply to the inputs and to the references.

`max_active` The largest number of requests in progress at the same time. The default is `10`.

`rpm` The largest number of requests per minute, or `NULL` (the default) for no limit. The requests then go out evenly, one each `60 / rpm` seconds.

`path` `NULL` (the default), or the path of a JSON file that keeps the finished answers, so that a long run that stops can continue without paying twice for an input (as `ellmer::batch_chat()` does). The folder must exist. The inputs that are not in the file go out in chunks of `4 * max_active`, and `decide()` adds the answers of each chunk to the file. A new call with the same `path` sends only the inputs that are not in the file and returns all rows, in the order of the inputs. Rows with an `.error` are not written, so a new call sends them again. With `path`, identical inputs are sent once.

The file holds a hash of the run and, for each finished input (keyed by a hash of the prepared text and image), its answers with their probabilities, `.model`, `.input_tokens`, `.cost` and `.seconds`. It holds no key and no input. The hash covers the provider, the server URL, the model, the request format, the image format, the questions (instructions, answers and descriptions), the prepared reference images and the image settings; a crop function enters through its body.

The keys and the hash of the run come from `rlang::hash()`, and both cover prepared JPEG bytes (of the inputs, and of the references). Another version of ImageMagick (the library under magick) can give other JPEG bytes for the same image, and another version of rlang can give other hashes. So a path file continues a run on the computer that wrote it, with the same versions of ImageMagick and rlang. Elsewhere, the hash of the run can differ, or the inputs are sent again.

With a setup that holds a calibrator, the file keeps the answers before calibration, and the hash does not cover the calibrator: `decide()` applies the calibrator after it reads the file. So a new call with a refitted calibrator, or with none, reuses the answers of the file.

A file that is not a path file of `decide()` (for example a setup file of `write_setup()`) stops the call before any request, also with `ignore_hash = TRUE`, so `decide()` never replaces it.

`ignore_hash` `FALSE` (the default) stops before any request when the hash in the `path` file differs from the hash of the run, that is, when the model, the questions, the references or the image settings changed. `TRUE` keeps the answers of the file with a warning and sends only the other inputs. The file gets the new hash when `decide()` adds answers to it; when the file holds every input, it does not change. Use `TRUE` only when you know that the change does not change the answers, for example the same model on a new server URL.

Details

One request goes out per input, in parallel (`httr2::req_perform_parallel()`), with at most `max_active` requests at a time and at most `rpm` requests per minute. Set `rpm` to the rate limit of your account: with no `rpm`, the first requests go out at full speed, and a provider with a rate limit then refuses some of them with HTTP 429.

Each request waits at most 300 seconds (5 minutes) for its response (`httr2::req_timeout()`). A request that takes longer gets a row error, so a server that stops responding does not hold the call. With `path`, a new call sends that input again.

A request that gets HTTP 429 or 503 is sent one more time, as the default retry policy of httr2 does (two tries). `decide()` sends all such requests again together, after the longest wait that their `Retry-After` headers ask for (1 second without the header, at most 60 seconds). A second failure stays as a row error. So a call never waits long for a server that keeps refusing.

An interrupt (Esc) keeps the rows that are done. After an interrupt while the requests are out, nothing is sent again, and each request that did not finish gets a row error. An interrupt during the wait before the second tries skips them; their rows get the error "Interrupted before a retry." With `path`, an interrupt also stops the chunks: the file keeps every finished answer, the inputs that were not sent get the row error of a request that did not finish, and a new call sends only the rest. Each write goes to a temporary file in the same folder first, which then replaces the file, so a crash during a write leaves the previous file whole. When the file cannot be replaced (for example, while a sync program holds it), `decide()` tries again twice, and warns at the end with the number of finished answers that the file does not hold.

No row is lost. A row has `NA` answers and a message in `.error` when the call fails (the message holds the error text of the provider word for word, the model name and the size of the request in bytes), when the response cannot be read, when an answer is not an allowed answer or is missing, when the probability of an allowed answer is missing, or when the score of a score question is not one number or its probabilities are all zero. With `path`, a new call sends such a row again. A row with an answered call keeps its `.input_tokens` and `.cost`. A `model`, `input_tokens` or `cost` field that is not one value gives the model of the decider or `NA`.

Because `.error` keeps the text of the provider word for word, it can hold a part of the key: OpenAI answers a wrong key with HTTP 401 and a text that shows the last four characters of the key. Do not share the `.error` texts of a call with a wrong key.

`decide()` reads the key and checks the input type of the model before any request. For a provider with a key (all deciders except `decider_systemone()`), it stops unless the key is one non-empty string; the message does not show the key. For a setup with a calibrator, it also checks before any request that probably and the packages of the calibrator (for example mgcv) are installed. It prepares each reference image once per call (a setup holds its references prepared) and each input image once. With `decider_systemone(image_format = "images")`, it writes the images with three color channels.

Each image goes out as a JPEG, which has no transparency. So each pixel keeps its color and loses its transparency, and transparent areas often become black (a transparent pixel usually holds black, as `"transparent"` does in magick).

SystemOne links reference images to the answers of choice and noul questions. A SystemOne score question has no place for them: they are sent, but not linked to a level. OpenAI puts them in the instructions of the question, and the image-list format names them in the state text.

`.cost` holds only a cost that the provider reports (OpenRouter does). When no row has a cost, `decide()` prints one message that says so.

`.seconds` is the total time of the response as `httr2::resp_timing()` gives it; for a request that was sent again, it is the time of the second try. It is `NA` when no response arrived.

Value

A tibble with one row per input, in the order of the inputs:

- For one question: `.pred_class`, a factor whose levels are the allowed answers, kept exactly (for example `"not sure"`); `.pred_<answer>`, the probability of each allowed answer; and `.score` for a score question. The probability columns have syntactic names: the answer cleaned as `janitor::make_clean_names(case = "none")` cleans it (ASCII letters, digits and `_`; `X` before a leading digit). For example, `"not sure"` gives `.pred_not_sure`, `"late / lost"` gives `.pred_late_lost` and `"50%"` gives `.pred_X50_percent`. probably and yardstick read these columns.
- For two or more questions: the same columns with the cleaned question name and `_` as a prefix, for example `quality_.pred_class`.
- `.model`: the model that the provider reports, or the model of the decider when the response has none.
- `.input_tokens`: the input tokens that the provider reports.
- `.cost`: the cost that the provider reports, or `NA`.
- `.seconds`: the time of the response.
- `.error`: `NA`, or the reason why the answers of the row are `NA`.

The attribute `requests` of the tibble is the number of requests that this call sent (0 when a `path` file holds all the answers). A second try after HTTP 429 or 503 counts as a request.

With a setup that holds a calibrator, the calibrated probabilities of `probably::cal_apply()` replace the `.pred_<answer>` columns, and `.pred_class` is the answer with the highest calibrated probability (its levels stay the allowed answers). `.score` stays as the model gave it. The rows with an `.error` stay `NA`.

See Also

The deciders (`decider_typesafe()` and the others), the questions (`question_noul()`, `question_choice()` and `question_score()`), `decision_setup()` to save the choices, and `vignette("getting-started")`.

Examples

```
## Not run:
# These examples need an API key and make paid calls, so they are not run on CRAN.
refund <- question_noul("Does the message ask for a refund?")
decide(decider_typesafe("jev-1.13.0"), refund, text = c(
  "Please refund the second charge.",
  "Where is my parcel?"
))

# Images of the package: two references and two inputs
dir <- system.file("extdata", "textiles", package = "decisions")
refs <- subset(textiles, role == "reference" & reference_rank == 1)
inputs <- subset(textiles, role == "input")[1:2, ]
```

```
quality <- question_choice("Inspect `inspection_image` for a flaw.", c(
  nominal = "A regular pattern.", defective = "A flaw in the pattern."
))
decide(
  decider_openai(), list(quality = quality),
  image = file.path(dir, inputs$file),
  references = list(
    nominal = file.path(dir, refs$file[refs$class == "nominal"]),
    defective = file.path(dir, refs$file[refs$class == "defective"])
  )
)

# A setup holds the decider, the questions and the references
setup <- decision_setup(decider_typesafe("jev-1.13.0"), list(refund = refund))
decide(setup, text = "Please refund the second charge.")

## End(Not run)
```

`decider_foundry` *Use Microsoft-Decision-1 on Microsoft Foundry*

Description

Makes a decider for a deployment of Microsoft-Decision-1 on Microsoft Foundry (SystemOne format).

Usage

```
decider_foundry(
  endpoint = Sys.getenv("AZURE_FOUNDRY_ENDPOINT"),
  deployment,
  credentials = NULL,
  input = NULL
)
```

Arguments

`endpoint`	The endpoint of the Foundry resource, for example `"https://<resource>.services.ai.azure.com"`. The default reads the environment variable `AZURE_FOUNDRY_ENDPOINT`.
`deployment`	The name of the deployment. It is the model name of the request.
`credentials`	A function with no arguments that returns the key. `NULL` (the default) reads the environment variable `AZURE_FOUNDRY_API_KEY` each time `decide()` sends requests. You generally should not need this argument. Set `AZURE_FOUNDRY_API_KEY` instead. The best place to set it is `.Renviron`. Open that file with `usethis::edit_r_environ()`. The usethis package is optional: you can also edit `.Renviron` by hand. Give a function for more control, for example to read the key from a secret store. The decider keeps the function, not the key.

`input` The input type that the model reads: `"text"`, `"image"` or `"text+image"`. `NULL` (the default) takes it from the internal table of input types; a model that is not in the table is not checked. `decide()` stops before any request when an input does not fit.

Details

You need an Azure subscription and a Foundry resource with a deployment of the model. Create them in Microsoft Foundry. Then set the environment variables `AZURE_FOUNDRY_ENDPOINT` (the endpoint of the resource) and `AZURE_FOUNDRY_API_KEY` (a key of the resource).

The requests go to `<endpoint>/providers/microsoft/v1/systemone` with the key in the `api-key` header. This release takes an API key only: a Microsoft Entra ID token, which goes in an `Authorization` header, does not work. This decider is not yet tested with a live call. Microsoft-Decision-1 reads text only.

Value

A decider object for `decide()`. It holds no key.

See Also

Other deciders: `decider_openai()`, `decider_openrouter()`, `decider_systemone()`, `decider_typesafe()`

Examples

```
decider_foundry("https://my-resource.services.ai.azure.com", "decision-1")
```

---

decider_openai *Use the OpenAI Decisions API*

Description

Makes a decider for a model of the OpenAI Decisions API (`https://api.openai.com`).

Usage

```
decider_openai(model = NULL, credentials = NULL, input = NULL)
```

Arguments

`model` The model name. `NULL` (the default) uses `"gpt-6-luna"` and prints a message that names it.

`credentials` A function with no arguments that returns the key. `NULL` (the default) reads the environment variable `OPENAI_API_KEY` each time `decide()` sends requests. You generally should not need this argument. Set `OPENAI_API_KEY` instead. The best place to set it is `.Renviron`. Open that file with `usethis::edit_r_environ()`. The usethis package is optional: you can also edit `.Renviron` by hand. Give a function for more control, for example to read the key from a secret store. The decider keeps the function, not the key.

input — The input type that the model reads: "text", "image" or "text+image". NULL (the default) takes it from the internal table of input types; a model that is not in the table is not checked. decide() stops before any request when an input does not fit.

Details

A ChatGPT Plus membership does not give access to the API. You need to sign up for a developer account (and pay for it) at the developer platform. Then set the environment variable OPENAI_API_KEY.

OpenAI does not report a cost, so .cost is NA in the output of decide().

Value

A decider object for decide(). It holds no key.

See Also

Other deciders: decider_foundry(), decider_openrouter(), decider_systemone(), decider_typesafe()

Examples

```
decider_openai("gpt-6-luna")
```

---

decider_openrouter *Use a decision model through OpenRouter*

Description

Makes a decider for a decision model on OpenRouter (https://openrouter.ai, alpha decisions endpoint), for example Jev or Microsoft-Decision-1.

Usage

```
decider_openrouter(model, credentials = NULL, input = NULL)
```

Arguments

model — The OpenRouter model id, for example "typesafe/jev-1.14" or "microsoft/microsoft-decision-1 Required.

credentials — A function with no arguments that returns the key. NULL (the default) reads the environment variable OPENROUTER_API_KEY each time decide() sends requests. You generally should not need this argument. Set OPENROUTER_API_KEY instead. The best place to set it is .Renviron. Open that file with usethis::edit_r_environ(). The usethis package is optional: you can also edit .Renviron by hand. Give a function for more control, for example to read the key from a secret store. The decider keeps the function, not the key.

input — The input type that the model reads: "text", "image" or "text+image". NULL (the default) takes it from the internal table of input types; a model that is not in the table is not checked. decide() stops before any request when an input does not fit.

Details

Sign up at https://openrouter.ai, then create a key on the keys page. Set the environment variable OPENROUTER_API_KEY.

OpenRouter documents no image field for decisions, so the table of input types gives every OpenRouter model the input type "text". OpenRouter reports the cost of each call, and .cost holds it.

Value

A decider object for decide(). It holds no key.

See Also

Other deciders: decider_foundry(), decider_openai(), decider_systemone(), decider_typesafe()

Examples

```
decider_openrouter("typesafe/jev-1.14")
```

---

decider_systemone *Use any SystemOne server*

Description

Makes a decider for a server that speaks SystemOne, for example Clef-Flash on an SGLang server or Laya on laya-serve, often on the computer of the user.

Usage

```
decider_systemone(
  base_url,
  model,
  image_format = c("assets", "images"),
  credentials = NULL,
  input = NULL
)
```

Arguments

base_url	The server root, for example "http://localhost:30000". Requests go to <base_url>/v1/systemone.
model	The model name, for example "Cloudflare/clef-flash", or a Laya checkpoint ("english", "multilingual", "typed-decisions"). Required.
image_format	How the server takes images: "assets" (the SystemOne assets object) or "images" (an ordered list of images, as the SGLang server of Clef-Flash needs).
credentials	A function with no arguments that returns a key. NULL (the default) sends no key, as a local server needs none. A hosted SystemOne server can need one: give a function, and the key goes in an Authorization: Bearer header. For more control, the function can read the key from a secret store.
input	The input type that the model reads: "text", "image" or "text+image". NULL (the default) takes it from the internal table of input types; a model that is not in the table is not checked. decide() stops before any request when an input does not fit.

Details

A local server such as Laya or Clef-Flash needs no key and no account. A hosted SystemOne server can need a key: give it with credentials.

With image_format = "images", decide() writes every image with three color channels, because the SGLang server cannot read a JPEG with one channel.

Value

A decider object for decide(). It holds no key.

See Also

Other deciders: decider_foundry(), decider_openai(), decider_openrouter(), decider_typesafe()

Examples

```
decider_systemone(
  "http://localhost:30000", "Cloudflare/clef-flash", image_format = "images"
)
decider_systemone("http://localhost:8000", "typed-decisions")
```

---

decider_typesafe *Use Jev through the TypeSafe API*

Description

Makes a decider for a Jev model on the TypeSafe API (https://api.typesafe.ai, SystemOne format).

Usage

```
decider_typesafe(model = NULL, credentials = NULL, input = NULL)
```

Arguments

model	The model name, for example "jev-1.13.0". NULL (the default) uses "jev-latest" and prints a message that names it.
credentials	A function with no arguments that returns the key. NULL (the default) reads the environment variable TYPESAFE_API_KEY each time decide() sends requests. You generally should not need this argument. Set TYPESAFE_API_KEY instead. The best place to set it is .Renviron. Open that file with usethis::edit_r_environ(). The usethis package is optional: you can also edit .Renviron by hand. Give a function for more control, for example to read the key from a secret store. The decider keeps the function, not the key.
input	The input type that the model reads: "text", "image" or "text+image". NULL (the default) takes it from the internal table of input types; a model that is not in the table is not checked. decide() stops before any request when an input does not fit.

Details

You need a TypeSafe API key. Get one in the TypeSafe console (https://console.typesafe.ai/), then set the environment variable TYPESAFE_API_KEY. The TypeSafe documentation is at https://docs.typesafe.ai/introduction.

Image input is experimental: TypeSafe reads images only with the header TypeSafe-Danger-Zone-Images: true, which decide() sends whenever a request carries an image. TypeSafe does not report a cost, so .cost is NA in the output of decide(). On 2026-10-23, TypeSafe refused images under 280 pixels on the longest side. So use upscale = TRUE for small images (see vignette("images")).

Value

A decider object for decide(). It holds no key.

See Also

Other deciders: decider_foundry(), decider_openai(), decider_openrouter(), decider_systemone()

Examples

```
decider_typesafe("jev-1.13.0")
# A function can give the key, for example from another variable:
decider_typesafe("jev-1.13.0", credentials = \() Sys.getenv("MY_JEV_KEY"))
```

`decision_setup` *Make a setup: one complete decision*

Description

A setup holds one complete decision: the decider, the questions, the reference images, the image settings and, if you want, a calibrator. `decide()` takes a setup in place of `decider` and `questions`. `write_setup()` saves it as one JSON file that R code, a Shiny app and an agent can share, and `read_setup()` reads it.

Usage

```
decision_setup(
  decider,
  questions,
  references = NULL,
  max_size = 768,
  upscale = FALSE,
  crop = "none",
  calibrator = NULL
)
```

Arguments

`decider`	A decider from `decider_typesafe()`, `decider_openai()`, `decider_foundry()`, `decider_openrouter()` or `decider_systemone()`.
`questions`	One question from `question_noul()`, `question_choice()` or `question_score()`, or a named list of questions, as in `decide()`.
`references`	Reference images for the answers, or `NULL` (the default), as in `decide()`: a named list in which each name is an allowed answer of the question and each element holds the images of that answer (paths, URLs or magick images). This release takes references with one question only.
`max_size`, `upscale`, `crop`	The image settings, as in `decide()`: `max_size` is `NULL` or one number of 1 or more, and `upscale` is `TRUE` or `FALSE`. They apply to the references now and to the inputs of `decide()`. A crop function works in R, but `write_setup()` cannot save it.
`calibrator`	`NULL` (the default), or a probably calibrator, for example from `probably::cal_estimate_isotonic()`, fitted on the output of `decide()` for this question: its truth is a factor whose levels are the answers, and its estimates are the `.pred_` columns of the answers. A setup with a calibrator has one question. `decide()` applies the calibrator (see the value of `decide()`). To add a calibrator to a setup that exists, see Details.

Details

decision_setup() prepares each reference image once, with the image settings of the setup (and with three color channels for decider_systemone(image_format = "images")), and keeps its JPEG bytes. So decide(), the file of write_setup() and read_setup() use the same bytes, and a run with path has the same hash whether it starts from a setup or from its pieces.

A model that reads only text cannot take references, and bad image settings are refused: the setup stops with an error before it reads the images.

To add a calibrator to a setup that exists, set its property: setup@calibrator <- calibrator. This works for a setup from decision_setup() and for a setup from read_setup(), whose references are prepared images that decision_setup() cannot take again. The setup checks the calibrator, as decision_setup() does, and its references, its settings and its run hash do not change. Then write_setup() stores the calibrator with that hash.

Value

A setup: an S7 object of class decision_setup with the properties decider, questions (a named list), references (the prepared JPEG bytes, or NULL), max_size, upscale, crop and calibrator. It holds no key.

See Also

decide() to use a setup, and score_setup() to score it.

Other setups: read_setup(), write_setup()

Examples

```
refund <- question_noul("Does the message ask for a refund?")
setup <- decision_setup(decider_typesafe("jev-1.13.0"), list(refund = refund))
setup
```

---

evaluate_once *Score the test rows one time*

Description

Runs decide() with a setup on the test rows and scores them as score_setup() does, but only once. It writes the result to path, and it refuses to run again while path exists, so the test rows are not tuned against.

Usage

```
evaluate_once(setup, text = NULL, image = NULL, truth, event = NULL, path)
```

Arguments

setup	A setup from decision_setup() or read_setup() with one question.
text, image	The inputs, as in decide().
truth	A factor or character vector with the true answer of each input. It has the length of the inputs, and its values are answers of the question.
event	The answer that counts as the event (the positive class) of the AUC. NULL (the default) gives "true" for a noul question and the second answer for a question with two answers (for c(normal = ..., defective = ...), "defective"). With three or more answers, event is required.
path	The path of the result file (an .rds file in a folder that exists). If it exists, evaluate_once() stops before any request.

Details

decide() keeps its state in paste0(path, ".state.json"). A run that stops continues from this file without paying twice, and each test row is scored once. When a row has an error, evaluate_once() returns the row with estimate = NA, does not write path, and warns with the failed rows that a new call sends only them again. So the test rows give one estimate only, and you cannot tune on them by repeated calls.

A test row can fail at each call, for example with an image that the provider refuses. Then the estimate cannot appear until that row has an answer or leaves the test rows. Read the reason in the .error column of answers. If a new call cannot fix the row, remove it from the test rows, call evaluate_once() again with the same path, and report the removed row with the result. The state file keeps the other answers, so they are not sent again.

Value

The one-row tibble of score_setup(). It is also written to path with saveRDS() when no row has an error. While a row has an error, estimate is NA: the estimate appears only in the complete result.

See Also

score_setup() for the tuning rows, and decision_setup() to make a setup.

Other scoring: score_setup()

Examples

```
## Not run:
# Needs an API key and makes paid calls, so it is not run on CRAN.
quality <- question_choice("Is the part defective?", c(
  nominal = "A clean part.", defective = "A part with a defect."
))
setup <- decision_setup(decider_typesafe("jev-1.13.0"), list(q = quality))
test_texts <- c("Clean surface.", "Dark streak.")
test_truth <- c("nominal", "defective")
evaluate_once(
  setup, text = test_texts, truth = test_truth,
  path = tempfile(fileext = ".rds")
```

```
)

## End(Not run)
```

`ohio_injuries` *Severe injury reports from Ohio, 2025*

Description

Narratives of 464 severe work injuries in Ohio, with the injury type that OSHA recorded. Use it to test a decision model on a text task with five groups of different size.

Usage

```
ohio_injuries
```

Format

A tibble with 464 rows and 11 columns:

id Character. The report ID of OSHA.

event_date Date. The date of the event.

city Character. The city of the event.

state Character. The state, always `"OHIO"`.

final_narrative Character. The narrative that OSHA published, with the whitespace trimmed and each line break changed to a space.

original_nature_title Character. The nature of the injury, as OSHA titles it.

collapsed_nature_title Factor. The nature of the injury in five groups, in this order: `amputation` (187 rows), `fracture` (242), `crushing injury` (14), `thermal burn` (15) and `concussion` (6).

original_event_title Character. The event, as OSHA titles it.

collapsed_event_title Factor. The OIICS 2.01 division of the event, from the first digit of its code, in code order.

original_source_title Character. The source of the injury, as OSHA titles it.

collapsed_source_title Factor. The OIICS 2.01 division of the source, from the first digit of its code, in code order.

Details

The data are a work of the U.S. government and are in the public domain. The reports cover employers under federal OSHA jurisdiction. The OIICS 2.01 divisions come from the Occupational Injury and Illness Classification System of the U.S. Bureau of Labor Statistics.

The five groups of `collapsed_nature_title` come from the titles of OSHA like this:

- `amputation`: "Amputations, avulsions, enucleations unspecified" and "Amputations involving bone loss".
- `fracture`: "Fractures".
- `crushing injury`: "Nonfatal 'crushing' injuries".
- `thermal burn`: the three titles "Thermal burns" (degree unspecified, second degree, third degree or higher).
- `concussion`: "Concussions".

Three kinds of report are not in the data on purpose. The title "Avulsions, enucleations without bone loss" is out, because OSHA does not count an avulsion as an amputation. The combined titles "Fractures and soft tissue injuries" and "Fractures and surface, flesh wounds" are out, because they name more than one injury. The data keep no other column of the download, for example the employer, the address and the flag for an amputation.

Source

Occupational Safety and Health Administration. (n.d.). *Severe injury reports* [Data set]. U.S. Department of Labor. Retrieved October 23, 2026, from https://www.osha.gov/severe-injury-reports

Filters: Year: 2025 and State: Ohio.

Examples

```
table(ohio_injuries$collapsed_nature_title)
ohio_injuries$final_narrative[1]
```

`ohio_recalls` *Food recalls of Ohio firms*

Description

348 food recalls of firms in Ohio, with the class that the U.S. Food and Drug Administration (FDA) gave to the recall. The class is an ordered answer, from low to high risk. Use it to test a decision model on a score question.

Usage

```
ohio_recalls
```

Format

A tibble with 348 rows and 8 columns:

recall_number Character. The recall number of the FDA.

event_id Character. The number of the recall event. One event can have more than one row, when its reasons or classes differ.

recall_initiation_date Date. The date on which the firm started the recall.

city Character. The city of the recalling firm.

state Character. The state of the recalling firm, always `"OH"`.

product_description Character. The description of the product, from the first product of the recall in the order of the API.

reason_for_recall Character. The reason for the recall, with the whitespace trimmed.

classification Ordered factor. The class of the FDA, from low to high risk: `Class III` < `Class II` < `Class I`. The data have 21 rows of Class III, 175 of Class II and 152 of Class I.

Details

openFDA data are in the public domain (CC0 1.0, see https://open.fda.gov/terms/).

The classes come from 21 CFR 7.3(m):

- Class I: a reasonable probability that the use of, or exposure to, the product will cause serious adverse health consequences or death.
- Class II: the use of, or exposure to, the product may cause temporary or medically reversible adverse health consequences, or the probability of serious adverse health consequences is remote.
- Class III: the use of, or exposure to, the product is not likely to cause adverse health consequences.

One recall event often lists many products with the same reason. The data keep one row for each unique event, trimmed reason and class (348 rows from 1,227 records), with the product of the first record in the order of the API. The data keep no firm name, address or postal code. Non-ASCII characters are changed to ASCII, for example an accented letter to its plain letter.

Source

U.S. Food and Drug Administration. (n.d.). *Food recall enforcement reports* [Data set]. openFDA. Retrieved October 23, 2026, from https://open.fda.gov/apis/food/enforcement/

Filter: recalling firm in Ohio (`state:"OH"`), all years. The API gave 1,227 records on 2026-10-23.

Examples

```
table(ohio_recalls$classification)
ohio_recalls$reason_for_recall[1]
```

---

question_choice *Ask a question with named choices*

Description

Ask a question with named choices

Usage

```
question_choice(instructions, choices)
```

Arguments

instructions	A string: the question to decide.
choices	A named character vector with at least 2 unique names. Each name is an allowed answer (it can contain spaces, such as "not sure"). Each value describes the answer. In the output of decide(), the probability column of an answer has a syntactic name (.pred_not_sure for "not sure"), so two names that give the same column ("not sure" and "not_sure") are refused.

Value

A question_choice object with the properties instructions and answers.

See Also

question_noul() for true or false, question_score() for ordered levels, and decide() to ask the question.

Other questions: question_noul(), question_score()

Examples

```
question_choice(
  "Which describes the part?",
  c(normal = "A clean part.", defective = "A visible defect.")
)
```

---

question_noul *Ask a question with true or false as the answer*

Description

A noul question has two answers, true and false. The model returns a probability for each.

Usage

```
question_noul(instructions, true = NULL, false = NULL)
```

Arguments

instructions	A string: the statement or question to decide.
true, false	Optional strings that describe the answers true and false: each is one string, or NULL (the default) or NA for no description.

Value

A question_noul object with the properties instructions and answers.

See Also

question_choice() for more than two answers, question_score() for ordered levels, and decide() to ask the question.

Other questions: question_choice(), question_score()

Examples

```
question_noul("Is the surface wet?")
question_noul("Is the surface wet?", true = "Liquid is visible.")
```

---

question_score *Ask a question with ordered score levels*

Description

Ask a question with ordered score levels

Usage

```
question_score(instructions, levels)
```

Arguments

instructions	A string: what to score.
levels	A named character vector with at least 2 unique names, in order from low to high. Each name is an allowed score and each value describes it. Two names that give the same probability column in the output of decide() are refused (see question_choice()).

Value

A question_score object with the properties instructions and answers.

See Also

question_noul() for true or false, question_choice() for unordered choices, and decide() to ask the question.

Other questions: question_choice(), question_noul()

Examples

```
question_score(
  "How severe is the defect?",
  c(none = "No defect.", minor = "Small defect.", major = "Large defect.")
)
```

read_setup *Read a setup file*

Description

Reads a JSON file of `write_setup()` and rebuilds the setup.

Usage

```
read_setup(path, credentials = NULL)
```

Arguments

`path`	The path of a setup file.
`credentials`	A function with no arguments that returns the key of the decider. `NULL` (the default) reads the environment variable of the provider each time `decide()` sends requests: `TYPESAFE_API_KEY`, `OPENAI_API_KEY`, `AZURE_FOUNDRY_API_KEY` or `OPENROUTER_API_KEY`. A `decider_systemone()` server then gets no key. A setup file holds no key and no credentials function.

Details

`read_setup()` stops when `version` is not `1`, and checks the questions and the names of the references as `decision_setup()` does. It does not check the input type of the model: `decide()` checks it before any request.

When the decider (provider, server URL, model, request format or image format), the questions, the references or the image settings of the file no longer give its `calibrator_hash` (for example, because an agent edited a description or the model), the calibrator was fitted for other answers. `read_setup()` then drops it with a warning: fit a new calibrator on answers of the new setup.

`unserialize()` restores the calibrator, which is an R object. As with `readRDS()`, read only setup files that you trust.

Value

A setup, as from `decision_setup()`. The reference images keep the bytes of the file: they are not prepared again.

See Also

Other setups: `decision_setup()`, `write_setup()`

Examples

```
refund <- question_noul("Does the message ask for a refund?")
setup <- decision_setup(decider_typesafe("jev-1.13.0"), list(refund = refund))
path <- tempfile(fileext = ".json")
write_setup(setup, path)
read_setup(path)
# A function can give the key, for example from another variable:
read_setup(path, credentials = \() Sys.getenv("MY_JEV_KEY"))
```

`rubric_answers` *Sample answers for a rubric*

Description

Twenty sample answers to the question "What does `decide()` do when one call of a batch fails?", with a label for each question of the rubric. We wrote the answers for the vignette "Grading short answers with a rubric".

Usage

```
rubric_answers
```

Format

A tibble with 20 rows and 8 columns:

answer_id Integer. The number of the answer.

answer Character. The text of the answer.

keeps_row Logical. The answer says that the failed input keeps its row in the output.

na_answers Logical. The answer says that the answers of the failed row are `NA`.

error_text Logical. The answer says that the reason of the failure is in the `.error` column.

others_continue Logical. The answer says that the other inputs of the batch still get their answers.

stops_all Logical. The answer says that `decide()` stops with an error, or returns nothing, when one call fails (a wrong claim).

drops_row Logical. The answer says that `decide()` removes the failed input from the output (a wrong claim).

Source

Written for this package. The labels are the labels of the author.

Examples

```
rubric_answers[1:3, c("answer_id", "keeps_row", "stops_all")]
```

score_setup *Score a setup on labeled rows*

Description

Runs `decide()` with a setup on inputs whose true answers you know, and scores the probabilities with the area under the ROC curve. Use it to compare candidate setups (descriptions, instructions, references) on the learning rows. A person or an agent reads the `estimate` of each setup and keeps the best.

Usage

```
score_setup(
  setup,
  text = NULL,
  image = NULL,
  truth,
  role = NULL,
  metric = "roc_auc",
  event = NULL,
  path = NULL
)
```

Arguments

`setup`	A setup from `decision_setup()` or `read_setup()` with one question.
`text`, `image`	The inputs, as in `decide()`.
`truth`	A factor or character vector with the true answer of each input. It has the length of the inputs, and its values are answers of the question.
`role`	`NULL` (the default, no check), or a vector with one value per input: `"learning"`, `"calibration"` or `"test"`. `score_setup()` stops before any request when a value is `"test"`: score the test rows one time, with `evaluate_once()`.
`metric`	The metric. This release has `"roc_auc"` only. For another metric, compute it from the `answers` column of the result with yardstick.
`event`	The answer that counts as the event (the positive class) of the AUC. `NULL` (the default) gives `"true"` for a noul question and the second answer for a question with two answers (for `c(normal = ..., defective = ...)`, `"defective"`). With three or more answers, `event` is required.
`path`	`NULL`, or a state file for `decide()` (see `path` in `decide()`). A new call with the same `path` sends only the inputs that the file does not hold.

Details

The AUC uses the probabilities that `decide()` returns, so it is the AUC of the calibrated probabilities when the setup has a calibrator. It is the Mann-Whitney statistic of the probability of `event`,

with average ranks for ties; it equals `yardstick::roc_auc_vec(truth, prob, event_level = "second")` when `event` is the second level of `truth`.

The function checks `setup`, `metric`, `event`, `truth` and `role` before it sends a request.

Value

A one-row tibble:

- `metric` and `estimate`: the metric and its value. `estimate` is `NA` with a warning when fewer than one row of each class remains.
- `n`: the number of rows in the AUC.
- `calls`: the number of requests that this call sent (0 for inputs read from `path`).
- `input_tokens`: the sum of the input tokens.
- `cost`: the sum of every cost that the provider reports (rows with an error included: the provider billed them), or `NA` when a row that used input tokens has no reported cost. It is 0 when no row reports a cost or input tokens.
- `seconds`: the sum of the response times.
- `errors`: the number of rows with an `.error` (for example a failed call, or an answer with a missing probability). They stay out of the AUC.
- `answers`: a list column that holds the output of `decide()`.

`input_tokens`, `cost` and `seconds` count every row, also the rows that come from the `path` file. So a new call with the same `path` can report tokens and a cost with `calls = 0`.

See Also

`evaluate_once()` for the one-time test, and `decision_setup()` to make a setup.

Other scoring: `evaluate_once()`

Examples

```
## Not run:
# Needs an API key and makes paid calls, so it is not run on CRAN.
quality <- question_choice("Is the part defective?", c(
  nominal = "A clean part.", defective = "A part with a defect."
))
setup <- decision_setup(decider_typesafe("jev-1.13.0"), list(q = quality))
score_setup(
  setup, text = c("Clean surface.", "Dark streak."),
  truth = c("nominal", "defective"), role = c("learning", "learning")
)

## End(Not run)
```

`textiles` *Simulated textile images*

Description

Fifty simulated images of a woven textile, 250 x 250 pixels. Half of them show a flaw in the pattern. Use them to test a decision model on an image task. The files are in the folder `extdata/textiles` of the installed package: `system.file("extdata", "textiles", package = "decisions")`.

Usage

```
textiles
```

Format

A tibble with 50 rows and 5 columns:

file Character. The name of the JPEG file in `extdata/textiles` of the installed package. The name starts with the folder of the source (`nominal_`, `local_` or `global_`), because two source files have the same name.

class Factor. The truth, with the levels `nominal` and `defective`. Each level has 25 images.

defect_type Factor. `nominal` (no flaw), `local` (a flaw in one place) or `global` (a flaw that changes the whole pattern).

role Character. `reference` (4 images, to show the model what each answer looks like) or `input` (46 images, to decide on).

reference_rank Integer. `1` or `2` for the references, two for each class, and `NA` for the inputs.

Source

Made with the R package spc4sts:

Bui, A. T., & Apley, D. W. (2021). spc4sts: Statistical process control for stochastic textured surfaces in R. *Journal of Quality Technology, 53*(3), 219-242. doi:10.1080/00224065.2019.1707730

The CRAN page of spc4sts, which links its reference manual: https://CRAN.R-project.org/package=spc4sts

The images are shared in the public repository https://github.com/fmegahed/qe_genai (MIT license). The 4 references are the images of rank 1 and 2 of each class among the images that the benchmark of that repository leaves open for training and design. The 46 inputs are 23 images of each class, drawn at random with a fixed seed from the same open images; none is a reference. No image that the benchmark keeps for testing is in the data. The script https://github.com/fmegahed/decisions/blob/main/data-raw/textiles.R builds the data.

Examples

```
table(textiles$role, textiles$class)
paths <- file.path(
  system.file("extdata", "textiles", package = "decisions"), textiles$file
)
file.exists(paths[1])
```

`write_setup` *Save a setup as one JSON file*

Description

Writes a setup to one JSON file that R code, a Shiny app and an agent can share. The file holds no key. An agent can edit the descriptions as text, and git shows the changes.

Usage

```
write_setup(setup, path)
```

Arguments

`setup`	A setup from `decision_setup()` or `read_setup()`.
`path`	The path of the JSON file, in a folder that exists. An existing file is replaced.

Details

The file holds one JSON object, indented with two spaces:

- `version`: `1`, the version of the layout.
- `decider`: `provider`, `base_url`, `model`, `format`, `image_format` (`null` for OpenAI and OpenRouter) and `input` (`null` when the input type is not checked). It holds no key and no credentials function: `read_setup()` gives the decider its credentials.
- `questions`: an object keyed by question name, in order. Each question has `type` (`"noul"`, `"choice"` or `"score"`), `instructions` and `answers`: an object keyed by answer, in order, whose values are the descriptions (`null` for none).
- `references`: `null`, or an object keyed by answer whose values are arrays of the prepared JPEG images, each in base64 on one line.
- `image_settings`: `max_size` (`null` to keep the size), `upscale` and `crop` (`"none"`, `"center"` or a box `[x, y, width, height]`).
- `calibrator`: `null`, or the calibrator as base64 of `serialize()`.
- `calibrator_hash`: `null`, or the hash of all that changes the answers of the model when the calibrator was stored: the decider (provider, server URL, model, request format and image format), the questions, the references and the image settings. It is the hash of a `path` file of `decide()` (see `read_setup()`).

A crop function cannot be saved: give `crop` as `"none"`, `"center"` or a box.

The file goes to a temporary file in the same folder first, which then replaces `path`, so a crash during the write leaves the previous file whole.

Value

path, invisibly.

See Also

Other setups: decision_setup(), read_setup()

Examples

```
refund <- question_noul("Does the message ask for a refund?")
setup <- decision_setup(decider_typesafe("jev-1.13.0"), list(refund = refund))
path <- tempfile(fileext = ".json")
write_setup(setup, path)
read_setup(path)
```

Index

∗ **datasets**
  ohio_injuries, 16
  ohio_recalls, 17
  rubric_answers, 22
  textiles, 25
∗ **deciders**
  decider_foundry, 7
  decider_openai, 8
  decider_openrouter, 9
  decider_systemone, 10
  decider_typesafe, 11
∗ **questions**
  question_choice, 18
  question_noul, 19
  question_score, 20
∗ **scoring**
  evaluate_once, 14
  score_setup, 23
∗ **setups**
  decision_setup, 13
  read_setup, 21
  write_setup, 26

decide, 3
decide(), *14*, *19, 20*
decider_foundry, 7
decider_foundry(), *9–12*
decider_openai, 8
decider_openai(), *8*, *10–12*
decider_openrouter, 9
decider_openrouter(), *8, 9*, *11, 12*
decider_systemone, 10
decider_systemone(), *8–10*, *12*
decider_typesafe, 11
decider_typesafe(), *6*, *8–11*
decision_setup, 13
decision_setup(), *6*, *15*, *21*, *24*, *27*

evaluate_once, 14
evaluate_once(), *24*

ohio_injuries, 16
ohio_recalls, 17

question_choice, 18
question_choice(), *6*, *20*
question_noul, 19
question_noul(), *6*, *19, 20*
question_score, 20
question_score(), *6*, *19, 20*

read_setup, 21
read_setup(), *14*, *27*
rubric_answers, 22

score_setup, 23
score_setup(), *14, 15*

textiles, 25

write_setup, 26
write_setup(), *14*, *21*