Package 'quallmer'

February 16, 2026

Type Package

Title Qualitative Analysis with Large Language Models

Version 0.3.0

Description Tools for AI-assisted qualitative data coding using large language models ('LLMs') via the 'ellmer' package, supporting providers including 'OpenAI', 'Anthropic', 'Google', 'Azure', and local models via 'Ollama'. Provides a 'codebook'-based workflow for defining coding instructions and applying them to texts, images, and other data. Includes built-in 'codebooks' for common applications such as sentiment analysis and policy coding, and functions for creating custom 'codebooks' for specific research questions. Supports systematic replication across models and settings, computing inter-coder reliability statistics including Krippendorff's alpha (Krippendorff 2019, <doi:10.4135/9781071878781>) and Fleiss' kappa (Fleiss 1971, <doi:10.1037/h0031619>), as well as gold-standard validation metrics including accuracy, precision, recall, and F1 scores following Sokolova and Lapalme (2009, <doi:10.1016/j.ipm.2009.03.002>). Provides audit trail functionality for documenting coding workflows following Lincoln and Guba's (1985, ISBN:0803924313) framework for establishing trustworthiness in qualitative research.

License GPL-3

URL `https://quallmer.github.io/quallmer/`

Depends R (>= 3.5.0), ellmer (>= 0.4.0)

Imports cli, dplyr, tidyr, digest, irr, lifecycle, rlang, stats, yardstick

Encoding UTF-8

LazyData true

RoxygenNote 7.3.3

Suggests ggplot2, janitor, knitr, rmarkdown, testthat (>= 3.0.0), kableExtra, mockery, quanteda, quanteda.tidy, tibble, withr

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author Seraphine F. Maerz [aut, cre] (ORCID: <https://orcid.org/0000-0002-7173-9617>),
Kenneth Benoit [aut] (ORCID: <https://orcid.org/0000-0002-0797-564X>)

Maintainer Seraphine F. Maerz <seraphine.maerz@unimelb.edu.au>

Repository CRAN

Date/Publication 2026-02-16 18:00:02 UTC

Contents

`accessors` *Accessor functions for quallmer objects*

Description

Functions to safely access and modify metadata from quallmer objects (`qlm_coded`, `qlm_comparison`, `qlm_validation`, `qlm_codebook`). These functions provide a stable API for accessing object metadata without directly manipulating internal attributes.

Metadata types

quallmer objects store metadata in three categories:

User metadata (`type = "user"`):

- `name`: Run identifier (settable)
- `notes`: Descriptive notes (settable)
- Plus any custom fields added via `as_qlm_coded(..., metadata = list(...))`

Object metadata (`type = "object"`):

- `call`: Function call that created the object
- `parent`: Parent run name (for replications)
- `batch`: Whether batch processing was used
- `chat_args`: Arguments passed to the LLM chat
- `execution_args`: Arguments for parallel/batch execution
- `n_units`: Number of coded units
- `input_type`: Type of input ("text", "image", or "human")
- `source`: Coding source ("human" or "llm")
- `is_gold`: Whether this is a gold standard

System metadata (`type = "system"`):

- `timestamp`: When the object was created
- `ellmer_version`: Version of ellmer package
- `quallmer_version`: Version of quallmer package
- `R_version`: Version of R

Functions

- `qlm_meta()`: Get metadata fields
- `qlm_meta<-()`: Set user metadata fields (only `name` and `notes`)
- `codebook()`: Extract codebook from coded objects
- `inputs()`: Extract original input data

See Also

- `qlm_code()` for creating coded objects
- `as_qlm_coded()` for converting human-coded data
- `qlm_trail()` for viewing coding history

Examples

```
# Create a coded object
texts <- c("I love this!", "Terrible.", "It's okay.")
coded <- qlm_code(
  texts,
  data_codebook_sentiment,
  model = "openai/gpt-4o-mini",
  name = "run1",
  notes = "Initial coding run"
)

# Access metadata
qlm_meta(coded, "name")              # Get run name
qlm_meta(coded, type = "user")       # Get all user metadata
qlm_meta(coded, type = "system")     # Get system metadata

# Modify user metadata
qlm_meta(coded, "name") <- "updated_run1"
qlm_meta(coded, "notes") <- "Revised notes"

# Extract components
codebook(coded)                      # Get the codebook
inputs(coded)                        # Get original texts

# Custom metadata from human coding
human_data <- data.frame(
  .id = 1:5,
  sentiment = c("pos", "neg", "pos", "neg", "pos")
)
human_coded <- as_qlm_coded(
  human_data,
  name = "coder_A",
  metadata = list(
    coder_name = "Dr. Smith",
    experience = "5 years"
  )
)

# Access custom metadata
qlm_meta(human_coded, "coder_name")  # "Dr. Smith"
qlm_meta(human_coded, type = "user") # All user fields
```

as_qlm_coded   *Convert coded data to qlm_coded format*

Description

Converts a data frame or quanteda corpus of coded data (human-coded or from external sources) into a `qlm_coded` object. This enables provenance tracking and integration with `qlm_compare()`, `qlm_validate()`, and `qlm_trail()` for coded data alongside LLM-coded results.

Usage

```
as_qlm_coded(
  x,
  id,
  name = NULL,
  is_gold = FALSE,
  codebook = NULL,
  texts = NULL,
  notes = NULL,
  metadata = list()
)

## S3 method for class 'data.frame'
as_qlm_coded(
  x,
  id,
  name = NULL,
  is_gold = FALSE,
  codebook = NULL,
  texts = NULL,
  notes = NULL,
  metadata = list()
)

## Default S3 method:
as_qlm_coded(
  x,
  id,
  name = NULL,
  is_gold = FALSE,
  codebook = NULL,
  texts = NULL,
  notes = NULL,
  metadata = list()
)
```

Arguments

`x`	A data frame or quanteda corpus object containing coded data. For data frames: Must include a column with unit identifiers (default `".id"`). For corpus objects: Document variables (docvars) are treated as coded variables, and document names are used as identifiers by default.
`id`	For data frames: Name of the column containing unit identifiers (supports both quoted and unquoted). Default is `NULL`, which looks for a column named `".id"`. Can be an unquoted column name (`id = doc_id`) or a quoted string (`id = "doc_id"`). For corpus objects: `NULL` (default) uses document names from `names(x)`, or specify a docvar name (quoted or unquoted) to use as identifiers.
`name`	Character. a string identifying this coding run (e.g., "Coder_A", "expert_rater", "Gold_Standard"). Default is `NULL`.
`is_gold`	Logical. If `TRUE`, marks this object as a gold standard for automatic detection by `qlm_validate()`. When a gold standard object is passed to `qlm_validate()`, the `gold =` parameter becomes optional. Default is `FALSE`.
`codebook`	Optional list containing coding instructions. Can include: `name` Name of the coding scheme `instructions` Text describing coding instructions

schema NULL (not used for human coding)

If NULL (default), a minimal placeholder codebook is created.

texts	Optional vector of original texts or data that were coded. Should correspond to the .id values in data. If provided, enables more complete provenance tracking.
notes	Optional character string with descriptive notes about this coding. Useful for documenting details when viewing results in qlm_trail(). Default is NULL.
metadata	Optional list of metadata about the coding process. Can include any relevant information such as:

coder_name Name of the human coder

coder_id Identifier for the coder

training Description of coder training

date Date of coding

The function automatically adds timestamp, n_units, notes, and source = "human".

Details

When printed, objects created with as_qlm_coded() display "Source: Human coder" instead of model information, clearly distinguishing human from LLM coding.

Gold Standards:

Objects marked with is_gold = TRUE are automatically detected by qlm_validate(), allowing simpler syntax:

```
# With is_gold = TRUE
gold <- as_qlm_coded(gold_data, name = "Expert", is_gold = TRUE)
qlm_validate(coded1, coded2, gold, by = "sentiment")  # gold = not needed!

# Without is_gold (or explicit gold =)
gold <- as_qlm_coded(gold_data, name = "Expert")
qlm_validate(coded1, coded2, gold = gold, by = "sentiment")
```

Value

A qlm_coded object (tibble with additional class and attributes) for provenance tracking. When is_gold = TRUE, the object is marked as a gold standard in its attributes.

See Also

qlm_code() for LLM coding, qlm_compare() for inter-rater reliability, qlm_validate() for validation against gold standards, qlm_trail() for provenance tracking.

Examples

```
# Basic usage with data frame (default .id column)
human_data <- data.frame(
  .id = 1:10,
  sentiment = sample(c("pos", "neg"), 10, replace = TRUE)
)

coder_a <- as_qlm_coded(human_data, name = "Coder_A")
coder_a

# Use custom id column with NSE (unquoted)
data_with_custom_id <- data.frame(
  doc_id = 1:10,
  sentiment = sample(c("pos", "neg"), 10, replace = TRUE)
)
coder_custom <- as_qlm_coded(data_with_custom_id, id = doc_id, name = "Coder_C")

# Or use quoted string
coder_custom2 <- as_qlm_coded(data_with_custom_id, id = "doc_id", name = "Coder_D")

# Create a gold standard from data frame
gold <- as_qlm_coded(
  human_data,
  name = "Expert",
  is_gold = TRUE
)

# Validate with automatic gold detection
coder_b_data <- data.frame(
  .id = 1:10,
  sentiment = sample(c("pos", "neg"), 10, replace = TRUE)
)
coder_b <- as_qlm_coded(coder_b_data, name = "Coder_B")

# No need for gold = when gold object is marked (NSE works for 'by' too)
qlm_validate(coder_a, coder_b, gold = gold, by = sentiment, level = "nominal")

# Create from corpus object (simplified workflow)
data("data_corpus_manifsentsUK2010sample")
crowd <- as_qlm_coded(
  data_corpus_manifsentsUK2010sample,
  is_gold = TRUE
)
# Document names automatically become .id, all docvars included

# Use a docvar as identifier with NSE (unquoted)
crowd_party <- as_qlm_coded(
  data_corpus_manifsentsUK2010sample,
  id = party,
  is_gold = TRUE
)

# Or use quoted string
crowd_party2 <- as_qlm_coded(
  data_corpus_manifsentsUK2010sample,
  id = "party",
  is_gold = TRUE
)
```

```
# With complete metadata
expert <- as_qlm_coded(
  human_data,
  name = "expert_rater",
  is_gold = TRUE,
  codebook = list(
    name = "Sentiment Analysis",
    instructions = "Code overall sentiment as positive or negative"
  ),
  metadata = list(
    coder_name = "Dr. Smith",
    coder_id = "EXP001",
    training = "5 years experience",
    date = "2024-01-15"
  )
)
```

---

codebook — *Extract codebook from quallmer objects*

Description

Extracts the codebook component from `qlm_coded`, `qlm_comparison`, and `qlm_validation` objects. The codebook is a constitutive part of the coding run, defining the coding instrument used.

Usage

```
codebook(x)
```

Arguments

x	A quallmer object (`qlm_coded`, `qlm_comparison`, or `qlm_validation`).

Details

The codebook is a core component of coded objects, analogous to `formula()` for `lm` objects. It specifies the coding instrument (instructions, schema, role) used in the coding run.

This function is an extractor for the codebook component, not a metadata accessor. For codebook metadata (name, instructions), use `qlm_meta()`.

Note: `qlm_codebook()` is the constructor for creating codebooks; `codebook()` is the extractor for retrieving them from coded objects.

Value

A `qlm_codebook` object, or `NULL` if no codebook is available.

See Also

- accessors for an overview of the accessor function system
- `qlm_codebook()` for creating codebooks
- `qlm_meta()` for extracting metadata
- `inputs()` for extracting input data

Examples

```
# Load example objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))
coded <- examples$example_coded_sentiment

# Extract codebook
cb <- codebook(coded)
cb

# Access codebook metadata
qlm_meta(cb, "name")
```

`data_codebook_immigration`

Immigration policy codebook based on Benoit et al. (2016)

Description

A `qlm_codebook` object defining instructions for annotating whether a text pertains to immigration policy and, if so, the stance toward immigration openness. This codebook replicates the crowd-sourced annotation task from Benoit et al. (2016) and is designed to work with data_corpus_manifsentsUK2010sample.

Usage

```
data_codebook_immigration
```

Format

A `qlm_codebook` object containing:

name Task name: "Immigration policy coding from Benoit et al. (2016)"

instructions Coding instructions for identifying whether sentences from UK 2010 election manifestos pertain to immigration policy, and if so, rating the policy position expressed

schema Response schema with two fields: `llm_immigration_label` (Enum: "Not immigration" or "Immigration" indicating whether the sentence relates to immigration policy), and `llm_immigration_position` (Integer from -1 to 1, where -1 = pro-immigration, 0 = neutral, and 1 = anti-immigration)

input_type "text"

levels Named character vector: llm_immigration_label = "nominal", llm_immigration_position = "ordinal"

References

Benoit, K., Conway, D., Lauderdale, B.E., Laver, M., & Mikhaylov, S. (2016). Crowd-sourced Text Analysis: Reproducible and Agile Production of Political Data. *American Political Science Review*, 110(2), 278–295. doi:10.1017/S0003055416000058

See Also

`qlm_codebook()`, `qlm_code()`, data_corpus_manifsentsUK2010sample

Examples

```
# View the codebook
data_codebook_immigration

## Not run:
# Use with UK manifesto sentences (requires API key)
if (requireNamespace("quanteda", quietly = TRUE)) {
  coded <- qlm_code(data_corpus_manifsentsUK2010sample,
                    data_codebook_immigration,
                    model = "openai/gpt-4o-mini")

  # Compare with crowd-sourced annotations
  crowd <- as_qlm_coded(
    data.frame(
      .id = docnames(data_corpus_manifsentsUK2010sample),
      docvars(data_corpus_manifsentsUK2010sample)
    ),
    is_gold = TRUE
  )

  qlm_validate(coded, gold = crowd)

}

## End(Not run)
```

`data_codebook_sentiment` *Sentiment analysis codebook for movie reviews*

Description

A `qlm_codebook` object defining instructions for sentiment analysis of movie reviews. Designed to work with data_corpus_LMRDsample but with an expanded polarity scale that includes a "mixed" category.

Usage

```
data_codebook_sentiment
```

Format

A qlm_codebook object containing:

name Task name: "Movie Review Sentiment"

instructions Coding instructions for analyzing movie review sentiment

schema Response schema with two fields: polarity (Enum of "neg", "mixed", or "pos") and rating (Integer from 1 to 10)

role Expert film critic persona

input_type "text"

See Also

qlm_codebook(), qlm_code(), qlm_compare(), data_corpus_LMRDsample

Examples

```
# View the codebook
data_codebook_sentiment


# Use with movie review corpus (requires API key)
coded <- qlm_code(data_corpus_LMRDsample[1:10],
                  data_codebook_sentiment,
                  model = "openai")

# Create multiple coded versions for comparison
coded1 <- qlm_code(data_corpus_LMRDsample[1:20],
                   data_codebook_sentiment,
                   model = "openai/gpt-4o-mini")
coded2 <- qlm_code(data_corpus_LMRDsample[1:20],
                   data_codebook_sentiment,
                   model = "openai/gpt-4o")

# Compare inter-rater reliability
comparison <- qlm_compare(coded1, coded2, by = "rating", level = "interval")
print(comparison)
```

---

data_corpus_LMRDsample   *Sample from Large Movie Review Dataset (Maas et al. 2011)*

Description

A sample of 100 positive and 100 negative reviews from the Maas et al. (2011) dataset for sentiment classification. The original dataset contains 50,000 highly polar movie reviews.

Usage

```
data_corpus_LMRDsample
```

Format

The corpus docvars consist of:

docnumber serial (within set and polarity) document number

rating user-assigned movie rating on a 1-10 point integer scale

polarity either neg or pos to indicate whether the movie review was negative or positive. See Maas et al (2011) for the cut-off values that governed this assignment.

Source

http://ai.stanford.edu/~amaas/data/sentiment/

References

Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). "Learning Word Vectors for Sentiment Analysis". The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).

See Also

data_codebook_sentiment for an example codebook and usage with this corpus

Examples

```
if (requireNamespace("quanteda", quietly = TRUE)) {
  # Inspect the corpus
  summary(data_corpus_LMRDsample)

  # Sample a few reviews
  head(data_corpus_LMRDsample, 3)
}
```

---

data_corpus_manifsentsUK2010sample
Sample of UK manifesto sentences 2010 crowd-annotated for immigration

Description

A corpus of sentences sampled from from publicly available party manifestos from the United Kingdom from the 2010 election. Each sentence has been rated in terms of its classification as pertaining to immigration or not and then on a scale of favorability or not toward open immigration policy (as the mean score of crowd coders on a scale of -1 (favours open immigration policy), 0 (neutral), or 1 (anti-immigration).

The sentences were sampled from the corpus used in Benoit et al. (2016) doi:10.1017/S0003055416000058, which contains more information on the crowd-sourced annotation approach.

Usage

```
data_corpus_manifsentsUK2010sample
```

Format

A corpus object. The corpus consists of 155 sentences randomly sampled from the party manifestos, with an attempt to balance the sentencs according to their categorisation as pertaining to immigration or not, as well as by party. The corpus contains the following document-level variables:

party factor; abbreviation of the party that wrote the manifesto.

partyname factor; party that wrote the manifesto.

year integer; 4-digit year of the election.

immigration_label Factor indicating whether the majority of crowd workers labelled a sentence as referring to immigration or not. The variable has missing values (`NA`) for all non-annotated manifestos.

immigration_mean numeric; the direction of statements coded as "Immigration" based on the aggregated crowd codings. The variable is the mean of the scores assigned by workers who coded a sentence and who allocated the sentence to the "Immigration" category. The variable ranges from -1 (Favorable and open immigration policy) to +1 ("Negative and closed immigration policy").

immigration_n integer; the number of coders who contributed to the mean score `immigration_mean`.

immigration_position integer; a thresholded version of `immigration_mean` coded as -1 (pro-immigration, mean < -0.5), 0 (neutral, -0.5 <= mean <= 0.5), or 1 (anti-immigration, mean > 0.5). Set to `NA` for non-immigration sentences.

References

Benoit, K., Conway, D., Lauderdale, B.E., Laver, M., & Mikhaylov, S. (2016). Crowd-sourced Text Analysis: Reproducible and Agile Production of Political Data. *American Political Science Review*, 100,(2), 278–295. doi:10.1017/S0003055416000058

Examples

```
if (requireNamespace("quanteda", quietly = TRUE)) {
  # Inspect the corpus
  summary(data_corpus_manifsentsUK2010sample)
}
```

data_corpus_ms2020sample   *Sample corpus of political speeches from Maerz & Schneider (2020)*

Description

A corpus of 100 speeches from the Maerz & Schneider (2020) corpus, balanced across regime types (50 autocracies, 50 democracies). This sample is included in the package for demos and testing. The full corpus of 4,740 speeches is available in the package's pkgdown examples folder.

Usage

```
data_corpus_ms2020sample
```

Format

A corpus object. The corpus consists of 100 speeches randomly sampled from 40 heads of government across 27 countries, balanced by regime type. The corpus contains the following document-level variables:

speaker Character. Name of the head of government.

country Character. Country name.

regime Factor. Regime type: "Democracy" or "Autocracy".

score Numeric. Original dictionary-based liberal-illiberal score.

date Date. Date of the speech.

title Character. Title of the speech.

References

Maerz, S. F., & Schneider, C. Q. (2020). Comparing public communication in democracies and autocracies: Automated text analyses of speeches by heads of government. *Quality & Quantity*, 54, 517-545. doi:10.1007/s11135019008857

Examples

```
if (requireNamespace("quanteda", quietly = TRUE)) {
  # Inspect the corpus
  summary(data_corpus_ms2020sample, n = 10)

  # Regime distribution
  table(data_corpus_ms2020sample$regime)

  # View a sample speech
  cat(data_corpus_ms2020sample[1])
}
```

`inputs` *Extract input data from qlm_coded objects*

Description

Extracts the original input data (texts or image paths) from `qlm_coded` objects. The inputs are the source material that was coded, constituting a core component of the coded object.

Usage

```
inputs(x)
```

Arguments

`x`	A `qlm_coded` object.

Details

The inputs are a core component of coded objects, representing the source material that was coded. Like `codebook()`, this is a component extractor rather than a metadata accessor.

The function name mirrors the `inputs` argument in `qlm_code()`, providing a direct conceptual mapping: what is passed in via `inputs =` is retrieved back via `inputs()`.

Value

The original input data: a character vector of texts (for text codebooks) or file paths to images (for image codebooks). If the original input had names, these are preserved.

See Also

- accessors for an overview of the accessor function system
- `qlm_code()` for creating coded objects
- `codebook()` for extracting the codebook
- `qlm_meta()` for extracting metadata

Examples

```
# Load example objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))
coded <- examples$example_coded_sentiment

# Extract inputs
texts <- inputs(coded)
texts
```

`qlm_code` *Code qualitative data with an LLM*

Description

Applies a codebook to input data using a large language model, returning a rich object that includes the codebook, execution settings, results, and metadata for reproducibility.

Usage

```
qlm_code(x, codebook, model, ..., batch = FALSE, name = NULL, notes = NULL)
```

Arguments

`x`	Input data: a character vector of texts (for text codebooks) or file paths to images (for image codebooks). Named vectors will use names as identifiers in the output; unnamed vectors will use sequential integers.
`codebook`	A codebook object created with `qlm_codebook()`. Also accepts deprecated `task()` objects for backward compatibility.
`model`	Provider (and optionally model) name in the form `"provider/model"` or `"provider"` (which will use the default model for that provider). Passed to the `name` argument of `ellmer::chat()`. Examples: `"openai/gpt-4o-mini"`, `"anthropic/claude-3-5-sonnet-202` `"ollama/llama3.2"`, `"openai"` (uses default OpenAI model).
`...`	Additional arguments passed to `ellmer::chat()`, `ellmer::parallel_chat_structured()`, or `ellmer::batch_chat_structured()`, based on argument name. Arguments recognized by `ellmer::parallel_chat_structured()` take priority when there are overlaps. Batch-specific arguments (`path`, `wait`, `ignore_hash`) are only used when `batch = TRUE`. Arguments not recognized by any function will generate a warning.
`batch`	Logical. If `TRUE`, uses `ellmer::batch_chat_structured()` instead of `ellmer::parallel_chat_struc` Batch processing is more cost-effective for large jobs but may have longer turnaround times. Default is `FALSE`. See `ellmer::batch_chat_structured()` for details.
`name`	Character string identifying this coding run. Default is `NULL`.
`notes`	Optional character string with descriptive notes about this coding run. Useful for documenting the purpose or rationale when viewing results in `qlm_trail()`. Default is `NULL`.

Details

Arguments in `...` are dynamically routed to either `ellmer::chat()`, `ellmer::parallel_chat_structured()`, or `ellmer::batch_chat_structured()` based on their names.

Progress indicators and error handling are provided by the underlying `ellmer::parallel_chat_structured()` or `ellmer::batch_chat_structured()` function. Set `verbose = TRUE` to see progress messages during coding. Retry logic for API failures should be configured through ellmer's options.

When `batch` = `TRUE`, the function uses `ellmer::batch_chat_structured()` which submits jobs to the provider's batch API. This is typically more cost-effective but has longer turnaround times. The `path` argument specifies where batch results are cached, `wait` controls whether to wait for completion, and `ignore_hash` can force reprocessing of cached results.

Value

A `qlm_coded` object (a tibble with additional attributes):

Data columns The coded results with a `.id` column for identifiers.

Attributes `data`, `input_type`, and `run` (list containing name, batch, call, codebook, chat_args, execution_args, metadata, parent).

The object prints as a tibble and can be used directly in data manipulation workflows. The `batch` flag in the `run` attribute indicates whether batch processing was used. The `execution_args` contains all non-chat execution arguments (for either parallel or batch processing).

See Also

`qlm_codebook()` for creating codebooks, `qlm_replicate()` for replicating coding runs, `qlm_compare()` and `qlm_validate()` for assessing reliability.

Examples

```
# Basic sentiment analysis
texts <- c("I love this product!", "Terrible experience.", "It's okay.")
coded <- qlm_code(texts, data_codebook_sentiment, model = "openai/gpt-4o-mini")
coded

# With named inputs (names become IDs in output)
texts_named <- c(review1 = "Great service!", review2 = "Very disappointing.")
coded2 <- qlm_code(texts_named, data_codebook_sentiment, model = "openai/gpt-4o-mini")
coded2
```

`qlm_codebook` *Define a qualitative codebook*

Description

Creates a codebook definition for use with `qlm_code()`. A codebook specifies what information to extract from input data, including the instructions that guide the LLM and the structured output schema.

Usage

```
qlm_codebook(
  name,
  instructions,
  schema,
  role = NULL,
  input_type = c("text", "image"),
  levels = NULL
)
```

Arguments

`name`	Name of the codebook (character).
`instructions`	Instructions to guide the model in performing the coding task.
`schema`	Structured output definition, e.g., created by `ellmer::type_object()`, `ellmer::type_array()`, or `ellmer::type_enum()`.
`role`	Optional role description for the model (e.g., "You are an expert annotator"). If provided, this will be prepended to the instructions when creating the system prompt.
`input_type`	Type of input data: `"text"` (default) or `"image"`.
`levels`	Optional named list specifying measurement levels for each variable in the schema. Names should match schema property names. Values should be one of `"nominal"`, `"ordinal"`, `"interval"`, or `"ratio"`. If `NULL` (default), levels are auto-detected from schema types using the following mapping: `type_boolean` and `type_enum` = nominal, `type_string` = nominal, `type_integer` = ordinal, `type_number` = interval.

Details

This function replaces `task()`, which is now deprecated. The returned object has dual class inheritance (`c("qlm_codebook", "task")`) to maintain backward compatibility.

Value

A codebook object (a list with class `c("qlm_codebook", "task")`) containing the codebook definition. Use with `qlm_code()` to apply the codebook to data.

See Also

`qlm_code()` for applying codebooks to data, data_codebook_sentiment for a predefined codebook example, `task()` for the deprecated function.

Examples

```
# Define a custom codebook
my_codebook <- qlm_codebook(
  name = "Sentiment",
  instructions = "Rate the sentiment from -1 (negative) to 1 (positive).",
  schema = type_object(
    score = type_number("Sentiment score from -1 to 1"),
    explanation = type_string("Brief explanation")
  )
)

# With a role
my_codebook_role <- qlm_codebook(
  name = "Sentiment",
  instructions = "Rate the sentiment from -1 (negative) to 1 (positive).",
  schema = type_object(
    score = type_number("Sentiment score from -1 to 1"),
    explanation = type_string("Brief explanation")
  ),
  role = "You are an expert sentiment analyst."
)

# With explicit measurement levels
my_codebook_levels <- qlm_codebook(
  name = "Sentiment",
  instructions = "Rate the sentiment from -1 (negative) to 1 (positive).",
  schema = type_object(
    score = type_number("Sentiment score from -1 to 1"),
    explanation = type_string("Brief explanation")
  ),
  levels = list(score = "interval", explanation = "nominal")
)


# Use with qlm_code() (requires API key)
texts <- c("I love this!", "This is terrible.")
coded <- qlm_code(texts, my_codebook, model = "openai/gpt-4o-mini")
coded
```

qlm_compare *Compare coded results for inter-rater reliability*

Description

Compares two or more data frames or qlm_coded objects to assess inter-rater reliability or agreement. This function extracts a specified variable from each object and computes reliability statistics using the irr package.

Usage

```
qlm_compare(
  ...,
  by,
  level = NULL,
  tolerance = 0,
  ci = c("none", "analytic", "bootstrap"),
  bootstrap_n = 1000
)
```

Arguments

`...`	Two or more data frames, `qlm_coded`, or `as_qlm_coded` objects to compare. These represent different "raters" (e.g., different LLM runs, different models, human coders, or human vs. LLM coding). Each object must have a `.id` column and the variable specified in `by`. Objects should have the same units (matching `.id` values). Plain data frames are automatically converted to `as_qlm_coded` objects.
`by`	Optional. Name of the variable(s) to compare across raters (supports both quoted and unquoted). If `NULL` (default), all coded variables are compared. Can be a single variable (`by = sentiment`), a character vector (`by = c("sentiment", "rating")`), or NULL to process all variables.
`level`	Optional. Measurement level(s) for the variable(s). Can be: • `NULL` (default): Auto-detect from codebook • Character scalar: Use same level for all variables • Named list: Specify level for each variable. Valid levels are `"nominal"`, `"ordinal"`, `"interval"`, or `"ratio"`.
`tolerance`	Numeric. Tolerance for agreement with numeric data. Default is 0 (exact agreement required). Used for percent agreement calculation.
`ci`	Confidence interval method: `"none"` No confidence intervals (default) `"analytic"` Analytic CIs where available (ICC, Pearson's r) `"bootstrap"` Bootstrap CIs for all metrics via resampling
`bootstrap_n`	Number of bootstrap resamples when `ci = "bootstrap"`. Default is 1000. Ignored when `ci` is `"none"` or `"analytic"`.

Details

The function merges the coded objects by their `.id` column and only includes units that are present in all objects. Missing values in any rater will exclude that unit from analysis.

Measurement levels and statistics:

- **Nominal**: For unordered categories. Computes Krippendorff's alpha, Cohen's/Fleiss' kappa, and percent agreement.
- **Ordinal**: For ordered categories. Computes Krippendorff's alpha (ordinal), weighted kappa (2 raters only), Kendall's W, Spearman's rho, and percent agreement.
- **Interval**: For continuous data with meaningful intervals. Computes Krippendorff's alpha (interval), ICC, Pearson's r, and percent agreement.

- **Ratio**: For continuous data with a true zero point. Computes the same measures as interval level, but Krippendorff's alpha uses the ratio-level formula which accounts for proportional differences.

Kendall's W, ICC, and percent agreement are computed using all raters simultaneously. For 3 or more raters, Spearman's rho and Pearson's r are computed as the mean of all pairwise correlations between raters.

Value

A `qlm_comparison` object (a tibble/data frame) with the following columns:

`variable`	Name of the compared variable
`level`	Measurement level used
`measure`	Name of the reliability metric
`value`	Computed value of the metric
`rater1`, `rater2`, **...**	Names of the compared objects (one column per rater)
`ci_lower`	Lower bound of confidence interval (only if `ci != "none"`)
`ci_upper`	Upper bound of confidence interval (only if `ci != "none"`)

The object has class `c("qlm_comparison", "tbl_df", "tbl", "data.frame")` and attributes containing metadata (`raters`, `n`, `call`).

Metrics computed by measurement level:

- **Nominal:** alpha_nominal, kappa (Cohen's/Fleiss'), percent_agreement
- **Ordinal:** alpha_ordinal, kappa_weighted (2 raters only), w (Kendall's W), rho (Spearman's), percent_agreement
- **Interval/Ratio:** alpha_interval/alpha_ratio, icc, r (Pearson's), percent_agreement

Confidence intervals:

- `ci = "analytic"`: Provides analytic CIs for ICC and Pearson's r only
- `ci = "bootstrap"`: Provides bootstrap CIs for all metrics via resampling

See Also

`qlm_validate()` for validation of coding against gold standards, `qlm_code()` for LLM coding, `as_qlm_coded()` for human coding.

Examples

```
# Load example coded objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))

# Compare two coding runs
comparison <- qlm_compare(
  examples$example_coded_sentiment,
  examples$example_coded_mini,
  by = "sentiment",
  level = "nominal"
)
print(comparison)

# Compare specific variables with explicit levels
qlm_compare(
  examples$example_coded_sentiment,
  examples$example_coded_mini,
  by = "sentiment"
)
```

qlm_meta *Get or set quallmer object metadata*

Description

Get or set metadata from `qlm_coded`, `qlm_codebook`, `qlm_comparison`, and `qlm_validation` objects. Metadata is organized into three types: user, object, and system. Only user metadata can be modified.

Usage

```
qlm_meta(x, field = NULL, type = c("user", "object", "system", "all"))

qlm_meta(x, field = NULL) <- value
```

Arguments

`x`	A quallmer object (`qlm_coded`, `qlm_codebook`, `qlm_comparison`, or `qlm_validation`).
`field`	Optional character string specifying a single metadata field to extract or set. If `NULL` (default), `qlm_meta()` returns all metadata of the specified type, and `qlm_meta<-()` expects `value` to be a named list.
`type`	Character string specifying the type of metadata to extract: `"user"` User-specified descriptive information (default). These fields are modifiable via `qlm_meta<-()`: `name` (run label) and `notes` (documentation). `"object"` Parameters defining how coding was executed. Read-only fields include: `batch`, `call`, `chat_args`, `execution_args`, `parent`, `n_units`, `input_type`. `"system"` Automatically captured environment information. Read-only fields include: `timestamp`, `ellmer_version`, `quallmer_version`, `R_version`. `"all"` Returns a named list combining all three types.
`value`	For `qlm_meta<-()`, the new value for the metadata field, or a named list of user metadata fields.

Details

Metadata is stratified into three types following the quanteda convention:

User metadata (`type = "user"`, default): User-specified descriptive information that can be modified via `qlm_meta<-()`. Fields: `name`, `notes`.

Object metadata (`type = "object"`): Parameters and intrinsic properties set at object creation time. Read-only. Fields vary by object type but typically include: `batch`, `call`, `chat_args`, `execution_args`, `parent`, `n_units`, `input_type`.

System metadata (`type = "system"`): Automatically captured environment and version information. Read-only. Fields: `timestamp`, `ellmer_version`, `quallmer_version`, `R_version`.

For `qlm_codebook` objects, user metadata includes `name` and `instructions` (the codebook instructions text), both of which can be modified.

Modification via `qlm_meta<-()` (assignment):

Only user metadata can be modified. For `qlm_coded`, `qlm_comparison`, and `qlm_validation` objects, modifiable fields are `name` and `notes`. For `qlm_codebook` objects, modifiable fields are `name` and `instructions`.

Object and system metadata are read-only and set at creation time. Attempting to modify these will produce an informative error.

Value

`qlm_meta()` returns the requested metadata (a named list or single value). `qlm_meta<-()` returns the modified object (invisibly).

See Also

- accessors for an overview of the accessor function system
- `codebook()` for extracting the codebook component
- `inputs()` for extracting input data

Examples

```
# Load example objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))
coded <- examples$example_coded_sentiment

# User metadata (default)
qlm_meta(coded)
qlm_meta(coded, "name")

# Object metadata
qlm_meta(coded, type = "object")
qlm_meta(coded, "call", type = "object")
qlm_meta(coded, "n_units", type = "object")

# System metadata
qlm_meta(coded, type = "system")
qlm_meta(coded, "timestamp", type = "system")
```

```
# All metadata
qlm_meta(coded, type = "all")

# Modify user metadata
qlm_meta(coded, "name") <- "updated_run"
qlm_meta(coded, "notes") <- "Analysis notes"

# Set multiple fields at once
qlm_meta(coded) <- list(name = "final_run", notes = "Final analysis")

## Not run:
# This will error - object and system metadata are read-only
qlm_meta(coded, "timestamp") <- Sys.time()

## End(Not run)
```

qlm_replicate *Replicate a coding task*

Description

Re-executes a coding task from a `qlm_coded` object, optionally with modified settings. If no overrides are provided, uses identical settings to the original coding.

Usage

```
qlm_replicate(
  x,
  ...,
  codebook = NULL,
  model = NULL,
  batch = NULL,
  name = NULL,
  notes = NULL
)
```

Arguments

`x`	A `qlm_coded` object.
`...`	Optional overrides passed to `qlm_code()`, such as `temperature` or `max_tokens`.
`codebook`	Optional replacement codebook. If `NULL` (default), uses the codebook from `x`.
`model`	Optional replacement model (e.g., `"openai/gpt-4o"`). If `NULL` (default), uses the model from `x`.
`batch`	Optional logical to override batch processing setting. If `NULL` (default), uses the batch setting from `x`. Set to `TRUE` to use batch processing or `FALSE` to use parallel processing, regardless of the original setting.

`name`	Optional name for this run. If `NULL`, defaults to the model name (if changed) or `"replication_N"` where N is the replication count.
`notes`	Optional character string with descriptive notes about this replication. Useful for documenting why this replication was run or what differs from the original. Default is `NULL`.

Value

A `qlm_coded` object with `run$parent` set to the parent's run name.

See Also

`qlm_code()` for initial coding, `qlm_compare()` for comparing replicated results.

Examples

```
# First create a coded object
texts <- c("I love this!", "Terrible.", "It's okay.")
coded <- qlm_code(texts, data_codebook_sentiment, model = "openai/gpt-4o-mini", name = "run1")

# Replicate with same model
coded2 <- qlm_replicate(coded, name = "run2")

# Compare results
qlm_compare(coded, coded2, by = "sentiment", level = "nominal")
```

`qlm_trail` *Create an audit trail from quallmer objects*

Description

Creates a complete audit trail documenting your qualitative coding workflow. Following Lincoln and Guba's (1985) concept of the audit trail for establishing trustworthiness in qualitative research, this function captures the full decision history of your AI-assisted coding process.

Usage

```
qlm_trail(..., path = NULL)
```

Arguments

`...`	One or more quallmer objects (`qlm_coded`, `qlm_comparison`, or `qlm_validation`). When multiple objects are provided, they will be used to reconstruct the complete workflow chain.
`path`	Optional base path for saving the audit trail. When provided, creates `{path}.rds` (complete archive) and `{path}.qmd` (human-readable report). If `NULL` (default), the trail is only returned without saving.

Details

Lincoln and Guba (1985, pp. 319-320) describe six categories of audit trail materials for establishing trustworthiness in qualitative research. The quallmer package operationalizes these for LLM-assisted text analysis:

Raw data Original texts stored in coded objects

Data reduction products Coded results from each run

Data reconstruction products Comparisons and validations

Process notes Model parameters, timestamps, decision history

Materials relating to intentions Function calls documenting intent

Instrument development information Codebook with instructions and schema

When `path` is provided, the function creates:

- `{path}.rds`: Complete trail object for R (reloadable with `readRDS()`)
- `{path}.qmd`: Quarto document with full audit trail documentation

Value

A `qlm_trail` object containing:

runs List of run information with coded data, ordered from oldest to newest

complete Logical indicating whether all parent references were resolved

References

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic Inquiry*. Sage.

See Also

`qlm_code()`, `qlm_replicate()`, `qlm_compare()`, `qlm_validate()`

Examples

```
# Load example coded objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))

# View audit trail from two coding runs
trail <- qlm_trail(
  examples$example_coded_sentiment,
  examples$example_coded_mini
)
print(trail)


# Save complete audit trail (creates .rds and .qmd files)
qlm_trail(
  examples$example_coded_sentiment,
  examples$example_coded_mini,
  path = tempfile("my_analysis")
)
```

qlm_validate *Validate coded results against a gold standard*

Description

Validates LLM-coded results from one or more `qlm_coded` objects against a gold standard (typically human annotations) using appropriate metrics based on measurement level. For nominal data, computes accuracy, precision, recall, F1-score, and Cohen's kappa. For ordinal data, computes accuracy and weighted kappa (linear weighting), which accounts for the ordering and distance between categories.

Usage

```
qlm_validate(
  ...,
  gold,
  by,
  level = NULL,
  average = c("macro", "micro", "weighted", "none"),
  ci = c("none", "analytic", "bootstrap"),
  bootstrap_n = 1000
)
```

Arguments

`...`	One or more data frames, `qlm_coded`, or `as_qlm_coded` objects containing predictions to validate. Must include a `.id` column and the variable(s) specified in `by`. Plain data frames are automatically converted to `as_qlm_coded` objects. Multiple objects will be validated separately against the same gold standard, and results combined with a `rater` column to distinguish them.
`gold`	A data frame, `qlm_coded`, or object created with `as_qlm_coded()` containing gold standard annotations. Must include a `.id` column for joining with objects in `...` and the variable(s) specified in `by`. Plain data frames are automatically converted. **Optional** when using objects marked with `as_qlm_coded(data, is_gold = TRUE)` - these are auto-detected.
`by`	Optional. Name of the variable(s) to validate (supports both quoted and unquoted). If `NULL` (default), all coded variables are validated. Can be a single variable (by = `sentiment`), a character vector (by = `c("sentiment", "rating")`), or NULL to process all variables.
`level`	Optional. Measurement level(s) for the variable(s). Can be: • `NULL` (default): Auto-detect from codebook

- Character scalar: Use same level for all variables
- Named list: Specify level for each variable

Valid levels are `"nominal"`, `"ordinal"`, or `"interval"`.

`average`	Character scalar. Averaging method for multiclass metrics (nominal level only): `"macro"` Unweighted mean across classes (default); `"micro"` Aggregate contributions globally (sum TP, FP, FN); `"weighted"` Weighted mean by class prevalence; `"none"` Return per-class metrics in addition to global metrics
`ci`	Confidence interval method: `"none"` No confidence intervals (default); `"analytic"` Analytic CIs where available (ICC, Pearson's r); `"bootstrap"` Bootstrap CIs for all metrics via resampling
`bootstrap_n`	Number of bootstrap resamples when `ci = "bootstrap"`. Default is 1000. Ignored when `ci` is `"none"` or `"analytic"`.

Details

The function performs an inner join between `x` and `gold` using the `.id` column, so only units present in both datasets are included in validation. Missing values (NA) in either predictions or gold standard are excluded with a warning.

Measurement levels:

- **Nominal**: Categories with no inherent ordering (e.g., topics, sentiment polarity). Metrics: accuracy, precision, recall, F1-score, Cohen's kappa (unweighted).
- **Ordinal**: Categories with meaningful ordering but unequal intervals (e.g., ratings 1-5, Likert scales). Metrics: Spearman's rho (`rho`, rank correlation), Kendall's tau (`tau`, rank correlation), and MAE (`mae`, mean absolute error). These measures account for the ordering of categories without assuming equal intervals.
- **Interval/Ratio**: Numeric data with equal intervals (e.g., counts, continuous measurements). Metrics: ICC (intraclass correlation), Pearson's r (linear correlation), MAE (mean absolute error), and RMSE (root mean squared error).

For multiclass problems with nominal data, the `average` parameter controls how per-class metrics are aggregated:

- **Macro averaging** computes metrics for each class independently and takes the unweighted mean. This treats all classes equally regardless of size.
- **Micro averaging** aggregates all true positives, false positives, and false negatives globally before computing metrics. This weights classes by their prevalence.
- **Weighted averaging** computes metrics for each class and takes the mean weighted by class size.
- **No averaging** (`average = "none"`) returns global macro-averaged metrics plus per-class breakdown.

Note: The `average` parameter only affects precision, recall, and F1 for nominal data. For ordinal data, these metrics are not computed.

Value

A `qlm_validation` object (a tibble/data frame) with the following columns:

`variable` Name of the validated variable

`level` Measurement level used

`measure` Name of the validation metric

`value` Computed value of the metric

`class` For nominal data: averaging method used (e.g., "macro", "micro", "weighted") or class label (when `average = "none"`). For ordinal/interval data: NA (averaging not applicable).

`rater` Name of the object being validated (from input names)

`ci_lower` Lower bound of confidence interval (only if `ci != "none"`)

`ci_upper` Upper bound of confidence interval (only if `ci != "none"`)

The object has class `c("qlm_validation", "tbl_df", "tbl", "data.frame")` and attributes containing metadata (`n`, `call`).

Metrics computed by measurement level:

- **Nominal:** accuracy, precision, recall, f1, kappa
- **Ordinal:** rho (Spearman's), tau (Kendall's), mae
- **Interval:** icc, r (Pearson's), mae, rmse

Confidence intervals:

- `ci = "analytic"`: Provides analytic CIs for ICC and Pearson's r only
- `ci = "bootstrap"`: Provides bootstrap CIs for all metrics via resampling

See Also

`qlm_compare()` for inter-rater reliability between coded objects, `qlm_code()` for LLM coding, `as_qlm_coded()` for converting human-coded data, `yardstick::accuracy()`, `yardstick::precision()`, `yardstick::recall()`, `yardstick::f_meas()`, `yardstick::kap()`, `yardstick::conf_mat()`

Examples

```
# Load example coded objects
examples <- readRDS(system.file("extdata", "example_objects.rds", package = "quallmer"))

# Validate against gold standard (auto-detected)
validation <- qlm_validate(
  examples$example_coded_mini,
  examples$example_gold_standard,
  by = "sentiment",
  level = "nominal"
)
print(validation)

# Explicit gold parameter (backward compatible)
validation2 <- qlm_validate(
  examples$example_coded_mini,
  gold = examples$example_gold_standard,
  by = "sentiment",
  level = "nominal"
)
print(validation2)
```

Index