Package: credgate (via r-universe)

November 2, 2026

Title Detect Indian Personal Data Before It Reaches Language Models

Version 0.1.0.9000

Description Detects personal data in text before it is sent to a large language model, with Indian identifiers built in: Aadhaar (validated with the Verhoeff checksum), Aadhaar Virtual ID, PAN, GSTIN, voter ID, passport, driving licence, CKYC, ration card and job card numbers, UPI IDs, IFSC codes, bank account and payment card numbers with expiry dates and verification codes, mobile numbers and Ayushman Bharat (ABHA) health IDs, plus rule-based detection of names and addresses in Latin and Indic scripts. Candidates are validated with checksums, scored from surrounding context and matched through Indic digits and invisible characters. The rules come from a language-neutral specification shared with the 'Python' package of the same name.

License Apache License (>= 2)

URL https://dwarpal.github.io/credgate/r/, https://github.com/dwarpal/credgate

BugReports https://github.com/dwarpal/credgate/issues

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Depends R (>= 4.3)

Imports cli, jsonlite, rlang, stringi, yaml

Suggests arrow, bigrquery, askpass, DBI, dbplyr, dplyr, duckdb, knitr, openssl, rmarkdown, RSQLite, sodium, sparklyr, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Config/pak/sysreqs libicu-dev

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

Date/Publication 2026-11-01 21:52:31 UTC

RemoteUrl https://github.com/dwarpal/credgate

RemoteRef HEAD

RemoteSha 30f417bfdb7c2ef819fc6cb4ebd0815161510824

RemoteSubdir pkg/credgate

Contents

bounded . . . 3
bq_scan_pii . . . 3
cg_audit_log . . . 5
cg_detectors . . . 6
cg_evaluate . . . 7
cg_new_token_key . . . 8
cg_normalise . . . 8
cg_policy . . . 9
cg_read_corpus . . . 10
cg_spec_path . . . 10
cg_vault . . . 11
detect_pii . . . 12
gstin_valid . . . 13
luhn_valid . . . 13
new_detector . . . 14
pii_inventory . . . 16
policy_decide . . . 17
preview . . . 18
redact . . . 18
redact_parquet . . . 19
redact_query . . . 20
register_detector . . . 21
restore . . . 22
scan_pii . . . 22
synth_indian_pii . . . 23
tokenise . . . 24
vault_merge . . . 25
vault_save . . . 26
verhoeff_valid . . . 27
write_inventory . . . 27

Index **29**

bounded *Wrap a pattern so it only matches whole tokens*

Description

The match may not be preceded or followed by a letter or digit, nor by a digit joined with a space or hyphen (so a 12-digit window inside a 16-digit card number is not reported as an Aadhaar number).

Usage

```
bounded(core)
```

Arguments

`core`	An ICU regular expression.

Value

A regular expression.

Examples

```
p <- bounded("[0-9]{4}")
stringi::stri_detect_regex(c("PIN 1234", "card 12345678"), p)
```

bq_scan_pii *Scan and inventory BigQuery tables*

Description

BigQuery samples the table and only the sample is downloaded; only the columns credgate can scan are read, and a dry run checks the cost of every query first. credgate makes no network calls of its own: it uses bigrquery, with your credentials (`bigrquery::bq_auth()`). The rules are shared with the Python package (`credgate.bigquery`; see the spec's "BigQuery tables").

Usage

```
bq_scan_pii(
  table,
  billing = NULL,
  sample = 10000L,
  min_rate = 0,
  detectors = NULL,
  sampling = "tablesample",
  max_bytes = 1e+09
)
```

```
bq_pii_inventory(
  table,
  source = NULL,
  billing = NULL,
  sample = 10000L,
  min_rate = 0,
  detectors = NULL,
  sampling = "tablesample",
  max_bytes = 1e+09
)

bq_inventory_dataset(
  dataset,
  source = NULL,
  billing = NULL,
  sample = 10000L,
  min_rate = 0,
  detectors = NULL,
  sampling = "tablesample",
  max_bytes = 1e+09
)
```

Arguments

`table`	A table: `"project.dataset.table"` or a `bigrquery::bq_table()`.
`billing`	Project billed for the queries; defaults to the table's project.
`sample`	Maximum number of rows sampled.
`min_rate`, `detectors`	As for `scan_pii()`.
`sampling`	`"tablesample"` (BigQuery's block sampling; only the sampled blocks are billed) or `"rand"` (reads and bills whole columns). Views and external tables always use `"rand"`.
`max_bytes`	Refuse any query estimated to process more bytes than this (default 1 GB). Also sent to BigQuery as the billing limit, except with TABLESAMPLE, where BigQuery would check it against the whole table.
`source`	Source name for the inventory; defaults to `"bigquery:<project>"`. The table is recorded as `"dataset.table"`.
`dataset`	A dataset: `"project.dataset"` or a `bigrquery::bq_dataset()`. Every table and view in it is scanned.

Value

`bq_scan_pii()` a scan report as from `scan_pii()`; `bq_pii_inventory()` and `bq_inventory_dataset()` an inventory as from `pii_inventory()`, with the table's row count and `sample_method` `"all"`, `"bigquery-tablesample"` or `"bigquery-rand"`.

Examples

```
## Not run:
bigrquery::bq_auth()
bq_scan_pii("my-project.crm.customers")
inv <- bq_inventory_dataset("my-project.crm")
write_inventory(inv, "inventory.csv")

## End(Not run)
```

`cg_audit_log` *Audit log of policy decisions*

Description

`cg_audit_log()` opens an append-only, hash-chained log; pass it to `policy_apply()` (`audit = log`) to record every decision, allowed or blocked, or record a decision yourself with `audit_record()`. Each record says who sent what kind of data where, under which policy revision and rule - never the text or the values. With an audit key (`key` or `CREDGATE_AUDIT_KEY`; not the token key), identifiers are recorded as keyed hashes, so whoever holds the key can check whether a given identifier was involved. Card verification codes are never hashed.

Usage

```
cg_audit_log(path, key = NULL, actor = NULL)

audit_record(log, decision)

audit_verify(path)

audit_read(path)
```

Arguments

`path`	The log file (JSON Lines); created if missing.
`key`	Audit key (64+ hex digits or raw); `NULL` uses `CREDGATE_AUDIT_KEY` if set, otherwise identifiers are not hashed.
`actor`	Who makes the requests; defaults to the operating-system user.
`log`	A log from `cg_audit_log()`.
`decision`	A decision from `policy_decide()` or `policy_apply()`.

Details

`audit_verify()` checks that no record was edited, deleted, inserted or reordered; keep copies of the returned `head` elsewhere to detect records removed from the end. `audit_read()` returns the records. The same files are written and verified by the Python package. Needs the openssl package.

Value

`cg_audit_log()`: a log. `audit_record()`: the record (a list), invisibly. `audit_verify()`: a list with `ok`, `records`, `head`, and for a broken log the failing `line` and the `error`. `audit_read()`: a data frame, one row per record, with the `items` as a list column.

Examples

```
policy <- cg_policy(list(
  policy = "credgate-policy", version = 1L, id = "example", revision = 1L,
  destinations = list(claude = list(tier = "frontier", provider = "anthropic")),
 rules = list(list(id = "no-ids", action = "redact", when = list(classes = list("identity_strong")))),
  default_action = "redact"
))
path <- tempfile(fileext = ".jsonl")
log <- cg_audit_log(path, actor = "analyst1")
policy_apply(policy, "PAN ABCPE1234F", "claude", audit = log)
audit_verify(path)
audit_read(path)[, c("seq", "actor", "destination", "allowed")]
```

---

cg_detectors *List registered detectors*

Description

List registered detectors

Usage

```
cg_detectors()
```

Value

A data frame with one row per detector.

Examples

```
cg_detectors()
```

`cg_evaluate` *Evaluate detection against labelled text*

Description

Measures how well `detect_pii()` finds the PII in documents where every PII span has been labelled: true positives, false positives, false negatives, precision, recall and F1, per type and overall. Use it to check credgate on your own (labelled) data. The rules are shared with the Python package (see the spec's "Evaluation" section).

Usage

```
cg_evaluate(
  documents,
  min_confidence = 0,
  match = c("exact", "overlap"),
  detectors = NULL
)
```

Arguments

`documents`	Labelled documents: a list of lists with `text` and `spans` (each span a list with `start`, `end` - 0-based, end-exclusive character offsets - and `type`), or the path of a JSON Lines file in that format.
`min_confidence`	Ignore predictions scoring below this value.
`match`	`"exact"` (same type and offsets) or `"overlap"` (same type, overlapping spans).
`detectors`	Detector ids to run; `NULL` runs all registered ones.

Value

A data frame of class `cg_evaluation` with columns `type`, `tp`, `fp`, `fn`, `precision`, `recall` and `f1`: one row per type, then `overall` (micro-averaged). Rates are `NA` when undefined.

Examples

```
docs <- list(
  list(text = "PAN: ABCPE1234F", spans = list(list(start = 5, end = 15, type = "pan"))),
  list(text = "order no 878245671885", spans = list())
)
cg_evaluate(docs)
cg_evaluate(docs, min_confidence = 0.5)
```

cg_new_token_key *Token keys for keyed tokens*

Description

`cg_new_token_key()` makes a new random key (64 hex digits) for `cg_vault()`. `cg_key_fingerprint()` returns 16 hex digits that identify a key without revealing it; saved keyed vaults record it, so opening one with the wrong key fails. Both need the openssl package.

Usage

```
cg_new_token_key()

cg_key_fingerprint(key = NULL)
```

Arguments

`key`	A key as hex digits or a raw vector; `NULL` uses `CREDGATE_TOKEN_KEY`.

Value

A string.

Examples

```
key <- cg_new_token_key()
cg_key_fingerprint(key)
```

cg_normalise *Normalise text for PII detection*

Description

Applies NFKC normalisation, maps Indic and Arabic-Indic digits to ASCII, maps dash variants to - and removes invisible characters (zero-width space/joiners, BOM, soft hyphen). Returns the normalised text together with an offset map back to the original.

Usage

```
cg_normalise(x)
```

Arguments

`x`	A character vector.

Value

A list with one element per element of `x`, each a list with `text` (the normalised string) and `map` (an integer vector giving, for each code point of `text`, the index of the original code point it came from, or `NULL` when the text is unchanged).

Examples

```
cg_normalise("Aadhaar: \u0968\u0969\u096a\u096b")
```

`cg_policy` *Load a policy*

Description

A policy decides, for one request to one destination, what happens to each identifier found in it: let it through, redact it, tokenise it, or block the whole request. Policies are YAML files written by InfoSec and compliance; the same file works in R and Python. Everything is checked when the policy is loaded - unknown fields, tiers, actions, types or classes are errors - so a policy that cannot be read correctly is never applied.

Usage

```
cg_policy(x)
```

Arguments

`x`	Path to a policy file (YAML), or the policy as a list.

Details

Destinations have a tier: `local` (this machine), `internal` (the organisation's own network or hosting) or `frontier` (an external model provider). Rules are tried in order and the first that matches an identifier decides its action; a rule's `when` can name particular `destinations`, `tiers`, `types`, `classes`, and `all_of` - types that must all occur in the request (for example a card number with its CVV and expiry date). Any `block` blocks the whole request. A destination that is not listed receives nothing: requests to it are refused.

Value

An object of class `cg_policy`.

See Also

`policy_decide()`, `policy_apply()`; the format is described in the spec (`cg_spec_path()`, README, "Policies").

Examples

```
policy <- cg_policy(list(
  policy = "credgate-policy", version = 1L, id = "example", revision = 1L,
  destinations = list(ollama = list(tier = "local"), claude = list(tier = "frontier")),
  rules = list(
    list(
      id = "card-with-cvv", action = "block",
      when = list(tiers = list("frontier"), all_of = list("card", "card_cvv"))
    ),
    list(id = "in-house", action = "allow", when = list(tiers = list("local")))
  ),
  default_action = "redact"
))
policy_apply(policy, "Card 4111 1111 1111 1111, CVV 123", "claude")
policy_apply(policy, "PAN ABCPE1234F", "claude")
```

---

cg_read_corpus *Read labelled documents from a JSON Lines file*

Description

Read labelled documents from a JSON Lines file

Usage

```
cg_read_corpus(path)
```

Arguments

path	Path of a JSON Lines file with one labelled document per line (see cg_evaluate()).

Value

A list of documents.

---

cg_spec_path *Path to the detection spec shipped with the package*

Description

The spec (detectors, engine parameters and conformance cases) is shared by every CredGate implementation. See its README.md for the schema.

Usage

```
cg_spec_path()
```

Value

The path of the spec directory.

Examples

```
list.files(cg_spec_path())
```

`cg_vault` *Create a token vault*

Description

A vault holds the mapping between tokens such as `[PAN_1]` and the values they stand for. It starts empty and grows as `tokenise()` meets new identifiers; the same identifier always gets the same token. The vault lives in memory only, and printing it masks the values.

Usage

```
cg_vault(key = NULL, keyed = FALSE)
```

Arguments

`key`	A token key: at least 64 hex digits (32 bytes), or a raw vector. Make one with `cg_new_token_key()`.
`keyed`	Use keyed tokens with the key from `CREDGATE_TOKEN_KEY`.

Details

With a token `key`, tokens are **keyed** (`[PAN_4qzm7k2x9tbc3]`): the same identifier gets the same token in every vault with that key - across sessions, datasets, R and Python, and parallel workers, whose vaults can be combined with `vault_merge()`. Without a key, tokens are numbered (`[PAN_1]`) and belong to this vault only. Keyed tokens need the openssl package.

Whoever holds the key can confirm a guessed value by computing its token, so keep the key like a password: from a key store or the environment variable, never in code or project files.

Value

An object of class `cg_vault`. It is modified in place by `tokenise()`, so it does not need to be reassigned.

Examples

```
vault <- cg_vault()
tokenise("PAN ABCPE1234F", vault)
vault

key <- cg_new_token_key()
a <- cg_vault(key)
b <- cg_vault(key)
tokenise("PAN ABCPE1234F", a) == tokenise("pan abcpe1234f", b)
```

---

detect_pii *Detect PII in text*

Description

Runs every registered detector over normalised text, validates candidates (checksums, code tables), scores them from nearby context and resolves overlapping matches (the more confident, then the longer, span wins - so a GSTIN is reported once, not also as the PAN inside it).

Usage

```
detect_pii(
  text,
  detectors = NULL,
  context = NULL,
  min_confidence = 0,
  window = NULL
)
```

Arguments

text	A character vector.
detectors	Detector ids to run; NULL runs all registered detectors (see cg_detectors()).
context	Optional extra context per element of text (recycled), treated as if it appeared next to every match. scan_pii() (planned) passes column names here, so a column called voter_id supports Voter ID matches.
min_confidence	Drop matches scoring below this value.
window	Number of characters either side of a match searched for context keywords; NULL uses the spec's context_window.

Value

A data frame of class cg_spans with one row per match: doc (index into text), start, end (character positions in the original text), type, class, value, canonical, confidence and reasons. Printing masks the values.

Examples

```
detect_pii("Customer PAN ABCPE1234F, GSTIN 27AAPFU0939F1ZV")
```

`gstin_valid` *GSTIN checksum*

Description

A GSTIN is 15 characters: a 2-digit state code, the holder's PAN, an entity number, the letter `Z` and a mod-36 check character. `gstin_valid()` checks structure and check character; `gstin_check_char()` computes the check character for the first 14 characters.

Usage

```
gstin_valid(x)

gstin_check_char(x)
```

Arguments

`x`	A character vector. Case is ignored.

Value

`gstin_valid()`: a logical vector. `gstin_check_char()`: a character vector (`NA` for invalid input).

Examples

```
gstin_check_char("27AAPFU0939F1Z")
```

`luhn_valid` *Luhn checksum (payment cards)*

Description

Luhn checksum (payment cards)

Usage

```
luhn_valid(x)
```

Arguments

`x`	A character vector of digit strings. Spaces and hyphens are ignored.

Value

A logical vector.

Examples

```
luhn_valid("4111 1111 1111 1111")
```

`new_detector` *Define a PII detector*

Description

Define a PII detector

Usage

```
new_detector(
  id,
  class,
  label,
  pattern,
  capture = NULL,
  validate = NULL,
  checks = NULL,
  base = 0.8,
  base_without_context = NULL,
  context = NULL,
  negative = NULL,
  context_required = FALSE,
  canonical = NULL,
  describe = NULL,
  ignore_case = FALSE,
  column_label = NULL,
  column_pattern = NULL
)
```

Arguments

`id`	Unique identifier, e.g. `"aadhaar"`.
`class`	Data class used by policies, e.g. `"identity_strong"`.
`label`	Human-readable name.
`pattern`	ICU regular expression for candidates, or several (their matches are combined). Wrap it with `bounded()` so it does not match inside a longer token.
`capture`	Name of a capture group whose span is reported instead of the whole match, or `NULL`. Lets a pattern match a cue such as `"Name:"` without reporting it; much faster than a lookbehind.

`validate` Function taking the matched strings (normalised text) and returning a logical vector; candidates returning `FALSE` are dropped.

`checks` Short description of what `validate` checks, reported as a reason for each match (e.g. `"Verhoeff checksum valid"`).

`base` Confidence (0-1) of a validated match without supporting context, or a function taking the matched strings and returning one confidence per match (e.g. higher for a mobile number written with `+91`).

`base_without_context` Confidence (0-1) used instead of `base` when no context keyword is found, or `NULL`. For patterns that are strong evidence with context but weak without it (a voter ID format alone could be a product code).

`context` Regex (case-insensitive) of keywords that raise confidence when they appear near the match, or `NULL`.

`negative` Regex (case-insensitive) of keywords that lower confidence, or `NULL`.

`context_required` If `TRUE`, matches without supporting context are dropped. Use for patterns too generic to trust on their own.

`canonical` Function mapping matched strings to a canonical form (used for consistent tokenisation). Defaults to removing spaces and hyphens and upper-casing.

`describe` Optional function returning extra reasons per match (e.g. the PAN holder type).

`ignore_case` Match the pattern case-insensitively.

`column_label` Regex (case-insensitive) that, when it matches the caller's extra context (a column name), makes that context the label for `column_pattern`, or `NULL`. For detectors whose `pattern` needs a label in the text, so that a column named `card_cvv` labels its values.

`column_pattern` Patterns used, reporting the whole match, only when `column_label` matches the extra context.

Value

An object of class `cg_detector`.

Examples

```
employee_id <- new_detector(
  id = "employee_id", class = "internal", label = "Employee ID",
  pattern = bounded("EMP[0-9]{6}"),
  context = "employee|staff"
)
employee_id$label
```

`pii_inventory` *Record where PII is, for an inventory*

Description

Scans a table with the same rules as `scan_pii()`, but records every column - including those where nothing was found - with the source and table names, the sample size and method, and when and by what it was scanned. No values are recorded, not even masked ones. Combine the inventories of several tables with `rbind()` and save them with `write_inventory()`; the file format is shared with the Python package (see the spec's "Inventory" section).

Usage

```
pii_inventory(
  data,
  source,
  table,
  sample = 10000L,
  min_rate = 0,
  detectors = NULL,
  seed = 42L
)
```

Arguments

`data`	A data frame, an Arrow `Table` or `Dataset` (for example from `arrow::open_dataset()`), or a database table from dbplyr (`dplyr::tbl(con, "calls")`, for DuckDB, Postgres and other DBI databases). Arrow and database tables are read one column at a time, in chunks, so they may be larger than memory; the sample is the same as for the table in memory. For a database, rows must come back in the same order on each read (DuckDB and SQLite do for a plain table). A Spark table from sparklyr is sampled inside the cluster (up to `sample` rows, randomly with `seed`) and only the sample is collected.
`source`	Name of the data source, chosen by you (e.g. `"bigquery:prod"`, `"hive"`).
`table`	Name of the table within the source (e.g. `"crm.customers"`, or a file path).
`sample`	Maximum number of values checked per column.
`min_rate`	Report a type only if it occurs in at least this share of sampled values (0 reports every type found).
`detectors`	Detector ids to run; `NULL` runs all registered ones.
`seed`	Random seed for sampling Spark tables; ignored otherwise.

Value

A data frame of class `cg_inventory` with one row per column and PII type found, and one row with `type = NA` for each column where none was found: `v`, `source`, `table`, `column`, `type`, `class`, `n_rows`

(when known without reading the table), n_sampled, n_matched, n_full, rate, mean_confidence, sample_method ("all" or "even", or "spark-sample" for Spark tables), scanned_at (UTC) and credgate (the version that scanned).

Examples

```
customers <- data.frame(
  cust_pan = c("ABCPE1234F", "AAACR5055K", "XYZPQ9876R"),
  notes = c("Called, Aadhaar 542229371182", "no issues", "mobile 9876543210"),
  balance = c(1520.5, 99.75, 0.25)
)
inv <- pii_inventory(customers, source = "crm", table = "customers")
inv
path <- tempfile(fileext = ".csv")
write_inventory(inv, path)
read_inventory(path)
```

---

policy_decide *Decide and apply a policy to a request*

Description

policy_decide() works out what the policy does with each identifier in a request to destination, without changing the text. policy_apply() also rewrites the text when the request is allowed: identifiers to redact become placeholders such as [PAN], identifiers to tokenise become tokens from vault. A blocked request gets no text: nothing may be sent.

Usage

```
policy_decide(policy, text, destination)

policy_apply(policy, text, destination, vault = NULL, audit = NULL)
```

Arguments

policy	A policy from cg_policy().
text	The texts of one request (for example the messages of a chat).
destination	Name of a destination in the policy; an unknown one is an error.
vault	A vault from cg_vault(), needed when the policy tokenises.
audit	An audit log from cg_audit_log(): the decision is recorded, blocked or not.

Value

An object of class cg_decision: a list with policy_id, policy_revision, destination, tier, provider, url, allowed, blocked_by ("destination-blocked" and rule ids), matches (a data frame of the identifiers with their action and deciding rule, "default" when no rule matched) and text (the rewritten texts, or NULL).

preview *Preview what would be redacted*

Description

Shows the redacted text together with each detected identifier (value masked), its confidence and the reasons it was detected - so you can check exactly what would leave the machine before sending anything.

Usage

```
preview(text, detectors = NULL, context = NULL, min_confidence = 0)
```

Arguments

`text`	A character vector.
`detectors`	Detector ids to run; `NULL` runs all registered detectors (see `cg_detectors()`).
`context`	Optional extra context per element of `text` (recycled), treated as if it appeared next to every match. `scan_pii()` (planned) passes column names here, so a column called `voter_id` supports Voter ID matches.
`min_confidence`	Drop matches scoring below this value.

Value

An object of class `cg_preview`: a list with `text` (the input), `redacted` (as from `redact()`) and `spans` (as from `detect_pii()`). Printing shows the redacted text and a masked table of the spans.

Examples

```
preview("Name: Priya Nair; PAN: ABCPN1234K; order no 878245671885")
```

redact *Redact PII from text*

Description

Replaces every detected identifier with a placeholder naming its type, for example `[PAN]` or `[PERSON_NAME]`. Everything else in the text is kept exactly. Redaction is irreversible; the rules are shared with the Python package (see the spec's "Redaction" section).

Usage

```
redact(text, detectors = NULL, context = NULL, min_confidence = 0)
```

Arguments

text	A character vector.
detectors	Detector ids to run; NULL runs all registered detectors (see cg_detectors()).
context	Optional extra context per element of text (recycled), treated as if it appeared next to every match. scan_pii() (planned) passes column names here, so a column called voter_id supports Voter ID matches.
min_confidence	Drop matches scoring below this value.

Value

A character vector the same length as text, with PII replaced. NA stays NA.

See Also

preview() to see what would be redacted and why.

Examples

```
redact("Name: Priya Nair; PAN: ABCPN1234K")
redact(c("UPI priya.n@oksbi", "nothing here"))
```

---

redact_parquet *Redact a Parquet file, batch by batch*

Description

Writes a copy of a Parquet file with identifiers in the chosen text columns replaced by placeholders such as [PAN]. The file is read in record batches, so it may be larger than memory; other columns and the schema are kept.

Usage

```
redact_parquet(source, destination, columns = NULL, batch_size = 10000L, ...)
```

Arguments

source, destination	Parquet files to read and write; they must differ.
columns	Columns to redact; NULL redacts every string column.
batch_size	Rows per batch.
...	Passed to redact(): detectors, min_confidence.

Value

The number of rows written, invisibly.

See Also

redact_query() for database tables, including DuckDB.

Examples

```
src <- tempfile(fileext = ".parquet")
dst <- tempfile(fileext = ".parquet")
arrow::write_parquet(data.frame(id = 1:2, notes = c("PAN ABCPE1234F", "no PII")), src)
redact_parquet(src, dst)
arrow::read_parquet(dst)
```

---

redact_query *Redact the result of a database query into a table*

Description

Runs query on a DBI connection, fetches the result chunk_size rows at a time, redacts the chosen text columns and appends each chunk to the table to. The result may be larger than memory. Works with any DBI database; with DuckDB, query can read files directly ("SELECT * FROM 'calls.parquet'"), and COPY to TO 'out.parquet' writes the redacted table back out.

Usage

```
redact_query(
  con,
  query,
  to,
  columns = NULL,
  chunk_size = 10000L,
  append = FALSE,
  to_con = con,
  ...
)
```

Arguments

con	A DBI connection.
query	SQL query whose result is redacted.
to	Name of the table to create (or, with append = TRUE, add to).
columns	Columns to redact; NULL redacts every text column.
chunk_size	Rows fetched at a time.
append	Append to an existing table to instead of creating it.

to_con	Connection to write to with; by default con. Most databases (Postgres, SQLite) cannot write on a connection while a query result is open on it, so with one connection the redacted chunks are kept in temporary files until the query has been read, then written in one transaction. With DuckDB, or with a second connection here, each chunk is written as it is read. (For SQLite, a second connection to the same file needs WAL mode: `PRAGMA journal_mode = WAL`.)
...	Passed to `redact()`: `detectors`, `min_confidence`.

Value

The number of rows written, invisibly.

Examples

```
con <- DBI::dbConnect(duckdb::duckdb())
DBI::dbWriteTable(con, "calls", data.frame(id = 1:2, notes = c("PAN ABCPE1234F", "no PII")))
redact_query(con, "SELECT * FROM calls", "calls_redacted")
DBI::dbReadTable(con, "calls_redacted")
DBI::dbDisconnect(con, shutdown = TRUE)
```

---

register_detector *Register a detector*

Description

Register a detector

Usage

```
register_detector(detector, overwrite = FALSE)
```

Arguments

detector	A detector created with `new_detector()`.
overwrite	Replace an existing detector with the same id.

Value

The detector id, invisibly.

Examples

```
register_detector(new_detector(
  id = "employee_id", class = "internal", label = "Employee ID",
  pattern = bounded("EMP[0-9]{6}")
))
detect_pii("Approved by EMP123456")$type
```

restore *Put the values behind tokens back into text*

Description

Replaces tokens made by `tokenise()` - for example in a model's answer - with the values they stand for. Tokens are recognised with any letter case, with spaces inside the brackets (`[ pan_1 ]`) and without brackets (`PAN_1`), because models do not always copy them exactly. Tokens the vault does not know are left unchanged.

Usage

```
restore(text, vault, strict = FALSE)
```

Arguments

`text`	A character vector.
`vault`	The vault used by `tokenise()`.
`strict`	If `TRUE`, it is an error when `text` contains a bracketed token the vault does not know (one the model invented or altered).

Value

A character vector like `text`.

Examples

```
vault <- cg_vault()
tokenise("Name: Priya Nair, PAN ABCPE1234F", vault)
restore("[PERSON_NAME_1] holds [pan_1].", vault)
```

scan_pii *Scan the columns of a data frame for PII*

Description

Reports, per column, which PII types occur and in what share of values - for example that `cust_id` looks like a PAN in 98% of rows. Useful on its own, before any data is shared, to find which columns need protection.

Usage

```
scan_pii(data, sample = 10000L, min_rate = 0, detectors = NULL, seed = 42L)
```

Arguments

data	A data frame, an Arrow Table or Dataset (for example from arrow::open_dataset()), or a database table from dbplyr (dplyr::tbl(con, "calls")), for DuckDB, Postgres and other DBI databases). Arrow and database tables are read one column at a time, in chunks, so they may be larger than memory; the sample is the same as for the table in memory. For a database, rows must come back in the same order on each read (DuckDB and SQLite do for a plain table). A Spark table from sparklyr is sampled inside the cluster (up to sample rows, randomly with seed) and only the sample is collected.
sample	Maximum number of values checked per column.
min_rate	Report a type only if it occurs in at least this share of sampled values (0 reports every type found).
detectors	Detector ids to run; NULL runs all registered ones.
seed	Random seed for sampling Spark tables; ignored otherwise.

Details

Each column is sampled (evenly spaced, so results are reproducible), and every sampled value is checked with detect_pii() using the column name as context: a column named voter_id supports Voter ID matches that would otherwise need a keyword in the text. Whole numbers are checked too (mobile and account numbers are often stored as numbers); other numbers, logicals and dates are skipped. The rules are shared with the Python package (see the spec's "Column scan" section).

Value

A data frame of class cg_scan with one row per column and type: column, type, class, n_sampled, n_matched (values with a match), n_full (values that are entirely the identifier), rate (n_matched / n_sampled), mean_confidence and a masked example.

Examples

```
customers <- data.frame(
  cust_pan = c("ABCPE1234F", "AAACR5055K", "XYZPQ9876R"),
  notes = c("Called, Aadhaar 542229371182", "no issues", "mobile 9876543210")
)
scan_pii(customers)
```

---

synth_indian_pii *Generate synthetic Indian identifiers*

Description

Produces random identifiers that pass the same structural and checksum validation as real ones, for tests, demos and detector benchmarks. They are random, not taken from any real record. A random number with a valid checksum can still coincide with a real one by chance, so treat generated data as test data and never as anonymised data.

Usage

```
synth_indian_pii(n = 10, seed = NULL)
```

Arguments

n	Number of rows.
seed	Optional random seed; the global RNG state is restored afterwards. Names are random combinations of common Indian first names and surnames; email addresses use the reserved example.com/example.org domains.

Value

A data frame with columns name, name_hi (Devanagari), aadhaar, aadhaar_vid, pan, gstin, card, voter_epic, passport, driving_licence, mobile, email, upi_id, ifsc, bank_account, abha_number, abha_address, pincode, address, ckyc, ration_card, nrega_job_card, card_cvv and card_expiry.

Examples

```
synth_indian_pii(3, seed = 1)
```

---

tokenise *Replace identifiers with tokens that can be restored*

Description

Replaces every detected identifier with a numbered token such as [PAN_1] or [PERSON_NAME_2], and records in vault which value each token stands for, so restore() can put the values back into a model's answer. The same identifier gets the same token however it is written (ABCPE1234F and abcpe1234f; +91 98765 43210 and 9876543210), across texts and across calls with the same vault.

Usage

```
tokenise(text, vault, detectors = NULL, context = NULL, min_confidence = 0)
```

Arguments

text	A character vector.
vault	A vault from cg_vault().
detectors	Detector ids to run; NULL runs all registered detectors (see cg_detectors()).
context	Optional extra context per element of text (recycled), treated as if it appeared next to every match. scan_pii() (planned) passes column names here, so a column called voter_id supports Voter ID matches.
min_confidence	Drop matches scoring below this value.

Value

A character vector like `text`, with identifiers replaced by tokens. `NA` stays `NA`.

See Also

`restore()`, `redact()` for irreversible placeholders.

Examples

```
vault <- cg_vault()
safe <- tokenise(c("PAN ABCPE1234F, mobile 9876543210", "pan abcpe1234f again"), vault)
safe
restore("The customer with [PAN_1] can be reached on [MOBILE_IN_1].", vault)
```

---

vault_merge *Combine keyed vaults*

Description

Adds the entries of `other` that `vault` does not have yet, for example from parallel workers that each tokenised part of a table with the same key. Both vaults must be keyed with the same key; it is an error if a token stands for different values in the two.

Usage

```
vault_merge(vault, other)
```

Arguments

`vault`, `other`	Keyed vaults from `cg_vault()`.

Value

`vault`, modified in place, invisibly.

vault_save *Save a vault to an encrypted file, and open it again*

Description

`vault_save()` writes the vault to a file encrypted with a passphrase, so tokens can be restored in a later session - in R or in Python, which read the same file. `vault_open()` reads it back; tokenising with the opened vault continues its numbering.

Usage

```
vault_save(vault, path, passphrase = NULL)

vault_open(path, passphrase = NULL, key = NULL)
```

Arguments

`vault`	A vault from `cg_vault()` or `vault_open()`.
`path`	File to write or read, conventionally ending in `.cgvault`.
`passphrase`	The passphrase, or `NULL` to use `CREDGATE_VAULT_PASSPHRASE` or a prompt.
`key`	For a keyed vault: its token key, or `NULL` to use `CREDGATE_TOKEN_KEY`. The key is never saved; opening fails if it differs from the one the vault was saved with.

Details

The passphrase comes from `passphrase`, else the environment variable `CREDGATE_VAULT_PASSPHRASE`, else a prompt (in an interactive session with the askpass package). It needs at least 12 characters and is never stored: without it the file cannot be opened. The file is written readable only by you; an existing file is replaced only if it is itself a vault file. Encryption uses libsodium (scrypt key derivation, XSalsa20-Poly1305) through the sodium package.

Value

`vault_save()` returns `path` invisibly; `vault_open()` returns a vault.

Examples

```
vault <- cg_vault()
tokenise("PAN ABCPE1234F", vault)
path <- tempfile(fileext = ".cgvault")
vault_save(vault, path, passphrase = "an example passphrase")

# later, possibly in another session or in Python:
vault <- vault_open(path, passphrase = "an example passphrase")
restore("Customer [PAN_1] called.", vault)
```

verhoeff_valid *Verhoeff checksum (used by Aadhaar and Aadhaar Virtual ID)*

Description

`verhoeff_valid()` checks whether a number (including its final check digit) passes the Verhoeff checksum. `verhoeff_check_digit()` computes the check digit to append to a number.

Usage

```
verhoeff_valid(x)

verhoeff_check_digit(x)
```

Arguments

`x`	A character vector of digit strings. Spaces and hyphens are ignored.

Value

`verhoeff_valid()`: a logical vector (`FALSE` for anything that is not all digits). `verhoeff_check_digit()`: a character vector of single digits (`NA` for invalid input).

Examples

```
verhoeff_check_digit("236")
verhoeff_valid("2363")
```

write_inventory *Write and read inventory files*

Description

`write_inventory()` saves an inventory from `pii_inventory()` as a CSV file, written exactly as the spec describes, so that R and Python write identical files for the same inventory. `read_inventory()` reads a file written by either and checks its format.

Usage

```
write_inventory(x, path)

read_inventory(path)
```

Arguments

`x`	An inventory from `pii_inventory()` (or several combined with `rbind()`).
`path`	File path.

Value

`write_inventory()` returns `path` invisibly; `read_inventory()` an inventory.

Index

audit_read *(cg_audit_log)*, 5
audit_record *(cg_audit_log)*, 5
audit_verify *(cg_audit_log)*, 5

bounded, 3
bounded(), *14*
bq_inventory_dataset *(bq_scan_pii)*, 3
bq_pii_inventory *(bq_scan_pii)*, 3
bq_scan_pii, 3

cg_audit_log, 5
cg_audit_log(), *17*
cg_detectors, 6
cg_detectors(), *12*, *18*, *19*, *24*
cg_evaluate, 7
cg_evaluate(), *10*
cg_key_fingerprint *(cg_new_token_key)*, 8
cg_new_token_key, 8
cg_new_token_key(), *11*
cg_normalise, 8
cg_policy, 9
cg_policy(), *17*
cg_read_corpus, 10
cg_spec_path, 10
cg_vault, 11
cg_vault(), *8*, *17*, *24–26*

detect_pii, 12
detect_pii(), *7*, *18*, *23*

gstin_check_char *(gstin_valid)*, 13
gstin_valid, 13

luhn_valid, 13

new_detector, 14
new_detector(), *21*

pii_inventory, 16
pii_inventory(), *4*, *27*
policy_apply *(policy_decide)*, 17
policy_apply(), *5*, *9*
policy_decide, 17
policy_decide(), *5*, *9*
preview, 18
preview(), *19*

read_inventory *(write_inventory)*, 27
redact, 18
redact(), *18*, *19*, *21*, *25*
redact_parquet, 19
redact_query, 20
redact_query(), *20*
register_detector, 21
restore, 22
restore(), *24*, *25*

scan_pii, 22
scan_pii(), *4*, *16*
synth_indian_pii, 23

tokenise, 24
tokenise(), *11*, *22*

vault_merge, 25
vault_merge(), *11*
vault_open *(vault_save)*, 26
vault_save, 26
verhoeff_check_digit *(verhoeff_valid)*, 27
verhoeff_valid, 27

write_inventory, 27
write_inventory(), *16*