Package: faissR (via r-universe)

July 10, 2026

Type Package

Title FAISS-Backed Nearest Neighbours, Graph Clustering, kNN Models, and k-Means

Version 0.99.15

Description Native nearest-neighbour search, graph construction, graph clustering, k-nearest neighbour prediction, and k-means helpers for large dimensionality reduction workflows in high-throughput biological data analysis, including single-cell, flow cytometry, imaging, and mass spectrometry applications. The package requires the FAISS C++ library for all builds, including CPU-only indexes, and can optionally use FAISS GPU indexes with NVIDIA cuVS integration or direct RAPIDS cuVS/CUDA indexes on NVIDIA machines. CPU-only systems do not need NVIDIA libraries, but NVIDIA CUDA/cuVS libraries are mandatory when a GPU-enabled build is explicitly requested. Supervised kNN models use knn(Xtrain, Ytrain) and predict(), or knn(Xtrain, Ytrain, Xtest) for immediate prediction. Class probabilities are returned through predict(type = ` ` prob") for kNN classification models.

License MIT + file LICENSE

URL https://github.com/tkcaccia/faissR

BugReports https://github.com/tkcaccia/faissR/issues

OS_type unix

biocViews Software, Infrastructure, GPU, SingleCell, FlowCytometry, MassSpectrometry, ImagingMassSpectrometry, Proteomics, Clustering, DimensionReduction, Classification

Encoding UTF-8

Imports methods, Rcpp

Suggests BiocStyle, knitr, Matrix, float, rmarkdown, testthat, withr

LinkingTo Rcpp

SystemRequirements C++20, Fortran, FAISS C++ development library (Debian/Ubuntu: libfaiss-dev), OpenMP runtime on macOS when FAISS headers include omp.h (for example Homebrew libomp). Optional GPU libraries are documented in the installation guide and are used only when a GPU-enabled build is explicitly requested.

Config/Bioconductor/UnsupportedPlatforms mac

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Config/pak/sysreqs libfaiss-dev

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

Date/Publication 2026-07-10 13:25:02 UTC

RemoteUrl https://github.com/BiocStaging/faissR

RemoteRef HEAD

RemoteSha f37ea97c5774200025b1480770b8ecbf1d2d7919

Contents

backend_info *Summarize native neighbour-search backend availability*

Description

'backend_info()' reports which 'faissR' nearest-neighbour backends can currently run. It never silently falls back from an explicit GPU request to CPU; this table is informational only.

Usage

```
backend_info()
```

Value

A data frame with one row per compiled/runtime backend family and columns describing availability, public call hints, public backend names, supported public method/metric summaries, non-public implementation route labels, device/runtime hints, and a short note. Use `nn_capabilities()` for the full method/backend/metric matrix.

Examples

```
info <- backend_info()
info[, c("backend", "available", "public_backends")]
```

candidate_knn *Select nearest neighbours from a candidate matrix*

Description

'candidate_knn()' computes exact top-k neighbours restricted to a supplied per-query candidate matrix. It is useful after an approximate candidate generation step, for NN-descent refinement, graph refinement, or landmark projection. The function does not generate candidates; it only scores and ranks the candidates that you pass in.

Usage

```
candidate_knn(
  data,
  candidates,
  points = data,
  k,
  backend = c("auto", "cpu", "cuda"),
  metric = c("euclidean", "cosine", "correlation", "inner_product"),
  n_threads = NULL,
  exclude_self = FALSE
)
```

Arguments

data	Numeric reference matrix with observations in rows.
candidates	Integer matrix of 1-based candidate reference row indices. It must have one row per query. Invalid, missing, zero, or out-of-range entries are ignored.
points	Numeric query matrix with observations in rows. Defaults to ‘data‘, i.e. self-query candidate KNN.
k	Number of neighbours to return from each candidate row.
backend	‘"auto"‘/‘"cpu"‘ for the general CPU implementation, ‘"cuda"‘ for the native CUDA row-candidate kernel. GPU backends currently require self-query candidates with ‘exclude_self = TRUE‘.
metric	‘"euclidean"‘, ‘"cosine"‘, ‘"correlation"‘, or ‘"inner_product"‘. Legacy metric aliases such as ‘"l2"‘, ‘"cor"‘, ‘"pearson"‘, and ‘"ip"‘ are rejected. Correlation is centered cosine similarity, not raw inner product. CPU inner-product scoring ranks by larger raw dot product, while returned ‘distances‘ are shifted within each query so the best returned dot product has distance ‘0‘. CUDA candidate scoring supports Euclidean directly, cosine/correlation through normalized Euclidean scoring, and raw inner-product scoring through a dedicated CUDA kernel mode that preserves the same shifted-distance convention as CPU.
n_threads	CPU threads for the CPU backend.
exclude_self	If ‘TRUE‘, remove each query row from its own candidate set. This is valid only for self-query candidate KNN.

Value

A ‘faissR_nn‘ object with ‘indices‘ and ‘distances‘. If a row has fewer than ‘k‘ valid unique candidates, remaining entries are ‘NA‘ and ‘Inf‘.

Examples

```
x <- scale(as.matrix(iris[, 1:4]))
rough <- nn(x, k = 10, backend = "cpu")
refined <- candidate_knn(x, rough$indices, k = 5, exclude_self = TRUE)
refined
```

---

cuda_available *Check whether the native CUDA backend is available*

Description

Check whether the native CUDA backend is available

Usage

```
cuda_available()
```

Value

'TRUE' when the package was built with CUDA support and the CUDA runtime reports at least one available device.

Examples

```
cuda_available()
```

---

cugraph_available *Check whether the RAPIDS libcugraph backend is available*

Description

Check whether the RAPIDS libcugraph backend is available for native CUDA graph clustering.

Usage

```
cugraph_available()
```

Value

TRUE when faissR was compiled and linked against RAPIDS libcugraph.

Examples

```
cugraph_available()
```

---

cuvs_available *Check whether the RAPIDS cuVS backend is available*

Description

Check whether the RAPIDS cuVS backend is available

Usage

```
cuvs_available()
```

Value

'TRUE' when faissR was compiled and linked against RAPIDS cuVS and the CUDA runtime reports at least one available device.

Examples

```
cuvs_available()
```

faiss_available *Check whether the real FAISS C++ backend is available*

Description

Check whether the real FAISS C++ backend is available

Usage

```
faiss_available()
```

Value

'TRUE' when faissR was compiled and linked against FAISS.

Examples

```
faiss_available()
```

faiss_gpu_available *Check whether FAISS GPU support is available*

Description

Check whether FAISS GPU support is available.

Usage

```
faiss_gpu_available()
```

Value

TRUE when faissR was compiled and linked against a FAISS build that reports GPU support.

Examples

```
faiss_gpu_available()
```

`fast_kmeans` *Fast k-means clustering*

Description

Run k-means using the fastest available backend. CPU acceleration uses FAISS when faissR was built with FAISS. CUDA acceleration first tries FAISS GPU, which can use NVIDIA/cuVS-enabled FAISS builds, then direct RAPIDS cuVS when available. Explicit GPU requests fail clearly instead of silently changing to CPU.

Usage

```
fast_kmeans(
  data,
  centers,
  backend = c("auto", "cpu", "cuda"),
  max_iter = "auto",
  n_init = "auto",
  tol = "auto",
  seed = 1L,
  n_threads = NULL,
  streaming_batch_size = 0L,
  init = c("kmeans++", "random"),
  tuning = c("auto", "fixed", "off", "none")
)
```

Arguments

`data`	Numeric matrix with observations in rows.
`centers`	Number of clusters.
`backend`	Device backend: ‘"auto"‘, ‘"cpu"‘, or ‘"cuda"‘. ‘"auto"‘ uses CUDA only when CUDA plus FAISS GPU k-means or direct cuVS k-means is compiled and available and the deterministic shape rule estimates enough work to offset GPU launch and host/device copy overhead; otherwise it resolves to CPU.
`max_iter`	Maximum number of Lloyd iterations, or ‘"auto"‘ for a deterministic shape-aware default computed by the compiled C++ tuning helper.
`n_init`	Number of random restarts where supported, or ‘"auto"‘ for a deterministic shape-aware default computed by the compiled C++ tuning helper.
`tol`	Single non-negative finite relative convergence tolerance where supported, or ‘"auto"‘ for a deterministic shape-aware default computed by the compiled C++ tuning helper.
`seed`	Random seed for CPU/statistics and FAISS paths. The current direct cuVS C API path does not expose an explicit seed in the stable params structure.
`n_threads`	Number of CPU threads for FAISS/statistics paths.

`streaming_batch_size`
: cuVS host-data streaming batch size. ‘0‘ lets cuVS choose its default.

`init`
: Initialization method, ‘"kmeans++"‘ or ‘"random"‘ where supported.

`tuning`
: Tuning policy. ‘"auto"‘ uses deterministic C++ rules based on ‘nrow(data)‘, ‘ncol(data)‘, and ‘centers‘ without running pilot searches. Small many-cluster jobs can use extra restarts when ‘n / centers‘ remains large enough, and cheap many-cluster jobs with few observations per center also use a small multistart budget for stability; large or high-dimensional jobs use cheaper iteration and tolerance defaults. ‘centers = 1‘ uses the exact column-mean solution for every backend request with ‘max_iter = 1‘, ‘n_init = 1‘, and ‘tol = 0‘, records ‘single_cluster_exact_mean‘, and stays on CPU because no iterative k-means backend can improve that solution. ‘centers = nrow(data)‘ uses the exact singleton assignment for every backend request and records ‘singleton_exact_identity‘; explicit CUDA requests are recorded as resolved by the exact faissR trivial route rather than launching GPU work. ‘"fixed"‘, ‘"off"‘, and ‘"none"‘ use the historical fixed defaults unless ‘max_iter‘, ‘n_init‘, or ‘tol‘ are explicitly supplied.

Value

A list with ‘cluster‘, ‘centers‘, ‘withinss‘, ‘tot.withinss‘, ‘size‘, ‘iter‘, ‘converged‘, ‘hit_max_iter‘, ‘backend‘, and ‘parameters‘. ‘backend‘ records the implementation that actually ran, while ‘parameters$requested_backend‘ and ‘parameters$resolved_backend‘ record the public backend request and device policy result. ‘parameters$tuning‘ records the deterministic k-means policy, stable ‘rule‘ label, shape metadata, and whether ‘max_iter‘, ‘n_init‘, and ‘tol‘ were auto-selected or supplied explicitly. The auto parameter, backend-policy, and final auto backend-selection rules are computed by compiled C++ helpers and record ‘tuning_source = "cpp"‘. ‘parameters$tuning$rule_detail‘ records the exact ‘n‘/‘p‘/‘centers‘/work values used to choose the rule. ‘parameters$tuning$effective‘ records the final values used after explicit overrides and ‘"auto"‘ defaults have been resolved; ‘parameters$tuning$effective_max_iter‘, ‘parameters$tuning$effective_n_init‘, and ‘parameters$tuning$effective_tol‘ expose the same values as flat fields for benchmark summaries. ‘parameters$tuning$backend_policy‘ records the deterministic shape rule used by ‘backend = "auto"‘ to decide whether CUDA has enough estimated work or float32 transfer size to offset transfer overhead. The policy keeps ‘nbytes‘ as the ordinary R double input footprint and records ‘gpu_transfer_nbytes‘ for the float32 data passed to FAISS/cuVS. The default thresholds can be overridden without adding pilot work by setting ‘options(faissR.kmeans_cuda_work_threshold = ...)‘, ‘options(faissR.kmeans_cuda_nbytes_threshold = ...)‘, ‘options(faissR.kmeans_cuda_large_n_threshold = ...)‘, or ‘options(faissR.kmeans_cuda_large_p_threshold = ...)‘, or ‘options(faissR.kmeans_cuda_min_n_per_center = ...)‘. ‘parameters$tuning$selection‘ stores the static no-pilot backend and effective-parameter decision used for benchmark auditing, including ‘explicit_backend‘ and ‘backend_decision‘ fields that distinguish an explicit ‘"cpu"‘/‘"cuda"‘ request from an automatic shape-policy choice, plus ‘runtime_decision‘ and CUDA k-means capability flags from the C++ selector. CUDA runs also record ‘parameters$cuda_provider_selection‘ as ‘"faiss_gpu"‘, ‘"direct_cuvs"‘, or ‘"direct_cuvs_after_faiss_gpu_unavailable_or_failed"‘; ‘parameters$backend_resolution_note‘ describes the provider route, and ‘parameters$faiss_gpu_error‘ is present when direct cuVS was used after a FAISS GPU route was unavailable or failed. ‘hit_max_iter‘ records whether the run reached the effective iteration cap, and ‘converged‘ is the corresponding conservative convergence flag used by benchmark summaries.

Examples

```
x <- scale(as.matrix(iris[, 1:4]))
fit <- fast_kmeans(x, centers = 3, backend = "cpu", n_threads = 2)
table(fit$cluster)
```

`gpu_knn_to_host` *Copy a GPU-resident KNN result to host matrices*

Description

'gpu_knn_to_host()' is an explicit diagnostic/conversion helper for 'faissR_gpu_knn' objects. It copies device 'indices' and 'distances' into ordinary R matrices. It is never called automatically by 'nn_gpu()'.

Usage

```
gpu_knn_to_host(x)
```

Arguments

x	A 'faissR_gpu_knn' object.

Value

A host 'faissR_nn' list with integer 'indices' and numeric 'distances'.

Examples

```
if (cuda_available()) {
  x <- matrix(rnorm(200), ncol = 4)
  gpu_knn <- nn_gpu(x, k = 5, exclude_self = TRUE)
  host_knn <- gpu_knn_to_host(gpu_knn)
  str(host_knn)
}
```

`graph_cluster` *Cluster a nearest-neighbour graph without igraph*

Description

`graph_cluster()` runs native faissR community detection on a KNN graph. The graph can be supplied as a precomputed `nn()` result or built from a matrix/data frame using any faissR nearest-neighbour backend, including FAISS and cuVS KNN backends. The CPU clustering backend is implemented in C++ and uses OpenMP when available; it does not depend on igraph. CUDA graph clustering uses native RAPIDS libcugraph for Louvain and Leiden when faissR was built against libcugraph. Random-walking currently remains CPU-only. CUDA graph clustering never calls Python and never silently falls back to CPU.

Usage

```
graph_cluster(
  graph,
  method = c("random_walking", "louvain", "leiden"),
  backend = c("auto", "cpu", "cuda"),
  k = 50L,
  graph_backend = "auto",
  graph_method = c("auto", "exact", "flat", "bruteforce", "grid", "hnsw", "ivf",
    "ivfpq", "vamana", "nsg", "nndescent", "ivfpq_fastscan", "cagra"),
  metric = c("euclidean", "cosine", "correlation", "inner_product"),
  tuning = c("auto", "cache", "pilot", "fixed", "off", "none"),
  target_recall = 0.99,
  cagra_implementation = NULL,
  cagra_build_algo = NULL,
  weight = c("auto", "snn", "adaptive", "distance", "binary"),
  mutual = FALSE,
  prune = 0,
  n_threads = NULL,
  n_runs = 1L,
  resolution = 1,
  n_clusters = NULL,
  objective_function = c("modularity"),
  n_iterations = 10L,
  steps = 4L,
  seed = NULL,
  ...
)
```

Arguments

graph	A numeric matrix/data frame, a KNN object returned by `nn()`, or an embedding object with a matrix `layout`.
method	One of `"random_walking"`, `"louvain"`, or `"leiden"`. `"random_walking"` uses a native random-walk label-propagation pass inspired by walktrap/random-walk clustering. `"louvain"` uses native modularity local moving. `"leiden"` adds a native refinement pass that splits disconnected communities after local moving. See References.
backend	Community-detection backend. `"auto"` uses CUDA when libcugraph is available for Louvain/Leiden and CPU otherwise; auto keeps `"random_walking"` on CPU. `"cpu"` uses native faissR C++/OpenMP code. `"cuda"` uses native RAPIDS libcugraph for Louvain and Leiden when libcugraph was detected at build time; random-walking is CPU-only. The `"auto"` backend decision is made by the compiled `graph_cluster_auto_backend_cpp()` selector and recorded in `parameters$backend_selection`.
k	Number of neighbours when `graph` is not already a KNN object.
graph_backend	Backend passed to `nn(..., exclude_self = TRUE)` for neighbour search.

graph_method
: Nearest-neighbour method passed to `nn(..., exclude_self = TRUE)` when `graph` is a matrix or embedding.

metric
: Distance metric passed to `nn(..., exclude_self = TRUE)` when `graph` is a matrix or embedding: `"euclidean"`, `"cosine"`, `"correlation"`, or `"inner_product"`. Legacy metric aliases such as `"l2"`, `"cor"`, `"pearson"`, and `"ip"` are rejected. Correlation is centered cosine similarity, not raw inner product. Inner-product graph construction ranks neighbours by larger raw dot product while reusing faissR's shifted smaller-is-better distance convention from `nn()`.

tuning
: Tuning policy passed to `nn(..., exclude_self = TRUE)` when `graph` is a matrix or embedding.

target_recall
: Speed/recall tier passed to `nn(..., exclude_self = TRUE)` when `graph` is a matrix or embedding. Use `0.9`, `0.95`, or `0.99`; CUDA `method = "auto"` uses it for Flat-vs-IVF selection, CUDA IVF uses it for probing defaults, and HNSW uses it for graph-search tiers.

cagra_implementation
: CUDA CAGRA provider passed to `nn(..., exclude_self = TRUE)` when graph KNN is computed here. `NULL` uses the global option; `"auto"`, `"faiss_gpu"`, or `"cuvs"` select the CAGRA provider for CUDA `graph_method = "cagra"` and CUDA-auto CAGRA routes.

cagra_build_algo
: Direct RAPIDS cuVS CAGRA graph-build algorithm passed to `nn(..., exclude_self = TRUE)` when graph KNN is computed here. `NULL` uses the global `faissR.cuvs_cagra_build_algo` option. This is a CAGRA construction parameter, not a fallback to a different public NN method.

weight
: KNN graph weighting used when a graph must be built.

mutual
: If `TRUE`, keep only reciprocal nearest-neighbour edges when a graph must be built from data or a KNN object.

prune
: Drop graph edges with weight less than or equal to this value when a graph must be built from data or a KNN object.

n_threads
: CPU threads. Used by KNN construction and native CPU clustering.

n_runs
: Number of independent native clustering runs. The result with the largest modularity is returned. Values greater than one can use several CPU cores.

resolution
: Modularity resolution for Louvain/Leiden-style scoring. Use a positive number for direct resolution control. When `n_clusters` is supplied and `resolution` is omitted or `NULL`, faissR seeds the bounded deterministic grid from the no-pilot graph-shape heuristic instead of the numeric default.

n_clusters
: Optional target number of communities for Louvain/Leiden. If supplied, faissR evaluates a bounded deterministic resolution grid on the already-built graph and keeps the result whose community count is closest to the target. The grid is centered from an explicitly requested `resolution`, or from an automatic seed when `resolution` is omitted or `NULL`; when graph size is known the automatic seed uses a no-pilot shape heuristic based on `n_clusters / sqrt(n_vertices)`. The search width is shape-aware: small graphs use a wider deterministic grid, while large graphs use fewer candidates to reduce repeated Louvain/Leiden passes. This is a convenience target, not a hard guarantee. If `n_clusters` is supplied

and method is omitted, faissR uses "louvain" as the target-count clustering method. The target must be a positive integer and cannot exceed the number of graph vertices.

objective_function
: Community objective. Only "modularity" is currently implemented by the native CPU and CUDA clustering paths; CPM is not accepted until the implementation can optimize and report it honestly.

n_iterations
: Native clustering iteration count.

steps
: Random-walk propagation depth.

seed
: Optional seed for reproducible repeated CPU runs.

...
: Reserved for future backend options.

Details

n_clusters is supplied directly to graph_cluster(). When a target count is available for Louvain or Leiden, faissR builds or reuses the graph once, evaluates a bounded deterministic resolution grid. Explicit numeric resolution values center the grid near that value and graph shape; omitted resolution or resolution = NULL uses the no-pilot target-count graph-shape seed directly. The grid width is shape-aware so large graph target searches use fewer deterministic candidates than small graph searches. faissR records the selected resolution, center, and search table in the returned object. If n_clusters is supplied and method is omitted, faissR uses "louvain" as the target-count clustering method. Passing n_clusters to explicit method = "random_walking" still errors. The target must be a positive integer and cannot exceed the number of graph vertices.

Method descriptions:

- "random_walking": native CPU random-walk label-propagation/community method inspired by walktrap-style random-walk clustering and local parallel random-walk literature. CUDA random-walking is not enabled yet.
- "louvain": native CPU Louvain modularity local-moving implementation, with optional RAPIDS libcugraph CUDA execution when faissR is built with cuGraph.
- "leiden": native CPU Leiden-style local moving plus refinement to split disconnected communities, with optional RAPIDS libcugraph CUDA execution when available.

Value

A faissR_graph_cluster list with membership, modularity, method, backend, graph edge list, parameters, and source acknowledgements. backend records the clustering implementation that actually ran, while parameters$requested_backend and parameters$resolved_backend record the public backend request and the device policy after resolving "auto". parameters$backend_selection records the compiled backend selector metadata, including runtime_decision, CUDA/libcugraph availability flags, and tuning_source = "cpp". When graph_cluster() builds a graph internally or receives a faissR_graph, parameters$graph_backend, parameters$graph_requested_backend, and parameters$graph_resolved_backend record the concrete KNN implementation, public graph backend request, and resolved KNN backend. parameters$n_vertices and parameters$n_edges record the clustered graph size for benchmark summaries. parameters$nn_metric_transform and parameters$nn_distance_transform preserve normalized metric conversion metadata from the KNN route that built the graph. parameters$nn_approximation, parameters$nn_faiss,

parameters$nn_cuvs, parameters$nn_spatial_index, and parameters$nn_auto_selection preserve compact KNN route metadata when the graph is built internally, including parameters$nn_target_recall for HNSW-backed graph builds. When a target community count is used, target_n_clusters, selected_resolution, target_gap, resolution_selection, and resolution_search record the requested target, selected resolution, final community-count gap, deterministic selection rule, and full resolution search table. The validated request is also stored as parameters$n_clusters_requested for benchmark summaries and as parameters$n_clusters for backward compatibility. The deterministic candidate center and bounded grid are computed by the compiled graph_resolution_candidates_cpp() helper and record resolution_selection$tuning_source = "cpp". parameters$resolution is NA for target-auto searches where n_clusters is supplied and resolution is omitted or NULL; the actual automatic center is stored in resolution_selection$candidate_center. parameters$resolution_source is "default", "user", or "target_auto" depending on how the resolution-grid seed was chosen.

References

Blondel VD, Guillaume JL, Lambiotte R, Lefebvre E. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment. 2008;2008(10):P10008.

Pons P, Latapy M. Computing communities in large networks using random walks. Journal of Graph Algorithms and Applications. 2006;10(2):191-218.

Traag VA, Waltman L, van Eck NJ. From Louvain to Leiden: guaranteeing well-connected communities. Scientific Reports. 2019;9:5233.

Sahu S. GVE-Leiden: Fast Leiden Algorithm for Community Detection in Shared Memory Setting. arXiv:2312.13936. Source repository: https://github.com/puzzlef/leiden-communities-openmp.

Sahu S. Heuristic-based Dynamic Leiden Algorithm for Efficient Tracking of Communities on Evolving Graphs. arXiv:2410.15451. Source repository: https://github.com/puzzlef/leiden-communities-openmp-h

Kapralov M, Lattanzi S, Nouri N, Tardos J. Efficient and Local Parallel Random Walks. arXiv:2112.00655.

CPU implementations are native faissR C++/OpenMP code. CUDA Louvain and Leiden use RAPIDS libcugraph when available at build time; cuGraph also provides GPU random-walk primitives that may support a future native random-walking adapter. See https://github.com/rapidsai/cugraph and https://docs.rapids.ai/api/cugraph/.

Examples

```
set.seed(1)
x <- rbind(matrix(rnorm(80, -2), ncol = 4), matrix(rnorm(80, 2), ncol = 4))
cl <- graph_cluster(x, method = "louvain", backend = "cpu", k = 8, graph_backend = "cpu")
table(cl$membership)
```

---

knn *Fit or apply a k-nearest-neighbour classifier or regressor*

Description

knn() is the high-level supervised kNN API. With Xtrain and Ytrain only, it returns a reusable model. With Xtest, it immediately predicts the query rows by fitting the model and calling predict() internally. The low-level nearest-neighbour search API remains nn(). For explicit CPU FAISS-backed method = "flat", "hnsw", "ivf", and "ivfpq" models, the fitted object stores a session-local NN index and predict() reuses it for compatible prediction calls. For IVF/IVFPQ, this reuses trained centroids, inverted lists, and indexed vectors; IVFPQ also reuses trained product-quantizer codebooks and compressed codes.

Usage

```
knn(
  Xtrain,
  Ytrain,
  Xtest = NULL,
  backend = c("auto", "cpu", "cuda"),
 method = c("auto", "exact", "flat", "bruteforce", "grid", "hnsw", "ivf", "ivfpq",
    "vamana", "nsg", "nndescent", "ivfpq_fastscan", "cagra"),
  metric = c("euclidean", "cosine", "correlation", "inner_product"),
  tuning = c("auto", "cache", "pilot", "fixed", "off", "none"),
  target_recall = 0.99,
  cagra_implementation = NULL,
  cagra_build_algo = NULL,
  task = c("auto", "classification", "regression"),
  k = 15L,
  n_threads = NULL,
  vote = c("majority", "weighted"),
  type = c("response", "prob"),
  ...
)
```

Arguments

Xtrain	Numeric training matrix or optional float::fl()/float32 matrix with observations in rows. Float32 inputs are preserved for nn() methods with direct float32 adapters.
Ytrain	Training labels or numeric response.
Xtest	Optional numeric or float32 query matrix. If supplied, knn() returns predictions for Xtest; otherwise it returns a fitted model.
backend	Device backend passed to nn(): "auto", "cpu", or "cuda". "auto" follows nn() backend/method/metric resolution, using CUDA only for validated CUDA combinations when CUDA/cuVS runtime support is available, and CPU otherwise.
method	Nearest-neighbour algorithm selector passed to nn(). "auto" chooses the most appropriate method for the selected backend. See ?nn for descriptions and references for "exact", "flat", "bruteforce", "grid", "hnsw", "ivf", "ivfpq", "ivfpq_fastscan", "vamana", "nsg", "nndescent", and "cagra".

`metric` Distance metric passed to `nn()`: `"euclidean"`, `"cosine"`, `"correlation"`, or `"inner_product"`. Legacy metric aliases such as `"l2"`, `"cor"`, `"pearson"`, and `"ip"` are rejected. Correlation is centered cosine similarity, not raw inner product. See `nn()` for metric/backend support details, including metric-aware CPU HNSW routing.

`tuning` Tuning policy passed to `nn()`. `"auto"` uses the deterministic default for the resolved method; `"cache"` and `"pilot"` opt into pilot tuning where implemented. FAISS GPU IVF pilot/cache tuning is Euclidean-only; non-Euclidean IVF routes use deterministic metric-aware defaults.

`target_recall` Speed/recall tier passed to `nn()`. Use `0.9`, `0.95`, or `0.99`. CUDA `method = "auto"` uses it for Flat-vs-IVF selection, CUDA IVF uses it for probing defaults, and HNSW uses it for graph-search tiers. CUDA HNSW metadata records that the available cuVS route is a CAGRA-to-HNSW wrapper.

`cagra_implementation` CUDA CAGRA provider passed to `nn()` for `method = "cagra"` or CUDA-auto routes that select CAGRA. `NULL` uses the global `faissR.cagra_implementation` option; `"auto"` uses the same deterministic shape-aware provider rule as `nn()`, while `"faiss_gpu"` or `"cuvs"` force one provider.

`cagra_build_algo` Direct RAPIDS cuVS CAGRA graph-build algorithm passed to `nn()` for direct cuVS CAGRA routes. `NULL` uses the global `faissR.cuvs_cagra_build_algo` option.

`task` `"auto"`, `"classification"`, or `"regression"`. `"auto"` treats numeric responses as regression and other response types as classification.

`k` Default number of neighbours used by `predict()` and by immediate predictions when `Xtest` is supplied.

`n_threads` CPU threads passed to `nn()` for CPU backends.

`vote` `"majority"` or `"weighted"` for immediate predictions.

`type` `"response"` for class/regression predictions or `"prob"` for class probability matrices from classification models.

`...` Reserved for future options.

Value

If `Xtest` is not supplied, a fitted `faissR_knn_model` object that stores the training data, response, task, backend, method, metric, tuning, `target_recall`, `k`, and CPU thread settings used by later `predict()` calls. Explicit CPU FAISS Flat/HNSW/IVF/IVFPQ models also store a session-local fitted index for compatible prediction calls; saved/reloaded models safely rebuild the same route when the external pointer is no longer valid. IVF metadata records whether trained centroids/inverted lists were reused and whether search used a query-specific `nprobe`; IVFPQ metadata also records product-quantizer codebook/code reuse. If `Xtest` is supplied, a factor for classification, a numeric vector for regression, or a numeric class-probability matrix when `type = "prob"`; prediction outputs carry `attr(result, "faissR_nn")` route metadata, approximation parameters, and auto-selection metadata from the underlying `nn()` call, including `batch_query = TRUE`, `query_n`, and `query_call_count = 1L` for the full-matrix prediction query.

Examples

```
x <- scale(as.matrix(iris[, 1:4]))
model <- knn(x, iris$Species, backend = "cpu", k = 5)
head(predict(model, x, k = 5))
head(predict(model, x, k = 5, type = "prob"))

pred <- knn(x, iris$Species, x, backend = "cpu", k = 5)
head(pred)
```

---

knn_graph *Build a native nearest-neighbour graph*

Description

knn_graph() turns a data matrix, a precomputed nn() result, or an embedding object with a matrix layout into a weighted nearest-neighbour graph. It returns a native faissR_graph edge-list object and does not require igraph.

Usage

```
knn_graph(
  data,
  knn = NULL,
  k = 50L,
  backend = c("auto", "cpu", "cuda"),
 nn_method = c("auto", "exact", "flat", "bruteforce", "grid", "hnsw", "ivf", "ivfpq",
    "vamana", "nsg", "nndescent", "ivfpq_fastscan", "cagra"),
  method = NULL,
  metric = c("euclidean", "cosine", "correlation", "inner_product"),
  tuning = c("auto", "cache", "pilot", "fixed", "off", "none"),
  target_recall = 0.99,
  cagra_implementation = NULL,
  cagra_build_algo = NULL,
  weight = c("auto", "snn", "adaptive", "distance", "binary"),
  mutual = FALSE,
  prune = 0,
  n_threads = NULL
)
```

Arguments

data	Numeric matrix/data frame, a KNN object returned by nn(), or an embedding object with a matrix layout.
knn	Optional precomputed KNN object returned by nn(). If supplied, data is ignored for neighbour search.
k	Number of non-self neighbours used in the graph.

backend — Device backend passed to nn(..., exclude_self = TRUE) when knn is not supplied: "auto", "cpu", or "cuda".

nn_method — Nearest-neighbour method passed to nn(..., exclude_self = TRUE) when knn is not supplied. The default "auto" uses the shape-aware selector for the chosen backend.

method — Alias for nn_method, matching the public method argument used by nn() and knn(). If both method and nn_method are supplied they must resolve to the same public nearest-neighbour method label.

metric — Distance metric passed to nn(..., exclude_self = TRUE) when knn is not supplied: "euclidean", "cosine", "correlation", or "inner_product". Legacy metric aliases such as "l2", "cor", "pearson", and "ip" are rejected. Correlation is centered cosine similarity, not raw inner product. Inner-product graph construction ranks neighbours by larger raw dot product while reusing faissR's shifted smaller-is-better distance convention from nn().

tuning — Tuning policy passed to nn(..., exclude_self = TRUE) when knn is not supplied.

target_recall — Speed/recall tier passed to nn(..., exclude_self = TRUE) when KNN is computed here. Use 0.9, 0.95, or 0.99; CUDA method = "auto" uses it for Flat-vs-IVF selection, CUDA IVF uses it for probing defaults, and HNSW uses it for graph-search tiers.

cagra_implementation — CUDA CAGRA provider passed to nn(..., exclude_self = TRUE) when KNN is computed here. NULL uses the global option; "auto" uses the same deterministic shape-aware provider rule as nn(), while "faiss_gpu" or "cuvs" force one provider.

cagra_build_algo — Direct RAPIDS cuVS CAGRA graph-build algorithm passed to nn(..., exclude_self = TRUE) when KNN is computed here. NULL uses the global faissR.cuvs_cagra_build_algo option. This setting applies to direct cuVS CAGRA and accepts "auto", "ivf_pq", "nn_descent", or "iterative_cagra_search".

weight — Graph weighting. "auto" uses SNN/Jaccard weights for input space and distance weights for embedding space. "snn" builds shared-nearest-neighbour Jaccard weights. "adaptive" uses a local radius kernel. "distance" uses 1 / (1 + distance). "binary" gives every edge weight 1.

mutual — If TRUE, keep only reciprocal nearest-neighbour edges.

prune — Drop edges with weight less than or equal to this value.

n_threads — CPU threads passed to nn() when KNN is computed here.

Value

A native faissR_graph edge-list object. The faissR_graph attribute stores graph-construction metadata: graph size, weighting, nearest-neighbour method, metric, tuning policy, requested/resolved KNN backends, and compact-relevant KNN result metadata such as nn_approximation, nn_faiss, nn_cuvs, nn_spatial_index, nn_auto_selection, nn_metric_transform, and nn_distance_transform. For precomputed KNN input, nn_backend prefers the KNN object's resolved backend when available, so benchmark metadata records concrete FAISS/cuVS routes rather than only the public requested backend.

Examples

```
x <- scale(as.matrix(iris[, 1:4]))
g <- knn_graph(x, k = 15, backend = "cpu")
cl <- graph_cluster(g, method = "louvain", backend = "cpu", n_clusters = 3)
table(cl$membership)
```

---

nn *Nearest neighbors from row-wise matrices*

Description

‘nn()‘ provides a package-native nearest-neighbor entry point compatible with the common ‘nn(data, points, k)‘ use case. The public API separates device selection from algorithm selection. ‘backend‘ is one of ‘"auto"‘, ‘"cpu"‘, or ‘"cuda"‘; ‘method‘ chooses the algorithm. For example, ‘backend = "cpu", method = "grid"‘ uses the CPU grid implementation, while ‘backend = "cuda", method = "grid"‘ uses the CUDA grid implementation. Invalid combinations stop clearly before computation; for example, ‘backend = "cpu", method = "cagra"‘ errors because CAGRA is CUDA-only.

Usage

```
nn(
  data,
  points = data,
  k = NULL,
  exclude_self = FALSE,
  backend = c("auto", "cpu", "cuda"),
 method = c("auto", "exact", "flat", "bruteforce", "grid", "hnsw", "ivf", "ivfpq",
    "vamana", "nsg", "nndescent", "ivfpq_fastscan", "cagra"),
  metric = c("euclidean", "cosine", "correlation", "inner_product"),
  tuning = c("auto", "cache", "pilot", "fixed", "off", "none"),
  target_recall = 0.99,
  cagra_implementation = NULL,
  cagra_build_algo = NULL,
  output = c("double", "float"),
  distances = NULL,
  n_threads = NULL
)
```

Arguments

data	Numeric matrix/data frame or optional ‘float::fl()‘/‘float32‘ object of reference observations in rows. FAISS CPU/GPU and RAPIDS cuVS nearest-neighbour routes use direct float-pointer adapters for float32 inputs. Native routes without a direct float32 adapter fail clearly instead of silently converting benchmark input back to R double.

`points` Numeric matrix/data frame or optional ‘float::fl()‘/‘float32‘ query object with observations in rows. Defaults to ‘data‘. A float32 query can be paired with an ordinary R double reference matrix on direct FAISS/cuVS float32 routes.

`k` Number of neighbors to return. ‘NULL‘ chooses the package’s automatic neighborhood size and includes the self-neighbor when ‘points‘ is ‘data‘.

`exclude_self` Logical; when ‘TRUE‘, remove each query row from its own neighbour list. This is valid only for self-query calls where ‘points‘ is omitted or identical to ‘data‘. Self-neighbour removal is passed to the compiled backend path rather than repaired by R-side post-processing. CUDA graph routes that do not yet expose compiled include-self output shaping require ‘exclude_self = TRUE‘ and fail clearly instead of reshaping in R.

`backend` Device backend: ‘"auto"‘, ‘"cpu"‘, or ‘"cuda"‘. ‘"auto"‘ uses a validated CUDA route only when the requested method/metric combination is supported and CUDA/cuVS runtime support is available, and otherwise resolves to CPU. Explicit ‘"cuda"‘ fails clearly when CUDA support or the selected CUDA combination is unavailable.

`method` Algorithm selector. ‘"auto"‘ chooses a shape-aware default for the selected backend. Other values include ‘"exact"‘, ‘"flat"‘, ‘"bruteforce"‘, ‘"grid"‘, ‘"hnsw"‘, ‘"ivf"‘, ‘"ivfpq"‘, ‘"vamana"‘, ‘"nsg"‘, ‘"nndescent"‘, ‘"ivfpq_fastscan"‘, and ‘"cagra"‘. Use these canonical lowercase method labels; resolved implementation labels such as ‘"faiss_hnsw"‘ or ‘"cuda_cuvs_cagra"‘ are not public ‘method‘ values. Unsupported backend/method combinations fail clearly; for example, ‘method = "cagra", backend = "cpu"‘ errors because CAGRA is CUDA-only, and CUDA ‘method = "ivfpq_fastscan"‘ accepts Euclidean/L2, cosine, correlation, and raw inner-product search. CPU FastScan accepts Euclidean, cosine, correlation, and raw inner product; cosine is implemented by row L2 normalization, correlation by row centering plus L2 normalization followed by FastScan L2 search, and raw inner product by FAISS FastScan IP. CUDA raw inner product applies the maximum-inner-product-to-L2 transform before direct cuVS 4-bit IVF-PQ search.

`metric` Distance metric. The intentionally small public set is ‘"euclidean"‘, ‘"cosine"‘, ‘"correlation"‘, and ‘"inner_product"‘. Legacy metric aliases such as ‘"l2"‘, ‘"cor"‘, ‘"pearson"‘, ‘"ip"‘, ‘"dot"‘, and ‘"innerproduct"‘ are rejected; use the canonical metric names. ‘"inner_product"‘ is the raw dot product, ‘"cosine"‘ is the dot product after row L2 normalization, and ‘"correlation"‘ is centered cosine similarity after subtracting each row mean and L2-normalizing each row. For ‘metric = "inner_product"‘, neighbours are ranked by larger raw dot product, but returned ‘distances‘ keep faissR’s smaller-is-better convention: within each query row the best returned dot product has distance ‘0‘, and lower dot products have larger shifted distances. ‘"euclidean"‘ is the validated high-performance default. ‘"cosine"‘ and ‘"correlation"‘ are implemented for exact CPU KNN, native 2D/3D grid search, FAISS CPU/GPU Flat, FAISS CPU/GPU IVF-Flat, FAISS CPU/GPU IVFPQ, FAISS CPU FastScan, CUDA cuVS IVFPQ FastScan for cosine and correlation, FAISS CPU HNSW, and native graph-refinement routes. FAISS approximate IP-capable routes use row L2 normalization for cosine and row centering plus L2 normalization for correlation before inner-product search; distances are returned as ‘1 - similarity‘. All-zero cosine rows

and constant correlation rows are zero-normalized edge cases: two zero-normalized rows have distance ‘0‘, while a zero-normalized row versus a nonzero row has distance ‘1‘. CPU FAISS Flat uses the exact CPU scorer for those rows to preserve deterministic small-‘k‘ tie handling; explicit CUDA routes error clearly instead of repairing those rows on CPU. CUDA FAISS/cuVS results carry ‘attr(result, "gpu_residency")‘ metadata with provider, index residency, host/device transfer strategy, query device reuse, and CPU fallback flags. CPU ‘method = "auto"‘ can use FAISS Flat for larger exact non-Euclidean query workloads, FAISS HNSW for large non-Euclidean self-search, native CPU NSG/Vamana refinement for selected larger self-KNN cases, and native CPU NN-descent for other large self-KNN cases. CPU ‘method = "hnsw"‘ uses FAISS HNSW for all metrics. ‘"inner_product"‘ is exact on native CPU routes and maps to FAISS Flat IP, FAISS IVF-Flat/IVFPQ IP, FAISS HNSW IP, native CPU NN-descent raw dot-product search, direct cuVS brute force through an exact MIPS-to-L2 transform, direct cuVS IVF/PQ through transformed approximate L2 indexes, CUDA CAGRA and CUDA cuVS HNSW through the same MIPS-to-L2 graph-search transform. CUDA ‘method = "nndescent"‘ uses direct cuVS NN-descent and does not support raw inner-product search. Direct cuVS NN-descent does not expose a safe raw-inner-product route in faissR. CUDA HNSW metadata records the available cuVS HNSW wrapper design. Unsupported backend combinations fail clearly instead of returning neighbours computed under a different metric.

`tuning` Tuning policy. ‘"auto"‘ uses deterministic no-pilot defaults for the resolved method, ‘"cache"‘ reuses/stores pilot results, ‘"pilot"‘ tunes for this call without persisting, ‘"fixed"‘ uses fixed defaults with tuning metadata, and ‘"off"‘/‘"none"‘ disables tuning. CPU Euclidean/cosine/correlation/inner-product ‘method = "exact"‘ uses compiled C++ policies for FAISS Flat query batch size metadata; it records ‘exact_recall_by_construction = TRUE‘ because exact search has no recall approximation parameter. CPU Euclidean/cosine/correlation/inner-product ‘method = "flat"‘ uses separate compiled C++ policies for FAISS Flat query batch size and fitted-index reuse, derived from the Flat CPU12 HPC tuning sweeps; it records the selected policy in ‘attr(result, "flat_tuning")‘ and also remains exact by construction. CPU Euclidean/cosine/correlation/inner-product ‘method = "bruteforce"‘ uses its own compiled C++ policy for FAISS Flat query batch size and fitted-index reuse, derived from the bruteforce CPU12 HPC tuning sweeps; it records the selected policy in ‘attr(result, "bruteforce_tuning")‘ and also remains exact by construction. CUDA Euclidean/cosine/correlation/inner-product ‘method = "exact"‘ uses compiled C++ policies for FAISS GPU Flat query batch size and GPU resource reuse, including the correlation summary in ‘benchmark_scripts/cuda_exact_correlation_shape_tuning_defaults_from_uploaded_results.csv‘ and the validation-pending inner-product seed table in ‘benchmark_scripts/cuda_exact_inner_product_sha it records the selected policy in ‘attr(result, "exact_tuning")‘. CUDA Euclidean/cosine/correlation/inner-product ‘method = "flat"‘ uses separate compiled policies for FAISS GPU Flat query batch size and GPU resource reuse, including the correlation summary in ‘benchmark_scripts/cuda_flat_correlation_shape_tuning_defaults_from_uploaded_results.csv‘ and the validation-pending inner-product seed table in ‘benchmark_scripts/cuda_flat_inner_product_shape it records the selected policy in ‘attr(result, "flat_tuning")‘. CUDA Euclidean/cosine/correlation/inner-product ‘method = "bruteforce"‘ use separate compiled C++ policies for cuVS

brute-force GPU query batch size and GPU resource reuse. Euclidean rows are derived from CUDA bruteforce HPC tuning sweeps; cosine, correlation, and inner-product rows initially reuse those batch/resource choices as metric-transform proxies until empirical metric-specific sweeps replace them. Correlation uses row-centering plus row normalization before cuVS L2 brute-force search; raw inner product uses an exact maximum-inner-product-to-L2 transform. The selected policy is recorded in ‘attr(result, "bruteforce_tuning")‘. CUDA Euclidean/cosine/correlation/inner-product ‘method = "ivf"‘ uses compiled C++ policies for FAISS GPU IVF-Flat ‘nlist‘ and ‘nprobe‘. Euclidean, cosine, and correlation rows come from measured CUDA IVF sweeps; raw inner product uses the validation-pending seed table in ‘benchmark_scripts/cuda_ivf_inner_product_shape_tuning_defa until ‘run_hpc_ivf_tuning_cuda_inner_product.sh‘ replaces it with measured IP rows. Seeded rows record ‘tuning_benchmark_target_met = FALSE‘. CUDA Euclidean/correlation/inner-product ‘method = "ivfpq"‘ uses compiled C++ policies for FAISS GPU IVF-PQ ‘nlist‘, ‘nprobe‘, ‘pq_m‘, and ‘pq_nbits‘. Raw inner product uses ‘benchmark_scripts/cuda_ivfpq_inner_product_shape_tuning_defaults_from_seeded_eu until ‘run_hpc_ivfpq_tuning_cuda_inner_product.sh‘ replaces it with measured IP rows. Seeded rows record ‘tuning_benchmark_target_met = FALSE‘. FAISS CPU HNSW uses deterministic no-pilot defaults based on ‘n‘, ‘p‘, ‘k‘, ‘metric‘, and ‘target_recall‘. Euclidean, cosine, correlation, and raw inner-product CPU FAISS HNSW use compiled HPC-derived lookup tables for shape groups, ‘k‘ buckets (‘15‘, ‘30‘, ‘50‘, and ‘100‘), and target recall; rows that did not meet the target on all shape-group datasets report ‘tuning_benchmark_target_met = FALSE‘. CUDA cuVS HNSW Euclidean, cosine, correlation, and raw inner product use separate compiled HPC-derived lookup tables keyed by dataset shape group, ‘k‘ bucket (‘15‘, ‘30‘, ‘50‘, and ‘100‘), and ‘target_recall‘; rows that did not meet the target report ‘tuning_benchmark_target_met = FALSE‘. Cosine uses row-normalized float32 Euclidean graph search, correlation uses centered row-normalized float32 Euclidean graph search, and raw inner product uses a maximum-inner-product-to-L2 transform before converting distances back to the public metric. Explicit ‘faissR.cuvs_*‘ HNSW/CAGRA options override those defaults. CPU Euclidean, cosine, correlation, and raw inner-product FAISS IVF use compiled shape/k/target-recall tables for ‘nlist‘ and ‘nprobe‘, and record ‘tuning_benchmark_target_met‘ so best-available partial rows are not confused with guaranteed recall. CUDA raw inner-product IVF has a validation-pending seed table from CUDA Euclidean IVF until the metric-specific sweep is rerun. CPU Euclidean, cosine, correlation, and raw inner-product FAISS IVFPQ use compiled shape/k/target-recall tables for ‘nlist‘, ‘nprobe‘, ‘pq_m‘, and ‘pq_nbits‘; CPU Euclidean, cosine, correlation, and raw inner-product native NSG uses compiled shape/k/target-recall tables for ‘r‘ and ‘graph_k‘; CUDA Euclidean/cosine/correlation/raw-inner-product native NSG uses compiled shape/k/target-recall tables for the same parameters. CUDA Euclidean/cosine rows are measured, while CUDA correlation and raw inner-product rows are validation-pending defaults seeded from measured CUDA cosine rows until the dedicated metric sweeps are rerun; CPU Euclidean, cosine, correlation, and raw inner-product native Vamana and CUDA Euclidean/cosine/correlation/raw-inner-product native Vamana use compiled shape/k/target-recall tables for ‘r‘, ‘search_l‘, and ‘alpha‘. CUDA Vamana correlation and raw-inner-product rows

are validation-pending defaults seeded from measured CUDA cosine rows until the dedicated metric sweeps are rerun; CUDA CAGRA cosine/correlation/raw-inner-product rows use validation-pending defaults seeded from the measured CUDA Euclidean CAGRA sweep until the metric-specific sweeps are rerun; optional FAISS GPU IVF ‘"cache"‘/‘"pilot"‘ tuning currently runs only for Euclidean IVF. CPU and CUDA HNSW use the selected ‘target_recall‘ value. CUDA cosine ‘method = "ivfpq_fastscan"‘ uses a shape/k/target policy seeded from the CUDA Euclidean FastScan sweep and records ‘tuning_benchmark_target_met = FALSE‘ until the corrected cosine sweep is rerun. The default is the 0.99 tier where feasible; ‘target_recall = 0.95‘ and ‘0.9‘ select faster, lower-recall HNSW tiers. Deterministic method defaults are computed by C++ ‘nn_tune_*_cpp()‘ helpers and record ‘tuning_source = "cpp"‘ in result metadata. Advanced tuning and cache knobs use ‘options(faissR.<name> = ...)‘.

`target_recall` Speed/recall tier. Use ‘0.9‘, ‘0.95‘, or ‘0.99‘. CUDA ‘method = "auto"‘ uses it to choose between Flat/brute force and IVF-Flat for Euclidean self-KNN. CPU Euclidean/cosine/correlation/inner-product ‘method = "ivf"‘ and CPU Euclidean/cosine/correlation/inner-product ‘method = "ivfpq"‘ use it for compiled ‘nlist‘/‘nprobe‘ or ‘nlist‘/‘nprobe‘/‘pq_m‘/‘pq_nbits‘ tiers; CUDA Euclidean/cosine/correlation ‘method = "ivf"‘ uses it for compiled ‘nlist‘/‘nprobe‘, CUDA Euclidean/correlation/raw-inner-product ‘method = "ivfpq"‘ uses it for compiled ‘nlist‘/‘nprobe‘/‘pq_m‘/‘pq_nbits‘, CPU Euclidean/cosine/correlation/inne
product ‘method = "nsg"‘ uses it for compiled ‘r‘/‘graph_k‘ tiers, CUDA Euclidean/cosine/correlation/inn
product ‘method = "nsg"‘ uses it for compiled ‘r‘/‘graph_k‘ tiers, CPU Euclidean/cosine/correlation/inner-product ‘method = "vamana"‘ and CUDA Euclidean/cosine/correlation/inner-product ‘method = "vamana"‘ use it for compiled ‘r‘/‘search_l‘/‘alpha‘ tiers, and CPU/CUDA HNSW use it for graph-search tiers. CPU IVF/IVFPQ and CUDA IVF/IVFPQ metadata records ‘tuning_benchmark_target_met‘ for benchmark-derived rows. CPU Euclidean/cosine/correlation/inner-product and CUDA Euclidean/cosine/correlation/i
product ‘method = "exact"‘ record the requested tier in exact tuning metadata, CPU Euclidean/cosine/correlation/inner-product and CUDA Euclidean/cosine/correlation/inner-product ‘method = "flat"‘ record it in flat-tuning metadata, and CPU Euclidean/cosine/correlation/inner-product ‘method = "bruteforce"‘ records it in bruteforce-tuning metadata, while recall remains 1.0 by construction. Lower values trade recall for speed where the selected method supports a recall/speed tier.

`cagra_implementation`
CUDA CAGRA provider for this call. ‘NULL‘ uses the global ‘options(faissR.cagra_implementation = ...)‘ value. ‘"auto"‘ uses a deterministic provider rule: compact high-dimensional self-KNN selects direct RAPIDS cuVS CAGRA when both providers are available, while FAISS GPU CAGRA remains the default for other shapes. ‘"faiss_gpu"‘ and ‘"cuvs"‘ force one provider for benchmarking. This argument affects only public ‘backend = "cuda", method = "cagra"‘ requests and CUDA-auto routes that select CAGRA.

`cagra_build_algo`
Direct RAPIDS cuVS CAGRA graph-build algorithm for this call. ‘NULL‘ uses ‘options(faissR.cuvs_cagra_build_algo = "auto")‘. For direct cuVS CAGRA, ‘"auto"‘ applies faissR’s deterministic shape-aware CAGRA build rule, choosing iterative CAGRA construction for compact high-dimensional self-KNN cases and IVF-PQ construction otherwise. ‘"ivf_pq"‘ requests the IVF-PQ graph builder, ‘"nn_descent"‘ requests cuVS NN-descent graph construction, and ‘"iterative_cagra_search"‘

requests cuVS iterative CAGRA graph building. This is a CAGRA construction parameter, not a fallback to a different public method; successful results record the selected value in ‘attr(result, "approximation")$cagra_build_algo‘.

`output` Distance storage type for the returned object. ‘"double"‘ returns the default R numeric distance matrix. ‘"float"‘ returns ‘distances‘ as a ‘float::fl()‘/‘float32‘ object and records ‘distance_type = "float32"‘ plus ‘attr(result, "distance_type") = "float32"‘; this requires the optional ‘float‘ package. When either ‘data‘ or ‘points‘ is a ‘float::fl()‘ matrix, FAISS Flat/IVF/IVFPQ/HNSW/FastScan/NSG/NNDescent, FAISS GPU Flat/IVF/IVFPQ/CAGRA, and RAPIDS cuVS brute-force/CAGRA/IVF/IVFPQ/NN-descent routes consume float32 input directly through C++ adapters. Methods without a direct float32 adapter now error instead of silently converting the benchmark input back to R double. Ordinary R double inputs with ‘output = "float"‘ also enter float-pointer routes for CPU FAISS Flat/IVF/IVFPQ/FastScan, cached CPU FAISS fitted indexes, FAISS GPU Flat/IVF/IVFPQ, and direct Euclidean RAPIDS cuVS brute-force/CAGRA/IVF/IVFPQ and IVFPQ FastScan routes. On direct FAISS/cuVS float routes, float distance output is constructed directly from backend float results instead of first materializing an R double distance matrix, except for routes that need R-side metric transformation or zero-row cosine/correlation correction.

`distances` Optional alias for ‘output‘, kept for callers that prefer ‘distances = "double"‘ or ‘distances = "float"‘ to describe the returned distance storage type.

`n_threads` Number of CPU worker threads for CPU backends. GPU backends ignore this argument.

Details

Method descriptions:

- ‘"auto"‘: shape-aware selector for the selected backend. CPU auto uses exact, grid, FAISS IVF, FAISS HNSW, native CPU NSG/Vamana refinement, or native CPU NN-descent depending on data shape and size. CUDA auto uses CUDA grid for 2D/3D Euclidean/cosine/correlation self-KNN. For Euclidean non-grid self-KNN, CUDA auto uses benchmark-derived shape/k/target-recall rules: large low-dimensional datasets use IVF-Flat, very large high-dimensional datasets use IVF only for lower recall tiers, and the high-recall default keeps exact FAISS GPU Flat/brute force for the other measured shapes. Non-grid cosine/correlation/IP auto routes stay on exact FAISS GPU Flat/cuVS brute force when available [1-3,5,13-15,22-23]. When ‘backend = "auto"‘ is combined with an explicit method, faissR first checks whether that exact method/metric has a runtime-capable CUDA route; otherwise it uses the CPU route when that method/metric is supported on CPU.
- ‘"exact"‘: exact nearest-neighbour search. CPU Euclidean exact uses FAISS Flat L2, CPU cosine exact uses normalized FAISS Flat cosine, and CPU correlation exact uses centered/normalized FAISS Flat correlation; these routes apply compiled exact tuning metadata for query batching. CUDA uses FAISS GPU Flat when the linked FAISS build reports GPU support; Euclidean, cosine, and correlation exact CUDA routes use compiled shape/k/target policies for GPU query batching. CUDA exact search can otherwise use direct cuVS brute force when available: Euclidean uses cuVS L2 directly, cosine/correlation use normalized Euclidean search, and inner product uses an exact maximum-inner-product-to-L2 transform [1-3,16].

- ‘"flat"‘: FAISS Flat exhaustive index. CPU and FAISS GPU support L2, IP, and normalized-IP cosine/correlation routes when available [1-2,16]. CPU Euclidean/cosine/correlation/inner-product Flat uses compiled metric/shape/k/target-recall policies for query batching and fitted-index reuse; CUDA Euclidean/cosine/correlation Flat uses compiled FAISS GPU Flat query-batch/resource policies. Result metadata stores the selected table row in ‘attr(result, "flat_tuning")‘.
- ‘"bruteforce"‘: exhaustive brute-force search. CPU Euclidean brute force uses FAISS Flat L2, CPU cosine brute force uses normalized FAISS Flat cosine, and CPU correlation brute force uses centered/normalized FAISS Flat correlation. It has its own compiled metric/shape/k/target tuning policy for FAISS query batching and fitted-index reuse, stored in ‘attr(result, "bruteforce_tuning")‘. On CUDA, RAPIDS cuVS brute force is preferred when available. CUDA Euclidean and cosine brute force use compiled shape/k/target-recall policies for GPU query batch size and resource reuse; cosine/correlation use normalized Euclidean search and inner product uses an exact maximum-inner-product-to-L2 transform around the cuVS L2 kernel [1-3,16].
- ‘"grid"‘: native exact 2D/3D spatial grid search for Euclidean, cosine, and correlation self-KNN. Cosine/correlation use normalized Euclidean grid search. Include-self output is finalized in compiled C++/CUDA code rather than by R-side matrix reshaping. Explicit grid requests error for higher-dimensional matrices; use ‘"auto"‘ to let faissR choose a non-grid method when appropriate.
- ‘"hnsw"‘: HNSW approximate graph-search index [3,5,16,22-23]. CPU uses FAISS HNSW. CUDA uses RAPIDS cuVS HNSW by building a CAGRA seed graph and converting it with ‘cuvsHnswFromCagraWithDataset‘ using the host dataset and cuVS CPU hierarchy; result metadata marks this as the cuVS wrapper design rather than a pure all-GPU HNSW search implementation. Default HNSW parameters are selected by compiled deterministic rules without pilot tuning. Euclidean, cosine, correlation, and raw inner-product CPU FAISS HNSW use HPC-derived lookup tables keyed by dataset shape group, ‘k‘ bucket (‘15‘, ‘30‘, ‘50‘, or ‘100‘), and ‘target_recall = 0.9‘, ‘0.95‘, or ‘0.99‘; CUDA cuVS HNSW uses metric-specific shape/k/recall lookup tables for its CAGRA seed graph and cuVS HNSW conversion route. CUDA raw inner product uses a maximum-inner-product-to-L2 transform and currently uses ‘benchmark_scripts/cuda_hnsw_inner_product_shape_tuning_defaults_from_seeded_euclidean_results.csv‘ until the dedicated CUDA IP sweep replaces the seed rows. Raw inner-product HNSW rows that did not reach the requested benchmark target report ‘tuning_benchmark_target_met = FALSE‘. Result approximation metadata records the selected ‘tuning_rule‘, ‘target_recall‘, ‘tuning_shape_group‘, ‘tuning_k_bucket‘, benchmark source fields, and ‘tuning_benchmark_target_met‘. When a CUDA HNSW benchmarked shape did not reach the requested target, ‘tuning_benchmark_basis‘ is recorded as ‘"target_not_reached_best_available"‘.
- ‘"ivf"‘: FAISS IVF-Flat inverted-file index, trading exhaustive search for coarse-list probing. It supports L2, raw IP, and normalized-IP cosine/correlation routes on CPU and FAISS GPU [1-2,16]. CPU Euclidean, cosine, correlation, and raw inner-product IVF record compiled shape/k/target ‘nlist‘/‘nprobe‘ tiers and ‘tuning_benchmark_target_met‘ in approximation metadata.
- ‘"ivfpq"‘: FAISS inverted-file index with product quantization, mainly for compressed-memory approximate search. It supports L2, raw IP, and normalized-IP cosine/correlation routes on CPU and FAISS GPU [1-2,6,16]. CPU Euclidean, cosine, correlation, and raw inner-product IVFPQ record compiled shape/k/target ‘nlist‘, ‘nprobe‘, ‘pq_m‘, and ‘pq_nbits‘ tiers plus ‘tuning_benchmark_target_met‘ in approximation metadata. IVFPQ is still a compression-first route, so best-available or target-not-reached rows should not be read as accuracy guarantees.

CUDA IVFPQ routes keep CUDA-oriented shape/k/target rows for Euclidean, correlation, and raw inner product; the raw-IP rows are seeded from the CUDA Euclidean IVFPQ table until the dedicated IP sweep replaces them and report ‘tuning_benchmark_target_met = FALSE‘.

- ‘"vamana"‘: DiskANN/Vamana-style robust-pruned candidate graph implemented in faissR [24]. CPU refines top-k within candidate rows using native CPU scoring; CUDA refines candidates with faissR’s native CUDA row-candidate kernel. Large high-dimensional CPU inputs use a deterministic FAISS HNSW seed before robust pruning; smaller inputs keep the exact seed. The first ‘k‘ seed neighbours are protected before robust pruning so pruning cannot discard neighbours already found by the seed generator. Robust pruning runs in compiled C++ over compact candidate storage before CPU or CUDA candidate refinement. cuVS Vamana is acknowledged for GPU build/serialization, but current cuVS documentation does not expose KNN search for this index. CPU Euclidean, cosine, correlation, and raw inner-product plus CUDA Euclidean/cosine/correlation/raw-inner-product ‘tuning = "auto"‘ select Vamana ‘r‘, ‘search_l‘, and ‘alpha‘ from compiled shape/k/target-recall tables and record whether the benchmark row reached the requested target. CUDA cosine rows come from measured normalized float32 Vamana sweeps; CUDA correlation and raw-inner-product rows are seeded from those cosine rows and report ‘tuning_benchmark_target_met = FALSE‘ until the dedicated metric sweeps are rerun. Cosine uses row-normalized Euclidean Vamana refinement and correlation uses row-centered, normalized Euclidean Vamana refinement; raw inner product uses shifted dot-product ordering. All keep metric-specific tuning tables.
- ‘"nsg"‘: Navigating Spreading-out Graph style approximate search [16,21,29]. CPU uses faissR’s native NSG-style self-KNN candidate graph for all public metrics to avoid unsafe linked-FAISS graph construction. CUDA uses faissR’s native NSG-style self-KNN candidate graph for all public metrics; cosine/correlation use normalized Euclidean search and raw inner product uses shifted dot-product distances. Large high-dimensional CPU inputs use a deterministic FAISS HNSW seed before NSG/MRNG-style pruning; smaller inputs keep the exact seed. The first ‘k‘ seed neighbours are protected before compiled C++ NSG/MRNG-style pruning over compact candidate storage. CPU Euclidean, cosine, correlation, and raw inner-product ‘tuning = "auto"‘ select NSG ‘r‘ and ‘graph_k‘ from compiled shape/k/target-recall tables and record whether the benchmark row reached the requested target.
- ‘"nndescent"‘: NN-descent approximate graph construction via faissR’s native CPU route, or direct cuVS on CUDA for Euclidean/L2 plus normalized cosine/correlation. CUDA raw inner-product NN-descent is not exposed because direct cuVS NN-descent does not support it. The native CPU route uses random-projection window seeds, flat row-major graph buffers, and fixed-width reverse-neighbour candidate storage in C++. CPU Euclidean, cosine, correlation, and raw inner-product ‘tuning = "auto"‘ select pool size, iterations, maximum candidates, and random projections from compiled shape/k/target-recall tables. FAISS NNDescent is experimental opt-in because linked FAISS builds can abort during graph construction [3-4,16].
- ‘"cagra"‘: CUDA-only graph-search method via FAISS GPU CAGRA/cuVS integration or direct RAPIDS cuVS CAGRA. By default faissR chooses FAISS GPU CAGRA when that route is available and otherwise direct cuVS CAGRA; set ‘options(faissR.cagra_implementation = "faiss_gpu")‘ or ‘"cuvs"‘ to force one provider for benchmarking. Availability preflights respect this forced provider for supported metrics, and approximation metadata records ‘cagra_provider‘ plus ‘cagra_provider_option‘. It supports Euclidean/L2, cosine/correlation through normalized Euclidean graph search, and raw inner product through a maximum-inner-product-to-L2 extra-dimension transform whose returned distances are converted back to faissR’s shifted inner-product convention. CUDA Euclidean CAGRA ‘tuning = "auto"‘ uses measured shape/k/target rows from ‘benchmark_scripts/cuda_cagra_euclidean_shape_tuning_defaults_from_uploaded_results.cs

CUDA cosine normalizes rows to float32, CUDA correlation row-centers then row-normalizes rows to float32, and CUDA raw inner product uses the maximum-inner-product-to-L2 transform; all three use validation-pending tables seeded from the measured Euclidean CAGRA sweep until the corrected metric-specific sweeps are rerun. The raw-inner-product seed table is ‘benchmark_scripts/cuda_cagra_inner_product_shape_tuning_defaults_from_seeded_euclidean_results.csv‘. Those seeded rows report ‘tuning_benchmark_target_met = FALSE‘ [3,13-16].

References are numbered as in ‘docs/references.md‘ in the GitHub repository.

Value

A list with integer matrix ‘indices‘, ‘distances‘, and stable metadata fields ‘index_base‘, ‘distance_type‘, ‘metric‘, and ‘backend_used‘. Float32 routes also record ‘input_layout‘ and ‘input_owns_data‘ so downstream packages can distinguish direct float32 payload use from one-time row-major conversion. Normalized Euclidean graph routes for cosine/correlation record ‘metric_transform‘ and ‘attr(result, "distance_transform")‘. Indices are 1-based. The requested backend/method, tuning policy, resolved backend, metric, exact/approximate flag, and self-query flag are stored in attributes including ‘attr(result, "requested_backend")‘, ‘attr(result, "requested_method")‘, ‘attr(result, "tuning")‘, and ‘attr(result, "resolved_backend")‘. Auto requests also include ‘attr(result, "auto_selection")‘, a static shape/k/metric decision record that records the predicted internal backend, public method class, device class, explicit backend/method flags, backend/method decision reasons, and does not run pilot tuning. CPU FAISS Flat/HNSW/IVF/IVFPQ/FastScan routes use a bounded session-local fitted-index cache for repeated raw ‘nn()‘ calls with matching data and parameters. CUDA ‘method = "ivfpq_fastscan"‘ also reuses a fitted cuVS IVF-PQ index, dataset device buffer, and cuVS resources through a separate bounded cache. Self-query uses the fitted dataset device buffer directly, and repeated separate-query calls can reuse one cached query device buffer with ‘options(faissR.cache_cuda_ivfpq_query_buffers = TRUE)‘. Metadata reports ‘persistent_index_cache‘, ‘index_cache_hit‘, ‘dataset_residency‘, ‘query_residency‘, and ‘query_host_to_device_copies‘. Disable CPU and CUDA fitted-index caches with ‘options(faissR.cache_fitted_nn_indexes = FALSE)‘, bound CPU memory with ‘options(faissR.cache_fitted_nn_indexes_max_entries = <n>)‘, or bound CUDA FastScan GPU memory with ‘options(faissR.cache_fitted_cuda_ivfpq_indexes_max_entries = <n>)‘. CPU Euclidean, cosine, correlation, and raw inner-product ‘method = "ivfpq_fastscan"‘ resolve ‘tuning = "auto"‘ in C++ from shape/k/target-recall defaults for ‘nlist‘, ‘nprobe‘, ‘pq_m‘, ‘refine_factor‘, and FastScan block size. Cosine uses row L2 normalization and correlation uses row centering plus L2 normalization before FastScan L2; raw inner product uses FastScan IP. CUDA FastScan cosine, correlation, and raw-inner-product auto policies are seeded from the CUDA Euclidean FastScan table until metric-specific HPC sweeps replace them, and those rows report ‘tuning_benchmark_target_met = FALSE‘.

References

Johnson J, Douze M, Jegou H. Billion-scale similarity search with GPUs. IEEE Transactions on Big Data. 2021;7:535-547.

Douze M, Guzhva A, Deng C, Johnson J, Szilvasy G, Mazaré PE, et al. The FAISS library. arXiv 2024. See also the FAISS C++ API documentation.

RAPIDS Development Team. RAPIDS cuVS: GPU-accelerated vector search and clustering. https://github.com/rapidsai/cuv

Dong W, Moses C, Li K. Efficient k-nearest neighbor graph construction for generic similarity measures. WWW 2011:577-586.

Malkov YA, Yashunin DA. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE TPAMI. 2020;42:824-836.

Jégou H, Douze M, Schmid C. Product quantization for nearest neighbor search. IEEE TPAMI. 2011;33:117-128.

NVIDIA, Meta, and FAISS documentation for FAISS GPU indexes backed by NVIDIA cuVS, including IVF and CAGRA integration.

Examples

```
x <- scale(as.matrix(iris[, 1:4]))
knn_euclidean <- nn(x, k = 16, metric = "euclidean", backend = "cpu")
knn_cosine <- nn(x, k = 16, metric = "cosine", backend = "cpu")
knn_correlation <- nn(x, k = 16, metric = "correlation", backend = "cpu")
knn_ip <- nn(x, k = 16, metric = "inner_product", backend = "cpu")
```

---

nn_capabilities *Nearest-neighbour method capabilities*

Description

`nn_capabilities()` returns the public method/backend/metric support table used by the nearest-neighbour API. It separates combinations that are supported by design from combinations that should be treated as expected skips in benchmarks.

Usage

```
nn_capabilities(runtime = FALSE)
```

Arguments

`runtime`	Logical; when `FALSE` (the default), report support by design without checking the current compiled/runtime libraries. When `TRUE`, add `resolved_backend`, `runtime_available`, `runtime_reason`, and `runtime_notes` columns for the current installation. `runtime_reason` uses stable labels such as `"available"`, `"unsupported_combination"`, `"missing_faiss"`, `"missing_faiss_gpu"`, `"missing_cuda"`, `"missing_cuda_route"`, and `"missing_cuvs"` for benchmark preflight tables.

Details

faissR exposes four public nearest-neighbour metrics: `"euclidean"`, `"cosine"`, `"correlation"`, and `"inner_product"`. Correlation is centered cosine similarity, whereas inner product is the raw dot product. For `metric = "inner_product"`, neighbours are ranked by larger raw dot product, but returned `distances` keep faissR's smaller-is-better convention by shifting each query row so the best returned dot product has distance `0`.

The capability table reports metric support only when the selected route computes neighbours under that metric. FAISS Flat, IVF-Flat, IVFPQ, and HNSW support cosine and correlation by using normalized inner-product searches. CAGRA and NNDescent support cosine and correlation by using

normalized Euclidean graph search. Native CPU NNDescent and faissR's native CUDA NNDescent candidate-refinement route also support raw inner-product self-KNN through faissR's shifted dot-product distance convention. Public CUDA CAGRA supports raw inner product through the same maximum-inner-product-to-L2 transform used by direct cuVS brute-force and IVF/PQ routes. Public CUDA HNSW uses RAPIDS cuVS HNSW from a CAGRA seed graph with a cuVS CPU hierarchy and records that wrapper design in metadata rather than presenting it as a pure all-GPU HNSW implementation. Public CPU NSG uses faissR's native NSG-style route for all metrics because linked FAISS graph builders can abort during graph construction; large high-dimensional CPU inputs use deterministic FAISS HNSW seeding before NSG/MRNG-style pruning. Direct cuVS NNDescent does not expose raw inner-product search. Public CUDA `method = "cagra"` can resolve to FAISS GPU CAGRA or direct cuVS CAGRA; `options(faissR.cagra_implementation = "faiss_gpu")` or `"cuvs"` forces one provider, while `"auto"` uses a deterministic shape rule: direct cuVS CAGRA is selected for compact high-dimensional self-KNN, and FAISS GPU CAGRA remains the default when both providers are available for other shapes. Availability preflights respect the forced provider for all CAGRA metrics, and returned approximate NN objects record `cagra_provider` plus `cagra_provider_option`.

Use `runtime = TRUE` for benchmark preflight checks that must distinguish a valid method/metric combination from a route that is unavailable in the current build, such as FAISS GPU rows on a CPU-only installation.

Value

A data frame with one row per public `method`, `backend` (`"auto"`, `"cpu"`, or `"cuda"`), and `metric` combination. Columns include `supported`, `exact`, `implementation`, and `notes`. If `runtime = TRUE`, runtime availability columns are appended.

Examples

```
caps <- nn_capabilities()
subset(caps, method == "flat" & supported)
runtime_caps <- nn_capabilities(runtime = TRUE)
```

---

nn_gpu *GPU-resident tuned nearest-neighbour search*

Description

'nn_gpu()' runs a CUDA nearest-neighbour search and returns a GPU-resident result object. Unlike 'nn()', the 'indices' and 'distances' are not copied back into R matrices. The returned object stores owning and non-owning external pointers so downstream C/C++ packages can consume the KNN output on the CUDA device.

Usage

```
nn_gpu(
  data,
  points = data,
```

```
  k = NULL,
  exclude_self = FALSE,
  method = c("auto", "exact", "flat", "bruteforce"),
  metric = c("euclidean", "cosine", "correlation", "inner_product"),
  tuning = c("auto", "cache", "pilot", "fixed", "off", "none"),
  target_recall = 0.99
)
```

Arguments

data	Numeric matrix/data frame or optional ‘float::fl()‘/‘float32‘ reference matrix.
points	Optional query matrix. Defaults to ‘data‘.
k	Number of neighbours.
exclude_self	Logical; remove each row from its own neighbour list for self-query calls.
method	‘"auto"‘, ‘"exact"‘, ‘"flat"‘, or ‘"bruteforce"‘. ‘"auto"‘ consults the compiled shape/k/metric/target-recall selector but currently returns GPU-resident exact-family buffers.
metric	‘"euclidean"‘, ‘"cosine"‘, ‘"correlation"‘, or ‘"inner_product"‘.
tuning	Tuning label to record. The current GPU-resident route is exact, so ‘target_recall‘ is metadata for the executed route; auto-selected approximate settings are recorded separately as ‘auto_preferred_tuning‘.
target_recall	Target recall label to record; use ‘0.9‘, ‘0.95‘, or ‘0.99‘.

Details

This is intentionally narrower than ‘nn()‘: for Euclidean and raw inner-product ‘method = "auto"‘, ‘"exact"‘, ‘"flat"‘, or ‘"bruteforce"‘, ‘nn_gpu()‘ uses a FAISS GPU direct ‘bfKnn‘ route and keeps the result buffers on the CUDA device when FAISS GPU is available. When FAISS was built with cuVS support, FAISS may dispatch the brute-force GPU distance primitive through cuVS internally. Raw inner-product results are converted on the CUDA device from FAISS similarities to faissR's shifted smaller-is-better distance. Cosine and correlation currently keep using the native CUDA GPU-resident exact route; they are transformed to normalized squared L2 on the C++ side and stored as ‘1 - similarity‘. Approximate cuVS/FAISS GPU methods currently fail clearly here because those provider result buffers still need provider-specific persistent GPU ownership. For ‘method = "auto"‘, ‘nn_gpu()‘ records the same C++ auto-selection metadata used by ‘nn()‘. If that policy would prefer an approximate CUDA method whose result buffers are not yet exposed as persistent GPU-resident objects, ‘nn_gpu()‘ keeps the exact-family GPU-resident route and records ‘auto_preferred_backend‘, ‘auto_preferred_method‘, and ‘auto_residency_constraint‘. It also records the actual exact-family ‘execution_tuning‘ used by the GPU-resident route and the ‘auto_preferred_tuning‘ row for the approximate method selected by the compiled policy, including inner-product CAGRA/IVF/graph settings when available.

Value

A ‘faissR_gpu_knn‘ list. It contains an owning ‘handle‘, non-owning ‘indices_ptr‘ and ‘distances_ptr‘, ‘n_query‘, ‘k‘, ‘index_base = 1L‘, ‘indices_type = "int32"‘, ‘distance_type = "float32"‘, ‘layout‘, ‘metric‘, and ‘backend_used‘. With ‘tuning = "auto"‘, it also includes ‘execution_tuning‘; with

‘method = "auto"‘, it includes ‘auto_preferred_tuning‘ when the compiled selector has a preferred CUDA method/tuning row.

Examples

```
if (cuda_available()) {
  x <- matrix(rnorm(200), ncol = 4)
  gpu_knn <- nn_gpu(x, k = 5, exclude_self = TRUE)
  print(gpu_knn)
}
```

`predict.faissR_knn_model` *Predict from a faissR kNN model*

Description

Predict from a faissR kNN model

Usage

```
## S3 method for class 'faissR_knn_model'
predict(
  object,
  newdata,
  k = NULL,
  backend = c("auto", "cpu", "cuda"),
  tuning = c("auto", "cache", "pilot", "fixed", "off", "none"),
  target_recall = NULL,
  cagra_implementation = NULL,
  cagra_build_algo = NULL,
  vote = c("majority", "weighted"),
  type = c("response", "prob"),
  ...
)
```

Arguments

`object`	A model returned by `knn()`.
`newdata`	Numeric or optional ‘float::fl()‘/‘float32‘ query matrix with observations in rows. Float32 queries are preserved for methods with direct float32 adapters.
`k`	Number of neighbours.
`backend`	Device backend used for this prediction call: ‘"auto"‘, ‘"cpu"‘, or ‘"cuda"‘. The fitted model’s method and metric are always reused.
`tuning`	Tuning policy used for this prediction call. ‘"auto"‘ uses the deterministic default for the resolved method; pilot/cache tuning is opt-in where implemented. FAISS GPU IVF pilot/cache tuning is Euclidean-only; non-Euclidean IVF routes use deterministic metric-aware defaults.

`target_recall` Optional speed/recall tier for this prediction call. ‘NULL‘ reuses the fitted model’s value; otherwise use ‘0.9‘, ‘0.95‘, or ‘0.99‘. It affects CUDA auto Flat-vs-IVF selection, CUDA IVF probing, and HNSW graph-search tiers when prediction needs a new NN search.

`cagra_implementation`
CUDA CAGRA provider for this prediction call. ‘NULL‘ reuses the fitted model’s setting, then the global option.

`cagra_build_algo`
Direct RAPIDS cuVS CAGRA graph-build algorithm for this prediction call. ‘NULL‘ reuses the fitted model’s setting, then the global option.

`vote` ‘"majority"‘ or ‘"weighted"‘ for classification; ‘"majority"‘ means an unweighted neighbour mean for regression.

`type` ‘"response"‘ for class/regression predictions or ‘"prob"‘ for class probability matrices from classification models.

`...` Reserved for future options.

Value

A factor for classification, a numeric vector for regression, or a numeric class-probability matrix when ‘type = "prob"‘. Outputs carry ‘attr(result, "faissR_nn")‘ route metadata, approximation parameters, and auto-selection metadata from the underlying `nn()` call. The prediction-time neighbour search is batched: the full ‘newdata‘ matrix is sent to the resolved FAISS/cuVS/native NN route in one call, and metadata records ‘batch_query‘, ‘query_n‘, and ‘query_call_count‘.

Index