Package 'rnndescent'

April 18, 2024

Type Package

Title Nearest Neighbor Descent Method for Approximate Nearest Neighbors

Version 0.1.5

Description The Nearest Neighbor Descent method for finding approximate nearest neighbors by Dong and co-workers (2010) <doi:10.1145/1963405.1963487>. Based on the 'Python' package 'PyNNDescent' <https://github.com/lmcinnes/pynndescent>.

License GPL (>= 3)

URL https://jlmelville.github.io/rnndescent/, https://github.com/jlmelville/rnndescent

BugReports https://github.com/jlmelville/rnndescent/issues

Imports dqrng, Matrix (>= 1.3-0), methods, Rcpp

Suggests covr, knitr, rmarkdown, testthat

LinkingTo BH, dqrng, Rcpp, sitmo

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.1

SystemRequirements C++17

NeedsCompilation yes

Author James Melville [aut, cre, cph],
Vitalie Spinu [ctb],
Ralf Stubner [ctb]

Maintainer James Melville <jlmelville@gmail.com>

Repository CRAN

Date/Publication 2024-04-18 15:12:39 UTC

R topics documented:

`brute_force_knn` *Find exact nearest neighbors by brute force*

Description

Returns the exact nearest neighbors of a dataset. A brute force search is carried out: all possible pairs of points are compared, and the nearest neighbors are returned.

Usage

```
brute_force_knn(
  data,
  k,
  metric = "euclidean",
  use_alt_metric = TRUE,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

`data`	Matrix of `n` items to generate neighbors for, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting `obs = "C"`, which should be more efficient. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse

matrices should be in `dgCMatrix` format. Dataframes will be converted to `numerical` matrix format internally, so if your data columns are `logical` and intended to be used with the specialized binary `metrics`, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).

`k` Number of nearest neighbors to return.

`metric` Type of distance calculation to use. One of:

- `"braycurtis"`
- `"canberra"`
- `"chebyshev"`
- `"correlation"` (1 minus the Pearson correlation)
- `"cosine"`
- `"dice"`
- `"euclidean"`
- `"hamming"`
- `"hellinger"`
- `"jaccard"`
- `"jensenshannon"`
- `"kulsinski"`
- `"sqeuclidean"` (squared Euclidean)
- `"manhattan"`
- `"rogerstanimoto"`
- `"russellrao"`
- `"sokalmichener"`
- `"sokalsneath"`
- `"spearmanr"` (1 minus the Spearman rank correlation)
- `"symmetrickl"` (symmetric Kullback-Leibler divergence)
- `"tsss"` (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- `"yule"`

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- `"cosine-preprocess"`: `cosine` with preprocessing.
- `"correlation-preprocess"`: `correlation` with preprocessing.

For non-sparse binary data passed as a `logical` matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- `"dice"`
- `"hamming"`
- `"jaccard"`

- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

n_threads Number of threads to use.

verbose If TRUE, log information to the console.

obs set to "C" to indicate that the input data orientation stores each observation as a column. The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Details

This method is accurate but scales poorly with dataset size, so use with caution with larger datasets. Having the exact neighbors as a ground truth to compare with approximate results is useful for benchmarking and determining parameter settings of the approximate methods.

Value

the nearest neighbor graph as a list containing:

- idx an n by k matrix containing the nearest neighbor indices.
- dist an n by k matrix containing the nearest neighbor distances.

Examples

```
# Find the 4 nearest neighbors using Euclidean distance
# If you pass a data frame, non-numeric columns are removed
iris_nn <- brute_force_knn(iris, k = 4, metric = "euclidean")

# Manhattan (l1) distance
iris_nn <- brute_force_knn(iris, k = 4, metric = "manhattan")

# Multi-threading: you can choose the number of threads to use: in real
# usage, you will want to set n_threads to at least 2
iris_nn <- brute_force_knn(iris, k = 4, metric = "manhattan", n_threads = 1)

# Use verbose flag to see information about progress
iris_nn <- brute_force_knn(iris, k = 4, metric = "euclidean", verbose = TRUE)
```

`brute_force_knn_query` *Query exact nearest neighbors by brute force*

Description

Returns the exact nearest neighbors of query data to the reference data. A brute force search is carried out: all possible pairs of reference and query points are compared, and the nearest neighbors are returned.

Usage

```
brute_force_knn_query(
  query,
  reference,
  k,
  metric = "euclidean",
  use_alt_metric = TRUE,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

`query`	Matrix of `n` query items, with observations in the rows and features in the columns. Optionally, the data may be passed with the observations in the columns, by setting `obs = "C"`, which should be more efficient. The `reference` data must be passed in the same orientation as `query`. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse matrices should be in `dgCMatrix` format. Dataframes will be converted to `numerical` matrix format internally, so if your data columns are `logical` and intended to be used with the specialized binary `metrics`, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
`reference`	Matrix of `m` reference items, with observations in the rows and features in the columns. The nearest neighbors to the queries are calculated from this data. Optionally, the data may be passed with the observations in the columns, by setting `obs = "C"`, which should be more efficient. The `query` data must be passed in the same format and orientation as `reference`. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse matrices should be in `dgCMatrix` format.
`k`	Number of nearest neighbors to return.
`metric`	Type of distance calculation to use. One of: • `"braycurtis"` • `"canberra"` • `"chebyshev"`

- "correlation" (1 minus the Pearson correlation)
- "cosine"
- "dice"
- "euclidean"
- "hamming"
- "hellinger"
- "jaccard"
- "jensenshannon"
- "kulsinski"
- "sqeuclidean" (squared Euclidean)
- "manhattan"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

n_threads	Number of threads to use.
verbose	If TRUE, log information to the console.
obs	set to "C" to indicate that the input query and reference orientation stores each observation as a column (the orientation must be consistent). The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Details

This is accurate but scales poorly with dataset size, so use with caution with larger datasets. Having the exact neighbors as a ground truth to compare with approximate results is useful for benchmarking and determining parameter settings of the approximate methods.

Value

the nearest neighbor graph as a list containing:

- idx an n by k matrix containing the nearest neighbor indices in reference.
- dist an n by k matrix containing the nearest neighbor distances to the items in reference.

Examples

```
# 100 reference iris items
iris_ref <- iris[iris$Species %in% c("setosa", "versicolor"), ]

# 50 query items
iris_query <- iris[iris$Species == "versicolor", ]

# For each item in iris_query find the 4 nearest neighbors in iris_ref
# If you pass a data frame, non-numeric columns are removed
# set verbose = TRUE to get details on the progress being made
iris_query_nn <- brute_force_knn_query(iris_query,
  reference = iris_ref,
  k = 4, metric = "euclidean", verbose = TRUE
)

# Manhattan (l1) distance
iris_query_nn <- brute_force_knn_query(iris_query,
  reference = iris_ref,
  k = 4, metric = "manhattan"
)
```

graph_knn_query *Query a search graph for nearest neighbors*

Description

Run queries against a search graph, to return nearest neighbors taken from the reference data used to build that graph.

Usage

```
graph_knn_query(
  query,
  reference,
  reference_graph,
  k = NULL,
  metric = "euclidean",
  init = NULL,
  epsilon = 0.1,
  max_search_fraction = 1,
  use_alt_metric = TRUE,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

query	Matrix of `n` query items, with observations in the rows and features in the columns. Optionally, the data may be passed with the observations in the columns, by setting `obs = "C"`, which should be more efficient. The `reference` data must be passed in the same orientation as `query`. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse matrices should be in `dgCMatrix` format. Dataframes will be converted to `numerical` matrix format internally, so if your data columns are `logical` and intended to be used with the specialized binary `metrics`, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
reference	Matrix of `m` reference items, with observations in the rows and features in the columns. The nearest neighbors to the queries are calculated from this data. Optionally, the data may be passed with the observations in the columns, by setting `obs = "C"`, which should be more efficient. The `query` data must be passed in the same format and orientation as `reference`. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse matrices should be in `dgCMatrix` format.
reference_graph	Search graph of the `reference` data. A neighbor graph, such as that output from `nnd_knn()` can be used, but preferably a suitably prepared sparse search graph should be used, such as that output by `prepare_search_graph()`.

k — Number of nearest neighbors to return. Optional if `init` is specified.

metric — Type of distance calculation to use. One of:

- `"braycurtis"`
- `"canberra"`
- `"chebyshev"`
- `"correlation"` (1 minus the Pearson correlation)
- `"cosine"`
- `"dice"`
- `"euclidean"`
- `"hamming"`
- `"hellinger"`
- `"jaccard"`
- `"jensenshannon"`
- `"kulsinski"`
- `"sqeuclidean"` (squared Euclidean)
- `"manhattan"`
- `"rogerstanimoto"`
- `"russellrao"`
- `"sokalmichener"`
- `"sokalsneath"`
- `"spearmanr"` (1 minus the Spearman rank correlation)
- `"symmetrickl"` (symmetric Kullback-Leibler divergence)
- `"tsss"` (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- `"yule"`

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- `"cosine-preprocess"`: `cosine` with preprocessing.
- `"correlation-preprocess"`: `correlation` with preprocessing.

For non-sparse binary data passed as a `logical` matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- `"dice"`
- `"hamming"`
- `"jaccard"`
- `"kulsinski"`
- `"matching"`
- `"rogerstanimoto"`
- `"russellrao"`
- `"sokalmichener"`

- "sokalsneath"
- "yule"

`init` Initial `query` neighbor graph to optimize. If not provided, `k` random neighbors are created. If provided, the input format must be one of:

1. A list containing:
   - `idx` an `n` by `k` matrix containing the nearest neighbor indices.
   - `dist` (optional) an `n` by `k` matrix containing the nearest neighbor distances.

   If `k` and `init` are specified as arguments to this function, and the number of neighbors provided in `init` is not equal to `k` then:
   - if `k` is smaller, only the `k` closest values in `init` are retained.
   - if `k` is larger, then random neighbors will be chosen to fill `init` to the size of `k`. Note that there is no checking if any of the random neighbors are duplicates of what is already in `init` so effectively fewer than `k` neighbors may be chosen for some observations under these circumstances.

   If the input distances are omitted, they will be calculated for you.
2. A random projection forest, such as that returned from `rpf_build()` or `rpf_knn()` with `ret_forest = TRUE`.

`epsilon` Controls trade-off between accuracy and search cost, as described by Iwasaki and Miyazaki (2018), by specifying a distance tolerance on whether to explore the neighbors of candidate points. The larger the value, the more neighbors will be searched. A value of 0.1 allows query-candidate distances to be 10% larger than the current most-distant neighbor of the query point, 0.2 means 20%, and so on. Suggested values are between 0-0.5, although this value is highly dependent on the distribution of distances in the dataset (higher dimensional data should choose a smaller cutoff). Too large a value of `epsilon` will result in the query search approaching brute force comparison. Use this parameter in conjunction with `max_search_fraction` and `prepare_search_graph()` to prevent excessive run time. Default is 0.1. If you set `verbose = TRUE`, statistics of the number of distance calculations will be logged which can help you tune `epsilon`.

`max_search_fraction`
Maximum fraction of the reference data to search. This is a value between 0 (search none of the reference data) and 1 (search all of the data if necessary). This works in conjunction with `epsilon` and will terminate the search early if the specified fraction of the reference data has been searched. Default is 1.

`use_alt_metric` If `TRUE`, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using `metric = "euclidean"`), then apply a correction at the end. Probably the only reason to set this to `FALSE` is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path. If a search forest is used for initialization via the `init` parameter, then the metric is fetched from there and this setting is ignored.

`n_threads` Number of threads to use.

`verbose` If `TRUE`, log information to the console.

obs — set to "C" to indicate that the input query and reference orientation stores each observation as a column (the orientation must be consistent). The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Details

A greedy beam search is used to query the graph, combining two search pruning strategies. The first, due to Iwasaki and Miyazaki (2018), only considers new candidates within a relative distance of the current furthest neighbor in the query's graph. The second, due to Harwood and Drummond (2016), puts a limit on the absolute number of distance calculations to carry out. See the epsilon and max_search_fraction parameters respectively.

Value

the approximate nearest neighbor graph as a list containing:

- idx a n by k matrix containing the nearest neighbor indices specifying the row of the neighbor in reference.
- dist a n by k matrix containing the nearest neighbor distances.

References

Harwood, B., & Drummond, T. (2016). Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 5713-5722).

Iwasaki, M., & Miyazaki, D. (2018). Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*.

Examples

```
# 100 reference iris items
iris_ref <- iris[iris$Species %in% c("setosa", "versicolor"), ]

# 50 query items
iris_query <- iris[iris$Species == "versicolor", ]

# First, find the approximate 4-nearest neighbor graph for the references:
iris_ref_graph <- nnd_knn(iris_ref, k = 4)

# For each item in iris_query find the 4 nearest neighbors in iris_ref.
# You need to pass both the reference data and the reference graph.
# If you pass a data frame, non-numeric columns are removed.
# set verbose = TRUE to get details on the progress being made
iris_query_nn <- graph_knn_query(iris_query, iris_ref, iris_ref_graph,
  k = 4, metric = "euclidean", verbose = TRUE
)

# A more complete example, converting the initial knn into a search graph
# and using a filtered random projection forest to initialize the search
```

```
# create initial knn and forest
iris_ref_graph <- nnd_knn(iris_ref, k = 4, init = "tree", ret_forest = TRUE)
# keep the best tree in the forest
forest <- rpf_filter(iris_ref_graph, n_trees = 1)
# expand the knn into a search graph
iris_ref_search_graph <- prepare_search_graph(iris_ref, iris_ref_graph)
# run the query with the improved graph and initialization
iris_query_nn <- graph_knn_query(iris_query, iris_ref, iris_ref_search_graph,
  init = forest, k = 4
)
```

---

k_occur *Quantify hubness of a nearest neighbor graph*

Description

k_occur returns a vector of the k-occurrences of a nearest neighbor graph as defined by Radovanovic and co-workers (2010). The k-occurrence of an object is the number of times it occurs among the k-nearest neighbors of objects in a dataset.

Usage

```
k_occur(idx, k = NULL, include_self = TRUE)
```

Arguments

idx	integer matrix containing the nearest neighbor indices, integers labeled starting at 1. Note that the integer labels do *not* have to refer to the rows of idx, for example if the nearest neighbor result is from querying one set of objects with respect to another (for instance from running graph_knn_query()). You may also pass a nearest neighbor graph object (e.g. the output of running nnd_knn()), and the indices will be extracted from it, or a sparse matrix in the same format as that returned by prepare_search_graph().
k	The number of closest neighbors to use. Must be between 1 and the number of columns in idx. By default, all columns of idx are used. Ignored if idx is sparse.
include_self	logical indicating whether the label i in idx is considered to be a valid neighbor when found in row i. By default this is TRUE. This can be set to FALSE when the the labels in idx refer to the row indices of idx, as in the case of results from nnd_knn(). In this case you may not want to consider the trivial case of an object being a neighbor of itself. In all other cases leave this set to TRUE.

Details

The k-occurrence can take values between 0 and the size of the dataset. The larger the k-occurrence for an object, the more "popular" it is. Very large values of the k-occurrence (much larger than k) indicates that an object is a "hub" and also implies the existence of "anti-hubs": objects that never appear as k-nearest neighbors of other objects.

The presence of hubs can reduce the accuracy of nearest-neighbor descent and other approximate nearest neighbor algorithms in terms of retrieving the exact k-nearest neighbors. However the appearance of hubs can still be detected in these approximate results, so calculating the k-occurrences for the output of nearest neighbor descent is a useful diagnostic step.

Value

a vector of length `max(idx)`, containing the number of times an object in `idx` was found in the nearest neighbor list of the objects represented by the row indices of `idx`.

References

Radovanovic, M., Nanopoulos, A., & Ivanovic, M. (2010). Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research*, *11*, 2487-2531. https://www.jmlr.org/papers/v11/radovanovic10a.html

Bratic, B., Houle, M. E., Kurbalija, V., Oria, V., & Radovanovic, M. (2019). The Influence of Hubness on NN-Descent. *International Journal on Artificial Intelligence Tools*, *28*(06), 1960002. doi:10.1142/S0218213019600029

Examples

```
iris_nbrs <- brute_force_knn(iris, k = 15)
iris_ko <- k_occur(iris_nbrs$idx)
# items 42 and 107 are not in 15 nearest neighbors of any other members of
# iris
which(iris_ko == 1) # they are only their own nearest neighbor
max(iris_ko) # most "popular" item appears on 29 15-nearest neighbor lists
which(iris_ko == max(iris_ko)) # it's iris item 64
# with k = 15, a maximum k-occurrence = 29 ~= 1.9 * k, which is not a cause
# for concern
```

---

merge_knn *Merge multiple approximate nearest neighbors graphs*

Description

`merge_knn` takes a list of nearest neighbor graphs and merges them into a single graph, with the same number of neighbors as the first graph. This is useful to combine the results of multiple different nearest neighbor searches: the output will be at least as accurate as the most accurate of the two input graphs, and ideally will be more accurate than either.

Usage

```
merge_knn(graphs, is_query = FALSE, n_threads = 0, verbose = FALSE)
```

Arguments

graphs	A list of nearest neighbor graphs to merge. Each item in the list should consist of a sub-list containing: • idx an n by k matrix containing the k nearest neighbor indices. • dist an n by k matrix containing k nearest neighbor distances. The number of neighbors can differ between graphs, but the merged result will have the same number of neighbors as the first graph in the list.
is_query	If TRUE then the graphs are treated as the result of a knn query, not a knn building process. Or: is the graph bipartite? This should be set to TRUE if nn_graphs are the results of using e.g. graph_knn_query() or random_knn_query(), and set to FALSE if these are the results of nnd_knn() or random_knn(). The difference is that if is_query = FALSE, if an index p is found in nn_graph1[i,], i.e. p is a neighbor of i with distance d, then it is assumed that i is a neighbor of p with the same distance. If is_query = TRUE, then i and p are indexes into two different datasets and the symmetry does not hold. If you aren't sure what case applies to you, it's safe (but potentially inefficient) to set is_query = TRUE.
n_threads	Number of threads to use.
verbose	If TRUE, log information to the console.

Value

a list containing:

- idx an n by k matrix containing the merged nearest neighbor indices.
- dist an n by k matrix containing the merged nearest neighbor distances.

The size of k in the output graph is the same as that of the first item in nn_graphs.

Examples

```
set.seed(1337)
# Nearest neighbor descent with 15 neighbors for iris three times,
# starting from a different random initialization each time
iris_rnn1 <- nnd_knn(iris, k = 15, n_iters = 1)
iris_rnn2 <- nnd_knn(iris, k = 15, n_iters = 1)
iris_rnn3 <- nnd_knn(iris, k = 15, n_iters = 1)

# Merged results should be an improvement over individual results
iris_mnn <- merge_knn(list(iris_rnn1, iris_rnn2, iris_rnn3))
sum(iris_mnn$dist) < sum(iris_rnn1$dist)
sum(iris_mnn$dist) < sum(iris_rnn2$dist)
sum(iris_mnn$dist) < sum(iris_rnn3$dist)
```

neighbor_overlap *Overlap between the indices of two nearest neighbor graphs*

Description

Calculates the mean average number of neighbors in common between the two graphs. The per-item overlap can also be returned. This function can be useful as a measure of accuracy of approximation algorithms, if the exact nearest neighbors are known, or as a measure of diversity of two different approximate graphs.

Usage

```
neighbor_overlap(idx1, idx2, k = NULL, ret_vec = FALSE)
```

Arguments

idx1	Indices of a nearest neighbor graph, i.e. a matrix of nearest neighbor indices. Can also be a list containing an idx element.
idx2	Indices of a nearest neighbor graph, i.e. a matrix of nearest neighbor indices. Can also be a list containing an idx element. This is considered to be the ground truth.
k	Number of neighbors to consider. If NULL, then the minimum of the number of neighbors in idx1 and idx2 is used.
ret_vec	If TRUE, also return a vector containing the per-item overlap.

Details

The graph format is the same as that returned by e.g. nnd_knn() and should be of dimensions n by k, where n is the number of points and k is the number of neighbors. If you pass a neighbor graph directly, the index matrix will be extracted if present. If the two graphs have different numbers of neighbors, then the smaller number of neighbors is used.

Value

The mean overlap between idx1 and idx2. If ret_vec = TRUE, then a list containing the mean overlap and the overlap of each item in is returned with names mean and overlaps, respectively.

Examples

```
set.seed(1337)
# Generate two random neighbor graphs for iris
iris_rnn1 <- random_knn(iris, k = 15)
iris_rnn2 <- random_knn(iris, k = 15)

# Overlap between the two graphs
mean_overlap <- neighbor_overlap(iris_rnn1, iris_rnn2)
```

```
# Also get a vector of per-item overlap
overlap_res <- neighbor_overlap(iris_rnn1, iris_rnn2, ret_vec = TRUE)
summary(overlap_res$overlaps)
```

---

nnd_knn *Find nearest neighbors using nearest neighbor descent*

Description

Uses the Nearest Neighbor Descent method due to Dong and co-workers (2011) to optimize an approximate nearest neighbor graph.

Usage

```
nnd_knn(
  data,
  k = NULL,
  metric = "euclidean",
  init = "rand",
  init_args = NULL,
  n_iters = NULL,
  max_candidates = NULL,
  delta = 0.001,
  low_memory = TRUE,
  weight_by_degree = FALSE,
  use_alt_metric = TRUE,
  n_threads = 0,
  verbose = FALSE,
  progress = "bar",
  obs = "R",
  ret_forest = FALSE
)
```

Arguments

data	Matrix of n items to generate neighbors for, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting obs = "C", which should be more efficient. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
k	Number of nearest neighbors to return. Optional if init is specified.
metric	Type of distance calculation to use. One of: • "braycurtis"

- "canberra"
- "chebyshev"
- "correlation" (1 minus the Pearson correlation)
- "cosine"
- "dice"
- "euclidean"
- "hamming"
- "hellinger"
- "jaccard"
- "jensenshannon"
- "kulsinski"
- "sqeuclidean" (squared Euclidean)
- "manhattan"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

init — Name of the initialization strategy or initial data neighbor graph to optimize. One of:

- "rand" random initialization (the default).
- "tree" use the random projection tree method of Dasgupta and Freund (2008).
- a pre-calculated neighbor graph. A list containing:
  - idx an n by k matrix containing the nearest neighbor indices.
  - dist (optional) an n by k matrix containing the nearest neighbor distances. If the input distances are omitted, they will be calculated for you.'

If k and init are specified as arguments to this function, and the number of neighbors provided in init is not equal to k then:

- if k is smaller, only the k closest values in init are retained.
- if k is larger, then random neighbors will be chosen to fill init to the size of k. Note that there is no checking if any of the random neighbors are duplicates of what is already in init so effectively fewer than k neighbors may be chosen for some observations under these circumstances.

init_args a list containing arguments to pass to the random partition forest initialization. See rpf_knn() for possible arguments. To avoid inconsistences with the tree calculation and subsequent nearest neighbor descent optimization, if you attempt to provide a metric or use_alt_metric option in this list it will be ignored.

n_iters Number of iterations of nearest neighbor descent to carry out. By default, this will be chosen based on the number of observations in data.

max_candidates Maximum number of candidate neighbors to try for each item in each iteration. Use relative to k to emulate the "rho" sampling parameter in the nearest neighbor descent paper. By default, this is set to k or 60, whichever is smaller.

delta The minimum relative change in the neighbor graph allowed before early stopping. Should be a value between 0 and 1. The smaller the value, the smaller the amount of progress between iterations is allowed. Default value of 0.001 means that at least 0.1% of the neighbor graph must be updated at each iteration.

low_memory If TRUE, use a lower memory, but more computationally expensive approach to index construction. If set to FALSE, you should see a noticeable speed improvement, especially when using a smaller number of threads, so this is worth trying if you have the memory to spare.

weight_by_degree

If TRUE, then candidates for the local join are weighted according to their in-degree, so that if there are more than max_candidates in a candidate list, candidates with a smaller degree are favored for retention. This prevents items with large numbers of edges crowding out other items and for high-dimensional data is likely to provide a small improvement in accuracy. Because this incurs a small extra cost of counting the degree of each node, and because it tends to delay early convergence, by default this is FALSE.

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

n_threads	Number of threads to use.
verbose	If TRUE, log information to the console.
progress	Determines the type of progress information logged if verbose = TRUE. Options are: • "bar": a simple text progress bar. • "dist": the sum of the distances in the approximate knn graph at the end of each iteration.
obs	set to "C" to indicate that the input data orientation stores each observation as a column. The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.
ret_forest	If TRUE and init = "tree" then the RP forest used to initialize the nearest neighbors will be returned with the nearest neighbor data. See the Value section for details. The returned forest can be used as part of initializing the search for new data: see rpf_knn_query() and rpf_filter() for more details.

Details

If no initial graph is provided, a random graph is generated, or you may also specify the use of a graph generated from a forest of random projection trees, using the method of Dasgupta and Freund (2008).

Value

the approximate nearest neighbor graph as a list containing:

- idx an n by k matrix containing the nearest neighbor indices.
- dist an n by k matrix containing the nearest neighbor distances.
- forest (if init = "tree" and ret_forest = TRUE only): the RP forest used to initialize the neighbor data.

References

Dasgupta, S., & Freund, Y. (2008, May). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing* (pp. 537-546). doi:10.1145/1374376.1374452.

Dong, W., Moses, C., & Li, K. (2011, March). Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World Wide Web* (pp. 577-586). ACM. doi:10.1145/1963405.1963487.

Examples

```
# Find 4 (approximate) nearest neighbors using Euclidean distance
# If you pass a data frame, non-numeric columns are removed
iris_nn <- nnd_knn(iris, k = 4, metric = "euclidean")
```

```
# Manhattan (l1) distance
iris_nn <- nnd_knn(iris, k = 4, metric = "manhattan")

# Multi-threading: you can choose the number of threads to use: in real
# usage, you will want to set n_threads to at least 2
iris_nn <- nnd_knn(iris, k = 4, metric = "manhattan", n_threads = 1)

# Use verbose flag to see information about progress
iris_nn <- nnd_knn(iris, k = 4, metric = "euclidean", verbose = TRUE)

# Nearest neighbor descent uses random initialization, but you can pass any
# approximation using the init argument (as long as the metrics used to
# calculate the initialization are compatible with the metric options used
# by nnd_knn).
iris_nn <- random_knn(iris, k = 4, metric = "euclidean")
iris_nn <- nnd_knn(iris, init = iris_nn, metric = "euclidean", verbose = TRUE)

# Number of iterations controls how much optimization is attempted. A smaller
# value will run faster but give poorer results
iris_nn <- nnd_knn(iris, k = 4, metric = "euclidean", n_iters = 2)

# You can also control the amount of work done within an iteration by
# setting max_candidates
iris_nn <- nnd_knn(iris, k = 4, metric = "euclidean", max_candidates = 50)

# Optimization may also stop early if not much progress is being made. This
# convergence criterion can be controlled via delta. A larger value will
# stop progress earlier. The verbose flag will provide some information if
# convergence is occurring before all iterations are carried out.
set.seed(1337)
iris_nn <- nnd_knn(iris, k = 4, metric = "euclidean", n_iters = 5, delta = 0.5)

# To ensure that descent only stops if no improvements are made, set delta = 0
set.seed(1337)
iris_nn <- nnd_knn(iris, k = 4, metric = "euclidean", n_iters = 5, delta = 0)

# A faster version of the algorithm is available that avoids repeated
# distance calculations at the cost of using more RAM. Set low_memory to
# FALSE to try it.
set.seed(1337)
iris_nn <- nnd_knn(iris, k = 4, metric = "euclidean", low_memory = FALSE)

# Using init = "tree" is usually more efficient than random initialization.
# arguments to the tree initialization method can be passed via the init_args
# list
set.seed(1337)
iris_nn <- nnd_knn(iris, k = 4, init = "tree", init_args = list(n_trees = 5))
```

---

prepare_search_graph *Convert a nearest neighbor graph into a search graph*

Description

Create a graph using existing nearest neighbor data to balance search speed and accuracy using the occlusion pruning and truncation strategies of Harwood and Drummond (2016). The resulting search graph should be more efficient for querying new data than the original nearest neighbor graph.

Usage

```
prepare_search_graph(
  data,
  graph,
  metric = "euclidean",
  use_alt_metric = TRUE,
  diversify_prob = 1,
  pruning_degree_multiplier = 1.5,
  prune_reverse = FALSE,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

data	Matrix of n items, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting obs = "C", which should be more efficient. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
graph	neighbor graph for data, a list containing: • idx an n by k matrix containing the nearest neighbor indices of the data in data. • dist an n by k matrix containing the nearest neighbor distances.
metric	Type of distance calculation to use. One of: • "braycurtis" • "canberra" • "chebyshev" • "correlation" (1 minus the Pearson correlation) • "cosine" • "dice" • "euclidean" • "hamming" • "hellinger" • "jaccard"

- "jensenshannon"
- "kulsinski"
- "sqeuclidean" (squared Euclidean)
- "manhattan"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

diversify_prob the degree of diversification of the search graph by removing unnecessary edges through occlusion pruning. This should take a value between 0 (no diversification) and 1 (remove as many edges as possible) and is treated as the probability of a neighbor being removed if it is found to be an "occlusion". If item p and q, two members of the neighbor list of item i, are closer to each other than they are to i, then the nearer neighbor p is said to "occlude" q. It is likely that q will

be in the neighbor list of p so there is no need to retain it in the neighbor list of i. You may also set this to NULL to skip any occlusion pruning. Note that occlusion pruning is carried out twice, once to the forward neighbors, and once to the reverse neighbors. Reducing this value will result in a more dense graph. This is similar to increasing the "alpha" parameter used by in the DiskAnn pruning method of Subramanya and co-workers (2014).

pruning_degree_multiplier
: How strongly to truncate the final neighbor list for each item. The neighbor list of each item will be truncated to retain only the closest d neighbors, where d = k * pruning_degree_multiplier, and k is the original number of neighbors per item in graph. Roughly, values larger than 1 will keep all the nearest neighbors of an item, plus the given fraction of reverse neighbors (if they exist). For example, setting this to 1.5 will keep all the forward neighbors and then half as many of the reverse neighbors, although exactly which neighbors are retained is also dependent on any occlusion pruning that occurs. Set this to NULL to skip this step.

prune_reverse
: If TRUE, prune the reverse neighbors of each item before the reverse graph diversification step using pruning_degree_multiplier. Because the number of reverse neighbors can be much larger than the number of forward neighbors, this can help to avoid excessive computation during the diversification step, with little overall effect on the final search graph. Default is FALSE.

n_threads
: Number of threads to use.

verbose
: If TRUE, log information to the console.

obs
: set to "C" to indicate that the input data orientation stores each observation as a column. The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Details

An approximate nearest neighbor graph is not very useful for querying via graph_knn_query(), especially if the query data is initialized randomly: some items in the data set may not be in the nearest neighbor list of any other item and can therefore never be returned as a neighbor, no matter how close they are to the query. Even those which do appear in at least one neighbor list may not be reachable by expanding an arbitrary starting list if the neighbor graph contains disconnected components.

Converting the directed graph represented by the neighbor graph to an undirected graph by adding an edge from item j to i if an edge exists from i to j (i.e. creating the mutual neighbor graph) solves the problems above, but can result in inefficient searches. Although the out-degree of each item is restricted to the number of neighbors the in-degree has no such restrictions: a given item could be very "popular" and in a large number of neighbors lists. Therefore mutualizing the neighbor graph can result in some items with a large number of neighbors to search. These usually have very similar neighborhoods so there is nothing to be gained from searching all of them.

To balance accuracy and search time, the following procedure is carried out:

1. The graph is "diversified" by occlusion pruning.

2. The reverse graph is formed by reversing the direction of all edges in the pruned graph.
3. The reverse graph is diversified by occlusion pruning.
4. The pruned forward and pruned reverse graph are merged.
5. The outdegree of each node in the merged graph is truncated.
6. The truncated merged graph is returned as the prepared search graph.

Explicit zero distances in the `graph` will be converted to a small positive number to avoid being dropped in the sparse representation. The one exception is the "self" distance, i.e. any edge in the `graph` which links a node to itself (the diagonal of the sparse distance matrix). These trivial edges aren't useful for search purposes and are always dropped.

Value

a search graph for `data` based on `graph`, represented as a sparse matrix, suitable for use with `graph_knn_query()`.

References

Harwood, B., & Drummond, T. (2016). Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 5713-5722).

Jayaram Subramanya, S., Devvrit, F., Simhadri, H. V., Krishnawamy, R., & Kadekodi, R. (2019). Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, *32*.

See Also

`graph_knn_query()`

Examples

```
# 100 reference iris items
iris_ref <- iris[iris$Species %in% c("setosa", "versicolor"), ]

# 50 query items
iris_query <- iris[iris$Species == "versicolor", ]

# First, find the approximate 4-nearest neighbor graph for the references:
ref_ann_graph <- nnd_knn(iris_ref, k = 4)

# Create a graph for querying with
ref_search_graph <- prepare_search_graph(iris_ref, ref_ann_graph)

# Using the search graph rather than the ref_ann_graph directly may give
# more accurate or faster results
iris_query_nn <- graph_knn_query(
  query = iris_query, reference = iris_ref,
  reference_graph = ref_search_graph, k = 4, metric = "euclidean",
  verbose = TRUE
)
```

random_knn *Find nearest neighbors by random selection*

Description

Create a neighbor graph by randomly selecting neighbors. This is not a useful nearest neighbor method on its own, but can be used with other methods which require initialization, such as `nnd_knn()`.

Usage

```
random_knn(
  data,
  k,
  metric = "euclidean",
  use_alt_metric = TRUE,
  order_by_distance = TRUE,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

`data`	Matrix of `n` items to generate random neighbors for, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting `obs = "C"`, which should be more efficient. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse matrices should be in `dgCMatrix` format. Dataframes will be converted to `numerical` matrix format internally, so if your data columns are `logical` and intended to be used with the specialized binary `metrics`, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
`k`	Number of nearest neighbors to return.
`metric`	Type of distance calculation to use. One of: • `"braycurtis"` • `"canberra"` • `"chebyshev"` • `"correlation"` (1 minus the Pearson correlation) • `"cosine"` • `"dice"` • `"euclidean"` • `"hamming"` • `"hellinger"` • `"jaccard"`

- "jensenshannon"
- "kulsinski"
- "sqeuclidean" (squared Euclidean)
- "manhattan"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

order_by_distance

If TRUE (the default), then results for each item are returned by increasing distance. If you don't need the results sorted, e.g. you are going to pass the results as initialization to another routine like nnd_knn(), set this to FALSE to save a small amount of computational time.

n_threads Number of threads to use.

verbose — If TRUE, log information to the console.

obs — set to "C" to indicate that the input data orientation stores each observation as a column. The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Value

a random neighbor graph as a list containing:

- idx an n by k matrix containing the nearest neighbor indices.
- dist an n by k matrix containing the nearest neighbor distances.

Examples

```
# Find 4 random neighbors and calculate their Euclidean distance
# If you pass a data frame, non-numeric columns are removed
iris_nn <- random_knn(iris, k = 4, metric = "euclidean")

# Manhattan (l1) distance
iris_nn <- random_knn(iris, k = 4, metric = "manhattan")

# Multi-threading: you can choose the number of threads to use: in real
# usage, you will want to set n_threads to at least 2
iris_nn <- random_knn(iris, k = 4, metric = "manhattan", n_threads = 1)

# Use verbose flag to see information about progress
iris_nn <- random_knn(iris, k = 4, metric = "euclidean", verbose = TRUE)

# These results can be improved by nearest neighbors descent. You don't need
# to specify k here because this is worked out from the initial input
iris_nn <- nnd_knn(iris, init = iris_nn, metric = "euclidean", verbose = TRUE)
```

---

random_knn_query *Query nearest neighbors by random selection*

Description

Run queries against reference data to return randomly selected neighbors. This is not a useful query method on its own, but can be used with other methods which require initialization.

Usage

```
random_knn_query(
  query,
  reference,
  k,
```

```
  metric = "euclidean",
  use_alt_metric = TRUE,
  order_by_distance = TRUE,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

query	Matrix of `n` query items, with observations in the rows and features in the columns. Optionally, the data may be passed with the observations in the columns, by setting `obs = "C"`, which should be more efficient. The `reference` data must be passed in the same orientation as `query`. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse matrices should be in `dgCMatrix` format. Dataframes will be converted to `numerical` matrix format internally, so if your data columns are `logical` and intended to be used with the specialized binary `metrics`, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
reference	Matrix of `m` reference items, with observations in the rows and features in the columns. The nearest neighbors to the queries are randomly selected from this data. Optionally, the data may be passed with the observations in the columns, by setting `obs = "C"`, which should be more efficient. The `query` data must be passed in the same orientation and format as `reference`. Possible formats are `base::data.frame()`, `base::matrix()` or `Matrix::sparseMatrix()`. Sparse matrices should be in `dgCMatrix` format.
k	Number of nearest neighbors to return.
metric	Type of distance calculation to use. One of: • `"braycurtis"` • `"canberra"` • `"chebyshev"` • `"correlation"` (1 minus the Pearson correlation) • `"cosine"` • `"dice"` • `"euclidean"` • `"hamming"` • `"hellinger"` • `"jaccard"` • `"jensenshannon"` • `"kulsinski"` • `"sqeuclidean"` (squared Euclidean) • `"manhattan"` • `"rogerstanimoto"` • `"russellrao"` • `"sokalmichener"`

- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

order_by_distance

If TRUE (the default), then results for each item are returned by increasing distance. If you don't need the results sorted, e.g. you are going to pass the results as initialization to another routine like graph_knn_query(), set this to FALSE to save a small amount of computational time.

n_threads Number of threads to use.

verbose If TRUE, log information to the console.

obs set to "C" to indicate that the input query and reference orientation stores each observation as a column (the orientation must be consistent). The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Value

an approximate nearest neighbor graph as a list containing:

- `idx` an n by k matrix containing the nearest neighbor indices.
- `dist` an n by k matrix containing the nearest neighbor distances.

Examples

```
# 100 reference iris items
iris_ref <- iris[iris$Species %in% c("setosa", "versicolor"), ]

# 50 query items
iris_query <- iris[iris$Species == "versicolor", ]

# For each item in iris_query find 4 random neighbors in iris_ref
# If you pass a data frame, non-numeric columns are removed
# set verbose = TRUE to get details on the progress being made
iris_query_random_nbrs <- random_knn_query(iris_query,
  reference = iris_ref,
  k = 4, metric = "euclidean", verbose = TRUE
)

# Manhattan (l1) distance
iris_query_random_nbrs <- random_knn_query(iris_query,
  reference = iris_ref,
  k = 4, metric = "manhattan"
)
```

---

rnnd_build *Build approximate nearest neighbors index and neighbor graph*

Description

This function builds an approximate nearest neighbors graph with convenient defaults, then prepares the index for querying new data, for later use with `rnnd_query()`. For more control over the process, please see the other functions in the package.

Usage

```
rnnd_build(
  data,
  k = 30,
  metric = "euclidean",
  use_alt_metric = TRUE,
  init = "tree",
  n_trees = NULL,
  leaf_size = NULL,
  max_tree_depth = 200,
```

```
  margin = "auto",
  n_iters = NULL,
  delta = 0.001,
  max_candidates = NULL,
  low_memory = TRUE,
  weight_by_degree = FALSE,
  n_search_trees = 1,
  pruning_degree_multiplier = 1.5,
  diversify_prob = 1,
  prune_reverse = FALSE,
  n_threads = 0,
  verbose = FALSE,
  progress = "bar",
  obs = "R"
)
```

Arguments

data	Matrix of n items to generate neighbors for, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting obs = "C", which should be more efficient. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
k	Number of nearest neighbors to build the index for. You can specify a different number when running rnnd_query, but the index is calibrated using this value so it's recommended to set k according to the likely number of neighbors you will want to retrieve. Optional if init is specified, in which case k can be inferred from the init data. If you do both, then the specified version of k will take precedence.
metric	Type of distance calculation to use. One of: • "braycurtis" • "canberra" • "chebyshev" • "correlation" (1 minus the Pearson correlation) • "cosine" • "dice" • "euclidean" • "hamming" • "hellinger" • "jaccard" • "jensenshannon" • "kulsinski"

- "sqeuclidean" (squared Euclidean)
- "manhattan"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

init Name of the initialization strategy or initial data neighbor graph to optimize. One of:

- "rand" random initialization (the default).
- "tree" use the random projection tree method of Dasgupta and Freund (2008).
- a pre-calculated neighbor graph. A list containing:
  - idx an n by k matrix containing the nearest neighbor indices.

- dist (optional) an n by k matrix containing the nearest neighbor distances. If the input distances are omitted, they will be calculated for you.'

If k and init are specified as arguments to this function, and the number of neighbors provided in init is not equal to k then:

- if k is smaller, only the k closest values in init are retained.
- if k is larger, then random neighbors will be chosen to fill init to the size of k. Note that there is no checking if any of the random neighbors are duplicates of what is already in init so effectively fewer than k neighbors may be chosen for some observations under these circumstances.

n_trees — The number of trees to use in the RP forest. A larger number will give more accurate results at the cost of a longer computation time. The default of NULL means that the number is chosen based on the number of observations in data. Only used if init = "tree".

leaf_size — The maximum number of items that can appear in a leaf. This value should be chosen to match the expected number of neighbors you will want to retrieve when running queries (e.g. if you want find 50 nearest neighbors set leaf_size = 50) and should not be set to a value smaller than 10. Only used if init = "tree".

max_tree_depth — The maximum depth of the tree to build (default = 200). If the maximum tree depth is exceeded then the leaf size of a tree may exceed leaf_size which can result in a large number of neighbor distances being calculated. If verbose = TRUE a message will be logged to indicate that the leaf size is large. However, increasing the max_tree_depth may not help: it may be that there is something unusual about the distribution of your data set under your chose metric that makes a tree-based initialization inappropriate. Only used if init = "tree".

margin — A character string specifying the method used to assign points to one side of the hyperplane or the other. Possible values are:

- "explicit" categorizes all distance metrics as either Euclidean or Angular (Euclidean after normalization), explicitly calculates a hyperplane and offset, and then calculates the margin based on the dot product with the hyperplane.
- "implicit" calculates the distance from a point to each of the points defining the normal vector. The margin is calculated by comparing the two distances: the point is assigned to the side of the hyperplane that the normal vector point with the closest distance belongs to.
- "auto" (the default) picks the margin method depending on whether a binary-specific metric such as "bhammming" is chosen, in which case "implicit" is used, and "explicit" otherwise: binary-specific metrics involve storing the data in a way that isn't very efficient for the "explicit" method and the binary-specific metric is usually a lot faster than the generic equivalent such that the cost of two distance calculations for the margin method is still faster.

Only used if init = "tree".

n_iters — Number of iterations of nearest neighbor descent to carry out. By default, this will be chosen based on the number of observations in data.

delta
: The minimum relative change in the neighbor graph allowed before early stopping. Should be a value between 0 and 1. The smaller the value, the smaller the amount of progress between iterations is allowed. Default value of `0.001` means that at least 0.1% of the neighbor graph must be updated at each iteration.

max_candidates
: Maximum number of candidate neighbors to try for each item in each iteration. Use relative to `k` to emulate the "rho" sampling parameter in the nearest neighbor descent paper. By default, this is set to `k` or `60`, whichever is smaller.

low_memory
: If `TRUE`, use a lower memory, but more computationally expensive approach to index construction. If set to `FALSE`, you should see a noticeable speed improvement, especially when using a smaller number of threads, so this is worth trying if you have the memory to spare.

weight_by_degree
: If `TRUE`, then candidates for the local join are weighted according to their in-degree, so that if there are more than `max_candidates` in a candidate list, candidates with a smaller degree are favored for retention. This prevents items with large numbers of edges crowding out other items and for high-dimensional data is likely to provide a small improvement in accuracy. Because this incurs a small extra cost of counting the degree of each node, and because it tends to delay early convergence, by default this is `FALSE`.

n_search_trees,
: the number of trees to keep in the search forest as part of index preparation. The default is `1`.

pruning_degree_multiplier
: How strongly to truncate the final neighbor list for each item. The neighbor list of each item will be truncated to retain only the closest `d` neighbors, where `d = k * pruning_degree_multiplier`, and `k` is the original number of neighbors per item in `graph`. Roughly, values larger than `1` will keep all the nearest neighbors of an item, plus the given fraction of reverse neighbors (if they exist). For example, setting this to `1.5` will keep all the forward neighbors and then half as many of the reverse neighbors, although exactly which neighbors are retained is also dependent on any occlusion pruning that occurs. Set this to `NULL` to skip this step.

diversify_prob
: the degree of diversification of the search graph by removing unnecessary edges through occlusion pruning. This should take a value between `0` (no diversification) and `1` (remove as many edges as possible) and is treated as the probability of a neighbor being removed if it is found to be an "occlusion". If item `p` and `q`, two members of the neighbor list of item `i`, are closer to each other than they are to `i`, then the nearer neighbor `p` is said to "occlude" `q`. It is likely that `q` will be in the neighbor list of `p` so there is no need to retain it in the neighbor list of `i`. You may also set this to `NULL` to skip any occlusion pruning. Note that occlusion pruning is carried out twice, once to the forward neighbors, and once to the reverse neighbors.

prune_reverse
: If `TRUE`, prune the reverse neighbors of each item before the reverse graph diversification step using `pruning_degree_multiplier`. Because the number of reverse neighbors can be much larger than the number of forward neighbors, this can help to avoid excessive computation during the diversification step, with little overall effect on the final search graph. Default is `FALSE`.

`n_threads` Number of threads to use.

`verbose` If `TRUE`, log information to the console.

`progress` Determines the type of progress information logged during the nearest neighbor descent stage when `verbose = TRUE`. Options are:

- `"bar"`: a simple text progress bar.
- `"dist"`: the sum of the distances in the approximate knn graph at the end of each iteration.

`obs` set to `"C"` to indicate that the input `data` orientation stores each observation as a column. The default `"R"` means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Details

The process of k-nearest neighbor graph construction using Random Projection Forests (Dasgupta and Freund, 2008) for initialization and Nearest Neighbor Descent (Dong and co-workers, 2011) for refinement. Index preparation, uses the graph diversification method of Harwood and Drummond (2016).

Value

the approximate nearest neighbor index, a list containing:

- `graph` the k-nearest neighbor graph, a list containing:
  - `idx` an n by k matrix containing the nearest neighbor indices.
  - `dist` an n by k matrix containing the nearest neighbor distances.
- Other list items are intended only for internal use by other functions such as `rnnd_query()`.

References

Dasgupta, S., & Freund, Y. (2008, May). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing* (pp. 537-546). doi:10.1145/1374376.1374452.

Dong, W., Moses, C., & Li, K. (2011, March). Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World Wide Web* (pp. 577-586). ACM. doi:10.1145/1963405.1963487.

Harwood, B., & Drummond, T. (2016). Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 5713-5722).

See Also

`rnnd_query()`

Examples

```
iris_even <- iris[seq_len(nrow(iris)) %% 2 == 0, ]
iris_odd <- iris[seq_len(nrow(iris)) %% 2 == 1, ]

# Find 4 (approximate) nearest neighbors using Euclidean distance
iris_even_index <- rnnd_build(iris_even, k = 4)
iris_odd_nbrs <- rnnd_query(index = iris_even_index, query = iris_odd, k = 4)
```

---

rnnd_knn *Find approximate nearest neighbors*

Description

This function builds an approximate nearest neighbors graph of the provided data using convenient defaults. It does not return an index for later querying, to speed the graph construction and reduce the size and complexity of the return value.

Usage

```
rnnd_knn(
  data,
  k = 30,
  metric = "euclidean",
  use_alt_metric = TRUE,
  init = "tree",
  n_trees = NULL,
  leaf_size = NULL,
  max_tree_depth = 200,
  margin = "auto",
  n_iters = NULL,
  delta = 0.001,
  max_candidates = NULL,
  weight_by_degree = FALSE,
  low_memory = TRUE,
  n_threads = 0,
  verbose = FALSE,
  progress = "bar",
  obs = "R"
)
```

Arguments

data	Matrix of n items to generate neighbors for, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting obs = "C", which should be more efficient. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse

matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).

k — Number of nearest neighbors to return. Optional if init is specified.

metric — Type of distance calculation to use. One of:

- "braycurtis"
- "canberra"
- "chebyshev"
- "correlation" (1 minus the Pearson correlation)
- "cosine"
- "dice"
- "euclidean"
- "hamming"
- "hellinger"
- "jaccard"
- "jensenshannon"
- "kulsinski"
- "sqeuclidean" (squared Euclidean)
- "manhattan"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"

- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

init Name of the initialization strategy or initial data neighbor graph to optimize. One of:

- "rand" random initialization (the default).
- "tree" use the random projection tree method of Dasgupta and Freund (2008).
- a pre-calculated neighbor graph. A list containing:
  - idx an n by k matrix containing the nearest neighbor indices.
  - dist (optional) an n by k matrix containing the nearest neighbor distances. If the input distances are omitted, they will be calculated for you.'

If k and init are specified as arguments to this function, and the number of neighbors provided in init is not equal to k then:

- if k is smaller, only the k closest values in init are retained.
- if k is larger, then random neighbors will be chosen to fill init to the size of k. Note that there is no checking if any of the random neighbors are duplicates of what is already in init so effectively fewer than k neighbors may be chosen for some observations under these circumstances.

n_trees The number of trees to use in the RP forest. A larger number will give more accurate results at the cost of a longer computation time. The default of NULL means that the number is chosen based on the number of observations in data. Only used if init = "tree".

leaf_size The maximum number of items that can appear in a leaf. This value should be chosen to match the expected number of neighbors you will want to retrieve when running queries (e.g. if you want find 50 nearest neighbors set leaf_size = 50) and should not be set to a value smaller than 10. Only used if init = "tree".

max_tree_depth The maximum depth of the tree to build (default = 200). If the maximum tree depth is exceeded then the leaf size of a tree may exceed leaf_size which can result in a large number of neighbor distances being calculated. If verbose = TRUE a message will be logged to indicate that the leaf size is large. However, increasing the max_tree_depth may not help: it may be that there is something unusual about the distribution of your data set under your chose metric that makes a tree-based initialization inappropriate. Only used if init = "tree".

margin
: A character string specifying the method used to assign points to one side of the hyperplane or the other. Possible values are:

  - "explicit" categorizes all distance metrics as either Euclidean or Angular (Euclidean after normalization), explicitly calculates a hyperplane and offset, and then calculates the margin based on the dot product with the hyperplane.
  - "implicit" calculates the distance from a point to each of the points defining the normal vector. The margin is calculated by comparing the two distances: the point is assigned to the side of the hyperplane that the normal vector point with the closest distance belongs to.
  - "auto" (the default) picks the margin method depending on whether a binary-specific metric such as "bhammming" is chosen, in which case "implicit" is used, and "explicit" otherwise: binary-specific metrics involve storing the data in a way that isn't very efficient for the "explicit" method and the binary-specific metric is usually a lot faster than the generic equivalent such that the cost of two distance calculations for the margin method is still faster.

  Only used if init = "tree".

n_iters
: Number of iterations of nearest neighbor descent to carry out. By default, this will be chosen based on the number of observations in data.

delta
: The minimum relative change in the neighbor graph allowed before early stopping. Should be a value between 0 and 1. The smaller the value, the smaller the amount of progress between iterations is allowed. Default value of 0.001 means that at least 0.1% of the neighbor graph must be updated at each iteration.

max_candidates
: Maximum number of candidate neighbors to try for each item in each iteration. Use relative to k to emulate the "rho" sampling parameter in the nearest neighbor descent paper. By default, this is set to k or 60, whichever is smaller.

weight_by_degree
: If TRUE, then candidates for the local join are weighted according to their in-degree, so that if there are more than max_candidates in a candidate list, candidates with a smaller degree are favored for retention. This prevents items with large numbers of edges crowding out other items and for high-dimensional data is likely to provide a small improvement in accuracy. Because this incurs a small extra cost of counting the degree of each node, and because it tends to delay early convergence, by default this is FALSE.

low_memory
: If TRUE, use a lower memory, but more computationally expensive approach to index construction. If set to FALSE, you should see a noticeable speed improvement, especially when using a smaller number of threads, so this is worth trying if you have the memory to spare.

n_threads
: Number of threads to use.

verbose
: If TRUE, log information to the console.

progress
: Determines the type of progress information logged during the nearest neighbor descent stage when verbose = TRUE. Options are:

  - "bar": a simple text progress bar.

- "dist": the sum of the distances in the approximate knn graph at the end of each iteration.

obs — set to "C" to indicate that the input data orientation stores each observation as a column. The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Details

The process of k-nearest neighbor graph construction using Random Projection Forests (Dasgupta and Freund, 2008) for initialization and Nearest Neighbor Descent (Dong and co-workers, 2011) for refinement. If you are sure you will not want to query new data then compared to rnnd_build() this function has the advantage of not storing the index, which can be very large.

Value

the approximate nearest neighbor index, a list containing:

- idx an n by k matrix containing the nearest neighbor indices.
- dist an n by k matrix containing the nearest neighbor distances.

References

Dasgupta, S., & Freund, Y. (2008, May). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing* (pp. 537-546). doi:10.1145/1374376.1374452.

Dong, W., Moses, C., & Li, K. (2011, March). Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World Wide Web* (pp. 577-586). ACM. doi:10.1145/1963405.1963487.

See Also

rnnd_build(), rnnd_query()

Examples

```
# Find 4 (approximate) nearest neighbors using Euclidean distance
iris_knn <- rnnd_knn(iris, k = 4)
```

rnnd_query *Query an index for approximate nearest neighbors*

Description

Takes a nearest neighbor index produced by rnnd_build() and uses it to find the nearest neighbors of a query set of observations, using a back-tracking search with the search size determined by the method of Iwasaki and Miyazaki (2018). For further control over the search effort, the total number of distance calculations can also be bounded, similar to the method of Harwood and Drummond (2016).

Usage

```
rnnd_query(
  index,
  query,
  k = 30,
  epsilon = 0.1,
  max_search_fraction = 1,
  init = NULL,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

index	A nearest neighbor index produced by rnnd_build().
query	Matrix of n query items, with observations in the rows and features in the columns. Optionally, the data may be passed with the observations in the columns, by setting obs = "C", which should be more efficient. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version). Sparse and non-sparse data cannot be mixed, so if the data used to build index was sparse, the query data must also be sparse. and vice versa.
k	Number of nearest neighbors to return.
epsilon	Controls trade-off between accuracy and search cost, as described by Iwasaki and Miyazaki (2018). Setting epsilon to a positive value specifies a distance tolerance on whether to explore the neighbors of candidate points. The larger the value, the more neighbors will be searched. A value of 0.1 allows query-candidate distances to be 10% larger than the current most-distant neighbor of the query point, 0.2 means 20%, and so on. Suggested values are between 0-0.5, although this value is highly dependent on the distribution of distances in the

dataset (higher dimensional data should choose a smaller cutoff). Too large a value of `epsilon` will result in the query search approaching brute force comparison. Use this parameter in conjunction with `max_search_fraction` to prevent excessive run time. Default is 0.1. If you set `verbose = TRUE`, statistics of the number of distance calculations will be logged which can help you tune `epsilon`.

`max_search_fraction`
: Maximum fraction of the reference data to search. This is a value between 0 (search none of the reference data) and 1 (search all of the data if necessary). This works in conjunction with `epsilon` and will terminate the search early if the specified fraction of the reference data has been searched. Default is 1.

`init`
: An optional matrix of `k` initial nearest neighbors for each query point.

`n_threads`
: Number of threads to use.

`verbose`
: If `TRUE`, log information to the console.

`obs`
: set to `"C"` to indicate that the input `data` orientation stores each observation as a column. The default `"R"` means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Value

the approximate nearest neighbor index, a list containing:

- `idx` an n by k matrix containing the nearest neighbor indices.
- `dist` an n by k matrix containing the nearest neighbor distances.

References

Harwood, B., & Drummond, T. (2016). Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 5713-5722).

Iwasaki, M., & Miyazaki, D. (2018). Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*. https://arxiv.org/abs/1810.07355

See Also

`rnnd_query()`

Examples

```
iris_even <- iris[seq_len(nrow(iris)) %% 2 == 0, ]
iris_odd <- iris[seq_len(nrow(iris)) %% 2 == 1, ]

iris_even_index <- rnnd_build(iris_even, k = 4)
iris_odd_nbrs <- rnnd_query(index = iris_even_index, query = iris_odd, k = 4)
```

rpf_build *Create a random projection forest nearest neighbor index*

Description

Builds a "forest" of Random Projection Trees (Dasgupta and Freund, 2008), which can later be searched to find approximate nearest neighbors.

Usage

```
rpf_build(
  data,
  metric = "euclidean",
  use_alt_metric = TRUE,
  n_trees = NULL,
  leaf_size = 10,
  max_tree_depth = 200,
  margin = "auto",
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

data	Matrix of n items to generate the index for, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting obs = "C", which should be more efficient. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
metric	Type of distance calculation to use. One of: • "braycurtis" • "canberra" • "chebyshev" • "correlation" (1 minus the Pearson correlation) • "cosine" • "dice" • "euclidean" • "hamming" • "hellinger" • "jaccard"

- "jensenshannon"
- "kulsinski"
- "sqeuclidean" (squared Euclidean)
- "manhattan"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

Note that if margin = "explicit", the metric is only used to determine whether an "angular" or "Euclidean" distance is used to measure the distance between split points in the tree.

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"). Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path. Only applies if the implicit margin method is used.

n_trees The number of trees to use in the RP forest. A larger number will give more accurate results at the cost of a longer computation time. The default of NULL means that the number is chosen based on the number of observations in data.

`leaf_size` The maximum number of items that can appear in a leaf. This value should be chosen to match the expected number of neighbors you will want to retrieve when running queries (e.g. if you want find 50 nearest neighbors set `leaf_size = 50`) and should not be set to a value smaller than `10`.

`max_tree_depth` The maximum depth of the tree to build (default = 200). If the maximum tree depth is exceeded then the leaf size of a tree may exceed `leaf_size` which can result in a large number of neighbor distances being calculated. If `verbose = TRUE` a message will be logged to indicate that the leaf size is large. However, increasing the `max_tree_depth` may not help: it may be that there is something unusual about the distribution of your data set under your chose `metric` that makes a tree-based initialization inappropriate.

`margin` A character string specifying the method used to assign points to one side of the hyperplane or the other. Possible values are:

- `"explicit"` categorizes all distance metrics as either Euclidean or Angular (Euclidean after normalization), explicitly calculates a hyperplane and offset, and then calculates the margin based on the dot product with the hyperplane.
- `"implicit"` calculates the distance from a point to each of the points defining the normal vector. The margin is calculated by comparing the two distances: the point is assigned to the side of the hyperplane that the normal vector point with the closest distance belongs to.
- `"auto"` (the default) picks the margin method depending on whether a binary-specific `metric` such as `"bhammming"` is chosen, in which case `"implicit"` is used, and `"explicit"` otherwise: binary-specific metrics involve storing the data in a way that isn't very efficient for the `"explicit"` method and the binary-specific metric is usually a lot faster than the generic equivalent such that the cost of two distance calculations for the margin method is still faster.

`n_threads` Number of threads to use.

`verbose` If `TRUE`, log information to the console.

`obs` set to `"C"` to indicate that the input `data` orientation stores each observation as a column. The default `"R"` means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Value

a forest of random projection trees as a list. Each tree in the forest is a further list, but is not intended to be examined or manipulated by the user. As a normal R data type, it can be safely serialized and deserialized with `base::saveRDS()` and `base::readRDS()`. To use it for querying pass it as the `forest` parameter of `rpf_knn_query()`. The forest does not store any of the `data` passed into build the tree, so if you are going to search the forest, you will also need to store the `data` used to build it and provide it during the search.

References

Dasgupta, S., & Freund, Y. (2008, May). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing* (pp. 537-546). doi:10.1145/1374376.1374452.

See Also

`rpf_knn_query()`

Examples

```
# Build a forest of 10 trees from the odd rows
iris_odd <- iris[seq_len(nrow(iris)) %% 2 == 1, ]
iris_odd_forest <- rpf_build(iris_odd, n_trees = 10)

iris_even <- iris[seq_len(nrow(iris)) %% 2 == 0, ]
iris_even_nn <- rpf_knn_query(
  query = iris_even, reference = iris_odd,
  forest = iris_odd_forest, k = 15
)
```

---

rpf_filter *Keep the best trees in a random projection forest*

Description

Reduce the size of a random projection forest, by scoring each tree against a k-nearest neighbors graph. Only the top N trees will be retained which allows for a faster querying.

Usage

```
rpf_filter(nn, forest = NULL, n_trees = 1, n_threads = 0, verbose = FALSE)
```

Arguments

`nn`	Nearest neighbor data in the dense list format. This should be derived from the same data that was used to build the `forest`.
`forest`	A random partition forest, e.g. created by `rpf_build()`, representing partitions of the same underlying data reflected in `nn`. As a convenient, this parameter is ignored if the `nn` list contains a `forest` entry, e.g. from running `rpf_knn()` or `nnd_knn()` with `ret_forest = TRUE`, and the forest value will be extracted from `nn`.
`n_trees`	The number of trees to retain. By default only the best-scoring tree is retained.
`n_threads`	Number of threads to use.
`verbose`	If `TRUE`, log information to the console.

Details

Trees are scored based on how well each leaf reflects the neighbors as specified in the nearest neighbor data. It's best to use as accurate nearest neighbor data as you can and it does not need to come directly from searching the `forest`: for example, the nearest neighbor data from running `nnd_knn()` to optimize the neighbor data output from an RP Forest is a good choice.

Rather than rely on an RP Forest solely for approximate nearest neighbor querying, it is probably more cost-effective to use a small number of trees to initialize the neighbor list for use in a graph search via `graph_knn_query()`.

Value

A forest with the best scoring `n_trees` trees.

See Also

`rpf_build()`

Examples

```
# Build a knn with a forest of 10 trees using the odd rows
iris_odd <- iris[seq_len(nrow(iris)) %% 2 == 1, ]
# also return the forest with the knn
rfknn <- rpf_knn(iris_odd, k = 15, n_trees = 10, ret_forest = TRUE)

# keep the best 2 trees:
iris_odd_filtered_forest <- rpf_filter(rfknn)

# get some new data to search
iris_even <- iris[seq_len(nrow(iris)) %% 2 == 0, ]

# search with the filtered forest
iris_even_nn <- rpf_knn_query(
  query = iris_even, reference = iris_odd,
  forest = iris_odd_filtered_forest, k = 15
)
```

`rpf_knn` *Find nearest neighbors using a random projection forest*

Description

Returns the approximate k-nearest neighbor graph of a dataset by searching multiple random projection trees, a variant of k-d trees originated by Dasgupta and Freund (2008).

Usage

```
rpf_knn(
  data,
  k,
  metric = "euclidean",
  use_alt_metric = TRUE,
  n_trees = NULL,
  leaf_size = NULL,
  max_tree_depth = 200,
  include_self = TRUE,
  ret_forest = FALSE,
  margin = "auto",
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

data	Matrix of n items to generate neighbors for, with observations in the rows and features in the columns. Optionally, input can be passed with observations in the columns, by setting obs = "C", which should be more efficient. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
k	Number of nearest neighbors to return. Optional if init is specified.
metric	Type of distance calculation to use. One of: • "braycurtis" • "canberra" • "chebyshev" • "correlation" (1 minus the Pearson correlation) • "cosine" • "dice" • "euclidean" • "hamming" • "hellinger" • "jaccard" • "jensenshannon" • "kulsinski" • "sqeuclidean" (squared Euclidean) • "manhattan" • "rogerstanimoto"

- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "spearmanr" (1 minus the Spearman rank correlation)
- "symmetrickl" (symmetric Kullback-Leibler divergence)
- "tsss" (Triangle Area Similarity-Sector Area Similarity or TS-SS metric)
- "yule"

For non-sparse data, the following variants are available with preprocessing: this trades memory for a potential speed up during the distance calculation. Some minor numerical differences should be expected compared to the non-preprocessed versions:

- "cosine-preprocess": cosine with preprocessing.
- "correlation-preprocess": correlation with preprocessing.

For non-sparse binary data passed as a logical matrix, the following metrics have specialized variants which should be substantially faster than the non-binary variants (in other cases the logical data will be treated as a dense numeric vector of 0s and 1s):

- "dice"
- "hamming"
- "jaccard"
- "kulsinski"
- "matching"
- "rogerstanimoto"
- "russellrao"
- "sokalmichener"
- "sokalsneath"
- "yule"

Note that if margin = "explicit", the metric is only used to determine whether an "angular" or "Euclidean" distance is used to measure the distance between split points in the tree.

use_alt_metric If TRUE, use faster metrics that maintain the ordering of distances internally (e.g. squared Euclidean distances if using metric = "euclidean"), then apply a correction at the end. Probably the only reason to set this to FALSE is if you suspect that some sort of numeric issue is occurring with your data in the alternative code path.

n_trees The number of trees to use in the RP forest. A larger number will give more accurate results at the cost of a longer computation time. The default of NULL means that the number is chosen based on the number of observations in data.

leaf_size The maximum number of items that can appear in a leaf. The default of NULL means that the number of leaves is chosen based on the number of requested neighbors k.

max_tree_depth The maximum depth of the tree to build (default = 200). If the maximum tree depth is exceeded then the leaf size of a tree may exceed leaf_size which can

result in a large number of neighbor distances being calculated. If `verbose = TRUE` a message will be logged to indicate that the leaf size is large. However, increasing the `max_tree_depth` may not help: it may be that there is something unusual about the distribution of your data set under your chose `metric` that makes a tree-based initialization inappropriate.

`include_self` If `TRUE` (the default) then an item is considered to be a neighbor of itself. Hence the first nearest neighbor in the results will be the item itself. This is a convention that many nearest neighbor methods and software adopt, so if you want to use the resulting knn graph from this function in downstream applications or compare with other methods, you should probably keep this set to `TRUE`. However, if you are planning on using the result of this as initialization to another nearest neighbor method (e.g. `nnd_knn()`), then set this to `FALSE`.

`ret_forest` If `TRUE` also return a search forest which can be used for future querying (via `rpf_knn_query()`) and filtering (via `rpf_filter()`). By default this is `FALSE`. Setting this to `TRUE` will change the output list to be nested (see the `Value` section below).

`margin` A character string specifying the method used to assign points to one side of the hyperplane or the other. Possible values are:

- `"explicit"` categorizes all distance metrics as either Euclidean or Angular (Euclidean after normalization), explicitly calculates a hyperplane and offset, and then calculates the margin based on the dot product with the hyperplane.
- `"implicit"` calculates the distance from a point to each of the points defining the normal vector. The margin is calculated by comparing the two distances: the point is assigned to the side of the hyperplane that the normal vector point with the closest distance belongs to.
- `"auto"` (the default) picks the margin method depending on whether a binary-specific `metric` such as `"bhammming"` is chosen, in which case `"implicit"` is used, and `"explicit"` otherwise: binary-specific metrics involve storing the data in a way that isn't very efficient for the `"explicit"` method and the binary-specific metric is usually a lot faster than the generic equivalent such that the cost of two distance calculations for the margin method is still faster.

`n_threads` Number of threads to use.

`verbose` If `TRUE`, log information to the console.

`obs` set to `"C"` to indicate that the input `data` orientation stores each observation as a column. The default `"R"` means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Value

the approximate nearest neighbor graph as a list containing:

- `idx` an n by k matrix containing the nearest neighbor indices.

- `dist` an n by k matrix containing the nearest neighbor distances.
- `forest` (if `ret_forest = TRUE`) the RP forest that generated the neighbor graph, which can be used to query new data.

k neighbors per observation are not guaranteed to be found. Missing data is represented with an index of `0` and a distance of `NA`.

References

Dasgupta, S., & Freund, Y. (2008, May). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing* (pp. 537-546). doi:10.1145/1374376.1374452.

See Also

`rpf_filter()`, `nnd_knn()`

Examples

```
# Find 4 (approximate) nearest neighbors using Euclidean distance
# If you pass a data frame, non-numeric columns are removed
iris_nn <- rpf_knn(iris, k = 4, metric = "euclidean", leaf_size = 3)

# If you want to initialize another method (e.g. nearest neighbor descent)
# with the result of the RP forest, then it's more efficient to skip
# evaluating whether an item is a neighbor of itself by setting
# `include_self = FALSE`:
iris_rp <- rpf_knn(iris, k = 4, n_trees = 3, include_self = FALSE)
# for future querying you may want to also return the RP forest:
iris_rpf <- rpf_knn(iris,
  k = 4, n_trees = 3, include_self = FALSE,
  ret_forest = TRUE
)
```

---

rpf_knn_query *Query a random projection forest index for nearest neighbors*

Description

Run queries against a "forest" of Random Projection Trees (Dasgupta and Freund, 2008), to return nearest neighbors taken from the reference data used to build the forest.

Usage

```
rpf_knn_query(
  query,
  reference,
  forest,
  k,
```

```
  cache = TRUE,
  n_threads = 0,
  verbose = FALSE,
  obs = "R"
)
```

Arguments

query	Matrix of n query items, with observations in the rows and features in the columns. Optionally, the data may be passed with the observations in the columns, by setting obs = "C", which should be more efficient. The reference data must be passed in the same orientation as query. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format. Dataframes will be converted to numerical matrix format internally, so if your data columns are logical and intended to be used with the specialized binary metrics, you should convert it to a logical matrix first (otherwise you will get the slower dense numerical version).
reference	Matrix of m reference items, with observations in the rows and features in the columns. The nearest neighbors to the queries are calculated from this data and should be the same data used to build the forest. Optionally, the data may be passed with the observations in the columns, by setting obs = "C", which should be more efficient. The query data must be passed in the same format and orientation as reference. Possible formats are base::data.frame(), base::matrix() or Matrix::sparseMatrix(). Sparse matrices should be in dgCMatrix format.
forest	A random partition forest, created by rpf_build(), representing partitions of the data in reference.
k	Number of nearest neighbors to return. You are unlikely to get good results if you choose a value substantially larger than the value of leaf_size used to build the forest.
cache	if TRUE (the default) then candidate indices found in the leaves of the forest are cached to avoid recalculating the same distance repeatedly. This incurs an extra memory cost which scales with n_threads. Set this to FALSE to disable distance caching.
n_threads	Number of threads to use. Note that the parallelism in the search is done over the observations in query not the trees in the forest. Thus a single observation will not see any speed-up from increasing n_threads.
verbose	If TRUE, log information to the console.
obs	set to "C" to indicate that the input data orientation stores each observation as a column. The default "R" means that observations are stored in each row. Storing the data by row is usually more convenient, but internally your data will be converted to column storage. Passing it already column-oriented will save some memory and (a small amount of) CPU usage.

Value

the approximate nearest neighbor graph as a list containing:

- `idx` an n by k matrix containing the nearest neighbor indices.
- `dist` an n by k matrix containing the nearest neighbor distances.

`k` neighbors per observation are not guaranteed to be found. Missing data is represented with an index of `0` and a distance of `NA`.

References

Dasgupta, S., & Freund, Y. (2008, May). Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing* (pp. 537-546). doi:10.1145/1374376.1374452.

See Also

`rpf_build()`

Examples

```
# Build a forest of 10 trees from the odd rows
iris_odd <- iris[seq_len(nrow(iris)) %% 2 == 1, ]
iris_odd_forest <- rpf_build(iris_odd, n_trees = 10)

iris_even <- iris[seq_len(nrow(iris)) %% 2 == 0, ]
iris_even_nn <- rpf_knn_query(
  query = iris_even, reference = iris_odd,
  forest = iris_odd_forest, k = 15
)
```

Index