Package 'BiocNeighbors'

September 23, 2021

Version 1.10.0

Date 2021-05-18

Title Nearest Neighbor Detection for Bioconductor Packages

Imports Rcpp, S4Vectors, BiocParallel, stats, methods, Matrix

Suggests testthat, BiocStyle, knitr, rmarkdown, FNN, RcppAnnoy, RcppHNSW

biocViews Clustering, Classification

Description Implements exact and approximate methods for nearest neighbor detection, in a framework that allows them to be easily switched within Bioconductor packages or workflows. Exact searches can be performed using the k-means for k-nearest neighbors algorithm or with vantage point trees. Approximate searches can be performed using the Annoy or HNSW libraries. Searching on either Euclidean or Manhattan distances is supported. Parallelization is achieved for all methods by using BiocParallel. Functions are also provided to search for all neighbors within a given distance.

License GPL-3

LinkingTo Rcpp, RcppHNSW

VignetteBuilder knitr

SystemRequirements C++11

RoxygenNote 7.1.1

git_url https://git.bioconductor.org/packages/BiocNeighbors

git_branch RELEASE_3_13

git_last_commit c6779d6

git_last_commit_date 2021-05-19

Date/Publication 2021-09-23

Author Aaron Lun [aut, cre, cph]

Maintainer Aaron Lun <infinite.monkeys.with.keyboards@gmail.com>

R topics documented:

`BiocNeighbors-package` *BiocNeighbors: Nearest Neighbor Detection for Bioconductor Packages*

Description

Implements exact and approximate methods for nearest neighbor detection, in a framework that allows them to be easily switched within Bioconductor packages or workflows. Exact searches can be performed using the k-means for k-nearest neighbors algorithm or with vantage point trees. Approximate searches can be performed using the Annoy or HNSW libraries. Searching on either Euclidean or Manhattan distances is supported. Parallelization is achieved for all methods by using BiocParallel. Functions are also provided to search for all neighbors within a given distance.

Author(s)

Maintainer: Aaron Lun <infinite.monkeys.with.keyboards@gmail.com> [copyright holder]

`AnnoyIndex` *The AnnoyIndex class*

Description

A class to hold indexing structures for the Annoy algorithm for approximate nearest neighbor identification.

Usage

```
AnnoyIndex(data, path, search.mult = 50, NAMES = NULL, distance = "Euclidean")
```

Arguments

`data`	A numeric matrix with data points in columns and dimensions in rows.
`path`	A string specifying the path to the index file.
`search.mult`	Numeric scalar, multiplier for the number of points to search.
`NAMES`	A character vector of sample names or `NULL`.
`distance`	A string specifying the distance metric to use.

Details

The AnnoyIndex class holds the indexing structure required to run the Annoy algorithm. Users should never need to call the constructor explicitly, but should generate instances of AnnoyIndex classes with `buildAnnoy`.

Users can get values from an AnnoyIndex object with the usual `[[` syntax. All parameters listed in the constructor can be extracted in this manner.

Value

An instance of the AnnoyIndex class.

Author(s)

Aaron Lun

See Also

`buildAnnoy`, for the index construction.

BiocNeighborIndex, for the parent class and its available methods.

Examples

```
example(buildAnnoy)
out[['path']]
bndistance(out)
str(bndata(out))
```

`AnnoyParam` *The AnnoyParam class*

Description

A class to hold parameters for the Annoy algorithm for approximate nearest neighbor identification.

Usage

```
AnnoyParam(
  ntrees = 50,
  directory = tempdir(),
  search.mult = ntrees,
  distance = "Euclidean"
)
```

Arguments

`ntrees`	Integer scalar, number of trees to use for index generation.
`directory`	String containing the path to the directory in which to save the index.
`search.mult`	Numeric scalar, multiplier for the number of points to search.
`distance`	String, the distance metric to use.

Details

The AnnoyParam class holds all parameters associated with running the Annoy algorithm. Most of these parameters are used to build the index - see `buildAnnoy` for details.

Users can get or set values with the usual `[[` syntax. All parameters listed in the constructor can be manipulated in this manner.

Value

An instance of the AnnoyParam class.

Author(s)

Aaron Lun

See Also

buildAnnoy, for the index construction.

findAnnoy and related functions, for the actual search.

BiocNeighborParam, for the parent class and its available methods.

Examples

```
(out <- AnnoyParam())
out[['ntrees']]

out[['ntrees']] <- 20L
out
```

BiocNeighborIndex *The BiocNeighborIndex class*

Description

A virtual class for indexing structures of different nearest-neighbor search algorithms.

Details

The BiocNeighborIndex class is a virtual base class on which other index objects are built. There are 4 concrete subclasses:

KmknnIndex: exact nearest-neighbor search with the KMKNN algorithm.

VptreeIndex: exact nearest-neighbor search with a VP tree.

AnnoyIndex: approximate nearest-neighbor search with the Annoy algorithm.

HnswIndex: approximate nearest-neighbor search with the HNSW algorithm.

These objects hold indexing structures for a given data set - see the associated documentation pages for more details. It also retains information about the input data as well as the sample names.

Methods

In the following code snippets, `x` and `object` are BiocNeighborIndex objects.

The main user-accessible methods are:

`show(object)`: Display the class and dimensions of `object`.

`dim(x)`: Return the dimensions of `x`, in terms of the matrix used to construct it.

`dimnames(x)`: Return the dimension names of `x`. Only the row names of the input matrix are stored, in the same order.

`x[[i]]`: Return the value of slot `i`, as used in the constructor for `x`.

More advanced methods (intended for developers of other packages) are:

`bndata(object)`: Return a numeric matrix containing the data used to construct `object`. Each column should represent a data point and each row should represent a variable (i.e., it is transposed compared to the usual input, for efficient column-major access in C++ code). Columns may be reordered from the input matrix according to `bnorder(object)`.

`bnorder(object)`: Return an integer vector specifying the new ordering of columns in `bndata(object)`. This generally only needs to be considered if `raw.index=TRUE`, see `?"BiocNeighbors-raw-index"`.

`bndistance(object)`: Return a string specifying the distance metric to be used for searching. This should be one of `"Euclidean"`, `"Manhattan"` or `"Cosine"`. Obviously, this should be the same as the distance metric used for constructing the index.

Author(s)

Aaron Lun

See Also

`KmknnIndex`, `VptreeIndex`, `AnnoyIndex`, and `HnswIndex` for direct constructors.

`buildIndex` for construction on an actual data set.

`findKNN` and `queryKNN` for dispatch.

`BiocNeighborParam` *The BiocNeighborParam class*

Description

A virtual class for specifying the type of nearest-neighbor search algorithm and associated parameters.

Details

The BiocNeighborParam class is a virtual base class on which other parameter objects are built. There are currently 4 concrete subclasses:

`KmknnParam`: exact nearest-neighbor search with the KMKNN algorithm.

`VptreeParam`: exact nearest-neighbor search with the VP tree algorithm.

`AnnoyParam`: approximate nearest-neighbor search with the Annoy algorithm.

`HnswParam`: approximate nearest-neighbor search with the HNSW algorithm.

These objects hold parameters specifying how each algorithm should be run on an arbitrary data set. See the associated documentation pages for more details.

Methods

In the following code snippets, `x` and `object` are BiocNeighborParam objects.

`show(object)`: Display the class and arguments of `object`.

`bndistance(object)`: Return a string specifying the distance metric to be used for searching. This should be one of `"Euclidean"`, `"Manhattan"` or `"Cosine"`.

`x[[i]]`: Return the value of slot `i`, as used in the constructor for `x`.

`x[[i]] <- value`: Set slot `i` to the specified `value`.

Author(s)

Aaron Lun

See Also

`KmknnParam`, `VptreeParam`, `AnnoyParam`, and `HnswParam` for constructors.

`buildIndex`, `findKNN` and `queryKNN` for dispatch.

BiocNeighbors-algorithms

Neighbor search algorithms

Description

This page provides an overview of the neighbor search algorithms available in **BiocNeighbors**.

K-means with k-nearest neighbors (KMKNN)

In the KMKNN algorithm (Wang, 2012), k-means clustering is first applied to the data points using the square root of the number of points as the number of cluster centers. The cluster assignment and distance to the assigned cluster center for each point represent the KMKNN indexing information. This speeds up the nearest neighbor search by exploiting the triangle inequality between cluster centers, the query point and each point in the cluster to narrow the search space. The advantage of the KMKNN approach is its simplicity and minimal overhead, resulting in performance improvements over conventional tree-based methods for high-dimensional data where most points need to be searched anyway. It is also trivially extended to find all neighbors within a threshold distance from a query point.

Vantage point (VP) trees

In a VP tree (Yianilos, 1993), each node contains a subset of points that is split into two further partitions. The split is determined by picking an arbitrary point inside that subset as the node center, computing the distance to all other points from the center, and taking the median as the "radius". The left child of this node contains all points within the median distance from the radius, while the right child contains the remaining points. This is applied recursively until all points resolve to individual nodes. The nearest neighbor search traverses the tree and exploits the triangle inequality

between query points, node centers and thresholds to narrow the search space. VP trees are often faster than more conventional KD-trees or ball trees as the former uses the points themselves as the nodes of the tree, avoiding the need to create many intermediate nodes and reducing the total number of distance calculations. Like KMKNN, it is also trivially extended to find all neighbors within a threshold distance from a query point.

Exhaustive search

The exhaustive search computes all pairwise distances between data and query points to identify nearest neighbors of the latter. It has quadratic complexity and is theoretically the worst-performing method; however, it has effectively no overhead from constructing or querying indexing structures, making it faster for in situations where indexing provides little benefit. This includes queries against datasets with few data points or very high dimensionality.

Approximate nearest neighbors Oh Yeah (Annoy)

The Annoy algorithm was developed by Erik Bernhardsson to identify approximate k-nearest neighbors in high-dimensional data. Briefly, a tree is constructed where a random hyperplane splits the points into two subsets at each internal node. Leaf nodes are defined when the number of points in a subset falls below a threshold (close to twice the number of dimensions for the settings used here). Multiple trees are constructed in this manner, each of which is different due to the random choice of hyperplanes. For a given query point, each tree is searched to identify the subset of all points in the same leaf node as the query point. The union of these subsets across all trees is exhaustively searched to identify the actual nearest neighbors to the query.

Hierarchical navigable small worlds (HNSW)

In the HNSW algorithm (Malkov and Yashunin, 2016), each point is a node in a "nagivable small world" graph. The nearest neighbor search proceeds by starting at a node and walking through the graph to obtain closer neighbors to a given query point. Nagivable small world graphs are used to maintain connectivity across the data set by creating links between distant points. This speeds up the search by ensuring that the algorithm does not need to take many small steps to move from one cluster to another. The HNSW algorithm extends this idea by using a hierarchy of such graphs containing links of different lengths, which avoids wasting time on small steps in the early stages of the search where the current node position is far from the query.

Distance metrics

All algorithms support neighbor searching by Euclidean, Manhattan and cosine distances. Cosine distances are implemented as the Euclidean distance between L2-normalized vectors. Note that KMKNN operates much more naturally with Euclidean distances, so your mileage may vary when using it with Manhattan distances.

Author(s)

Aaron Lun, using code from the **cydar** package for the KMKNN implementation; from Steve Hanov, for the VP tree implementation; **RcppAnnoy**, for the Annoy implementation; and **RcppHNSW**, for the HNSW implementation.

References

Wang X (2012). A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. *Proc Int Jt Conf Neural Netw*, 43, 6:2351-2358.

Hanov S (2011). VP trees: A data structure for finding stuff fast. http://stevehanov.ca/blog/index.php?id=130

Yianilos PN (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, 311-321.

Bernhardsson E (2018). Annoy. https://github.com/spotify/annoy

Malkov YA, Yashunin DA (2016). Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. *arXiv*. https://arxiv.org/abs/1603.09320

`BiocNeighbors-raw-index`

Reporting raw indices

Description

An overview of what raw indices mean for neighbor-search implementations that contain a rearranged matrix in the BiocNeighborIndex object.

What are raw indices?

Consider the following call:

```
index <- buildKmknn(vals)
out <- findKmknn(precomputed=index, k=k, raw.index=TRUE)
```

This yields the same output as:

```
PRE <- bndata(index)
out2 <- findKmknn(X=t(PRE), k=k)
```

When `raw.index=TRUE` in the first call, the indices in `out$index` matrix can be imagined to refer to *columns* of `PRE` in the second call. Moreover, all function arguments that previously referred to rows of `X` (e.g., `subset`) are now considered to refer to columns of `PRE`.

The same reasoning applies to all functions where `precomputed` can be specified in place of `X`. This includes query-based searches (e.g., `queryKmknn`) and range searches (`rangeFindKmknn`).

Motivation

Setting `raw.index=TRUE` is intended for scenarios where the reordered data in `precomputed` is used elsewhere. By returning indices to the reordered data, the user does not need to hold onto the original data and/or switch between the original ordering and that in `precomputed`. This simplifies downstream code and provides a slight speed boost by avoiding the need for re-indexing.

Neighbor search implementations can only return raw indices if their index construction involves transposing `X` and reordering its columns. This tends to be the case for most implementations as transposition allows efficient column-major distance calculations and reordering improves data locality. Both the KMKNN and VP tree implementations fulfill these requirements and thus have the `raw.index` option.

Note that setting `raw.index=TRUE` makes little sense when `precomputed` is not specified. When `precomputed=NULL`, a temporary index will be constructed that is not visible in the calling scope. As index construction may be stochastic, the raw indices will not refer to anything that is meaningful to the end-user.

Author(s)

Aaron Lun

See Also

`findKmknn` and `findVptree` for examples where raw indices are used.

Examples

```
vals <- matrix(rnorm(100000), ncol=20)
index <- buildKmknn(vals)
out <- findKmknn(precomputed=index, raw.index=TRUE, k=5)
alt <- findKmknn(t(bndata(index)), k=5)
head(out$index)
head(alt$index)
```

BiocNeighbors-ties *Handling tied distances*

Description

Interpreting the warnings when distances are tied in an exact nearest neighbor (NN) search.

The problem of ties

The most obvious problem with ties is that it may affect the identity of the reported neighbors. The various NN search functions will return a constant number of neighbors for each data point. If the kth neighbor is tied with the k+1th neighbor, this requires an arbitrary decision about which data point to retain in the NN set. A milder issue is that the order of the neighbors within the set is arbitrary, which may be important for certain algorithms.

As such, a warning will be raised if tied distances are detected among the k+1 NNs for any of the exact NN search methods. We only consider exact ties at double precision - previous versions of this package would account for numerical imprecision, but this is no longer the case. No warning is given for the approximate methods as their use already implies that a certain degree of inaccuracy is acceptable.

Interaction with random seeds

In general, the exact NN search algorithms in this package are fully deterministic despite the use of stochastic steps during index construction. The only exception occurs when there are tied distances to neighbors, at which point the order and/or identity of the k-nearest neighboring points is not well-defined. This is because, in the presence of ties, the output will depend on the ordering of points in the constructed index from `buildKmknn` or `buildVptree`.

Users should set the seed to guarantee consistent (albeit arbitrary) results across different runs of the function. However, note that the exact selection of tied points depends on the numerical precision of the system. Thus, even after setting a seed, there is no guarantee that the results will be reproducible across machines (especially Windows)!

Turning off the warnings

It may ocassionally be appropriate to disable the warnings by setting `warn.ties=FALSE`. The most obvious scenario is when `get.index=FALSE`, i.e., we are only interested in the distances to the neighbors. In such cases, the presence of ties does not matter as changes to the identity of tied neighbors do not affect the returned distances (which, for ties, are equal by definition). Similarly, if the seed is set prior to the search, the warnings are unnecessary as the output is fully deterministic.

Author(s)

Aaron Lun

See Also

`findKmknn` and `findVptree` for examples where tie warnings are produced.

Examples

```
vals <- matrix(0, nrow=10, ncol=20)
out <- findKmknn(vals, k=5)
```

---

buildAnnoy *Build an Annoy index*

Description

Build an Annoy index and save it to file in preparation for a nearest-neighbors search.

Usage

```
buildAnnoy(
  X,
  transposed = FALSE,
  ntrees = 50,
  directory = tempdir(),
```

```
  search.mult = ntrees,
  fname = tempfile(tmpdir = directory, fileext = ".idx"),
  distance = c("Euclidean", "Manhattan", "Cosine")
)
```

Arguments

X	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
transposed	Logical scalar indicating whether X is transposed, i.e., rows are variables and columns are data points.
ntrees	Integer scalar specifying the number of trees to build in the index.
directory	String containing the path to the directory in which to save the index file.
search.mult	Numeric scalar specifying the multiplier for the number of points to search.
fname	String containing the path to the index file.
distance	String specifying the type of distance to use.

Details

This function is automatically called by findAnnoy and related functions. However, it can be called directly by the user to save time if multiple queries are to be performed to the same X.

It is advisable to change directory to a location that is amenable to parallel read operations on HPC file systems. Of course, if index files are manually constructed, the user is also responsible for their clean-up after all calculations are completed.

The ntrees parameter controls the trade-off between accuracy and computational work. More trees provide greater accuracy at the cost of more computational work (both in terms of the indexing time and search speed in downstream functions).

The search.mult controls the parameter known as search_k in the original Annoy documentation. Specifically, search_k is defined as k * search.mult where k is the number of nearest neighbors to identify in downstream functions. This represents the number of points to search exhaustively and determines the run-time balance between speed and accuracy. The default search.mult=ntrees is based on the Annoy library defaults. Note that this parameter is not actually used in the index construction itself, and is only included here so that the output index fully parametrizes the search.

Technically, the index construction algorithm is stochastic but, for various logistical reasons, the seed is hard-coded into the C++ code. This means that the results of the Annoy neighbor searches will be fully deterministic for the same inputs, even though the theory provides no such guarantees.

Value

An AnnoyIndex object containing a path to the index file, plus additional parameters for the search.

Author(s)

Aaron Lun

See Also

AnnoyIndex, for details on the output class.

findAnnoy and queryAnnoy, for dependent functions.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
out <- buildAnnoy(Y)
out
```

`buildExhaustive` *Prepare data for an exhaustive search*

Description

Transform data in preparation for an exhaustive (i.e., brute-force) search.

Usage

```
buildExhaustive(
  X,
  transposed = FALSE,
  distance = c("Euclidean", "Manhattan", "Cosine")
)
```

Arguments

`X`	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
`transposed`	Logical scalar indicating whether `X` is transposed, i.e., rows are variables and columns are data points.
`distance`	String specifying the type of distance to use.

Details

This algorithm is largely provided as a baseline for comparing against the other algorithms. On rare occasions, it may actually be useful in, e.g., very high-dimensional data where the indexing step of other algorithms adds computational overhead for no benefit.

Value

An ExhaustiveIndex object containing indexed data.

Author(s)

Allison Vuong

See Also

ExhaustiveIndex, for details on the output class.

`findExhaustive` and `queryExhaustive`, for dependent functions.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
out <- buildExhaustive(Y)
out
```

`buildHnsw` *Build a HNSW index*

Description

Build a HNSW index and save it to file in preparation for a nearest-neighbors search.

Usage

```
buildHnsw(
  X,
  transposed = FALSE,
  nlinks = 16,
  ef.construction = 200,
  directory = tempdir(),
  ef.search = 10,
  fname = tempfile(tmpdir = directory, fileext = ".idx"),
  distance = c("Euclidean", "Manhattan", "Cosine")
)
```

Arguments

`X`	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
`transposed`	Logical scalar indicating whether `X` is transposed, i.e., rows are variables and columns are data points.
`nlinks`	Integer scalar specifying the number of bi-directional links for each element.
`ef.construction`	Integer scalar specifying the size of the dynamic list during index construction.
`directory`	String containing the path to the directory in which to save the index file.
`ef.search`	Integer scalar specifying the size of the dynamic list to use during neighbor searching.
`fname`	String containing the path to the index file.
`distance`	String specifying the type of distance to use.

Details

This function is automatically called by `findHnsw` and related functions. However, it can be called directly by the user to save time if multiple queries are to be performed to the same `X`.

It is advisable to change `directory` to a location that is amenable to parallel read operations on HPC file systems. Of course, if index files are manually constructed, the user is also responsible for their clean-up after all calculations are completed.

Larger values of `nlinks` improve accuracy at the expense of speed and memory usage. Larger values of `ef.construction` improve index quality at the expense of indexing time.

The value of `ef.search` controls the accuracy of the neighbor search at run time. Larger values improve accuracy at the expense of a slower search. In `findHnsw` and `queryHnsw`, this is always lower-bounded at `k`, the number of nearest neighbors to identify. Note that this parameter is not actually used in the index construction itself, and is only included here so that the output index fully parametrizes the search.

Technically, the index construction algorithm is stochastic but, for various logistical reasons, the seed is hard-coded into the C++ code. This means that the results of the HNSW neighbor searches will be fully deterministic for the same inputs, even though the theory provides no such guarantees.

Value

An AnnoyIndex object containing a path to the index file, plus additional parameters for the search.

Author(s)

Aaron Lun

See Also

HnswIndex, for details on the output class.

`findHnsw` and `queryHnsw`, for dependent functions.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
out <- buildHnsw(Y)
out
```

`buildIndex` *Build a nearest-neighbor index*

Description

Build indices for nearest-neighbor searching with different algorithms.

Usage

```
buildIndex(X, ..., BNPARAM)
```

Arguments

X	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
...	Further arguments to be passed to individual methods. This is guaranteed to include transposed.
BNPARAM	A BiocNeighborParam object specifying the type of index to be constructed. This defaults to a KmknnParam object if no argument is supplied.

Details

Supplying a KmknnParam object as BNPARAM will dispatch to buildKmknn.

Supplying a VptreeParam object as BNPARAM will dispatch to buildVptree.

Supplying an AnnoyParam object as BNPARAM will dispatch to buildAnnoy.

Supplying an HnswParam object as BNPARAM will dispatch to buildHnsw.

Value

An instance of a BiocNeighborIndex subclass, containing indexing structures for the specified algorithm.

Author(s)

Aaron Lun

See Also

buildKmknn, buildVptree, buildAnnoy and buildHnsw for specific methods.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
(k.out <- buildIndex(Y))
(a.out <- buildIndex(Y, BNPARAM=AnnoyParam()))
```

---

buildKmknn *Pre-cluster points with k-means*

Description

Perform k-means clustering in preparation for a KMKNN nearest-neighbors search.

Usage

```
buildKmknn(
  X,
  transposed = FALSE,
  distance = c("Euclidean", "Manhattan", "Cosine"),
  ...
)
```

Arguments

X	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
transposed	Logical scalar indicating whether X is transposed, i.e., rows are variables and columns are data points.
distance	String specifying the type of distance to use.
...	Further arguments to pass to kmeans.

Details

This function is automatically called by findKmknn and related functions. However, it can be called directly by the user to save time if multiple queries are to be performed to the same X.

Points in X are reordered to improve data locality during the nearest-neighbor search. Specifically, points in the same cluster are contiguous and ordered by increasing distance from the cluster center.

After k-means clustering, the function will store the coordinates of the cluster center in the output object. In addition, it records a list of extra information of length equal to the number of clusters. Each entry corresponds a cluster (let's say cluster j) and is a list of length 2. The first element is an integer scalar containing the zero-index of the first point in the reordered data matrix that is assigned to j. The second element is a numeric vector containing the distance of each point in the cluster from the cluster center.

Value

A KmknnIndex object containing indexing structures for the KMKNN search.

Author(s)

Aaron Lun

See Also

kmeans, for optional arguments.

KmknnIndex for details on the output class.

findKmknn, queryKmknn and findNeighbors, for dependent functions.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
out <- buildKmknn(Y)
out
```

`buildVptree` *Build a VP tree*

Description

Build a vantage point tree in preparation for a nearest-neighbors search.

Usage

```
buildVptree(
  X,
  transposed = FALSE,
  distance = c("Euclidean", "Manhattan", "Cosine")
)
```

Arguments

`X`	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
`transposed`	Logical scalar indicating whether `X` is transposed, i.e., rows are variables and columns are data points.
`distance`	String specifying the type of distance to use.

Details

This function is automatically called by `findVptree` and related functions. However, it can be called directly by the user to save time if multiple queries are to be performed to the same `X`.

Points in `X` are reordered to improve data locality during the nearest-neighbor search. Specifically, points in the same cluster are contiguous and ordered by increasing distance from the cluster center.

The function also reports a list containing four vectors of equal length describing the structure of the VP tree. Each parallel element specifies a node:

- The first integer vector specifies the column index of `data` of the current node.
- The second integer vector specifies the column index of the left child of the current node,
- The third integer vector specifies the column index of the right child of the current node.
- The fourth numeric vector specifies the radius of the current node.

All indices here are zero-based, with child values set to -1 for leaf nodes.

Value

A VptreeIndex object containing indexing structures for the VP-tree search.

Author(s)

Aaron Lun

See Also

VptreeIndex, for details on the output class.

`findVptree` and `queryVptree`, for dependent functions.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
out <- buildVptree(Y)
out
```

ExhaustiveIndex *The ExhaustiveIndex class*

Description

A class to hold the data for exact nearest neighbor identification.

Usage

```
ExhaustiveIndex(data, NAMES = NULL, distance = "Euclidean")
```

Arguments

`data`	A numeric matrix with data points in columns and dimensions in rows.
`NAMES`	A character vector of sample names or `NULL`.
`distance`	A string specifying the distance metric to use.

Details

Users should never need to call the constructor explicitly, but should generate instances of ExhaustiveIndex classes with `buildExhaustive`.

Users can get values from an ExhaustiveIndex object with the usual `[[` syntax. All parameters listed in the constructor can be extracted in this manner.

Value

An ExhaustiveIndex object.

See Also

buildExhaustive, for the index construction.

BiocNeighborIndex, for the parent class and its available methods.

Examples

```
example(buildExhaustive)
out[['distance']]
bndistance(out)
```

ExhaustiveParam *The ExhaustiveParam class*

Description

A class to hold parameters for the exhaustive algorithm for exact nearest neighbor identification.

Usage

```
ExhaustiveParam(distance = "Euclidean")
```

Arguments

distance	A string specifying the distance metric to use.

Value

An instance of the ExhaustiveParam class.

Author(s)

Allison Vuong

See Also

buildExhaustive, for the index construction.

findExhaustive and related functions, for the actual search.

BiocNeighborParam, for the parent class and its available methods.

Examples

```
(out <- ExhaustiveParam())
```

findKNN *Find k-nearest neighbors*

Description

Find the k-nearest neighbors for each point in a data set, using exact or approximate algorithms.

Usage

```
findKNN(X, k, ..., BNINDEX, BNPARAM)
```

Arguments

X	A numeric data matrix where rows are points and columns are dimensions. This can be missing if BNINDEX is supplied.
k	An integer scalar specifying the number of nearest neighbors to search for.
...	Further arguments to pass to individual methods. This is guaranteed to include subset, get.index, get.distance, last, warn.ties, raw.index and BPPARAM. See ?"findKNN-functions" for more details.
BNINDEX	A BiocNeighborIndex object containing precomputed index information. This can be missing if X and BNPARAM is supplied, see Details.
BNPARAM	A BiocNeighborParam object specifying the algorithm to use. This can be missing if BNINDEX is supplied, see Details.

Details

The class of BNINDEX and BNPARAM will determine dispatch to specific methods. Only one of these arguments needs to be defined to resolve dispatch. However, if both are defined, they cannot specify different algorithms.

If BNINDEX is supplied, X does not need to be specified. In fact, any value of X will be ignored as all necessary information for the search is already present in BNINDEX. Similarly, any parameters in BNPARAM will be ignored.

If both BNINDEX and BNPARAM are missing, the function will default to the KMKNN algorithm by setting BNPARAM=KmknnParam().

Value

A list is returned containing index, an integer matrix of neighbor identities; and distance, a numeric matrix of distances to those neighbors. See ?"findKNN-functions" for more details.

Author(s)

Aaron Lun

See Also

findExhaustive, findKmknn, findVptree, findAnnoy and findHnsw for specific methods.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
str(k.out <- findKNN(Y, k=10))
str(a.out <- findKNN(Y, k=10, BNPARAM=AnnoyParam()))

e.dex <- buildExhaustive(Y)
str(k.out2 <- findKNN(Y, k=10, BNINDEX=e.dex))
str(k.out3 <- findKNN(Y, k=10, BNINDEX=e.dex, BNPARAM=ExhaustiveParam()))

k.dex <- buildKmknn(Y)
str(k.out2 <- findKNN(Y, k=10, BNINDEX=k.dex))
str(k.out3 <- findKNN(Y, k=10, BNINDEX=k.dex, BNPARAM=KmknnParam()))

a.dex <- buildAnnoy(Y)
str(a.out2 <- findKNN(Y, k=10, BNINDEX=a.dex))
str(a.out3 <- findKNN(Y, k=10, BNINDEX=a.dex, BNPARAM=AnnoyParam()))
```

---

findKNN-functions *Find nearest neighbors*

Description

Find the nearest neighbors of each point in a dataset, using a variety of algorithms.

Usage

```
findAnnoy(
  X,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  subset = NULL,
  raw.index = NA,
  warn.ties = NA,
  ...
)

findHnsw(
  X,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
```

```
  precomputed = NULL,
  subset = NULL,
  raw.index = NA,
  warn.ties = NA,
  ...
)

findKmknn(
  X,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  subset = NULL,
  raw.index = FALSE,
  warn.ties = TRUE,
  ...
)

findVptree(
  X,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  subset = NULL,
  raw.index = FALSE,
  warn.ties = TRUE,
  ...
)

findExhaustive(
  X,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  subset = NULL,
  raw.index = FALSE,
  warn.ties = TRUE,
  ...
)
```

Arguments

X	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
k	A positive integer scalar specifying the number of nearest neighbors to retrieve.
get.index	A logical scalar indicating whether the indices of the nearest neighbors should be recorded.
get.distance	A logical scalar indicating whether distances to the nearest neighbors should be recorded.
last	An integer scalar specifying the number of furthest neighbors for which statistics should be returned.
BPPARAM	A BiocParallelParam object indicating how the search should be parallelized.
precomputed	A BiocNeighborIndex object of the appropriate class, generated from X. For findExhaustive, this should be a ExhaustiveIndex from buildExhaustive; For findKmknn, this should be a KmknnIndex from buildKmknn; for findVptree, this should be a VptreeIndex from buildVptree; for findAnnoy, this should be a AnnoyIndex from buildAnnoy; and for findHnsw, this should be a HnswIndex from buildHnsw.
subset	A vector indicating the rows of X for which the nearest neighbors should be identified.
raw.index	A logial scalar indicating whether raw column indices should be returned, see ?"BiocNeighbors-raw-index". This argument is ignored for findAnnoy and findHnsw.
warn.ties	Logical scalar indicating whether a warning should be raised if any of the k+1 neighbors have tied distances. This argument is ignored for findAnnoy and findHnsw.
...	Further arguments to pass to the respective build* function for each algorithm. This includes distance, a string specifying whether "Euclidean", "Manhattan" or "Cosine" distances are to be used.

Details

All of these functions identify points in X that are the k nearest neighbors of each other point. findAnnoy and findHnsw perform an approximate search, while findKmknn and findVptree are exact. The upper bound for k is set at the number of points in X minus 1.

By default, nearest neighbors are identified for all data points within X. If subset is specified, nearest neighbors are only detected for the points in the subset. This yields the same result as (but is more efficient than) subsetting the output matrices after running findKmknn with subset=NULL.

Turning off get.index or get.distance will not return the corresponding matrices in the output. This may provide a slight speed boost when these returned values are not of interest. Using BPPARAM will also split the search across multiple workers, which should increase speed proportionally (in theory) to the number of cores.

Setting last will return indices and/or distances for the k -last + 1-th closest neighbor to the k-th neighbor. This can be used to improve memory efficiency, e.g., by only returning statistics for the k-th nearest neighbor by setting last=1. Note that this is entirely orthogonal to subset.

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X (e.g., with buildKmknn). The resulting BiocNeighborIndex object can be supplied as precomputed to multiple function calls, avoiding the need to repeat index construction in each call. Note that when precomputed is supplied, the value of X is completely ignored.

For exact methods, see comments in ?"BiocNeighbors-ties" regarding the warnings when tied distances are observed. For approximate methods, see comments in buildAnnoy and buildHnsw about the (lack of) randomness in the search results.

Value

A list is returned containing:

- index, if get.index=TRUE. This is an integer matrix where each row corresponds to a point (denoted here as i) in X. The row for i contains the row indices of X that are the nearest neighbors to point i, sorted by increasing distance from i.
- distance, if get.distance=TRUE. This is a numeric matrix where each row corresponds to a point (as above) and contains the sorted distances of the neighbors from i.

Each matrix contains last columns. If subset is not NULL, each row of the above matrices refers to a point in the subset, in the same order as supplied in subset.

See ?"BiocNeighbors-raw-index" for an explanation of the output when raw.index=TRUE for the functions that support it.

Author(s)

Aaron Lun

See Also

buildExhaustive, buildKmknn, buildVptree, buildAnnoy, or buildHnsw to build an index ahead of time.

See ?"BiocNeighbors-algorithms" for an overview of the available algorithms.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)

out <- findExhaustive(Y, k=8)
head(out$index)
head(out$distance)

out1 <- findKmknn(Y, k=8)
head(out1$index)
head(out1$distance)

out2 <- findVptree(Y, k=8)
head(out2$index)
head(out2$distance)

out3 <- findAnnoy(Y, k=8)
```

```
head(out3$index)
head(out3$distance)

out4 <- findHnsw(Y, k=8)
head(out4$index)
head(out4$distance)
```

`findMutualNN` *Find mutual nearest neighbors*

Description

Find mutual nearest neighbors (MNN) across two data sets.

Usage

```
findMutualNN(
  data1,
  data2,
  k1,
  k2 = k1,
  BNINDEX1 = NULL,
  BNINDEX2 = NULL,
  BNPARAM = KmknnParam(),
  BPPARAM = SerialParam()
)
```

Arguments

`data1`	A numeric matrix containing points in the rows and variables/dimensions in the columns.
`data2`	A numeric matrix like `data1` for another dataset with the same variables/dimensions.
`k1`	Integer scalar specifying the number of neighbors to search for in `data1`.
`k2`	Integer scalar specifying the number of neighbors to search for in `data2`.
`BNINDEX1`	A BiocNeighborIndex object containing a pre-built index for `data1`.
`BNINDEX2`	A BiocNeighborIndex object containing a pre-built index for `data2`.
`BNPARAM`	A BiocNeighborParam object specifying the neighbour search algorithm to use. This should be consistent with the class of `BNINDEX1` and `BNINDEX2`, if either are specified.
`BPPARAM`	A BiocParallelParam object specifying how parallelization should be performed.

Details

For each point in dataset 1, the set of k2 nearest points in dataset 2 is identified. For each point in dataset 2, the set of k1 nearest points in dataset 1 is similarly identified. Two points in different datasets are considered to be part of an MNN pair if each point lies in the other's set of neighbors. This concept allows us to identify matching points across datasets, which is useful for, e.g., batch correction.

Any values for the BNINDEX1 and BNINDEX2 arguments should be equal to the output of buildIndex for the respective matrices, using the algorithm specified with BNPARAM. These arguments are only provided to improve efficiency during repeated searches on the same datasets (e.g., for comparisons between all pairs). The specification of these arguments should not, generally speaking, alter the output of the function.

Value

A list containing the integer vectors first and second, containing row indices from data1 and data2 respectively. Corresponding entries in first and second specify a MNN pair consisting of the specified rows from each matrix.

Author(s)

Aaron Lun

See Also

queryKNN for the underlying neighbor search code.

fastMNN and related functions from the **batchelor** package, from which this code was originally derived.

Examples

```
B1 <- matrix(rnorm(10000), ncol=50) # Batch 1
B2 <- matrix(rnorm(10000), ncol=50) # Batch 2
out <- findMutualNN(B1, B2, k1=20)
head(out$first)
head(out$second)
```

---

findNeighbors *Find all neighbors in range*

Description

Find all neighbors within a given distance for each point in a data set.

Usage

```
findNeighbors(X, threshold, ..., BNINDEX, BNPARAM)
```

Arguments

X	A numeric data matrix where rows are points and columns are dimensions. This can be missing if BNINDEX is supplied.
threshold	A numeric scalar or vector specifying the maximum distance for considering neighbors.
...	Further arguments to pass to specific methods. This is guaranteed to include subset, get.index, get.distance BPPARAM and raw.index. See ?"findNeighbors-functions" for more details.
BNINDEX	A BiocNeighborIndex object containing precomputed index information. This can be missing if X and BNPARAM is supplied, see Details.
BNPARAM	A BiocNeighborParam object specifying the algorithm to use. This can be missing if BNINDEX is supplied, see Details.

Details

The class of BNINDEX and BNPARAM will determine the dispatch to specific functions. Only one of these arguments needs to be defined to resolve dispatch. However, if both are defined, they cannot specify different algorithms.

If BNINDEX is supplied, X does not need to be specified. In fact, any value of X will be ignored as all necessary information for the search is already present in BNINDEX. Similarly, any parameters in BNPARAM will be ignored.

If both BNINDEX and BNPARAM are missing, the function will default to the KMKNN algorithm by setting BNPARAM=KmknnParam().

Value

A list is returned containing index, a list of integer vectors specifying the identities of the neighbors of each point; and distance, a list of numeric vectors containing the distances to those neighbors. See ?"findNeighbors-functions" for more details.

Author(s)

Aaron Lun

See Also

rangeFindKmknn and rangeFindVptree for specific methods.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
k.out <- findNeighbors(Y, threshold=3)
a.out <- findNeighbors(Y, threshold=3, BNPARAM=VptreeParam())

k.dex <- buildKmknn(Y)
k.out2 <- findNeighbors(Y, threshold=3, BNINDEX=k.dex)
k.out3 <- findNeighbors(Y, threshold=3, BNINDEX=k.dex, BNPARAM=KmknnParam())
```

```
v.dex <- buildVptree(Y)
v.out2 <- findNeighbors(Y, threshold=3, BNINDEX=v.dex)
v.out3 <- findNeighbors(Y, threshold=3, BNINDEX=v.dex, BNPARAM=VptreeParam())
```

`findNeighbors-functions`

Find all neighbors in range

Description

Find all neighboring data points within a certain distance of each point.

Usage

```
rangeFindExhaustive(
  X,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  subset = NULL,
  raw.index = FALSE,
  ...
)

rangeFindKmknn(
  X,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  subset = NULL,
  raw.index = FALSE,
  ...
)

rangeFindVptree(
  X,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  subset = NULL,
```

```
  raw.index = FALSE,
  ...
)
```

Arguments

X	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
threshold	A positive numeric scalar specifying the maximum distance at which a point is considered a neighbor. Alternatively, a vector containing a different distance threshold for each point.
get.index	A logical scalar indicating whether the indices of the neighbors should be recorded.
get.distance	A logical scalar indicating whether distances to the neighbors should be recorded.
BPPARAM	A BiocParallelParam object indicating how the search should be parallelized.
precomputed	A BiocNeighborIndex object of the appropriate class, generated from X. For rangeFindExhaustive, this should be a ExhaustiveIndex from rangeFindExhaustive. For rangeFindKmknn, this should be a KmknnIndex from rangeFindKmknn. For rangeFindVptree, this should be a VptreeIndex from rangeFindVptree.
subset	A vector indicating the rows of X for which the nearest neighbors should be identified.
raw.index	A logial scalar indicating whether raw column indices should be returned, see ?"BiocNeighbors-raw-index".
...	Further arguments to pass to the respective build* function for each algorithm. This includes distance, a string specifying whether "Euclidean", "Manhattan" or "Cosine" distances are to be used.

Details

This function identifies all points in X that within threshold of each point in X. For Euclidean distances, this is equivalent to identifying all points in a hypersphere centered around the point of interest. The exact implementation can either use the KMKNNN approach or a VP tree.

By default, a search is performed for each data point in X, but it can be limited to a specified subset of points with subset. This yields the same result as (but is more efficient than) subsetting the output matrices after running findNeighbors with subset=NULL.

If threshold is a vector, each entry is assumed to specify a (possibly different) threshold for each point in X. If subset is also specified, each entry is assumed to specify a threshold for each point in subset. An error will be raised if threshold is a vector of incorrect length.

Turning off get.index or get.distance will provide a slight speed boost and reduce memory usage when these returned values are not of interest. If both get.index=FALSE and get.distance=FALSE, an integer vector containing the number of neighbors to each point is returned instead. This is more memory efficient when the identities of/distances to the neighbors are not required.

Using BPPARAM will parallelize the search across points, which usually provides a linear increase in speed.

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X (e.g., with buildKmknn). The resulting BiocNeighborIndex object can be supplied as precomputed

to multiple function calls, avoiding the need to repeat index construction in each call. Note that when precomputed is supplied, the value of X is ignored.

Value

A list is returned containing:

- index, if get.index=TRUE. This is a list of integer vectors where each entry corresponds to a point (denoted here as i) in X. The vector for i contains the set of row indices of all points in X that lie within threshold of point i. Points in each vector are not ordered, and i will always be included in its own set.
- distance, if get.distance=TRUE. This is a list of numeric vectors where each entry corresponds to a point (as above) and contains the distances of the neighbors from i. Elements of each vector in distance match to elements of the corresponding vector in index.

If get.index=FALSE and get.distance=FALSE, an integer vector is returned instead containing the number of neighbors to i.

If subset is not NULL, each entry of the above lists corresponds to a point in the subset, in the same order as supplied in subset.

See ?"BiocNeighbors-raw-index" for an explanation of the output when raw.index=TRUE.

Author(s)

Aaron Lun

See Also

buildExhaustive, buildKmknn or buildVptree to build an index ahead of time.

See ?"BiocNeighbors-algorithms" for an overview of the available algorithms.

Examples

```
Y <- matrix(runif(100000), ncol=20)
out <- rangeFindKmknn(Y, threshold=3)
out2 <- rangeFindVptree(Y, threshold=3)
out3 <- rangeFindExhaustive(Y, threshold=3)
```

HnswIndex *The HnswIndex class*

Description

A class to hold indexing structures for the HNSW algorithm for approximate nearest neighbor identification.

Usage

```
HnswIndex(data, path, ef.search = 10, NAMES = NULL, distance = "Euclidean")
```

Arguments

data	A numeric matrix with data points in columns and dimensions in rows.
path	A string specifying the path to the index file.
ef.search	Integer scalar specifying the size of the dynamic list at run time.
NAMES	A character vector of sample names or NULL.
distance	A string specifying the distance metric to use.

Details

The HnswIndex class holds the indexing structure required to run the HNSW algorithm. Users should never need to call the constructor explicitly, but should generate instances of HnswIndex classes with buildHnsw.

Users can get values from an HnswIndex object with the usual [[syntax. All parameters listed in the constructor can be extracted in this manner.

Value

An instance of the HnswIndex class.

Author(s)

Aaron Lun

See Also

buildHnsw, to build the index.

BiocNeighborIndex, for the parent class and its available methods.

Examples

```
example(buildHnsw)
out[['path']]
```

HnswParam *The HnswParam class*

Description

A class to hold parameters for the Hnsw algorithm for approximate nearest neighbor identification.

Usage

```
HnswParam(
  nlinks = 16,
  ef.construction = 200,
  directory = tempdir(),
  ef.search = 10,
  distance = "Euclidean"
)
```

Arguments

nlinks	Integer scalar, number of bi-directional links per element for index generation.
ef.construction	Integer scalar, size of the dynamic list for index generation.
directory	String specifying the directory in which to save the index.
ef.search	Integer scalar, size of the dynamic list for neighbor searching.
distance	A string specifying the distance metric to use.

Details

The HnswParam class holds any parameters associated with running the HNSW algorithm. This generally relates to building of the index - see buildHnsw for details.

Users can get or set values with the usual [[syntax. All parameters listed in the constructor can be manipulated in this manner.

Value

An instance of the HnswParam class.

Author(s)

Aaron Lun

See Also

buildHnsw, for the index construction.

findHnsw and related functions, for the actual search.

BiocNeighborParam, for the parent class and its available methods.

Examples

```
(out <- HnswParam())
out[['nlinks']]

out[['nlinks']] <- 20L
out
```

KmknnIndex *The KmknnIndex class*

Description

A class to hold indexing structures for the KMKNN algorithm for exact nearest neighbor identification.

Usage

```
KmknnIndex(data, centers, info, order, NAMES = NULL, distance = "Euclidean")
```

Arguments

data	A numeric matrix where columns correspond to data points and rows correspond to dimensions.
centers	A numeric matrix containing coordinates for cluster centroids, with clusters in columns and dimensions in rows.
info	A list containing additional information for each cluster, see buildKmknn for details.
order	An integer vector of length equal to ncol(data), specifying the order of points in x relative to the original data matrix.
NAMES	A character vector of sample names or NULL.
distance	A string specifying the distance metric to use.

Details

The KmknnIndex class holds the indexing structure required to run the KMKNN algorithm. Users should never need to call the constructor explicitly, but should generate instances of KmknnIndex classes with buildKmknn.

Users can get values from an HnswIndex object with the usual [[syntax. All parameters listed in the constructor can be extracted in this manner.

Value

An instance of the KmknnIndex class.

Author(s)

Aaron Lun

See Also

buildKmknn, to build the index.

BiocNeighborIndex, for the parent class and its available methods.

Examples

```
example(buildKmknn)
out[['centers']]
out[['info']]
```

KmknnParam *The KmknnParam class*

Description

A class to hold parameters for the KMKNN algorithm for exact nearest neighbor identification.

Usage

```
KmknnParam(..., distance = "Euclidean")
```

Arguments

`...`	Arguments to be passed to `kmeans`.
`distance`	A string specifying the distance metric to use.

Details

The KmknnParam class holds any parameters associated with running the KMKNN algorithm. Currently, this relates to tuning of the k-means step - see `buildKmknn` for details.

Users can get or set values from an KmknnParam object with the usual `[[` syntax. All parameters listed in `...` are available via `x[['kmeans.args']]`.

Value

An instance of the KmknnParam class.

Author(s)

Aaron Lun

See Also

`buildKmknn`, for the index construction.

`findKmknn` and related functions, for the actual search.

BiocNeighborParam, for the parent class and its available methods.

Examples

```
(out <- KmknnParam(iter.max=100))
out[['kmeans.args']]
```

queryKNN *Query k-nearest neighbors*

Description

Find the k-nearest neighbors in one data set for each point in another query data set, using exact or approximate algorithms.

Usage

```
queryKNN(X, query, k, ..., BNINDEX, BNPARAM)
```

Arguments

X	A numeric data matrix where rows are points and columns are dimensions. This can be missing if BNINDEX is supplied.
query	A numeric query matrix where rows are points and columns are dimensions.
k	An integer scalar specifying the number of nearest neighbors to search for.
...	Further arguments to pass to specific methods. This is guaranteed to include subset, get.index, get.distance, last, transposed, warn.ties, raw.index and BPPARAM. See ?"queryKNN-functions" for more details.
BNINDEX	A BiocNeighborIndex object containing precomputed index information. This can be missing if X and BNPARAM is supplied, see Details.
BNPARAM	A BiocNeighborParam object specifying the algorithm to use. This can be missing if BNINDEX is supplied, see Details.

Details

The class of BNINDEX and BNPARAM will determine dispatch to specific methods. Only one of these arguments needs to be defined to resolve dispatch. However, if both are defined, they cannot specify different algorithms.

If BNINDEX is supplied, X does not need to be specified. In fact, any value of X will be ignored as all necessary information for the search is already present in BNINDEX. Similarly, any parameters in BNPARAM will be ignored.

If both BNINDEX and BNPARAM are missing, the function will default to the KMKNN algorithm by setting BNPARAM=KmknnParam().

Value

A list is returned containing index, an integer matrix of neighbor identities; and distance, a numeric matrix of distances to those neighbors. See ?"queryKNN-functions" for more details.

Author(s)

Aaron Lun

See Also

queryExhaustive, queryKmknn, queryVptree, queryAnnoy and queryHnsw for specific methods.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
Z <- matrix(rnorm(10000), ncol=20)
str(k.out <- queryKNN(Y, Z, k=10))
str(a.out <- queryKNN(Y, Z, k=10, BNPARAM=AnnoyParam()))

e.dex <- buildExhaustive(Y)
str(k.out2 <- queryKNN(Y,Z,  k=10, BNINDEX=e.dex))
str(k.out3 <- queryKNN(Y,Z,  k=10, BNINDEX=e.dex, BNPARAM=ExhaustiveParam()))

k.dex <- buildKmknn(Y)
str(k.out2 <- queryKNN(Y,Z,  k=10, BNINDEX=k.dex))
str(k.out3 <- queryKNN(Y,Z,  k=10, BNINDEX=k.dex, BNPARAM=KmknnParam()))

a.dex <- buildAnnoy(Y)
str(a.out2 <- queryKNN(Y,Z,  k=10, BNINDEX=a.dex))
str(a.out3 <- queryKNN(Y,Z,  k=10, BNINDEX=a.dex, BNPARAM=AnnoyParam()))
```

---

queryKNN-functions *Query nearest neighbors*

Description

Query a dataset for nearest neighbors of points in another dataset, using a variety of algorithms.

Usage

```
queryAnnoy(
  X,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
  subset = NULL,
  raw.index = NA,
  warn.ties = NA,
  ...
)
```

```
queryHnsw(
  X,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
  subset = NULL,
  raw.index = NA,
  warn.ties = NA,
  ...
)

queryKmknn(
  X,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
  subset = NULL,
  raw.index = FALSE,
  warn.ties = TRUE,
  ...
)

queryVptree(
  X,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
  subset = NULL,
  raw.index = FALSE,
  warn.ties = TRUE,
  ...
)
```

```
queryExhaustive(
  X,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  last = k,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
  subset = NULL,
  raw.index = FALSE,
  warn.ties = TRUE,
  ...
)
```

Arguments

X	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
query	A numeric matrix of query points, containing different data points in the rows but the same number and ordering of dimensions in the columns.
k	A positive integer scalar specifying the number of nearest neighbors to retrieve.
get.index	A logical scalar indicating whether the indices of the nearest neighbors should be recorded.
get.distance	A logical scalar indicating whether distances to the nearest neighbors should be recorded.
last	An integer scalar specifying the number of furthest neighbors for which statistics should be returned.
BPPARAM	A BiocParallelParam object indicating how the search should be parallelized.
precomputed	A BiocNeighborIndex object of the appropriate class, generated from X. For findExhaustive, this should be a ExhaustiveIndex from buildExhaustive; For findKmknn, this should be a KmknnIndex from buildKmknn; for findVptree, this should be a VptreeIndex from buildVptree; for findAnnoy, this should be a AnnoyIndex from buildAnnoy; and for findHnsw, this should be a HnswIndex from buildHnsw.
transposed	A logical scalar indicating whether the query is transposed, in which case query is assumed to contain dimensions in the rows and data points in the columns.
subset	A vector indicating the rows of query (or columns, if transposed=TRUE) for which the nearest neighbors should be identified.
raw.index	A logial scalar indicating whether raw column indices should be returned, see ?"BiocNeighbors-raw-index". This argument is ignored for findAnnoy and findHnsw.
warn.ties	Logical scalar indicating whether a warning should be raised if any of the k+1 neighbors have tied distances. This argument is ignored for findAnnoy and findHnsw.

... Further arguments to pass to the respective build* function for each algorithm. This includes distance, a string specifying whether "Euclidean", "Manhattan" or "Cosine" distances are to be used.

Details

All of these functions identify points in X that are the k nearest neighbors of each point in query. queryAnnoy performs an approximate search, while queryExhaustive, queryKmknn and queryVptree are exact. This requires both X and query to have the same number of dimensions. Moreover, the upper bound for k is set at the number of points in X.

By default, nearest neighbors are identified for all data points within query. If subset is specified, nearest neighbors are only detected for the query points in the subset. This yields the same result as (but is more efficient than) subsetting the output matrices after running queryKmknn on the full query.

If transposed=TRUE, this function assumes that query is already transposed, which saves a bit of time by avoiding an unnecessary transposition. Turning off get.index or get.distance may also provide a slight speed boost when these returned values are not of interest. Using BPPARAM will also split the search by query points across multiple processes.

Setting last will return indices and/or distances for the k -last + 1-th closest neighbor to the k-th neighbor. This can be used to improve memory efficiency, e.g., by only returning statistics for the k-th nearest neighbor by setting last=1. Note that this is entirely orthogonal to subset.

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X (e.g., with buildKmknn). The resulting BiocNeighborIndex object can be supplied as precomputed to multiple function calls, avoiding the need to repeat index construction in each call. Note that when precomputed is supplied, the value of X is ignored.

For exact methods, see comments in ?"BiocNeighbors-ties" regarding the warnings when tied distances are observed. For approximate methods, see comments in buildAnnoy and buildHnsw about the (lack of) randomness in the search results.

Value

A list is returned containing:

- index, if get.index=TRUE. This is an integer matrix where each row corresponds to a point (denoted here as i) in query. The row for i contains the row indices of X that are the nearest neighbors to point i, sorted by increasing distance from i.
- distance, if get.distance=TRUE. This is a numeric matrix where each row corresponds to a point (as above) and contains the sorted distances of the neighbors from i.

Each matrix contains last columns. If subset is not NULL, each row of the above matrices refers to a point in the subset, in the same order as supplied in subset.

See ?"BiocNeighbors-raw-index" for an explanation of the output when raw.index=TRUE for the functions that support it.

Author(s)

Aaron Lun

See Also

buildExhaustive, buildKmknn, buildVptree, or buildAnnoy to build an index ahead of time.

See ?"BiocNeighbors-algorithms" for an overview of the available algorithms.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
Z <- matrix(rnorm(20000), ncol=20)

out <- queryExhaustive(Y, query=Z, k=5)
head(out$index)
head(out$distance)

out1 <- queryKmknn(Y, query=Z, k=5)
head(out1$index)
head(out1$distance)

out2 <- queryVptree(Y, query=Z, k=5)
head(out2$index)
head(out2$distance)

out3 <- queryAnnoy(Y, query=Z, k=5)
head(out3$index)
head(out3$distance)

out4 <- queryHnsw(Y, query=Z, k=5)
head(out4$index)
head(out4$distance)
```

---

queryNeighbors *Query all neighbors*

Description

Find all neighbors in one data set that are in range of each point in another query data set.

Usage

```
queryNeighbors(X, query, threshold, ..., BNINDEX, BNPARAM)
```

Arguments

X	A numeric data matrix where rows are points and columns are dimensions. This can be missing if BNINDEX is supplied.
query	A numeric query matrix where rows are points and columns are dimensions.
threshold	A numeric scalar or vector specifying the maximum distance for considering neighbors.

...	Further arguments to pass to specific methods. This is guaranteed to include subset, get.index, get.distance BPPARAM and raw.index. See ?"queryNeighbors-functions" for more details.
BNINDEX	A BiocNeighborIndex object containing precomputed index information. This can be missing if X and BNPARAM is supplied, see Details.
BNPARAM	A BiocNeighborParam object specifying the algorithm to use. This can be missing if BNINDEX is supplied, see Details.

Details

The class of BNINDEX and BNPARAM will determine dispatch to specific methods. Only one of these arguments needs to be defined to resolve dispatch. However, if both are defined, they cannot specify different algorithms.

If BNINDEX is supplied, X does not need to be specified. In fact, any value of X will be ignored as all necessary information for the search is already present in BNINDEX. Similarly, any parameters in BNPARAM will be ignored.

If both BNINDEX and BNPARAM are missing, the function will default to the KMKNN algorithm by setting BNPARAM=KmknnParam().

Value

A list is returned containing index, a list of integer vectors specifying the identities of the neighbors of each point; and distance, a list of numeric vectors containing the distances to those neighbors. See ?"queryNeighbors-functions" for more details.

Author(s)

Aaron Lun

See Also

rangeQueryKmknn and rangeQueryVptree for specific methods.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
Z <- matrix(rnorm(10000), ncol=20)
k.out <- queryNeighbors(Y, Z, threshold=3)
v.out <- queryNeighbors(Y, Z, threshold=3, BNPARAM=VptreeParam())

k.dex <- buildKmknn(Y)
k.out2 <- queryNeighbors(Y,Z,  threshold=3, BNINDEX=k.dex)
k.out3 <- queryNeighbors(Y,Z,  threshold=3, BNINDEX=k.dex, BNPARAM=KmknnParam())

v.dex <- buildVptree(Y)
v.out2 <- queryNeighbors(Y,Z,  threshold=3, BNINDEX=v.dex)
v.out3 <- queryNeighbors(Y,Z,  threshold=3, BNINDEX=v.dex, BNPARAM=VptreeParam())
```

queryNeighbors-functions
Query neighbors in range

Description

Find all neighboring data points within a certain distance of a query point.

Usage

```
rangeQueryExhaustive(
  X,
  query,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
  subset = NULL,
  raw.index = FALSE,
  ...
)

rangeQueryKmknn(
  X,
  query,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
  subset = NULL,
  raw.index = FALSE,
  ...
)

rangeQueryVptree(
  X,
  query,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  BPPARAM = SerialParam(),
  precomputed = NULL,
  transposed = FALSE,
```

```
  subset = NULL,
  raw.index = FALSE,
  ...
)
```

Arguments

X	A numeric matrix where rows correspond to data points and columns correspond to variables (i.e., dimensions).
query	A numeric matrix of query points, containing different data points in the rows but the same number and ordering of dimensions in the columns.
threshold	A positive numeric scalar specifying the maximum distance at which a point is considered a neighbor. Alternatively, a vector containing a different distance threshold for each query point.
get.index	A logical scalar indicating whether the indices of the neighbors should be recorded.
get.distance	A logical scalar indicating whether distances to the neighbors should be recorded.
BPPARAM	A BiocParallelParam object indicating how the search should be parallelized.
precomputed	A BiocNeighborIndex object of the appropriate class, generated from X. For rangeFindExhaustive, this should be a ExhaustiveIndex from rangeFindExhaustive. For rangeFindKmknn, this should be a KmknnIndex from rangeFindKmknn. For rangeFindVptree, this should be a VptreeIndex from rangeFindVptree.
transposed	A logical scalar indicating whether the query is transposed, in which case query is assumed to contain dimensions in the rows and data points in the columns.
subset	A vector indicating the rows of query (or columns, if transposed=TRUE) for which the neighbors should be identified.
raw.index	A logial scalar indicating whether raw column indices should be returned, see ?"BiocNeighbors-raw-index".
...	Further arguments to pass to the respective build* function for each algorithm. This includes distance, a string specifying whether "Euclidean", "Manhattan" or "Cosine" distances are to be used.

Details

This function identifies points in X that are neighbors (i.e., within a distance threshold) of each point in query. The exact implementation can either use the KMKNNN approach or a VP tree. This requires both X and query to have the same number of variables.

By default, neighbors are identified for all data points within query. If subset is specified, neighbors are only detected for the query points in the subset. This yields the same result as (but is more efficient than) subsetting the output matrices after running queryNeighbors on the full query.

If threshold is a vector, each entry is assumed to specify a (possibly different) threshold for each point in query. If subset is also specified, each entry is assumed to specify a threshold for each point in subset. An error will be raised if threshold is a vector of incorrect length.

Turning off get.index or get.distance will provide a slight speed boost and reduce memory usage when those returned values are not of interest. If both get.index=FALSE and get.distance=FALSE,

an integer vector containing the number of neighbors to each point is returned instead, which is more memory efficient when the identities of/distances to the neighbors are not required.

If `transposed=TRUE`, this function assumes that `query` is already transposed, which saves a bit of time by avoiding an unnecessary transposition. Using `BPPARAM` will also split the search by query points across multiple processes.

If multiple queries are to be performed to the same `X`, it may be beneficial to build the index from `X` (e.g., with `buildKmknn`). The resulting BiocNeighborIndex object can be supplied as `precomputed` to multiple function calls, avoiding the need to repeat index construction in each call. Note that when `precomputed` is supplied, the value of `X` is ignored.

Value

A list is returned containing:

- `index`, if `get.index=TRUE`. This is a list of integer vectors where each entry corresponds to a point (denoted here as i) in `query`. The vector for i contains the set of row indices of all points in `X` that lie within `threshold` of point i. Points in each vector are not ordered, and i will always be included in its own set.
- `distance`, if `get.distance=TRUE`. This is a list of numeric vectors where each entry corresponds to a point (as above) and contains the distances of the neighbors from i. Elements of each vector in `distance` match to elements of the corresponding vector in `index`.

If `get.index=FALSE` and `get.distance=FALSE`, an integer vector is returned instead containing the number of neighbors to i.

If `subset` is not `NULL`, each entry of the above lists refers to a point in the subset, in the same order as supplied in `subset`.

See ?"BiocNeighbors-raw-index" for an explanation of the output when `raw.index=TRUE`.

Author(s)

Aaron Lun

See Also

`buildKmknn` or `buildVptree` to build an index ahead of time.

See ?"BiocNeighbors-algorithms" for an overview of the available algorithms.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
Z <- matrix(rnorm(20000), ncol=20)

out <- rangeQueryKmknn(Y, query=Z, threshold=1)
head(out$index)
head(out$distance)

out2 <- rangeQueryVptree(Y, query=Z, threshold=1)
head(out2$index)
head(out2$distance)
```

```
out3 <- rangeQueryExhaustive(Y, query=Z, threshold=1)
head(out3$index)
head(out3$distance)
```

VptreeIndex *The VptreeIndex class*

Description

A class to hold the vantage point tree for exact nearest neighbor identification.

Usage

```
VptreeIndex(data, nodes, order, NAMES = NULL, distance = "Euclidean")
```

Arguments

data	A numeric matrix with data points in columns and dimensions in rows.
nodes	A list of vectors specifying the structure of the VP tree.
order	An integer vector of length equal to ncol(data), specifying the order of observations.
NAMES	A character vector of sample names or NULL.
distance	A string specifying the distance metric to use.

Details

The VptreeIndex class holds the indexing structure required to run the VP tree algorithm. Users should never need to call the constructor explicitly, but should generate instances of VptreeIndex classes with buildVptree.

Users can get values from a VptreeIndex object with the usual [[syntax. All parameters listed in the constructor can be extracted in this manner.

Value

An instance of the VptreeIndex class.

Author(s)

Aaron Lun

See Also

buildVptree, for the index construction.

BiocNeighborIndex, for the parent class and its available methods.

Examples

```
example(buildVptree)
str(out[["nodes"]])
```

`VptreeParam` *The VptreeParam class*

Description

A class to hold parameters for the VP tree algorithm for exact nearest neighbor identification.

Usage

```
VptreeParam(distance = "Euclidean")
```

Arguments

`distance`	A string specifying the distance metric to use.

Value

An instance of the VptreeParam class.

Author(s)

Aaron Lun

See Also

`buildVptree`, for the index construction.

`findVptree` and related functions, for the actual search.

BiocNeighborParam, for the parent class and its available methods.

Examples

```
(out <- VptreeParam())
```

Index