Package 'BiocNeighbors'

April 6, 2026

Version 2.5.4

Date 2026-02-13

Title Nearest Neighbor Detection for Bioconductor Packages

Imports Rcpp, methods

Suggests Matrix, DelayedArray, beachmat, BiocParallel, testthat, BiocStyle, knitr, rmarkdown

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, vantage point trees, or an exhaustive search. Approximate searches can be performed using the Annoy or HNSW libraries. Each search can be performed with a variety of different distance metrics, parallelization, and variable numbers of neighbors. Range-based searches (to find all neighbors within a certain distance) are also supported.

License GPL-3

LinkingTo Rcpp, assorthead, beachmat

VignetteBuilder knitr

SystemRequirements C++17

RoxygenNote 7.3.3

Encoding UTF-8

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

git_branch devel

git_last_commit 2d1b405

git_last_commit_date 2026-02-12

Repository Bioconductor 3.23

Date/Publication 2026-04-06

Author Aaron Lun [aut, cre, cph]

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

Contents

`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, vantage point trees, or an exhaustive search. Approximate searches can be performed using the Annoy or HNSW libraries. Each search can be performed with a variety of different distance metrics, parallelization, and variable numbers of neighbors. Range-based searches (to find all neighbors within a certain distance) are also supported.

Author(s)

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

AnnoyParam *The AnnoyParam class*

Description

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

Usage

```
AnnoyParam(
  ntrees = 50,
  search.mult = ntrees,
  distance = c("Euclidean", "Manhattan", "Cosine")
)

## S4 method for signature 'AnnoyParam'
defineBuilder(BNPARAM)
```

Arguments

ntrees	Integer scalar, number of trees to use for index generation.
search.mult	Numeric scalar, multiplier for the number of points to search.
distance	String specifying the distance metric to use. Cosine distances are implemented as Euclidean distances on L2-normalized coordinates.
BNPARAM	An AnnoyParam instance.

Details

The Approximate nearest neighbors Oh Yeah (Annoy) algorithm is based on recursive hyperplane partitions. 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.

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

The `AnnoyParam` constructor returns an instance of the AnnoyParam class.

The `defineBuilder` method returns a list that can be used in `buildIndex` to construct an Annoy index.

Author(s)

Aaron Lun

See Also

BiocNeighborParam, for the parent class and its available methods.

`https://github.com/spotify/annoy`, for details on the underlying algorithm.

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. Developers should define subclasses for their own `buildIndex` and/or `defineBuilder` methods.

Details

In general, the internal structure of a BiocNeighborIndex class is arbitrary and left to the discretion of the developer. If an arbitrary structure is used, the associated methods should be written for all downstream generics like `findKNN`, etc.

Alternatively, developers may choose to derive from the BiocNeighborGenericIndex class. This expects:

- A `ptr` slot containing an external pointer that refers to a `BiocNeighbors::Prebuilt` object (see definition in `system.file("include", "BiocNeighbors.h", package="BiocNeighbors")`).
- A `names` slot containing a character vector with the names of the observations, or `NULL` if no names are available. This is used by `subset=` in the various `find*` generics.

In this case, no additional methods are required for the downstream generics.

Author(s)

Aaron Lun

`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 5 concrete subclasses in **BiocNeighbors**:

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

`VptreeParam`**:** Exact nearest-neighbor search with the tree algorithm.

`ExhaustiveParam`**:** Exact nearest-neighbor search via brute-force.

`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.

buildIndex *Build a nearest-neighbor index*

Description

Build indices for nearest-neighbor searching with different algorithms.

Usage

```
buildIndex(X, BNPARAM, transposed = FALSE, ...)

## S4 method for signature 'missing'
buildIndex(X, BNPARAM, transposed = FALSE, ...)

## S4 method for signature 'NULL'
buildIndex(X, BNPARAM, transposed = FALSE, ...)

## S4 method for signature 'BiocNeighborParam'
buildIndex(X, BNPARAM, transposed = FALSE, ...)

## S4 method for signature 'list'
buildIndex(X, BNPARAM, transposed = FALSE, ..., .check.nonfinite = TRUE)
```

Arguments

X	A numeric matrix or matrix-like object where rows correspond to data points and columns correspond to variables (i.e., dimensions).
BNPARAM	A BiocNeighborParam object specifying the type of index to be constructed. If NULL or missing, this defaults to a KmknnParam object. Alternatively, this may be a list returned by defineBuilder.
transposed	Logical scalar indicating whether X is transposed, i.e., rows are variables and columns are data points.
...	Further arguments to be passed to individual methods. If a method accepts arguments here, it should prefix these arguments with the algorithm name to avoid conflicts, e.g., vptree.foo.bar.
.check.nonfinite	Boolean indicating whether to check for non-finite values in X. This can be set to FALSE for greater efficiency.

Details

Each buildIndex method is expected to return an instance of a BiocNeighborIndex subclass. The structure of this subclass is arbitrary and left to the discretion of the method developer.

If a buildIndex method returns a BiocNeighborGenericIndex subclass, the index can be used with the existing methods for findKnnFromIndex, etc. without further effort. Otherwise, developers are responsible for defining methods for their subclass in each of the relevant generics.

Users should assume that the index is not serializable (i.e., saved or transferred between processes) via R's usual mechanisms. If the index must be saved to disk, consider using saveIndex instead.

Value

A BiocNeighborIndex object can be used in findKNN and related functions as the X= argument, or in findKnnFromIndex and related generics as the BNINDEX= argument.

Author(s)

Aaron Lun

Examples

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

---

defineBuilder *Define an index builder*

Description

Define a builder object that can construct C++ indices for neighbor searches.

Usage

```
defineBuilder(BNPARAM)

## S4 method for signature 'NULL'
defineBuilder(BNPARAM)

## S4 method for signature 'missing'
defineBuilder(BNPARAM)
```

Arguments

BNPARAM	A BiocNeighborParam object specifying the type of index to be constructed. If NULL, this defaults to a KmknnParam object.

Details

The external pointer returned in builder should refer to a BiocNeighbors::Builder object, see the definition in system.file("include", "BiocNeighbors.h", package="BiocNeighbors") for details. If a developer defines a defineBuilder method for a search algorithm, they do not have to define a new buildIndex method. The existing buildIndex methods will automatically create an instance of the appropriate BiocNeighborGenericIndex subclass based on class, which can be immediately used in all generics (e.g., findKNN, queryNeighbors) without further effort.

Note that the pointer returned by `defineBuilder` should *not* be used as the `ptr` in the BiocNeighborIndex subclasses. The `ptr` slot is expected to contain a pointer referring to a `BiocNeighbors::Prebuilt` object, as returned by the default `buildIndex`. Using the pointer from `builder` will probably crash the R session.

Needless to say, users should not attempt to serialize the external pointer returned by this generic. Attempting to use a deserialized pointer in `buildIndex` will cause the R session to crash.

Value

List containing:

- `builder`, a pointer to a builder instance that can be used to construct a prebuilt index in `buildIndex`.
- `class`, the constructor for a BiocNeighborGenericIndex subclass that accepts `ptr` and `names` arguments.

Author(s)

Aaron Lun

See Also

`defineBuilder,KmknnParam-method`, `defineBuilder,VptreeParam-method`, `defineBuilder,AnnoyParam-method` and `defineBuilder,HnswParam-method` for specific methods.

Examples

```
(out <- defineBuilder())
(out2 <- defineBuilder(AnnoyParam()))
```

ExhaustiveParam *The ExhaustiveParam class*

Description

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

Usage

```
ExhaustiveParam(distance = c("Euclidean", "Manhattan", "Cosine"))

## S4 method for signature 'ExhaustiveParam'
defineBuilder(BNPARAM)
```

Arguments

distance	String specifying the distance metric to use. Cosine distances are implemented as Euclidean distances on L2-normalized coordinates.
BNPARAM	An ExhaustiveParam instance.

Details

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.

All that said, this algorithm is largely provided as a baseline for comparing against the other algorithms.

Value

The `ExhaustiveParam` constructor returns an instance of the ExhaustiveParam class.

The `defineBuilder` method returns an external pointer that can be used in `buildIndex` to construct an exhaustive index.

Author(s)

Allison Vuong

See Also

BiocNeighborParam, for the parent class and its available methods.

Examples

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

---

findDistanceFromIndex *Distance to the k-th nearest neighbor*

Description

Find the distance to the k-th nearest neighbor for each point in a dataset.

Usage

```
findDistanceFromIndex(BNINDEX, k, num.threads = 1, subset = NULL, ...)

## S4 method for signature 'BiocNeighborGenericIndex'
findDistanceFromIndex(BNINDEX, k, num.threads = 1, subset = NULL, ...)

findDistance(X, k, num.threads = 1, subset = NULL, ..., BNPARAM = NULL)
```

Arguments

BNINDEX	A BiocNeighborIndex object, typically created by buildIndex.
k	A positive integer scalar specifying the number of nearest neighbors to retrieve. Alternatively, an integer vector of length equal to the number of points in X, specifying the number of neighbors to identify for each point. If subset is provided, this should have length equal to the length of subset. Users should wrap this vector in an AsIs class to distinguish length-1 vectors from integer scalars. All k should be less than or equal to the number of points in X minus 1, otherwise the former will be capped at the latter with a warning.
num.threads	Integer scalar specifying the number of threads to use for the search.
subset	An integer, logical or character vector specifying the indices of points in X for which the nearest neighbors should be identified. This yields the same result as (but is more efficient than) subsetting the output matrices after computing neighbors for all points.
...	For findDistanceFromIndex, further arguments to pass to individual methods. If a method accepts arguments here, it should prefix these arguments with the algorithm name to avoid conflicts, e.g., vptree.foo.bar. For findDistance, further arguments to pass to findDistanceFromIndex. These are also passed to buildIndex when X is not an external pointer.
X	A numeric matrix or matrix-like object where rows correspond to data points and columns correspond to variables (i.e., dimensions). Alternatively, a prebuilt BiocNeighborIndex object from buildIndex.
BNPARAM	A BiocNeighborParam object specifying how the index should be constructed. If NULL, this defaults to a KmknnParam. Ignored if X contains a prebuilt index.

Details

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X with buildIndex. The resulting pointer object can be supplied as X to multiple findDistance calls, avoiding the need to repeat index construction in each call.

Value

Numeric vector of length equal to the number of points in X (or subset, if provided), containing the distance from each point to its k-th nearest neighbor. This is equivalent to but more memory efficient than using findKNN and subsetting to the last distance.

Author(s)

Aaron Lun

See Also

buildIndex, to build an index ahead of time.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
out <- findDistance(Y, k=8)
summary(out)
```

`findKnnFromIndex` *Find k-nearest neighbors*

Description

Find the k-nearest neighbors of each point in a dataset.

Usage

```
findKnnFromIndex(
  BNINDEX,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  ...
)

## S4 method for signature 'BiocNeighborGenericIndex'
findKnnFromIndex(
  BNINDEX,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  ...
)

findKNN(
  X,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  ...,
  BPPARAM = NULL,
  BNPARAM = NULL
)
```

Arguments

BNINDEX	A BiocNeighborIndex object, typically created by buildIndex.
k	A positive integer scalar specifying the number of nearest neighbors to retrieve. Alternatively, an integer vector of length equal to the number of points in X, specifying the number of neighbors to identify for each point. If subset is provided, this should have length equal to the length of subset. Users should wrap this vector in an AsIs class to distinguish length-1 vectors from integer scalars. All k should be less than or equal to the number of points in X minus 1, otherwise the former will be capped at the latter with a warning.
get.index	A logical scalar indicating whether the indices of the nearest neighbors should be recorded. Setting this to FALSE improves efficiency if the indices are not of interest. Alternatively, if k is an integer scalar, this may be a string containing "normal" or "transposed". The former is the same as TRUE, while the latter returns the index matrix in transposed format.
get.distance	A logical scalar indicating whether distances to the nearest neighbors should be recorded. Setting this to FALSE improves efficiency if the distances are not of interest. Alternatively, if k is an integer scalar, this may be a string containing "normal" or "transposed". The former is the same as TRUE, while the latter returns the distance matrix in transposed format.
num.threads	Integer scalar specifying the number of threads to use for the search.
subset	An integer, logical or character vector specifying the indices of points in X for which the nearest neighbors should be identified. This yields the same result as (but is more efficient than) subsetting the output matrices after computing neighbors for all points.
...	For findKnnFromIndex, further arguments to pass to individual methods. If a method accepts arguments here, it should prefix these arguments with the algorithm name to avoid conflicts, e.g., vptree.foo.bar. For findKNN, further arguments to pass to findKnnFromIndex. These are also passed to buildIndex when X is not an external pointer.
X	A numeric matrix or matrix-like object where rows correspond to data points and columns correspond to variables (i.e., dimensions). Alternatively, a prebuilt BiocNeighborIndex object from buildIndex.
BPPARAM	Soft-deprecated, use num.threads instead.
BNPARAM	A BiocNeighborParam object specifying how the index should be constructed. If NULL, this defaults to a KmknnParam. Ignored if X contains a prebuilt index.

Value

List containing index (if get.index is not FALSE) and distance (if get.distance is not FALSE).

- If get.index=TRUE or "normal" and k is an integer scalar, index is an integer matrix with k columns where each row corresponds to a point (denoted here as i) in X. The i-th row contains

the indices of points in X that are the nearest neighbors to point i, sorted by increasing distance from i. i will *not* be included in its own set of nearest neighbors.

If get.index=FALSE or "transposed" and k is an integer scalar, index is as described above but transposed, i.e., the i-th column contains the indices of neighboring points in X.

- If get.distance=TRUE or "normal" and k is an integer scalar, distance is a numeric matrix of the same dimensions as index. The i-th row contains the distances of neighboring points in X to the point i, sorted in increasing order.

  If get.distance=FALSE or "transposed" and k is an integer scalar, distance is as described above but transposed, i.e., the i-th column contains the distances to neighboring points in X.
- If get.index is not FALSE and k is an integer vector, index is a list of integer vectors where each vector corresponds to a point (denoted here as i) in X. The i-th vector has length k[i] and contains the indices of points in X that are the nearest neighbors to point i, sorted by increasing distance from i.
- If get.distance is not FALSE and k is an integer vector, distance is a list of numeric vectors of the same lengths as those in index. The i-th vector contains the distances of neighboring points in X to the point i, sorted in increasing order.

Author(s)

Aaron Lun

See Also

buildIndex, to build an index ahead of time.

findDistance, to efficiently obtain the distance to the k-th nearest neighbor.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
out <- findKNN(Y, k=8)
head(out$index)
head(out$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, ...)
```

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 pre-built index for data1. If NULL, this is constructed from data1 within the internal queryKNN call.
BNINDEX2	A pre-built index for data2. If NULL, this is constructed from data2 within the internal queryKNN call.
...	Other arguments to be passed to the underlying queryKNN calls, e.g., BNPARAM, .

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)
```

`findNeighborsFromIndex` *Find neighbors within a threshold distance*

Description

Find all neighbors within a threshold distance of each point of a dataset.

Usage

```
findNeighborsFromIndex(
  BNINDEX,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  ...
)

## S4 method for signature 'BiocNeighborGenericIndex'
findNeighborsFromIndex(
  BNINDEX,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  ...
)

findNeighbors(
  X,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  ...,
  BPPARAM = NULL,
  BNPARAM = NULL
)
```

Arguments

BNINDEX	A BiocNeighborIndex object, typically created by buildIndex.
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.
num.threads	Integer scalar specifying the number of threads to use for the search.
subset	An integer, logical or character vector specifying the indices of points in X for which the nearest neighbors should be identified. This yields the same result as (but is more efficient than) subsetting the output matrices after computing neighbors for all points.
...	For findNeighborsFromIndex, further arguments to pass to individual methods. If a method accepts arguments here, it should prefix these arguments with the algorithm name to avoid conflicts, e.g., vptree.foo.bar. For findNeighbors, further arguments to pass to findNeighborsFromIndex. These are also passed to buildIndex when X is not an external pointer.
X	A numeric matrix or matrix-like object where rows correspond to data points and columns correspond to variables (i.e., dimensions). Alternatively, a prebuilt BiocNeighborIndex object from buildIndex.
BPPARAM	Soft-deprecated, use num.threads instead.
BNPARAM	A BiocNeighborParam object specifying how the index should be constructed. If NULL, this defaults to a KmknnParam. Ignored if X contains a prebuilt index.

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. Not all implementations support this search mode, but we can use KmknnParam and VptreeParam.

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.

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X with buildIndex. The resulting pointer object can be supplied as X to multiple findNeighbors calls, avoiding the need to repeat index construction in each call.

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. Neighbors for i are sorted by increasing distance.

- 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 both get.index=FALSE and get.distance=FALSE, an integer vector is returned of length equal to the number of observations. The i-th entry contains the number of neighbors of i within threshold.

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

Author(s)

Aaron Lun

See Also

buildIndex, to build an index ahead of time.

Examples

```
Y <- matrix(runif(100000), ncol=20)
out <- findNeighbors(Y, threshold=1)
summary(lengths(out$index))
```

---

getLoadGenericIndexRegistry

Load indices from extension packages

Description

Extend loadIndex to work with algorithms from **BiocNeighbors** extension packages.

Usage

```
getLoadGenericIndexRegistry()

registerLoadGenericIndexClass(name, class)

registerLoadIndexFunction(name, fun)
```

Arguments

name	String containing the name of the neighbor search algorithm, e.g., "knncolle::Vptree".
class	Function that accepts a single argument (the external pointer to a knncolle::Prebuilt instance) and returns an instance of the appropriate BiocNeighborGenericIndex subclass.
fun	Function that accepts a directory path (see loadIndex) plus any number of additional arguments via ..., and returns an instance of the BiocNeighborIndex subclass corresponding to name.

Details

If the extension developer implements their search algorithm in the form of C++ subclasses of the various **knncolle** interfaces, they can call `getLoadGenericIndexRegistry` in their package's `.onLoad` to access the registry and add a loading method for their algorithm. They should also call `registerLoadGenericIndexClass` to specify a constructor function for the corresponding BiocNeighborGenericIndex subclass.

If the extension developer implements their search algorithm by subclassing **BiocNeighbors** generics at the R level, they should call `registerLoadIndexFunction` in their package's `.onLoad`. This function is responsible for reading the on-disk contents from the specified directory and using them to construct a BiocNeighborIndex instance.

Value

For `getLoadGenericIndexRegistry`, an external pointer to the global `knncolle::load_prebuilt_registry()` object in C++.

For `registerLoadGenericIndexClass`, the name and class are registered with `loadIndex`.

For `registerLoadIndexFunction`, the name and function are registered with `loadIndex`.

Author(s)

Aaron Lun

Examples

```
getLoadGenericIndexRegistry()
registerLoadGenericIndexClass("knncolle_annoy::Annoy", AnnoyIndex)
```

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,
  ef.search = 10,
  distance = c("Euclidean", "Manhattan", "Cosine")
)

## S4 method for signature 'HnswParam'
defineBuilder(BNPARAM)
```

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.
ef.search	Integer scalar, size of the dynamic list for neighbor searching.
distance	String specifying the distance metric to use. Cosine distances are implemented as Euclidean distances on L2-normalized coordinates.
BNPARAM	A HsnwParam instance.

Details

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.

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, where larger values improve accuracy at the expense of a slower 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

The `HnswParam` constructor returns an instance of the HnswParam class.

The `defineBuilder` method returns an external pointer that can be used in `buildIndex` to construct a HNSW index.

Author(s)

Aaron Lun

See Also

BiocNeighborParam, for the parent class and its available methods.

`https://github.com/nmslib/hnswlib`, for details on the underlying algorithm.

Examples

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

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

KmknnParam *The KmknnParam class*

Description

A class to hold parameters for the k-means k-nearest-neighbors (KMKNN) algorithm for exact nearest neighbor identification.

Usage

```
KmknnParam(..., distance = c("Euclidean", "Manhattan", "Cosine"))

## S4 method for signature 'KmknnParam'
defineBuilder(BNPARAM)
```

Arguments

...	Further arguments, ignored.
distance	String specifying the distance metric to use. Cosine distances are implemented as Euclidean distances on L2-normalized coordinates.
BNPARAM	A KmknnParam instance.

Details

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.

Note that KMKNN operates much more naturally with Euclidean distances. Computational efficiency may not be optimal when using it with other choices of `distance`, though the results will still be exact.

Value

The KmknnParam constructor returns an instance of the KmknnParam class.

The defineBuilder method returns a list that can be used in buildIndex to construct a KMKNN index.

Author(s)

Aaron Lun, using code from the **cydar** package.

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.

See Also

BiocNeighborParam, for the parent class and its available methods.

Examples

```
(out <- KmknnParam(iter.max=100))
```

---

loadIndex *Load a prebuilt search index from disk*

Description

Load a BiocNeighborIndex object from its on-disk representation back into the current R session.

Usage

```
loadIndex(dir, ...)
```

Arguments

dir	String containing the path to a directory in which the original index was saved. This should be the same as the argument passed to saveIndex.
...	Additional arguments passed to specific methods.

Details

As discussed in saveIndex, it is expected that the on-disk representation is loaded in the same R environment that was used to save it.

Developers are directed to getLoadGenericIndexRegistry to add loading functions for their own search algorithms.

Value

A BiocNeighborIndex object, created from files inside `dir`.

Author(s)

Aaron Lun

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
k.out <- buildIndex(Y)
k.nn <- findKNN(k.out, k=5)

tmp <- tempfile()
dir.create(tmp)
saveIndex(k.out, tmp)

reloaded <- loadIndex(tmp)
re.nn <- findKNN(reloaded, k=5)
identical(k.nn, re.nn)
```

`queryDistanceFromIndex` *Distance to the k-th nearest neighbor to query points*

Description

Query a reference dataset to determine the distance to the k-th nearest neighbor of each point in a query dataset.

Usage

```
queryDistanceFromIndex(
  BNINDEX,
  query,
  k,
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...
)

## S4 method for signature 'BiocNeighborGenericIndex'
queryDistanceFromIndex(
  BNINDEX,
  query,
  k,
```

```
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...,
  .check.nonfinite = TRUE
)

queryDistance(
  X,
  query,
  k,
  num.threads = 1,
  ...,
  subset = NULL,
  transposed = FALSE,
  BNPARAM = NULL
)
```

Arguments

BNINDEX	A BiocNeighborIndex object, typically created by buildIndex.
query	A numeric matrix or matrix-like object of query points, containing the same number of columns as X.
k	A positive integer scalar specifying the number of nearest neighbors to retrieve. Alternatively, an integer vector of length equal to the number of points in query, specifying the number of neighbors to identify for each point. If subset is provided, this should have length equal to the length of subset. Users should wrap this vector in an AsIs class to distinguish length-1 vectors from integer scalars. All k should be less than or equal to the number of points in X, otherwise the former will be capped at the latter with a warning.
num.threads	Integer scalar specifying the number of threads to use for the search.
subset	An integer, logical or character vector indicating the rows of query (or columns, if transposed=TRUE) for which the nearest neighbors should be identified.
transposed	A logical scalar indicating whether query is transposed, in which case it contains dimensions in the rows and data points in the columns. For queryKNN, settting transposed=TRUE also indicates that X is also transposed.
...	For queryDistanceFromIndex, further arguments to pass to individual methods. If a method accepts arguments here, it should prefix these arguments with the algorithm name to avoid conflicts, e.g., vptree.foo.bar. For queryDistance, further arguments to pass to queryDistanceFromIndex. These are also passed to buildIndex when X is not an external pointer.
.check.nonfinite	Boolean indicating whether to check for non-finite values in query. This can be set to FALSE for greater efficiency.

X	The reference dataset to be queried. This should be a numeric matrix or matrix-like object where rows correspond to reference points and columns correspond to variables (i.e., dimensions). Alternatively, a prebuilt BiocNeighborIndex object from buildIndex.
BNPARAM	A BiocNeighborParam object specifying how the index should be constructed. If NULL, this defaults to a KmknnParam. Ignored if X contains a prebuilt index.

Details

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X with buildIndex. The resulting pointer object can be supplied as X to multiple queryKNN calls, avoiding the need to repeat index construction in each call.

Value

Numeric vector of length equal to the number of points in query (or subset, if provided), containing the distance from each point to its k-th nearest neighbor. This is equivalent to but more memory efficient than using queryKNN and subsetting to the last distance.

Author(s)

Aaron Lun

See Also

buildIndex, to build an index ahead of time.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
Z <- matrix(rnorm(20000), ncol=20)
out <- queryDistance(Y, query=Z, k=5)
head(out)
```

---

queryKnnFromIndex *Query k-nearest neighbors*

Description

Query a reference dataset for the k-nearest neighbors of each point in a query dataset.

Usage

```
queryKnnFromIndex(
  BNINDEX,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...
)

## S4 method for signature 'BiocNeighborGenericIndex'
queryKnnFromIndex(
  BNINDEX,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...,
  .check.nonfinite = TRUE
)

queryKNN(
  X,
  query,
  k,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...,
  BPPARAM = NULL,
  BNPARAM = NULL
)
```

Arguments

BNINDEX	A BiocNeighborIndex object, typically created by buildIndex.
query	A numeric matrix or matrix-like object of query points, containing the same number of columns as X.
k	A positive integer scalar specifying the number of nearest neighbors to retrieve.

Alternatively, an integer vector of length equal to the number of points in query, specifying the number of neighbors to identify for each point. If subset is provided, this should have length equal to the length of subset. Users should wrap this vector in an AsIs class to distinguish length-1 vectors from integer scalars.

All k should be less than or equal to the number of points in X, otherwise the former will be capped at the latter with a warning.

get.index — A logical scalar indicating whether the indices of the nearest neighbors should be recorded. Setting this to FALSE improves efficiency if the indices are not of interest.

Alternatively, if k is an integer scalar, this may be a string containing "normal" or "transposed". The former is the same as TRUE, while the latter returns the index matrix in transposed format.

get.distance — A logical scalar indicating whether distances to the nearest neighbors should be recorded. Setting this to FALSE improves efficiency if the distances are not of interest.

Alternatively, if k is an integer scalar, this may be a string containing "normal" or "transposed". The former is the same as TRUE, while the latter returns the distance matrix in transposed format.

num.threads — Integer scalar specifying the number of threads to use for the search.

subset — An integer, logical or character vector indicating the rows of query (or columns, if transposed=TRUE) for which the nearest neighbors should be identified.

transposed — A logical scalar indicating whether query is transposed, in which case it contains dimensions in the rows and data points in the columns. For queryKNN, settting transposed=TRUE also indicates that X is also transposed.

... — For queryKnnFromIndex, further arguments to pass to individual methods. If a method accepts arguments here, it should prefix these arguments with the algorithm name to avoid conflicts, e.g., vptree.foo.bar.

For queryKNN, further arguments to pass to queryKnnFromIndex. These are also passed to buildIndex when X is not an external pointer.

.check.nonfinite — Boolean indicating whether to check for non-finite values in query. This can be set to FALSE for greater efficiency.

X — The reference dataset to be queried. This should be a numeric matrix or matrix-like object where rows correspond to reference points and columns correspond to variables (i.e., dimensions). Alternatively, a prebuilt BiocNeighborIndex object from buildIndex.

BPPARAM — Soft-deprecated, use num.threads instead.

BNPARAM — A BiocNeighborParam object specifying how the index should be constructed. If NULL, this defaults to a KmknnParam. Ignored if X contains a prebuilt index.

Details

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X with buildIndex. The resulting pointer object can be supplied as X to multiple queryKNN calls, avoiding the need to repeat index construction in each call.

Value

List containing `index` (if `get.index` is not `FALSE`) and `distance` (if `get.distance` is not `FALSE`).

- If `get.index=TRUE` or `"normal"` and `k` is an integer scalar, `index` is an integer matrix with `k` columns where each row corresponds to a point (denoted here as i) in `query`. The i-th row contains the indices of points in `X` that are the nearest neighbors to point i, sorted by increasing distance from i.

  If `get.index=FALSE` or `"transposed"` and `k` is an integer scalar, `index` is as described above but transposed, i.e., the `i`-th column contains the indices of neighboring points in `X`.

- If `get.distance=TRUE` or `"normal"` and `k` is an integer scalar, `distance` is a numeric matrix of the same dimensions as `index`. The i-th row contains the distances of neighboring points in `X` to the point i, sorted in increasing order.

  If `get.distance=FALSE` or `"transposed"` and `k` is an integer scalar, `distance` is as described above but transposed, i.e., the `i`-th column contains the distances to neighboring points in `X`.

- If `get.index` is not `FALSE` and `k` is an integer vector, `index` is a list of integer vectors where each vector corresponds to a point (denoted here as i) in `X`. The i-th vector has length `k[i]` and contains the indices of points in `X` that are the nearest neighbors to point i, sorted by increasing distance from i.

- If `get.distance` is not `FALSE` and `k` is an integer vector, `distance` is a list of numeric vectors of the same lengths as those in `index`. The i-th vector contains the distances of neighboring points in `X` to the point i, sorted in increasing order.

Author(s)

Aaron Lun

See Also

`buildIndex`, to build an index ahead of time.

`queryDistance`, to obtain the distance from each query point to its k-th nearest neighbor.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
Z <- matrix(rnorm(20000), ncol=20)
out <- queryKNN(Y, query=Z, k=5)
head(out$index)
head(out$distance)
```

queryNeighborsFromIndex *Query neighbors within a threshold distance*

Description

Find all points in a reference dataset that lie within a threshold distance of each point in a query dataset.

Usage

```
queryNeighborsFromIndex(
  BNINDEX,
  query,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...
)

## S4 method for signature 'BiocNeighborGenericIndex'
queryNeighborsFromIndex(
  BNINDEX,
  query,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...,
  .check.nonfinite = TRUE
)

queryNeighbors(
  X,
  query,
  threshold,
  get.index = TRUE,
  get.distance = TRUE,
  num.threads = 1,
  subset = NULL,
  transposed = FALSE,
  ...,
```

```
  BPPARAM = NULL,
  BNPARAM = NULL
)
```

Arguments

BNINDEX	A BiocNeighborIndex object, typically created by buildIndex.
query	A numeric matrix or matrix-like object of query points, containing the same number of columns as X.
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.
num.threads	Integer scalar specifying the number of threads to use for the search.
subset	An integer, logical or character vector indicating the rows of query (or columns, if transposed=TRUE) for which the nearest neighbors should be identified.
transposed	A logical scalar indicating whether query is transposed, in which case it contains dimensions in the rows and data points in the columns. For queryKNN, settting transposed=TRUE also indicates that X is also transposed.
...	For queryNeighborsFromIndex, further arguments to pass to individual methods. If a method accepts arguments here, it should prefix these arguments with the algorithm name to avoid conflicts, e.g., vptree.foo.bar. For queryNeighbors, further arguments to pass to queryNeighborsFromIndex. These are also passed to buildIndex when X is not an external pointer.
.check.nonfinite	Boolean indicating whether to check for non-finite values in query. This can be set to FALSE for greater efficiency.
X	The reference dataset to be queried. This should be a numeric matrix or matrix-like object where rows correspond to reference points and columns correspond to variables (i.e., dimensions). Alternatively, a prebuilt BiocNeighborIndex object from buildIndex.
BPPARAM	Soft-deprecated, use num.threads instead.
BNPARAM	A BiocNeighborParam object specifying how the index should be constructed. If NULL, this defaults to a KmknnParam. Ignored if X contains a prebuilt index.

Details

This function identifies all points in X that within threshold of each point in query. For Euclidean distances, this is equivalent to identifying all points in a hypersphere centered around the point of interest. Not all implementations support this search mode, but we can use KmknnParam and VptreeParam.

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.

If multiple queries are to be performed to the same X, it may be beneficial to build the index from X with buildIndex. The resulting pointer object can be supplied as X to multiple queryKNN calls, avoiding the need to repeat index construction in each call.

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. Neighbors for i are sorted by increasing distance from i.
- 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 both get.index=FALSE and get.distance=FALSE, an integer vector is returned of length equal to the number of observations. The i-th entry contains the number of neighbors of i within threshold.

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

Author(s)

Aaron Lun

See Also

buildIndex, to build an index ahead of time.

Examples

```
Y <- matrix(rnorm(100000), ncol=20)
Z <- matrix(rnorm(20000), ncol=20)
out <- queryNeighbors(Y, query=Z, threshold=3)
summary(lengths(out$index))
```

---

saveIndex *Save a nearest-neighbor index*

Description

Save an index for nearest-neighbor searching to disk.

Usage

```
saveIndex(BNINDEX, dir, ...)

## S4 method for signature 'BiocNeighborGenericIndex'
saveIndex(BNINDEX, dir, ...)
```

Arguments

BNINDEX	A BiocNeighborIndex object representing a pre-built index, typically from buildIndex.
dir	String containing the path to a directory in which to save the index. This directory should already exist.
...	Further arguments to pass to specific methods.

Details

Files generated by saveIndex are not guaranteed to be portable across architectures, compilers, or even versions of **BiocNeighbors**. An index saved in this manner is only intended to be read back to the same R environment on the same machine.

Value

One or more files are created on disk inside dir. These can be used to reconstitute BNINDEX by calling loadIndex.

Author(s)

Aaron Lun

Examples

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

tmp <- tempfile()
dir.create(tmp)
saveIndex(k.out, tmp)
list.files(tmp, recursive=TRUE)
```

VptreeParam *The VptreeParam class*

Description

A class to hold parameters for the vantage point (VP) tree algorithm for exact nearest neighbor identification.

Usage

```
VptreeParam(distance = c("Euclidean", "Manhattan", "Cosine"))

## S4 method for signature 'VptreeParam'
defineBuilder(BNPARAM)
```

Arguments

distance	String specifying the distance metric to use. Cosine distances are implemented as Euclidean distances on L2-normalized coordinates.
BNPARAM	A VptreeParam instance.

Details

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.

Value

The `VptreeParam` constructor returns an instance of the VptreeParam class.

The `defineBuilder` method returns an external pointer that can be used in `buildIndex` to construct a VP tree index.

Author(s)

Aaron Lun

References

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.

See Also

BiocNeighborParam, for the parent class and its available methods.

`https://stevehanov.ca/blog/index.php?id=130`, for a description of the algorithm.

Examples

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

Index