Package 'nabor'

July 22, 2025

Type Package

Title Wraps 'libnabo', a Fast K Nearest Neighbour Library for Low Dimensions

Version 0.5.0

Author Stephane Mangenat (for 'libnabo'), Gregory Jefferis

Maintainer Gregory Jefferis <jefferis@gmail.com>

Description An R wrapper for 'libnabo', an exact or approximate k nearest neighbour library which is optimised for low dimensional spaces (e.g. 3D). 'libnabo' has speed and space advantages over the 'ANN' library wrapped by package 'RANN'. 'nabor' includes a knn function that is designed as a drop-in replacement for 'RANN' function nn2. In addition, objects which include the k-d tree search structure can be returned to speed up repeated queries of the same set of target points.

License BSD_3_clause + file LICENSE

Copyright libnabo is copyright 2010--2011, Stephane Magnenat, ASL, ETHZ, Switzerland <stephane at magnenat dot net>

URL https://github.com/jefferis/nabor

https://github.com/ethz-asl/libnabo

BugReports https://github.com/jefferis/nabor/issues

Depends R (>= 3.0.2)

Imports Rcpp (>= 0.11.2), methods

LinkingTo Rcpp, RcppEigen (>= 0.3.2.2.0), BH (>= 1.54.0-4)

Suggests testthat, RANN

RoxygenNote 6.0.1

NeedsCompilation yes

Repository CRAN

Date/Publication 2018-07-11 16:00:02 UTC

Contents

`nabor-package` *Wrapper for libnabo K Nearest Neighbours C++ library*

Description

R package **nabor** wraps the libnabo library, a fast K Nearest Neighbour library for low-dimensional spaces written in templated C++. The package provides both a standalone function (see `knn` for basic queries along an option to produce an object containing the k-d tree search (see `WKNN`) structure when making multiple queries against the same target points.

Details

libnabo uses the same approach as the ANN library (wrapped in R package `RANN`) but is generally faster and with a smaller memory footprint. Furthermore since it is templated on the underlying scalar type for coordinates (among other things), we have provided both float and double coordinate implementations of the classes wrapping the search tree structures. See the github repository and Elsenberg et al paper below for details.

References

Elseberg J, Magnenat S, Siegwart R and Nuechter A (2012). "Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration." _Journal of Software Engineering for Robotics (JOSER)_, *3*(1), pp. 2-12. ISSN 2035-3928.

See Also

`knn`, `WKNN`

`kcpoints` *List of 3 matrices containing 3D points from Drosophila neurons*

Description

This R list contains 3 skeletonized *Drosophila* Kenyon cells as `dotprops` objects. Original data is due to Chiang et al. 2011, who have generously shared their raw data at `http://flycircuit.tw`. Image registration and further processing was carried out by Greg Jefferis.

References

[1] Chiang A.S., Lin C.Y., Chuang C.C., Chang H.M., Hsieh C.H., Yeh C.W., Shih C.T., Wu J.J., Wang G.T., Chen Y.C., Wu C.C., Chen G.Y., Ching Y.T., Lee P.C., Lin C.Y., Lin H.H., Wu C.C., Hsu H.W., Huang Y.A., Chen J.Y., et al. (2011). Three-dimensional reconstruction of brain-wide wiring networks in Drosophila at single-cell resolution. Curr Biol 21 (1), 1–11.

---

knn *Find K nearest neighbours for multiple query points*

Description

Find K nearest neighbours for multiple query points

Usage

```
knn(data, query = data, k, eps = 0, searchtype = 1L, radius = 0)
```

Arguments

`data`	Mxd matrix of M target points with dimension d
`query`	Nxd matrix of N query points with dimension d (nb `data` and `query` must have same dimension). If missing defaults to `data` i.e. a self-query.
`k`	an integer number of nearest neighbours to find
`eps`	An approximate error bound. The default of 0 implies exact matching.
`searchtype`	A character vector or integer indicating the search type. The default value of `1L` is equivalent to "auto". See details.
`radius`	Maximum radius search bound. The default of 0 implies no radius bound.

Details

If `searchtype="auto"`, the default, knn uses a k-d tree with a linear heap when `k < 30` nearest neighbours are requested (equivalent to `searchtype="kd_linear_heap"`), a k-d tree with a tree heap otherwise (equivalent to `searchtype="kd_tree_heap"`). `searchtype="brute"` checks all point combinations and is intended for validation only.

Integer values of searchtype should be the 1-indexed position in the vector `c("auto", "brute", "kd_linear_heap", "kd_tree_heap")`, i.e. a value between 1L and 4L.

The underlying libnabo does not have a signalling value to identify indices for invalid query points (e.g. those containing an `NA`). In this situation, the index returned by libnabo will be 0 and `knn` will therefore return an index of 1. However the distance will be `Inf` signalling a failure to find a nearest neighbour.

When radius>0.0 and no point is found within the search bound, the index returned will be 0 but the reported distance will be Inf (in contrast RANN::nn2 returns 1.340781e+154).

Value

A list with elements `nn.idx` (1-indexed indices) and `nn.dists` (distances), both of which are N x k matrices. See details for the results obtained with1 invalid inputs.

Examples

```
## Basic usage
# load sample data consisting of list of 3 separate 3d pointets
data(kcpoints)

# Nearest neighbour in first pointset of all points in second pointset
nn1 <- knn(data=kcpoints[[1]], query=kcpoints[[2]], k=1)
str(nn1)

# 5 nearest neighbours
nn5 <-knn(data=kcpoints[[1]], query=kcpoints[[2]], k=5)
str(nn5)

# Self match within first pointset, all distances will be 0
nnself1 <- knn(data=kcpoints[[1]], k=1)
str(nnself1)

# neighbour 2 will be the nearest point
nnself2 <- knn(data=kcpoints[[1]], k=2)

## Advanced usage
# nearest neighbour with radius bound
nn1.rad <- knn(data=kcpoints[[1]], query=kcpoints[[2]], k=1, radius=5)
str(nn1.rad)

# approximate nearest neighbour with 10% error bound
nn1.approx <- knn(data=kcpoints[[1]], query=kcpoints[[2]], k=1, eps=0.1)
str(nn1.approx)

# 5 nearest neighbours, brute force search
nn5.b <-knn(data=kcpoints[[1]], query=kcpoints[[2]], k=5, searchtype='brute')
stopifnot(all.equal(nn5.b, nn5))

# 5 nearest neighbours, brute force search (specified by int)
nn5.b2 <-knn(data=kcpoints[[1]], query=kcpoints[[2]], k=5, searchtype=2L)
stopifnot(all.equal(nn5.b2, nn5.b))
```

`WKNNF-class` *Wrapper classes for k-NN searches enabling repeated queries of the same tree*

Description

`WKNNF` and `WKNND` are reference classes that wrap C++ classes of the same name that include a space-efficient k-d tree along with the target data points. They have `query` methods with exactly the same

interface as the knn function. One important point compared with knn - they must be intialised with floating point data and you are responsible for this - see storage.mode) and the example below.

Details

WKNNF expects and returns matrices in R's standard (double, 8 bytes) data type but uses floats internally. WKNND uses doubles throughout. When retaining large numbers of points, the WKNNF objects will have a small memory saving, especially if tree building is delayed.

The constructor for WKNN objects includes a logical flag indicating whether to build the tree immediately (default: TRUE) or (when FALSE) to delay building the tree until a query is made (this happens automatically when required).

Performance

The use of WKNN objects will incur a performance penalty for single queries of trees with < ~1000 data points. This is because of the overhead associated with the R wrapper class. It therefore makes sense to use knn in these circumstances.

If you wish to make repeated queries of the same target data, then using WKNN objects can give significant advantages. If you are going to make repeated queries with the same set of query points (presumably against different target data), you can obtain benefits in some cases by converting the query points into WKNNF objects without building the trees.

See Also

knn

Examples

```
## Basic usage
# load sample data consisting of list of 3 separate 3d pointets
data(kcpoints)
# build a tree and query it with two different sets of points
w1 <- WKNNF(kcpoints[[1]])
w1q2 <- w1$query(kcpoints[[2]], k=5, eps=0, radius=0)
str(w1q2)
w1q3 <- w1$query(kcpoints[[3]], k=5, eps=0, radius=0)
# note that there will be small difference between WKNNF and knn due to loss
# of precision in the double to float conversion when a WKNNF tree is
# built and queried.
stopifnot(all.equal(
 knn(data=kcpoints[[1]], query=kcpoints[[2]], k=5, eps=0, radius=0),
 w1q2, tolerance=1e-6))

## storage mode: must be double
m=matrix(1:24, ncol=3)
storage.mode(m)
# this will generate an error unless we change to a double
w=tools::assertCondition(WKNND(m), "error")
storage.mode(m)="double"
w=WKNND(matrix(m, ncol=3))
```

```
## construct wrapper objects but delay tree construction
w1 <- WKNNF(kcpoints[[1]], FALSE)
# query triggers tree construction
w1q2 <- w1$query(kcpoints[[2]], k=5, eps=0, radius=0)
str(w1q2)

## queries using wrapper objects
wkcpoints <-lapply(kcpoints, WKNNF, FALSE)
# query all 3 point sets against first
# this will trigger tree construction only for pointset 1
qall <- lapply(wkcpoints,
  function(x) wkcpoints[[1]]$queryWKNN(x$.CppObject, k=5, eps=0, radius=0))
str(qall)
```

Index