Using Annoy in package C++ code

Aaron Lun[a]

[a]https://github.com/LTLA

This version was compiled on November 16, 2020

This note shows how to use the Annoy library for *Approximate Nearest Neighbours (Oh Yeah)* from C++ code using the headers provided by the RcppAnnoy package.

Rcpp | Annoy | Approximate Nearest Neighbours

Setting up your package

The **Annoy** C++ library (Bernhardsson, 2018) implements a quick and simple method for *approximate nearest neighbor (oh yeah)* searching. The **RcppAnnoy** package (Eddelbuettel, 2018) provides a centralized resource for developers to use this code in their own R packages by relying on **Rcpp** (Eddelbuettel and Balamuta, 2018; Eddelbuettel *et al.*, 2018). To use **Annoy** in C++ code, simply put in your `DESCRIPTION` the line

```
LinkingTo: RcppAnnoy
```

and the header files will be available for inclusion into your package's source files. Note that **Annoy** is a header-only library so no additional commands are necessary for the linker.

Including the header files

Obviously, the header files need to be `included` in any C++ source file that uses **Annoy**. A few macros also need to be added to handle Windows-specific behaviour and to ensure that error messages are printed through R. Version number comparison macros help in conditioning changes on a particular version. Since release 0.0.17 all this is now expressed centrally in a header in the package so users can just use this one-liner:

```
#include "RcppAnnoy.h"
```

Defining the search type

The `AnnoyIndex` template class can accommodate different data types, distance metrics, random number generators, and threading policies (where the latter are a choice between sequential or multithreaded). Here, we will consider the most common application of a nearest-neighbor search on floating-point data with Euclidean distance. We `typedef` the type and realized template for convenience:

```
typedef float ANNOYTYPE;

typedef
AnnoyIndex<int, ANNOYTYPE, Euclidean, Kiss64Random,
           AnnoyIndexThreadedBuildPolicy>
MyAnnoyIndex;
```

Note that we use `float` by default, rather than the more conventional `double`. This is chosen for speed and to be consistent with the original Python implementation.

The **Annoy** library uses random number generation during index creation (via the `Kiss64Random` class), with a seed that is separate from R's RNG seed. By default, the seed is fixed and results will be "deterministic" in the sense that repeated runs on the same data will yield the same result. They will also be unresponsive to the state of R's RNG seed. The seed used by `AnnoyIndex` can be specified by the `set_seed` method, which should be called before adding items to the index.

Building an index

Let's say we have an `Rcpp::NumericMatrix` named `mat`, where each row corresponds to a sample and each column corresponds to a dimension/variable.

```
const size_t nsamples=mat.nrow();
const size_t ndims=mat.ncol();
```

It is simple to build a `MyAnnoyIndex` containing the data in this matrix. Note the copy from the double-precision matrix into a `float` vector before calling `add_item()`.

```
MyAnnoyIndex obj(ndims);
// from <vector>
std::vector<ANNOYTYPE> tmp(ndims);
for (size_t i=0; i<nsamples; ++i) {
    Rcpp::NumericMatrix::Row cr=mat.row(i);
    // from <algorithm>
    std::copy(cr.begin(), cr.end(), tmp.begin());
    obj.add_item(i, tmp.data());
}
obj.build(50);
```

The `build()` method accepts an integer argument specifying the number of trees to use to construct the index. Indices with more trees are larger (in memory and on file) but yield greater search accuracy.

The index can also be saved to file via

```
obj.save("annoy.index");
```

and reloaded in some other context:

```
MyAnnoyIndex obj2(ndims);
obj2.load("annoy.index"); // same as 'obj'.
```

This is helpful for parallelization across workers running in different R sessions. It also allows us to avoid rebuilding the index in applications where the same data set is to be queried multiple times.

Searching for nearest neighbors

Let's say that we want to find the `K` (approximate) nearest neighbors of sample `c` in the original data set used to construct `obj`. To do this, we write:

```
std::vector<int> neighbor_index;
std::vector<ANNOYTYPE> neighbor_dist;
obj.get_nns_by_item(c, K + 1, -1, &neighbor_index,
                    &neighbor_dist);
```

Upon return, the `neighbor_index` vector will be filled with the sample numbers of the `K` nearest neighbors (i.e., rows of the original `mat`, in this case). The `neighbor_dist` vector will be filled with the distances to each of those neighbors. Note that:

- We ask for the `K+1` nearest neighbors, as the set returned in `neighbor_index` will usually include `c` itself. This should be taken into consideration when the results are used in downstream calculations.
- The returned neighbors are sorted by increasing distance from `c`. However, note that `c` itself may not necessarily be at the start if there is another point with the same coordinates.
- `get_nns_by_item()` requires pointers to the vectors rather than the vectors themselves. If the pointer to the output vector for distances is `NULL`, distances will not be returned. This provides a slight performance boost when only the identities of the neighbors are of interest.
- The `-1` is the default value for a tuning parameter that specifies how many samples should be collected from the trees for exhaustive distance calculations. This defaults to the number of trees multiplied by the number of requested neighbors; larger values will increase accuracy at the cost of speed.

Another application is to query the index for the neighbors of a new sample given its coordinates. Assuming we have a `float*` to an array of coordinates of length `ndims`, we do:

```
obj.get_nns_by_vector(query, K+1, -1,
                      &neighbor_index,
                      &neighbor_dist);
```

Further information

The Annoy repository is the canonical source of all things Annoying. Questions or issues related to the **Annoy** C++ library itself should be posted there. Any issues specific to the **RcppAnnoy** interface should be posted at its separate Github repository. An example of using the Annoy library via **RcppAnnoy** is available in the **BiocNeighbors** package (Lun, 2018).

References

Bernhardsson E (2018). *Annoy: Approximate Nearest Neighbors in C++/Python*. Python package version 1.13.0, URL https://pypi.org/project/annoy/.

Eddelbuettel D (2018). *RcppAnnoy: Rcpp Bindings for Annoy, a Library for Approximate Nearest Neighbors*. R package version 0.0.10, URL http://CRAN.R-Project.org/package=RcppAnnoy.

Eddelbuettel D, Balamuta JJ (2018). "Extending R with C++: A Brief Introduction to Rcpp." *The American Statistician*, **72**(1). doi:10.1080/00031305.2017.1375990.

Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russel N, Chambers J, Bates D (2018). *Rcpp: Seamless R and C++ Integration*. R package version 0.12.19, URL http://CRAN.R-Project.org/package=Rcpp.

Lun A (2018). *BiocNeighbors: Nearest Neighbor Detection for Bioconductor Packages*. R package version 0.99.22.