spotify / annoy

Code Issues 49 Pull requests 4 Actions Projects Wiki Security Insights

Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk

Apache-2.0 license

Code of conduct

12.3k stars 1.2k forks 320 watching 21 Branches 27 Tags Activity

Public repository

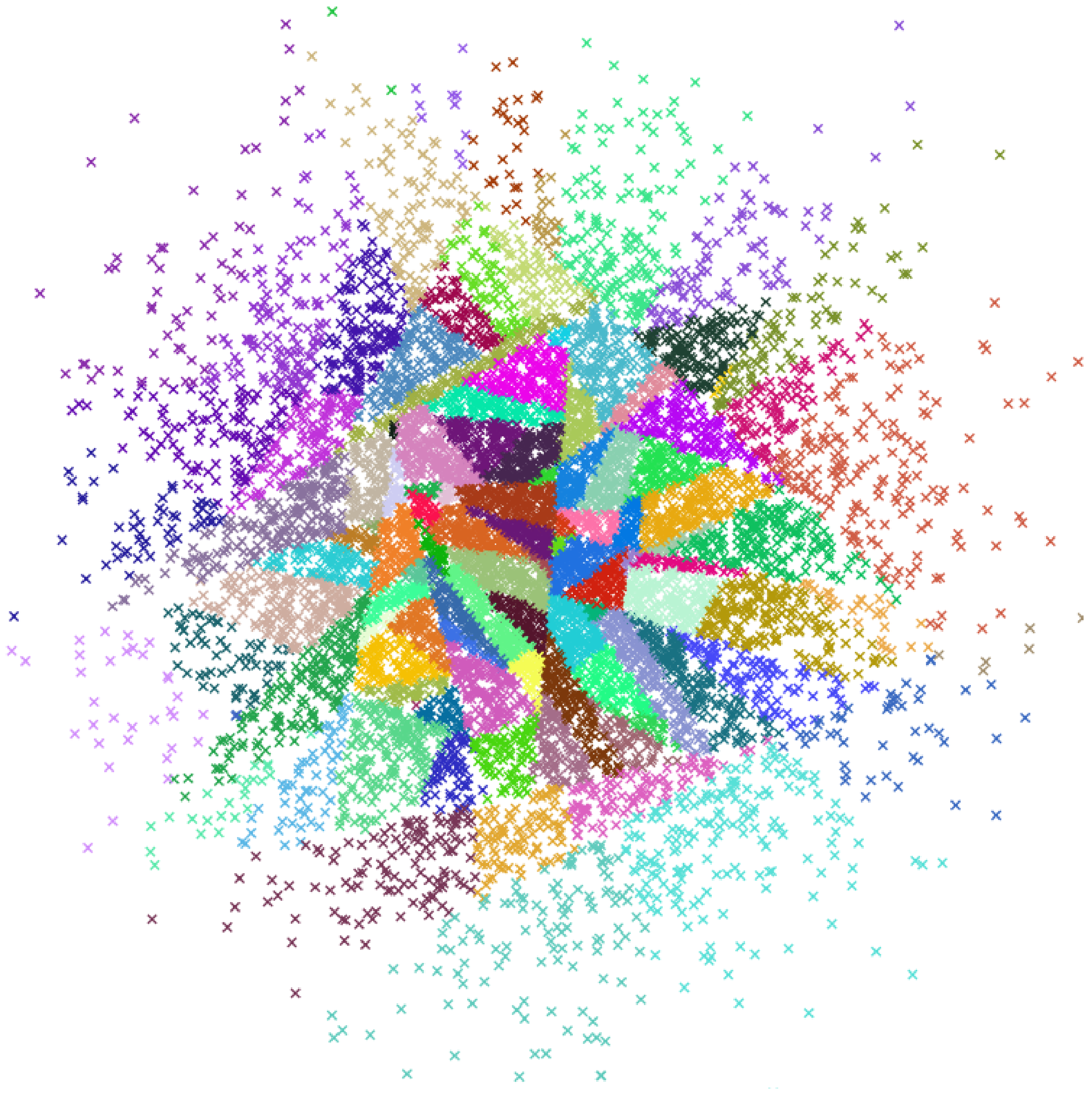
main 21 Branches 27 Tags Go to file t Go to file + Add file Code ...

erikbern Merge pull request #628 from pkorobov/fix-dot-recall ✓ 4 months ago

.github/workflows	pytest verbose	8 months ago
annoy	Fix 646	7 months ago
debian	removed boost from debian/control and ...	8 years ago
examples	Adding namespace	9 months ago
src	Merge branch 'main' into fix-dot-recall	4 months ago
test	Fix another merge conflict (remove self fr...	4 months ago
.gitignore	Improve .gitignore coverage of files creat...	4 years ago
CMakeLists.txt	Fix cmake	10 months ago
LICENSE	Sign the Apache license for Spotify	2 years ago
MANIFEST.in	Bump to 1.6.2 and include the correct files	8 years ago
README.rst	Merge branch 'main' into fix-dot-recall	4 months ago
README_GO.rst	Update README_GO.rst	last year
README_Lua.md	Update READMEs and Lua rockspec.	3 years ago
RELEASE.md	version 1.16.2	4 years ago
ann.png	Converted documentation to RST, cropp...	9 years ago
annoy-dev-1.rockspec	add Lua bindings, rockspec and test	7 years ago
setup.cfg	run GloVe f=25 as a part of the test suite	8 years ago

setup.py	Version 1.17.3	6 months ago
tox.ini	increase verbosity so we can see which te...	3 years ago

Annoy



README Code of conduct Apache-2.0 license

are close to a given query point. It also creates large read-only file-based data structures that are mmapped into memory so that many processes may share the same data.

Install

To install, simply do `pip install --user annoy` to pull down the latest version from PyPI.

For the C++ version, just clone the repo and `#include "annoylib.h"` .

Background

There are some other libraries to do nearest neighbor search. Annoy is almost as fast as the fastest libraries, (see below), but there is actually another feature that really sets Annoy apart: it has the ability to **use static files as indexes**. In particular, this means you can **share index across processes**. Annoy also decouples creating indexes from loading them, so you can pass around indexes as files and map them into memory quickly. Another nice thing of Annoy is that it tries to minimize memory footprint so the indexes are quite small.

Why is this useful? If you want to find nearest neighbors and you have many CPU's, you only need to build the index once. You can also pass around and distribute static files to use in production environment, in Hadoop jobs, etc. Any process will be able to load (mmap) the index into memory and will be able to do lookups immediately.

We use it at Spotify for music recommendations. After running matrix factorization algorithms, every user/item can be represented as a vector in f-dimensional space. This library helps us search for similar users/items. We have many millions of tracks in a high-dimensional space, so memory usage is a prime concern.

Annoy was built by Erik Bernhardsson in a couple of afternoons during Hack Week.

Summary of features

- Euclidean distance, Manhattan distance, cosine distance, Hamming distance, or Dot (Inner) Product distance
- Cosine distance is equivalent to Euclidean distance of normalized vectors = sqrt(2-2*cos(u, v))
- Works better if you don't have too many dimensions (like <100) but seems to perform surprisingly well even up to 1,000 dimensions
- Small memory usage
- Lets you share memory between multiple processes
- Index creation is separate from lookup (in particular you can not add more items once the tree has been created)
- Native Python support, tested with 2.7, 3.6, and 3.7.
- Build index on disk to enable indexing big datasets that won't fit into memory (contributed by Rene Hollander)

Python code example

```
from annoy import AnnoyIndex
import random

f = 40  # Length of item vector that will be indexed

t = AnnoyIndex(f, 'angular')
for i in range(1000):
    v = [random.gauss(0, 1) for z in range(f)]
    t.add_item(i, v)

t.build(10) # 10 trees
t.save('test.ann')

# ...

u = AnnoyIndex(f, 'angular')
u.load('test.ann') # super fast, will just mmap the file
print(u.get_nns_by_item(0, 1000)) # will find the 1000 nearest neighbors
```

Right now it only accepts integers as identifiers for items. Note that it will allocate memory for max(id)+1 items because it assumes your items are numbered 0 … n-1. If you need other id's, you will have to keep track of a map yourself.

Full Python API

- `AnnoyIndex(f, metric)` returns a new index that's read-write and stores vector of `f` dimensions. Metric can be `"angular"`, `"euclidean"`, `"manhattan"`, `"hamming"`, or `"dot"`.
- `a.add_item(i, v)` adds item `i` (any nonnegative integer) with vector `v`. Note that it will allocate memory for `max(i)+1` items.
- `a.build(n_trees, n_jobs=-1)` builds a forest of `n_trees` trees. More trees gives higher precision when querying. After calling `build`, no more items can be added. `n_jobs` specifies the number of threads used to build the trees. `n_jobs=-1` uses all available CPU cores.
- `a.save(fn, prefault=False)` saves the index to disk and loads it (see next function). After saving, no more items can be added.
- `a.load(fn, prefault=False)` loads (mmaps) an index from disk. If prefault is set to True, it will pre-read the entire file into memory (using mmap with MAP_POPULATE). Default is False.
- `a.unload()` unloads.
- `a.get_nns_by_item(i, n, search_k=-1, include_distances=False)` returns the `n` closest items. During the query it will inspect up to `search_k` nodes which defaults to `n_trees * n` if not provided. `search_k` gives you a run-time tradeoff between better accuracy and speed. If you set `include_distances` to `True`, it will return a 2 element tuple with two lists in it: the second one containing all corresponding distances.
- `a.get_nns_by_vector(v, n, search_k=-1, include_distances=False)` same but query by vector `v`.
- `a.get_item_vector(i)` returns the vector for item `i` that was previously added.
- `a.get_distance(i, j)` returns the distance between items `i` and `j`. NOTE: this used to return the *squared* distance, but has been changed as of Aug 2016.
- `a.get_n_items()` returns the number of items in the index.
- `a.get_n_trees()` returns the number of trees in the index.
- `a.on_disk_build(fn)` prepares annoy to build the index in the specified file instead of RAM (execute before adding items, no need to save after build)
- `a.set_seed(seed)` will initialize the random number generator with the given seed. Only used for building up the tree, i. e. only necessary to pass this before adding the items. Will have no effect after calling a.build(n_trees) or a.load(fn).

Notes:

- There's no bounds checking performed on the values so be careful.
- Annoy uses Euclidean distance of normalized vectors for its angular distance, which for two vectors u,v is equal to `sqrt(2(1-cos(u,v)))`

The C++ API is very similar: just `#include "annoylib.h"` to get access to it.

Tradeoffs

There are just two main parameters needed to tune Annoy: the number of trees `n_trees` and the number of nodes to inspect during searching `search_k`.

- `n_trees` is provided during build time and affects the build time and the index size. A larger value will give more accurate results, but larger indexes.
- `search_k` is provided in runtime and affects the search performance. A larger value will give more accurate results, but will take longer time to return.

If `search_k` is not provided, it will default to `n * n_trees` where `n` is the number of approximate nearest neighbors. Otherwise, `search_k` and `n_trees` are roughly independent, i.e. the value of `n_trees` will not affect search time if `search_k` is held constant and vice versa. Basically it's recommended to set `n_trees` as large as possible given the amount of memory you can afford, and it's recommended to set `search_k` as large as possible given the time constraints you have for the queries.

You can also accept slower search times in favour of reduced loading times, memory usage, and disk IO. On supported platforms the index is prefaulted during `load` and `save`, causing the file to be pre-emptively read from disk into memory. If you set `prefault` to `False`, pages of the mmapped index are instead read from disk and cached in memory on-demand, as necessary for a search to complete. This can significantly increase early search times but may be better suited for systems with low memory compared to index size, when few queries are executed against a loaded index, and/or when large areas of the index are unlikely to be relevant to search queries.

How does it work

Using random projections and by building up a tree. At every intermediate node in the tree, a random hyperplane is chosen, which divides the space into two subspaces. This hyperplane is chosen by sampling two points from the subset and taking the hyperplane equidistant from them.

We do this k times so that we get a forest of trees. k has to be tuned to your need, by looking at what tradeoff you have between precision and performance.

Hamming distance (contributed by Martin Aumüller) packs the data into 64-bit integers under the hood and uses built-in bit count primitives so it could be quite fast. All splits are axis-aligned.

Dot Product distance (contributed by Peter Sobot and Pavel Korobov) reduces the provided vectors from dot (or "inner-product") space to a more query-friendly cosine space using a method by Bachrach et al., at Microsoft Research, published in 2014.

More info

- Dirk Eddelbuettel provides an R version of Annoy.
- Andy Sloane provides a Java version of Annoy although currently limited to cosine and read-only.
- Pishen Tsai provides a Scala wrapper of Annoy which uses JNA to call the C++ library of Annoy.
- Atsushi Tatsuma provides Ruby bindings for Annoy.
- There is experimental support for Go provided by Taneli Leppä.
- Boris Nagaev wrote Lua bindings.
- During part of Spotify Hack Week 2016 (and a bit afterward), Jim Kang wrote Node bindings for Annoy.
- Min-Seok Kim built a Scala version of Annoy.
- hanabi1224 built a read-only Rust version of Annoy, together with **dotnet, jvm and dart** read-only bindings.
- Presentation from New York Machine Learning meetup about Annoy
- Annoy is available as a conda package on Linux, OS X, and Windows.
- ann-benchmarks is a benchmark for several approximate nearest neighbor libraries. Annoy seems to be fairly competitive, especially at higher precisions:

ANN benchmarks

Source code

It's all written in C++ with a handful of ugly optimizations for performance and memory usage. You have been warned :)

The code should support Windows, thanks to Qiang Kou and Timothy Riley.

To run the tests, execute python setup.py nosetests. The test suite includes a big real world dataset that is downloaded from the internet, so it will take a few minutes to execute.

Discuss

Releases 20

1.17.2 – Fix memory leak Latest
on Apr 10

+ 19 releases

Packages

No packages published

Used by 3.1k

+ 3,109

Contributors 65

+ 51 contributors

Languages

● C++ 49.1% ● Python 31.1% ● Lua 10.5% ● Go 4.4% ● C 3.4% ● SWIG 1.0% ● CMake 0.5%