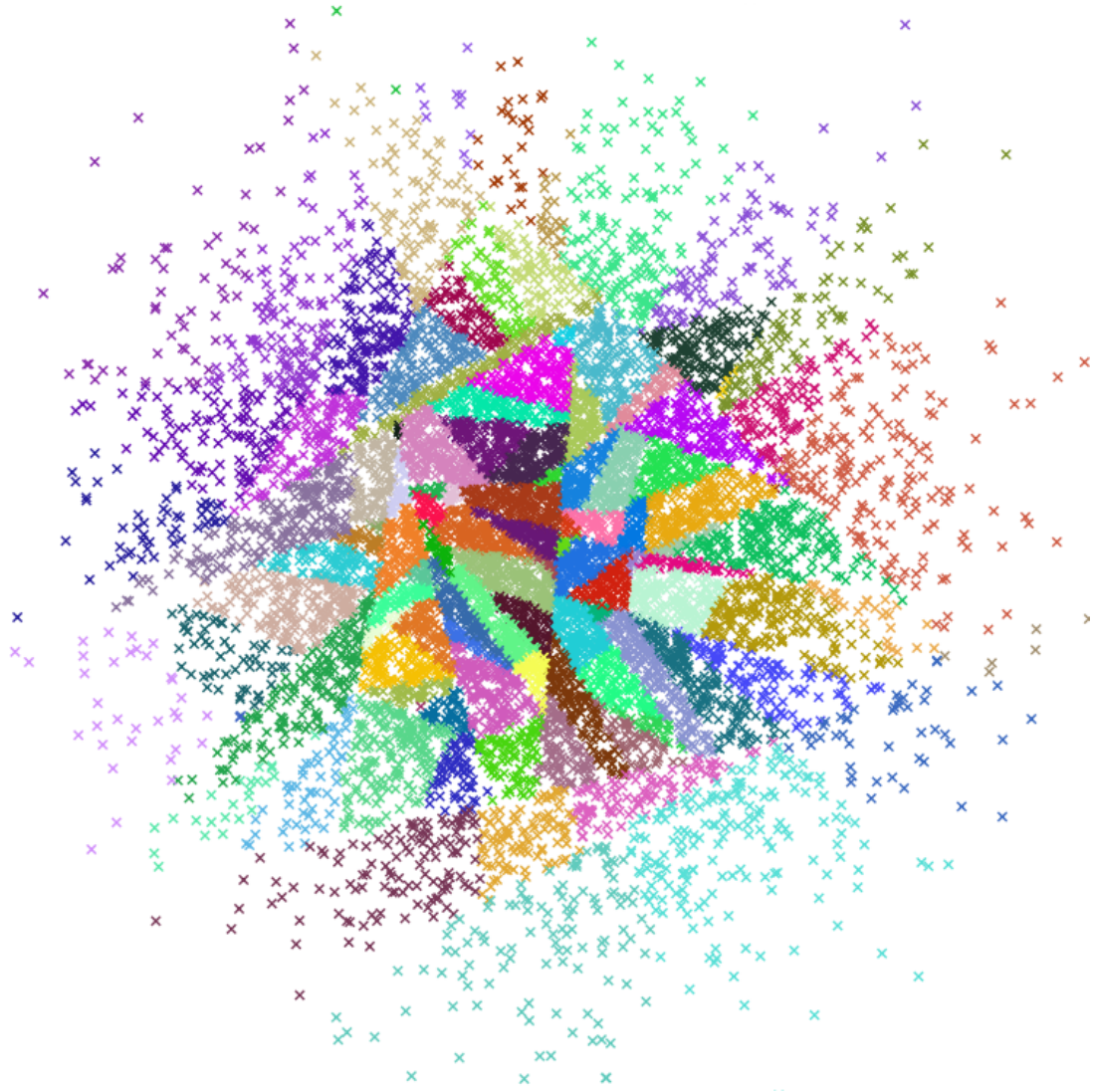


build passing windows - Failing pypi v1.10.0

Annoy (Approximate Nearest Neighbors Oh Yeah) is a C++ library with Python bindings to search for points in space that 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 `sudo pip install 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 the RAM to fit 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, or Hamming 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 2.6, 2.7, 3.3, 3.4, 3.5

Python code example

```
from annoy import AnnoyIndex
import random

f = 40
t = AnnoyIndex(f)  # Length of item vector that will be indexed
for i in xrange(1000):
    v = [random.gauss(0, 1) for z in xrange(f)]
    t.add_item(i, v)

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

# ...

u = AnnoyIndex(f)
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='angular')` returns a new index that's read-write and stores vector of `f` dimensions. Metric can be `"angular"`, `"euclidean"`, `"manhattan"`, or `"hamming"`.
- `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)` builds a forest of `n_trees` trees. More trees gives higher precision when querying. After calling `build`, no more items can be added.
- `a.save(fn)` saves the index to disk.
- `a.load(fn)` loads (mmaps) an index from disk.
- `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.

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 parameters you can use 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. a 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.

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.

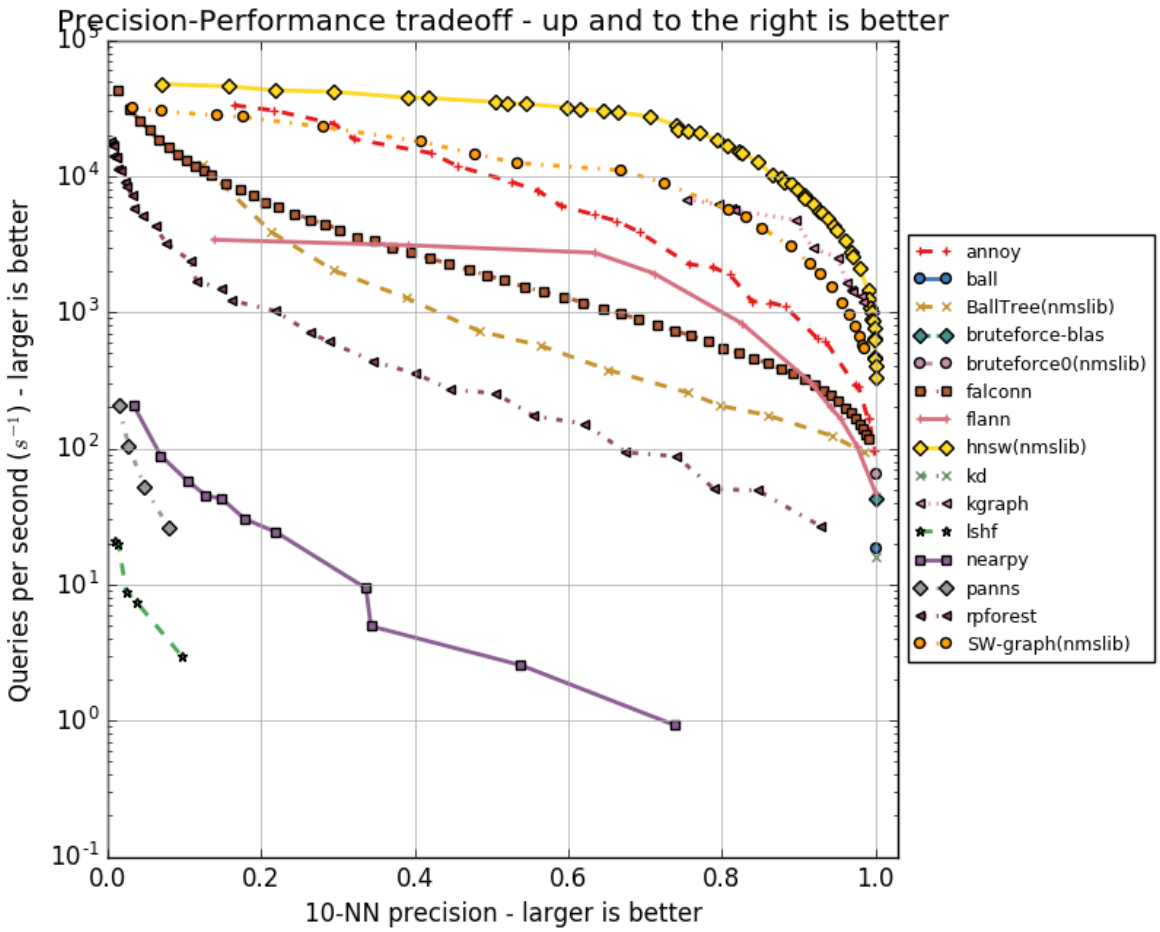
We do this k times so that we get a forest of trees. k has to be tuned to your need, by looking at what tradeoff you have between precision and performance.

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.

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.
- 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.
- Presentation from New York Machine Learning meetup about Annoy
- Radim Řehůřek's blog posts comparing Annoy to a couple of other similar Python libraries: Intro, Contestants, Querying

- ann-benchmarks is a benchmark for several approximate nearest neighbor libraries. Annoy seems to be fairly competitive, especially at higher precisions:



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

Feel free to post any questions or comments to the annoy-user group. I'm @fulhack on Twitter.