推荐系统的近似近邻

Approximate Nearest Neighbours for Recommender Systems (/approximate-nearest-neighbours-for-recommender-systems/)

Faiss是Facebook的AI团队开源的一套用于做聚类或者相似性搜索的软件库，底层是用C++实现。Faiss因为超级优越的性能，被广泛应用于推荐相关的业务当中

One challenge that recommender systems face is in quickly generating a list of the best recommendations to show for the user. These days many libraries can quickly train models that can handle millions of users and millions of items, but the naive solution for evaluating these models involves ranking every single item for every single user which can be extremely expensive. As an example, my implicit (https://github.com/benfred/implicit) recommendation library can train a model on the last.fm dataset (http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-360K.html) in 24 seconds on my desktop - but takes over an hour to use that model to generate recommendations for each user.

This post is about evaluating a couple of different approximate nearest neighbours libraries to speed up making recommendations made by matrix factorization models. In particular, the libraries I'm looking at are Annoy (https://github.com/spotify/annoy), NMSLib (https://github.com/searchivarius/nmslib) and Faiss (https://github.com/facebookresearch/faiss).

I've used Annoy successfully for a couple different projects now in the past - but was recently intrigued when I read that NMSLib can be up to 10x faster (https://github.com/erikbern/ann-benchmarks) when using its Hierarchical Navigable Small World Graph (https://arxiv.org/abs/1603.09320) (HNSW) index option. I also wanted to try out Faiss after reading the blog post (https://code.facebook.com/posts/1373769912645926/faiss-a-library-for-efficient-similarity-search/) that Facebook Research wrote about it - where they claimed that the GPU enabled version of Faiss was the fastest available option.

Both NMSLib and Faiss turn out to be extremely good at this task, and I've added code to implicit (https://github.com/benfred/implicit/pull/51) to use these libraries for generating recommendations.

Maximum Inner Product Search 最大内积搜索

One problem with using most of these approximate nearest neighbour libraries is that the predictor for most latent factor matrix factorization models is the inner product - which isn't supported out of the box by Annoy and NMSLib.

Unfortunately, getting the top nearest neighbours by the inner product is trickier than using proper distance metrics like Euclidean or Cosine distance. The challenge is that the inner product doesn't form a proper metric space (https://en.wikipedia.org/wiki/Metric_space): since the similarity scores for the inner product are unbounded this means that a point might not be its own nearest neighbour. This violates the triangle inequality and invalidates some common approaches for approximate nearest neighbours.

Luckily the paper "Speeding Up the Xbox Recommender System Using a Euclidean Transformation for Inner-Product Spaces" (https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/XboxInnerProduct.pdf) explains how to transform the inner product search so that it can be done on top of a Cosine based nearest neighbours lookup. This involves adding an extra dimension to each item factor, such that each row in the item factors matrix has the same norm. When querying, this extra dimension is set to 0 - which means that the cosine will be proportional to the dot product after this transformation has taken place, and enables us to use Annoy and NMSLib for generating recommendations here.

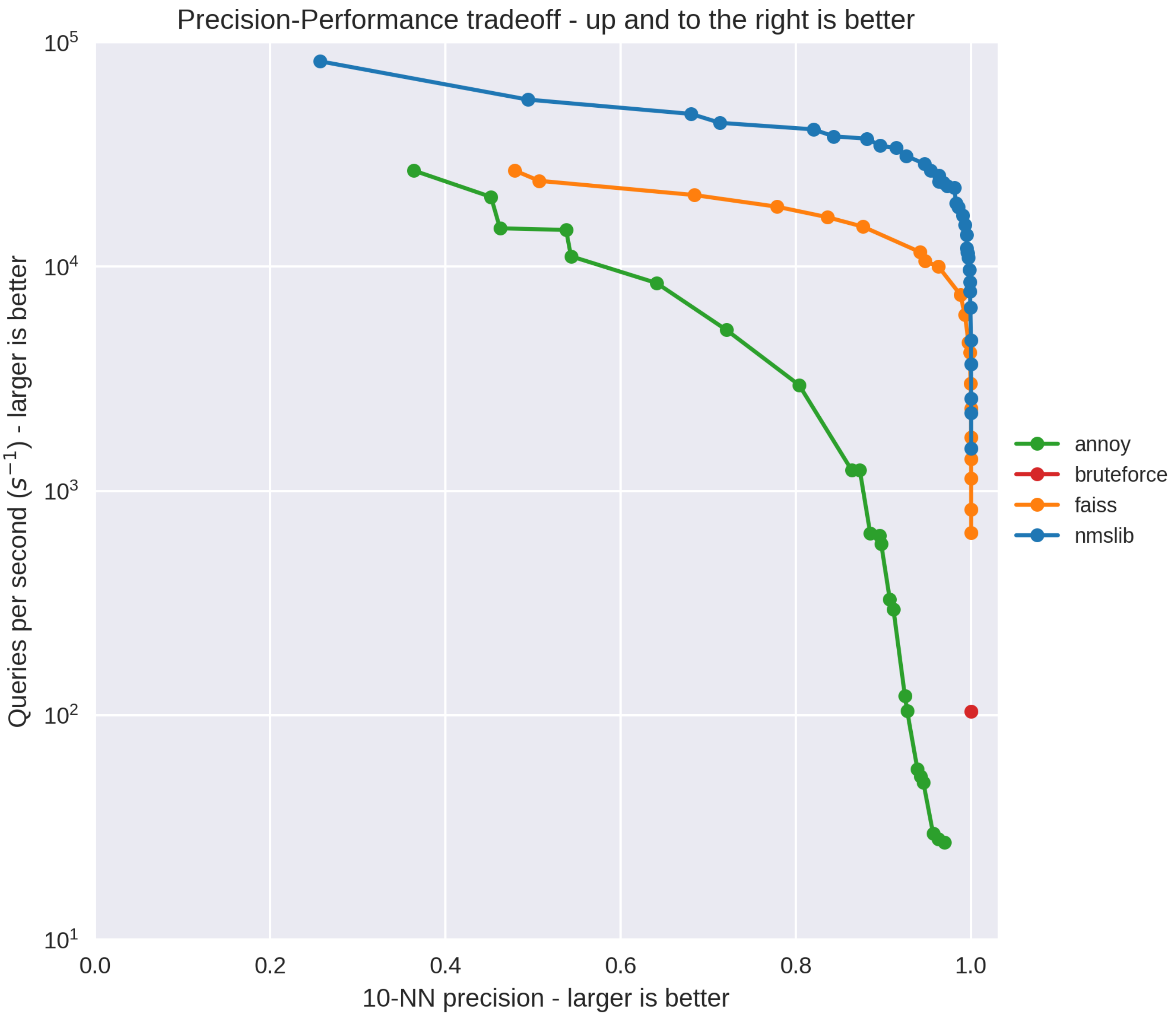
Measuring the Performance

Each library has several parameters that control how much effort to spend in doing the nearest neighbours search. Spending more time usually leads to better results, but the challenge here is to get good results quickly.

To measure these tradeoffs, I'm using the ann-benchmarks (https://github.com/erikbern/ann-benchmarks) package. For each ANN library, this package builds an index with different parameters and then records what percentage of the correct neighbours are returned and how long it takes to query for them. Using this data it then plots out the Pareto Frontier (https://en.wikipedia.org/wiki/Pareto_efficiency#Pareto_frontier) showing the optimal boundary between query time efficiency and accuracy.

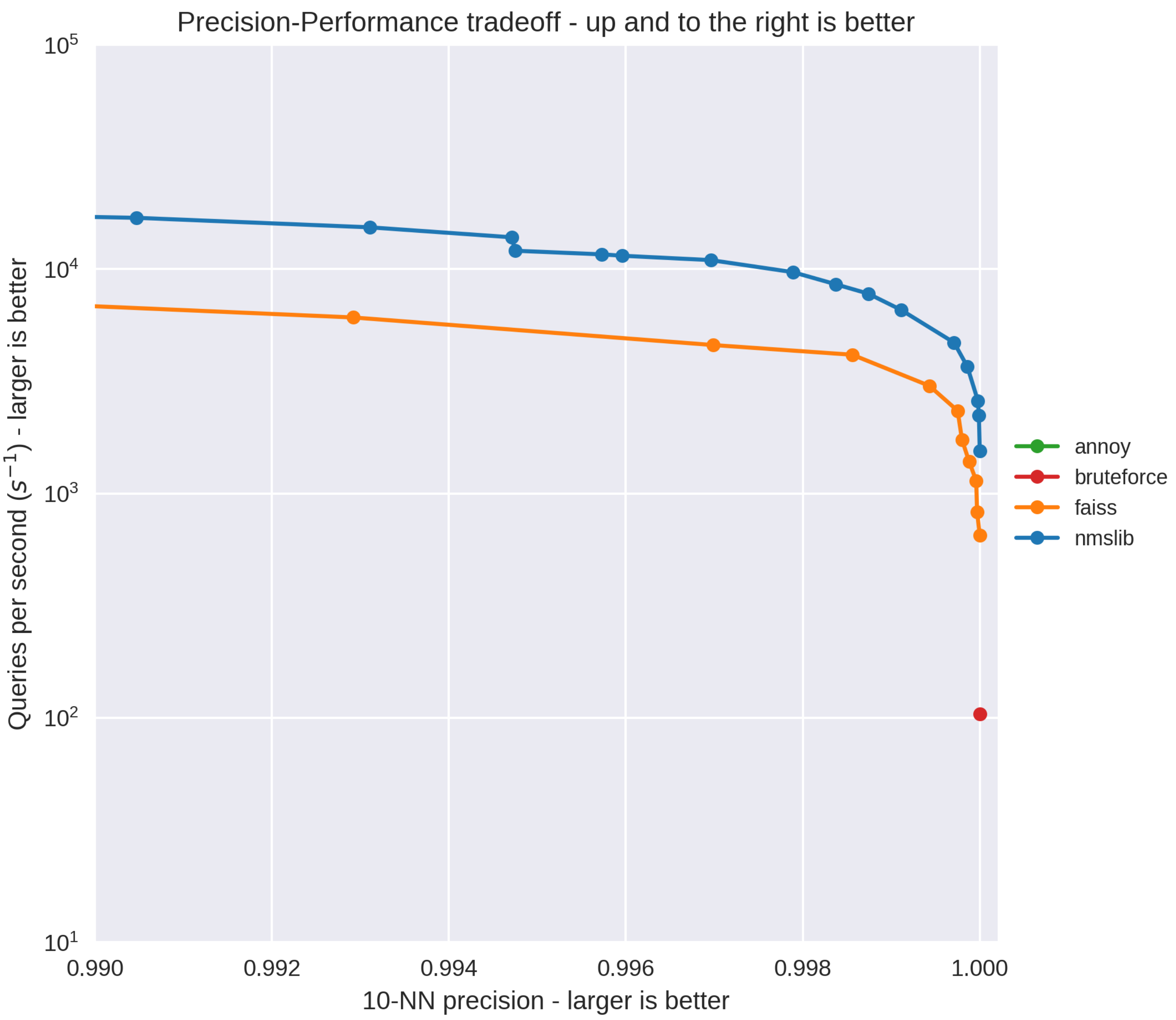
To use ann-benchmarks on this inner product search for matrix factorization models, I wrote a script (https://github.com/benfred/bens-blog-code/blob/master/approximate_als/create_ann_benchmarks_data.py) that trains a 50 factor ALS model on the last.fm dataset (http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/lastfm-360K.html), augments the item factors and user factors as described above and then dumps out the train/test set out in a format that ann-benchmarks expects.

With all this done, here are the results for each different library:



The big takeaway here is that the HNSW index from NMSLib substantially outperforms both Annoy and Faiss. Annoy seems to do extremely poorly on this test, which is surprising to me since on a Glove dataset using Cosine distance (https://github.com/erikbern/ann-benchmarks/pull/35) both Faiss and Annoy performed similarly on my system.

While NMSLib also outperforms FAISS, this difference starts to shrink at higher precision levels. Zooming in on to look at just 99% precision and above, and you can see this effect in action:

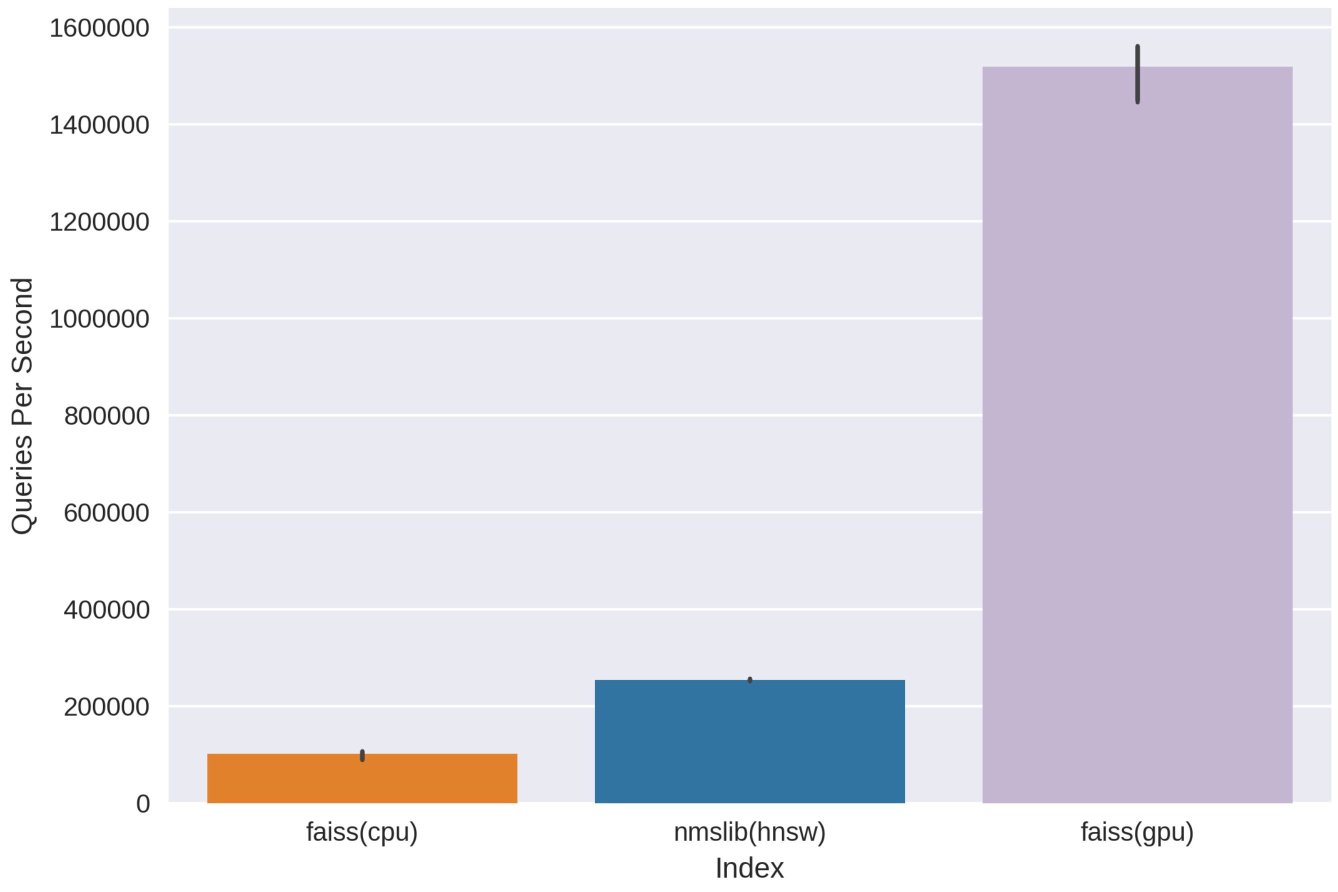


Recommendations on the GPU with Faiss

One of the cool things about Faiss is that it allows offloading computation onto the GPU - which the previous benchmarks I included weren't trying out. Also, the Faiss documentation (https://github.com/facebookresearch/faiss/wiki/Indexing-1M-vectors) indicates that providing queries one at a time will be 2x slower than doing batch queries, so I also thought it would be worth comparing batch querying speed instead of single thread sequential querying like in the above test.

To test out the GPU implementation of Faiss, I selected the fastest parameters for each index that had at least 99% precision in the previous test. I then wrote a little script (https://github.com/benfred/bens-blog-code/blob/master/approximate_als/test_batch_queries.py) to test the batch operation of NMSLib versus both batch CPU and GPU querying on Faiss with these parameters.

Both NMSLib and Faiss can parallelize the queries onto all the CPU cores, which on my system is a Core i7-7820x which has 8 cores/16 threads. The GPU I'm testing on is a NVidia GTX 1080 Ti (https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/), which has 3584 Cuda Cores. While this is a bit of an apples to oranges comparison, its still kind of interesting to take a look at:



As expected, the querying rates on the CPU in batch mode are roughly 16x higher for both Faiss and NMSLib, showing that they can both effectively parallelize onto all the available cores. NMSLib remains over twice as fast for querying on the CPU - but both numbers are completely dwarfed by the insane rates by Faiss on the GPU.

I only get 100 queries per second when computing the exact brute force nearest neighbours on a single core with this dataset. This means to compute the recommendations for each of the 360 thousand users takes around an hour. For comparison, NMSLib is getting 200,000 QPS and the GPU version of Faiss is getting 1,500,000 QPS. Instead of an hour, the NMSLib takes 1.6 seconds to return all the nearest neighbours, and the GPU variant of Faiss only takes 0.23 seconds - and both of them are still returning 99% of the relevant neighbours for each query.

Other Considerations

If you don't have access to a GPU on your system - NMSLib is by far and away the best choice here.

Both NMSLib and Annoy are easily installable in python: `pip install nmslib` and `pip install annoy` will install each respectively on most systems for all recent versions of python. However, Faiss requires manually editing makefiles to build, and then manually installing the resulting binaries into your python distribution.

While Annoy's performance is the worst at this particular task, it performs much better with cosine based lookup (like when computing similar items). Also, the indices are all memory mapped from file, which makes it much more suitable if you have multiple python processes serving up requests.

Finally in the interest of full disclosure, I should probably note that I've been contributing to NMSLib recently. I've been mainly working on things like improving the Python bindings, adding CI, and adding Python documentation. However, I don't think this has biased these results at all: I've been contributing to NMSLib because I've been impressed by its performance, rather than making claims about its performance because I've been contributing to it.

All the test code for this post is on my github (https://github.com/benfred/bens-blog-code/tree/master/approximate_als). I've also added support for NMSLib and Faiss (https://github.com/benfred/implicit/pull/51) to my implicit (https://github.com/benfred/implicit) recommender systems library for generating recommendations.

Published on 11 October 2017