

Approximate Nearest Neighbors (ANN)

Andrew Li (ayl2159)

Summary

Vector Search is a retrieval technique that is used to find similarities between words or other modalities based on conceptual meaning. Text, images, audio, or video are first converted into vector embeddings in a high dimensional space, and then items that are similar to each other map to nearby points in this space.

K nearest neighbors is a brute force solution to find the mathematical distance between the query vector and every other vector in the database. Since we have to touch every vector, this runs in $O(N)$ linear time.

In production with millions to billions of vectors, this may be too inefficient. To solve the speed problem, vector tools look to approximate nearest neighbors (ANN). It trades the exact accuracy guarantee of finding the closest neighbor, in exchange for being much faster. The algorithm organizes the high dimensional space into data structures, such as a graph, in order to scan a subset of the dataset instead of the whole dataset. This runs in $O(\log n)$ time.

Serial Algorithm

We will first implement Approximate Nearest Neighbors using Random Projection Trees in high-dimensional vector spaces. It partitions data using a divide and conquer strategy that works well for high dimensional data.

Here is what's done in the algorithm:

1. High dimensional vectors are projected to a lower dimensional space. This preserves spatial relationships while avoiding the curse of dimensionality.
2. At each tree node, the points there are projected onto a random line. We find the median value, and split the dataset into two child trees based on that point. (Think of it as split based on a decision boundary / hyperplane)
3. We do this binary splitting recursively to achieve $O(\log n)$ time.

This is all done in an existing reference repository: [ocramz/rp-tree](https://github.com/ocramz/rp-tree) (on Hackage as [rp-tree](https://github.com/ocramz/rp-tree)). It implements:

- Vector types based on unboxed contiguous arrays (`Data.Vector.Storable` / `Data.Vector.Unboxed`).
- Random hyperplane generation and dot-product projection splits.

- Recursive tree construction (`tree`, `forest`).
- Nearest-neighbor lookup and tree traversal.

Therefore, we can import `rp-tree` and focus on the parallel engine on top of it.

Parallelization Plan & Anticipated Difficulties

First, we'll build a forest of T trees concurrently by mapping independent random generator seeds across capabilities with `parMap rdeepseq` or `Control.Monad.Par`.

We'll evaluate the subtrees recursively with depth-bounded sparks (`rparWith rdeepseq`) to evaluate left and right subtrees in parallel.

We'll also use chunk holdout query batches across GHC worker capabilities using `parListChunk` in order to process data in parallel.

Some difficulties might arise around correctness; ANN is supposed to be a randomized algorithm (where the hyperplane we choose is random, so we may split the absolute nearest neighbor into a different child tree), so I'm worried there will be some parallelization strategies that may affect accuracy. That will need to be closely monitored.

We'll also need to see how many sparks to spawn, since there is a scheduling cost to them. There are many settings and numbers to tune in general.

Benchmarking

We can run the exact same dataset through both serial and parallel versions where $N = 100$, 1000, and 10,000 points.

We can then analyze the speedup across `-N1`, `-N2`, `-N4`, `-N8`, `-N16` and analyze spark behavior via ThreadScope.

Input Data

There are ANN source repositories on github, and in general vector datasets (such as GloVe); but synthetic data is also straightforward to generate, as all we need is a collection of vectors. This can be done with synthetic Gaussian mixtures.

Resources

- <https://github.com/ocramz/rp-tree> - reference sequential ann implementation

- <https://github.com/erikbern/ann-benchmarks> - ann input data
- Google Gemini for information on the ANN algorithm, and as a search for useful Haskell modules.