KNN Optimization Methods

Denys Kaliuzhnyi [1] Pavel Chizhov [1]

[1]*Institute of Computer Science, University of Tartu*

UNIVERSITY OF TARTU 1632

Introduction

K Nearest Neighbours (KNN) is one of the simplest, yet very efficient in many cases, methods of machine learning. The main problem is, a KNN model is very slow at making predictions, because it needs to somehow compare the queried input with all the training data to find the closest data points. Therefore, there is a need for predictions optimization, where efficient searching data structures may be of help.

In this project we aim to implement and compare two methods of KNN optimization: K-d tree and Ball tree. We are going to describe and visualize them, and compare their performance in a variety of situations.

Baseline

As a baseline we will consider a simple brute-force algorithm, when we compare the prediction input with all the data we have in the training set.

Obviously, the fitting time is $O(1)$, because we only consume the data. This approach allows to get k nearest neighbours in $O(dn)$ time, where d is number of dimensions and n is training set size. The time does not depend on k, since the selection of nearest neighbours is done with partition procedure, which takes $O(n)$ time, regardless of k.

K-d Tree

K-d tree [1] is a multi-dimensional binary search tree. On each level a split is done by a median element in a certain dimension, which are sequentially interleaved. Thus, each node is an axis-aligned hyperplane that splits its own segment of space.

A full k-d tree is built in $O(nlog^2n)$ time: it needs sorting (which is done in $O(nlogn)$ time) to find medians on each of log_2n levels. A full k-d tree is a massive data structure, because on lower levels it has plenty of splits, so the leaves are usually made to contain not 1 but L elements, which we will further refer to as a *leaf size*. Therefore, the building time changes to $O(nlognlog\frac{n}{L})$.

The search in a k-d tree takes $O(logn)$ time, as in a usual search tree. However, search is not enough to find nearest neighbour: the queried element can be close to the hyperplane, which potentially obliges the algorithm to visit neighbouring leaves as well. Thus, in worst-case scenario search time can inflate up to $O(n)$.

Figure 1 illustrates a 2-d tree with red and blue splitting lines and neighbour search in this tree. The dark leaves are the ones that were visited.

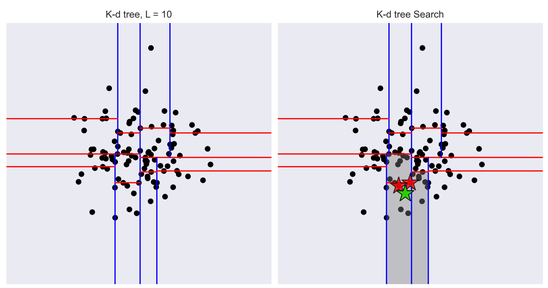

Figure 1. K-d Tree.

Multiple neighbours support

Support of multiple neighbours in our optimization implementations is done with **priority queue** data structure. It contains k current minimums ordered in a heap with the following operations provided:

- **Building** in $O(1)$ time, since the queue is initialized with infs.
- **Pushpop** (insertion with size preservation) in $O(logk)$ time.
- k **minimums** in $O(1)$ time, because the size of the queue always remains k.

Ball Tree

Likewise, ball tree [2] is a binary search tree aimed to efficiently store multidimensional data. It is constructed under the similar logic as k-d tree, but the main difference is that splits are no longer axis-aligned. Instead, each division is associated with random vector that approximates direction of the greatest spread. Data is projected onto that line and then median value divides original points into two halves. The second distinctive aspect of ball trees is that points are encapsulated into hyperspheres.

With regards to building time complexity, it needs $O(dn)$ time to perform data projection as well as calculating ball radius and $O(nlogn)$ time to sort projected values. Repeating that at every of $logn$ levels yields an $O(logn(dn+nlogn))$ total complexity. If *leaf size* $L>1$ is taken into account, we obtain $O(log\frac{n}{L}(dn+nlogn))$ time.

Even though building ball tree is much more costly compared to k-d tree, the search as well as query neighbors time are the same - $O(logn)$ and $O(n)$ respectively. The latter is an upper estimate, but commonly $logn$ takes place. However, a constant factor of a ball tree is higher due to more costly distance operations.

Figure 2 shows how ball tree visually covers 2-D points. Circles of the same gray intensity are balls of one level and those outlined in blue were visited in the query.

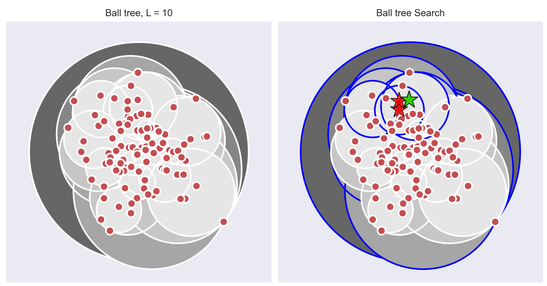

Figure 2. Ball Tree.

Testing Environment

All the three methods of neighbour search were tested and compared along a variety of parameters: dataset size n, dimensionality d, neighbours number k and leaf size L for k-d and ball trees.

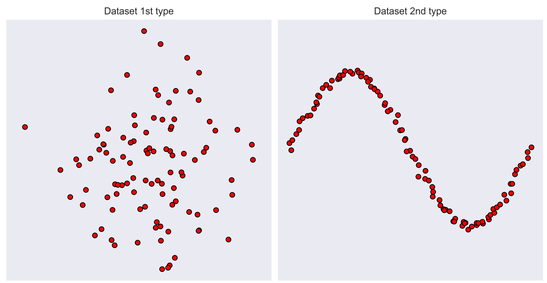

Figure 3. Datasets.

Special attention was paid to the data: we performed testing on two kinds of datasets: Gaussian noise and a sinusoidal cloud (Figure 3). Along with this, we tested algorithms on the multi-dimensional datasets with useless features - some of the features were just noise.

Project repository

The results demonstrated in this report are only a part of the work. The full code and collection of the results can be found on github.

Results

In this section we are presenting a part of the results, which, in our opinion, leads to the most valuable inferences.

Figures 4 and 5 demonstrate trials over dimensionality. In Figure 4 we see that k-d tree in general works much faster, regardless of growing number of dimensions. However, if we add useless features (Figure 5), k-d tree loses its excellence, whereas Ball tree improves its performance significantly.

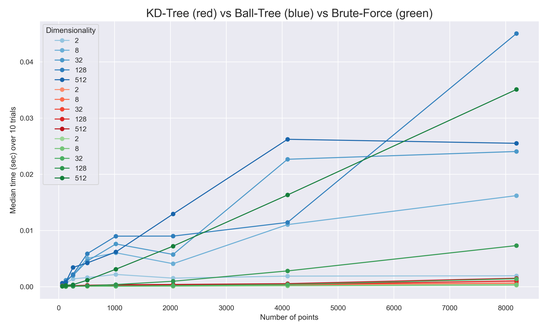

Figure 4. Dimensionality testing on Dataset 1.

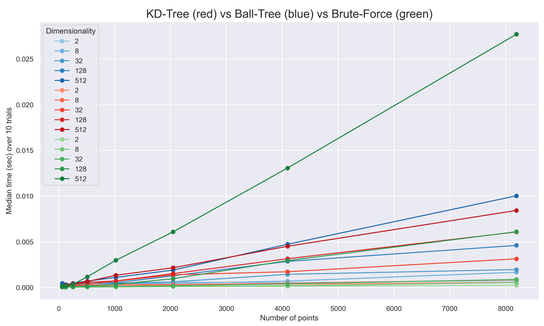

Figure 5. Dimensionality testing with dummy features on Dataset 2

Growing number of neighbours affects significantly both optimization methods (Figure 6), because they both support a priority queue, while Brute-force approach, as expected, does not depend on k at all.

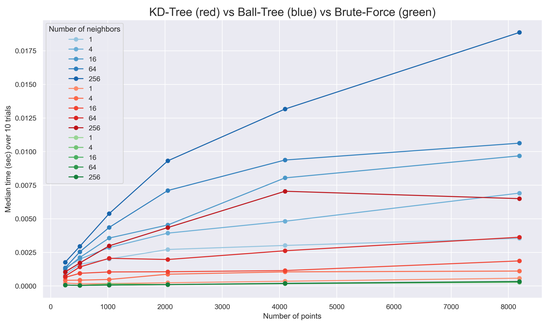

Figure 6. Number of neighbours testing on Dataset 1

References

[1] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[2] K. Weinberger. Machine learning lecture 28. *Cornell University*, 2018.