Assignment 2, HPPS

Silas Nøstvik, vld897
Tobias Sædder Schiolborg, bnp997

Københavns Universitet
Machine learning og datavidenskab
High Performance Programmering og Systemer
Hold 03

December 11, 2020

This report investigates and evaluates two approaches for the implementation of the k-NN learning algorithms: bruteforce and k-d tree.

Implementation

The following section will focus on the implementation of the incomplete files provided.

The file "io.c" functions "read_points", "read_indexes", "write_points", and "write_indexes" has been taken from the lab session 2-l-2.

From the file "util.c" the functions "distance" and "insert_if_closer" has been implemented.

The implementation of "distance" is relatively straight-forward, but has surprisingly been causing a few issues. This is due to us realizing that comparing squared distances works just as well in KNN (i.e. dropping the sqrt operation to avoid a difficult computation), but as the distance-function is put to use in multiple places in the existing code, we use actual norms instead.

The implementation of "insert_if_closer" is a tad more complicated; the array "closest" is designed to consist of the indexes to the k nearest neighbours of a query in increasing order of distance, which makes it easier to add a new element, because the elements in "closest", which are larger than the candidate, can just be pushed back in the array, thus making room for the candidate and removing the largest element. The loop, which push the rest of the elements back in the array, starts exactly where the first loop, that finds room for the candidate, left, thus increasing the locally. The distances themselves are not stored along with the indexes. Originally, we implemented the array using flat indexes, but, as per our prior experience, we realized that "insert_if_closer" was implemented elsewhere in the existing code which expected row-major indexes, so we conformed to the row-major indexing.

The bruteforce-knn algorithm has been implemented more or less directly from the pseudo-algorithm using calls to "insert_if_closer".

The following functions for the k-d tree implementation have been implemented: "kdtree_create_node", "kdtree_free_node" and "kdtree_knn_node". When the k-d tree is created, "kdtree_create_node" is called, and it pretty much follows the given procedure. To find the median, the indexes are sorted with respect to their given points along the given axis. This is done similarly to in lab 2-I-2 with "hpps_quicksort", so the comparison function "cmp_indexes" and the struct "sort_env" have been created.

In the end of the given procedure, it tells us to go directly to both the left and right node. Instead of this, only when the sub-array of points left/right of the median are non-empty, then the left/right nodes are visited. To create the sub-arrays, the corresponding indexes are copied to some freshly allocated arrays, which are freed directly after creating the new nodes. This leads to "kdtree_free_node", that simply, recursively removes/frees the left and right nodes in the tree.

The "kdtree_knn_node" function also follows the given procedure, where "insert_if_closer" again is used to handle the insertion of points in "closest". The only difference from the procedure is that the radius (most distance point in "closest") only is computed when "closest" is updated.

Correctness

Multiple approaches has been taking to check the correctness of the implementation. One of the more basic, but also more intuitive methods have been training the algorithms on 2D points and visually inspecting the results, e.g. on Figure 1 and Figure 2.

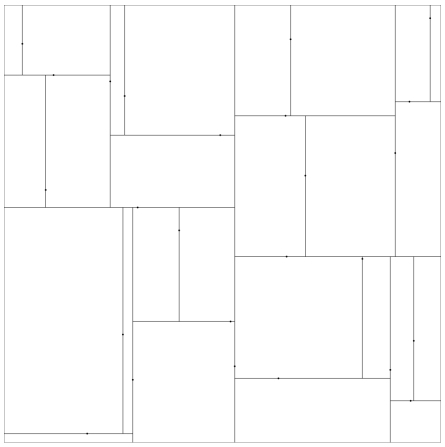


Figure 1: K-d tree, 25 points

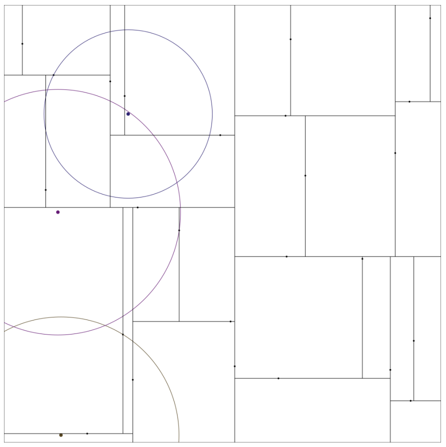


Figure 2: K-d tree, 25 points, 3 queries, 3-NN encircled

Test results

The test program is the file "tests.c" which performs a testing of both the bruteforce- and the k-d tree algorithm. The executable will be build by the "make" call in terminal.

For larger datasets with $d > 2$ a visual inspection is not viable. Instead the resulting indexes for each query is verified by computing the distance between the farthest NN and the query. Then the distances between the query and all points are compared to this distance, and if less than k points qualify (equal or less than the distance of the farthest NN) the test fails. If not the test passes, i.e. the indexes where in fact the nearest neighbours. One might imagine that the farthest neighbour is tied in distance with a point which is not in "closest" . In this case the test also passes. These ties are very rare, however, as the distance is a floating point.

The k-d tree algorithm should produce the same results as the bruteforce algorithm. Ties for distance may cause inconsistencies, but as previously explained ties are rare and will hence not be considered. The comparison between the output indexes of bruteforce and kd-tree are performed using the Unix "cmp" tool.

The test output can be seen below:

```
Test 1: n_points=10, n_queries=4, d=2, k=2
Number of queries handled correctly by knn-bruteforce: 4/4
knn-bruteforce and knn-kdtree output comparison: SUCCES

Test 2: n_points=15000, n_queries=4, d=500, k=2
Number of queries handled correctly by knn-bruteforce: 4/4
knn-bruteforce and knn-kdtree output comparison: SUCCES

Test 3: n_points=500, n_queries=200, d=500, k=10
Number of queries handled correctly by knn-bruteforce: 200/200
knn-bruteforce and knn-kdtree output comparison: SUCCES

Test 4: n_points=10, n_queries=1000, d=500, k=10
Number of queries handled correctly by knn-bruteforce: 1000/1000
knn-bruteforce and knn-kdtree output comparison: SUCCES

Test 5: n_points=500, n_queries=200, d=1, k=1
Number of queries handled correctly by knn-bruteforce: 200/200
knn-bruteforce and knn-kdtree output comparison: SUCCES
```

Run-time of bruteforce vs k-d tree

Table 1 is a overview of the performances of the bruteforce algorithm compared to the k-d tree algorithm using input data of various sizes and dimensions for the training process using different values for k and different amounts of query points.

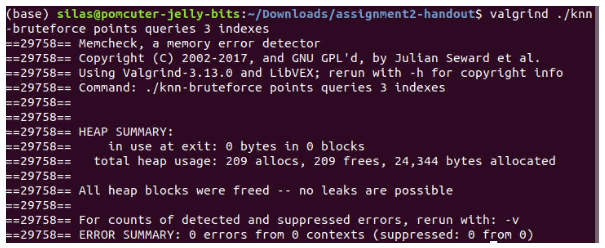
The k-d tree algorithm should have time complexity $O(k \log n)$, while the bruteforce algorithm is more along the lines of $O(n)$. We see that even though the testing is performed on relatively high amounts of data points and the k-d tree thus should win out, this does not happen for the case of a single query. This might have something to do with the initial prize of having to build the k-d tree (higher constant costs). As the amount of queries is increased (for $n = 10^5$) the performance ratio seems to converge on a factor 2 (refer to row 3 and 4). This is lower than expected, as two processes of time complexity $O(k \log n)$ and $O(n)$ for $n = 10^5$ could have a performance difference of ratio of (for similar leading constants): $\frac{10^5}{\log 10^5} = \frac{10^5}{5 \log 10}$. We cannot with certainty explain why the performance ratio is nowhere near this level, but we would expect the ratio increase with the number of data points.

	Test	points	dimension	queries	k	run-time (sec)	ratio
1	bruteforce kdtree	100,000	10	1	10	0.334 1.101	0.303
2	bruteforce kdtree	100,000	10	10	10	3.119 2.431	1.283
3	bruteforce kdtree	100,000	10	100	10	35.482 17.276	2.054
4	bruteforce kdtree	100,000	10	1000	10	351.55 173.79	2.023
5	bruteforce kdtree	1,000,000	10	1	10	3.137 16.951	0.185
6	bruteforce kdtree	1,000,000	10	10	10	31.368 20.275	1.547
7	bruteforce kdtree	100,000	10	10	100	29.321 23.031	1.273
8	bruteforce kdtree	100,000	100	10	10	32.559 35.298	0.922

Table 1: Run-times of bruteforce- vs. k-d tree approach

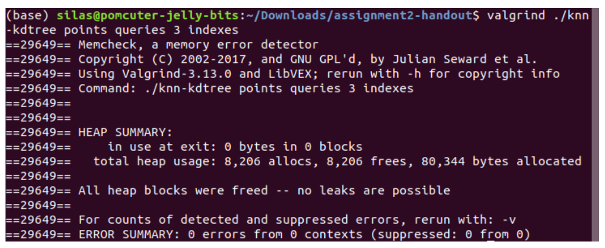
Memory leak

Using Valgrind no memory leaks were detected, refer to Figure 3 and Figure 4.



```
(base) silas@pomcuter-jelly-bits:~/Downloads/assignment2-handout$ valgrind ./knn
-bruteforce points queries 3 indexes
==29758== Memcheck, a memory error detector
==29758== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29758== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==29758== Command: ./knn-bruteforce points queries 3 indexes
==29758==
==29758==
==29758== HEAP SUMMARY:
==29758==     in use at exit: 0 bytes in 0 blocks
==29758==   total heap usage: 209 allocs, 209 frees, 24,344 bytes allocated
==29758==
==29758== All heap blocks were freed -- no leaks are possible
==29758==
==29758== For counts of detected and suppressed errors, rerun with: -v
==29758== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 3: Valgrind output for bruteforce call



```
(base) silas@pomcuter-jelly-bits:~/Downloads/assignment2-handout$ valgrind ./knn
-kdtree points queries 3 indexes
==29649== Memcheck, a memory error detector
==29649== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29649== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==29649== Command: ./knn-kdtree points queries 3 indexes
==29649==
==29649==
==29649== HEAP SUMMARY:
==29649==     in use at exit: 0 bytes in 0 blocks
==29649==   total heap usage: 8,206 allocs, 8,206 frees, 80,344 bytes allocated
==29649==
==29649== All heap blocks were freed -- no leaks are possible
==29649==
==29649== For counts of detected and suppressed errors, rerun with: -v
==29649== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4: Valgrind output for kd-tree call

Improvement reflection

Our implementations produce results that seem to be correct, but they are by no means perfect. This section will discuss what could have been done better, given more time:

The ”closest” array could have been implemented more efficiently. As of now, the distances for each of the k NN are not saved along with their indexes and so have to be recomputed with every point. As a result our implementation of bruteforce has an asymptotic complexity more along the lines of $O(k \cdot N)$, rather than $O(n)$ (if we did not recompute distances).

A way of implementing the storage of the distances could be to define a struct type that contains an index and a distance. ”closest” could then point to an array of this type, rather than array of indexes.

The sorting done in k-d tree to find the median is done for each node in the tree, so even though the array getting sorted keeps getting smaller, it could be a good idea to sort the indexes for each axis before beginning the creation of the k-d tree. This would likely decrease the run-time, particularly when the number of points are very large with respect to the number of dimensions. This would, however, require a couple of extra copies of the indexes array, one for each dimension.

Another improvement would have been more extensive testing. For instance, the bruteforce testing is very dependent on the indexes being sorted in increasing order. It would make sense to test if this can be relied on to be correct.

A higher focus on error-handling could make the software more reliable.