# ANN Parameter Analysis

In this file, the ANN algorithms' results will be analyzed, depending on the parameters given. Keep in mind that these algorithms have an element of randomness, so results may differ among executions (with the same parameters). The tests are all executed with N = 1 (amount of nearest neighbors kNN returns) and R = 5000 (radius of range search) although the last one is irrelevant. The calculations on accuracy and performance are all done via the following python script: scripts/getAverage.py. This script accepts 3 arguments: the output file LSH or Hypercube produced, the number of queries on that file, the type of algorithm used (LSH or HC)

## Locality Sensitive Hashing

The parameters that can be modified are the following: - W (window size) - L (number of hash tables) - k (number of hash functions that create amplified hash function)

#### Window Size (W)

Let us assume constant values for L and k. In these examples L = 7 and k = 5. 1. W = 40000
- Time Ratio (average time for LSH kNN / average time for brute force kNN) : 0.6182 - Accuracy (mean distance for brute force kNN / mean distance for LSH kNN): 99.394%
We can safely assume that LSH with those parameters is extremely accurate. However, the time consumed to query the items does not differ much from the brute force implementation. 1. W = 10000
- Time Ratio: 0.1265 - Accuracy: 72.537%
When W equals 10000, we notice significant change from 40000. The time ratio has dropped to 0.12 (from 0.6) and the accuracy has dropped to 72% (from 99%) 1. W = 4000
- Time Ratio: 0.1256 - Accuracy: 56.105%
With W = 4000 although the algorithm may seem fine timewise (almost 1/10 of brute force and a bit lower from W=10000), the accuracy drops dramatically to nearly 56%. 1. W = Average Item Distance * Constant.
Through brute force, the average item distance of the items is calculated (for a portion of the training set). This distance is then multiplied by a constant C. Average Item Distance is approximately 33000 (calculated from 5% of the training set) 1. C = 1 (Window Size = 33184) - Time Ratio: 0.5133 - Accuracy: 99.122%
This execution is similar to W = 40000. Accuracy drops by a little, but time ratio drops by 0.10 2. C = 2 (Window Size = 66369) - Time Ratio: 2.4336 - Accuracy: 99.989%
With a window size of 66369 we notice that LSH is more time consuming than brute force! Accuracy is extremely high, however this implementation is utterly useless. There is always a trade-off between accuracy and speed. The best values for W (as examined above) are 40000, 10000 and Average Item Distance * 1. These values achieve a balanced trade-off and it is up to the user to select which fits his model best.

#### Number of Hash Tables (L)

Let us assume constant values for W and k. In these examples k = 5 and W = Average Item Distance * 1 (~=33000) 1. L = 1 - Time Ratio: 0.1161 - Accuracy: 90.228%
Almost 1/10 faster than brute force, with an acceptable accuracy. 2. L = 2 - Time Ratio: 0.1466 - Accuracy: 93.383%
A little bit slower than the previous example, accuracy has improved by almost 3% 3. L = 4 - Time Ratio: 0.1562 - Accuracy: 95.006%
Very small time difference than L=2, yet accuracy has improved by almost 2% 4. L = 6 - Time Ratio: 0.3575 - Accuracy: 98.270
Time has almost doubled. Accuracy has improved by more than 3% 5. L = 8 - Time Ratio: 0.8860 - Accuracy: 99.621%
This value of L seems to be a slight "overkill" as speed is close to brute force. Accuracy, however, is at a very respectable 99.621%
All values of L presented above are respectable in terms of balanced accuracy vs performance. For calculating the best value of k we will use L = 4 as it is overall fast and also highly accurate.

#### Number of Hash Functions in Amplified Hash Function (k)

Let us assume constant values for L and W. In the following examples L = 4 and W = Average Item Distance * 1 (~=33000) 1. k = 2 - Time Ratio: 1.0734 - Accuracy: 99.851%
Performance is worse than brute force, accuracy is close to 100%. Not acceptable. 2. k = 3 - Time Ratio: 0.4442 - Accuracy: 98.165
Performance is a lot better than k = 2 and accuracy is less than 2% lower. 3. k = 4 - Time Ratio: 0.2814 - Accuracy: 96.648
Performance is almost half of k = 3, accuracy has dropped almost the same amount. 4. k = 5 - Time Ratio: 0.2959 - Accuracy: 96.736
Performance and accuracy are almost the same as in k = 4. 5. k = 6 - Time Ratio: 0.3851 - Accuracy: 98.353
Performance and accuracy are near to the results of k = 3 indicating that there is no linear correlation between k and accuracy or performance.

#### Summary

Disregarding some edge cases, we can clearly see that the user can parameterize LSH to the way they see fit. For example a really fast combination could be W = 10000, k = 4, L = 1 with Time Ratio = 0.01 (!) yet its accuracy is at 47%. On the contrary, a very accurate combination (without exceeding brute force times) is W = Average Item Distance * 1, k = 5, L = 7 with a time ratio of 0.88 (close to brute force) but accuracy of 99.621%. Finally, an "all-around" combination (fast with relatively high accuracy) is W = Average Item Distance * 1, L = 4 and k = 5.

## Hypercube

The parameters that can be modified are the following: + Window Size (W) + Number of Hash Functions - Dimension of Hypercube (k) + F random number range (F_Range) + Item threshold (M) + Probes threshold (probes)

#### Window Size (W)

Let us assume constant values for k, F_Range, M, probes. In the following examples k = 13, F_Range = (100, 200), M = 20, probes = 3. 1. W = 400 - Time Ratio: 0.3474 - Accuracy: 41.561%
Comparing to LSH implementations around the same performance, the accuracy is very low. 2. W = 4000 - Time Ratio: 0.3447 - Accuracy: 41.561%
The results seem equal to the ones with W = 400. 3. W = 40000 - Time Ratio: 0.3381 - Accuracy: 45.195%
Accuracy has improved by 4% yet still remains low. Performance is almost equal to before.
4. W = Average Item Distance * Constant Average Item Distance is approximately ~33000 1. C = 1 - Time Ratio: 0.3344 - Accuracy: 45.195%
Results seem equal to the ones with W = 40000. We have noticed that in both LSH and Hypercube the perfect window size is Average Item Distance * 1. 2. C = 2 - Time Ratio: 0.3366 - Accuracy: 45.195%
Results yet again seem equal.

We can see that as window size increases, so does the accuracy of the model (yet performance remains relatively the same). We will be using a window size of

Average Item Distance * 1 for our future experiments.

#### Thresholds (M and probes)

We are going to examine those two parameters simultaneously as they are correlated with each other (having a small M would negate any probe after the starting, also having a huge M would be negated by having a small amount of probe threshold). In the following examples k = 13, W = Average Item Distance * 1, F_Range = (100, 200)

1. M = 100, probes = 2
   - Time Ratio: 0.3360
   - Accuracy: 53.912%
     Accuracy has increased significantly (from M = 20, probes = 3), performance remains relatively the same, so we will examine higher values for M and probes.
2. M = 500, probes = 4
   - Time Ratio: 0.3446
   - Accuracy: 64.397%
     Accuracy increased by nearly 10% (!), performance seems equal.
3. M = 1000, probes = 5
   - Time Ratio: 0.3452
   - Accuracy: 69.032%
     Accuracy increased by 5%, performance equal. So far it seems that as we increase those thresholds, accuracy increases by a lot while performance stays the same.
4. M = 4000, probes = 10
   - Time Ratio: 0.4100
   - Accuracy: 79.593%
     This is the first time performance has gotten worse (only by a small amount). Accuracy is starting to reach acceptable levels.
5. M = 10000, probes = 20
   - Time Ratio: 0.6487
   - Accuracy: 86.501%
     This time performance has increased yet again and is reaching brute force levels. Accuracy has increased yet again by 6%, however still remains below 90%

From the above experiments we can clearly see that the first 3 experiments simply increase accuracy yet keeping performance equal. So we may assume that the best values for M and probes are above 1000 and 5 respectively. After that, the correlation between accuracy and performance becomes increasingly more visible. For the rest of our experiments on other parameters, we will keep M = 4000 and probes = 10.

#### Number of Hash Functions - Hypercube Dimension (k)

Now we are going to take a look at k. The values we will use for other parameters will be: W = Average Item Distance * 1, M = 4000, probes = 10, F_Range = (100, 200)

1. k = 2
   - Time Ratio: 0.0902
   - Accuracy: 79.593%
     Performance has increased dramatically, accuracy seems around equal
2. k = 4
   - Time Ratio: 0.1668
   - Accuracy: 88.358%
     This is by far the best hypercube experiment we have had. Accuracy is the highest so far and performance is the second best so far!
3. k = 6
   - Time Ratio: 0.1102
   - Accuracy: 79.593%
     The results seem the same with the ones from k = 2 (although performance is a little worse, we may assume it is equal since there is always a small error)
4. k = 8
   - Time Ratio: 0.1203
   - Accuracy: 79.593%
     Yet again, results seem to be the same.
5. k = 10
   - Time Ratio: 0.1452
   - Accuracy: 79.593%
     Accuracy is the same, performance is getting incrementally worse
6. k = 16
   - Time Ratio: 2.506
   - Accuracy: 79.593%
     Accuracy still remains equal, performance has worsened dramatically (2.5 times the time of brute force)

In these experiments all values of k apart from k = 4 have the same accuracy. Among those, k = 2 is the fastest. For k = 4, accuracy has increased significantly and performance remains balanced, this is the value that we will use for our next and final parameter.

#### F random number range (F_Range)

This is a range instead of a single value. There has to be careful consideration for this range because it can affect the way items are distributed among vertices. A "bad" range could mean that all items are inserted into a small subsection of vertices. On the other hand, a "good" range could mean that items are distributed uniformly among all vertices.

1. F_Range = $(0, 2^{32/k})$
   - Time Ratio: 0.1344
   - Accuracy: 83.344%
     Performance is at a respectable level, accuracy is relatively low
2. F_Range = $(2^{32/k}/4, 2^{32/k} * (3/4))$
   - Time Ratio: 0.1403
   - Accuracy: 84.653

It seems that H values are not uniformly distributed among $(0, 2^{32/k})$. So picking a smaller range inside that seems to have worked in our favor.

3. F_Range = (meanOfHashValues-deviationOfHashValues, meanOfHashValues+deviationOfHashValues)
    - Time Ratio: 0.1774
    - Accuracy: 87.743
      Since F_Range's job is to distribute hash values evenly onto hypercube vertices, it is natural to think that the middle value of the range should be the mean of the dataset and the distance from it should be the standard deviation of the dataset. Indeed we can see that accuracy is at its most among F_Range experiments. Performance has dropped, but only by a small amount. Overall, this is arguably one of the best F_Ranges that can be set up so as to achieve high accuracy with low performance cost. In the example dataset, this range is around (136, 188) although keep in mind that the results are based on randomness.

**Summary**

As was the case with LSH, Hypercube is heavily parameterizable. The user can pick any combination of parameters to achieve his goal by either tuning down accuracy for the sake of higher performance or vice versa. A good "all-around" combination is W = Average Item Distance * 1, M = 4000, probes = 10, k = 4 and F_Range = (mean-deviation, mean+deviation).

# User-Tweaking

The user can obviously change the values of the aforementioned parameters easily.

**For LSH:**

L and k can be given as command line arguments while W needs to be set inside exc1/LSHMain.cpp. There is a helper function called "averageDistance" which calculates the Average Item Distance for a portion of the training dataset.

**For Hypercube:**

M, probes and k can be provided as command line arguments while W and F_Range needs to be set inside exc1/HCMain.cpp. As mentioned above there is a helper function for the Average Item Distance. Alongside that, there exists another helper function to calculate the Mean and Deviation of a portion of the dataset which is applied on a number of LSH Hash Functions.