Computing Pairwise Distance Between Data

(Python Only)

**Loops vs Matrix operation methods**

Loops are very expensive techniques to use in Python, in terms of running time. Instead we can take advantage of the Numpy Library available for python. This library already implemented some very efficient methods which can handle very important and complex matrix or vector operations. The methods are already compiled and implemented in C so the loops are executed inside them.

In order to verify this claim, we will build a simple program to find the Hamming Distance, which will be solved using both methods, respectively using loops and the Numpy built-in methods. Afterwards, the results will be measured and plotted to make it easier for us to support our claim.

# Setting up the test

The idea behind the Hamming Distance between two vectors is quite straight forward. Below is the formula used to find the Hamming distance between two vectors **p** and **q** with N features or parameters.

𝑛

𝐻𝐷 − Qi|

𝑖

Given that the vectors being compared for their Hamming distance consist on Bit values only, the hamming distance is the sum of the “bits” that are different from one vector to the respective bits of the other one.

Our program will store these values in an ZxZ matrix where Z is the total number of the vectors. Each Z[i][j] will have the Hamming Distance between vector i and vector j.

The loop solution for this problem consists on two nested **for**-loops which will help us loop through every Z[i][j] cell.

|  |
| --- |
| **def hamming\_dist\_loop**(X):  N = np.zeros([len(X), len(X)])    **for** i **in** range(len(X)):  **for** j **in** range(i+1, len(X)):    **for** c **in** range(len(X[0])): **if** X[i][c] != X[j][c]:  N[i][j] += 1  N[j][i] = N[i][j] **return** N |

On the left, we can see the loop-solution method. The X is a 2D array where each row X[i] represents one vector. The array N is where every distance within any two vectors will be stored.

The nested **for-**loops iterate through the N array. The third for loop is the one that iterates through each value of the vectors being compared. Since the distance from X[i] to any other X[k] is the same as the distance from X[k] to X[i] we only need to (𝑙𝑒𝑛𝑔𝑡ℎ(𝑋))2

make only calculations. We implement this

in the last two lines of the method. 2

The fast solution for this problem consists on the implementation of several methods build-in the Numpy library.

|  |
| --- |
| **def hamming\_dist\_fast**(X):  N = np.zeros([len(X), len(X)])    **for** row **in** range(len(X)):  N[row] = np.sum(np.subtract(X,X[row])!=0,axis=1)    **return** N |

**Note** that np is an abbreviation for Numpy.

This method only uses one for loop which iterates through every row of the N array. In this case, instead of individually calculating every respective parameter with each other, through the Numpy methods, we can perform calculations directly with the rows. In only line of code, we do the following:

-for each row

-Subtract the row from every row of the original array(X – in this case)

-For each row, sum up the number of “places” where the difference is not zero. This will happen every time the values between two vectors are not the same. -Find the sum of every row (returns a vector with size [length(X) x 1]

-Assign the sum previously found to the “row’th” position in the N array

# Measuring and profiling the results

Now that we built the methods, we should prepare a fair test which will show which one of the methods is the fastest.

To compare these two methods, we will be giving them the same input to calculate. We will generate a matrix with random integer consisting on 0’s or 1’s.

There will be given 15 test matrixes with the following dimensions:

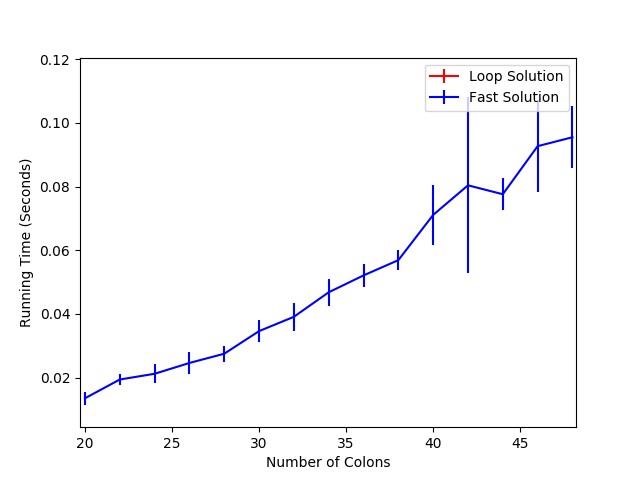
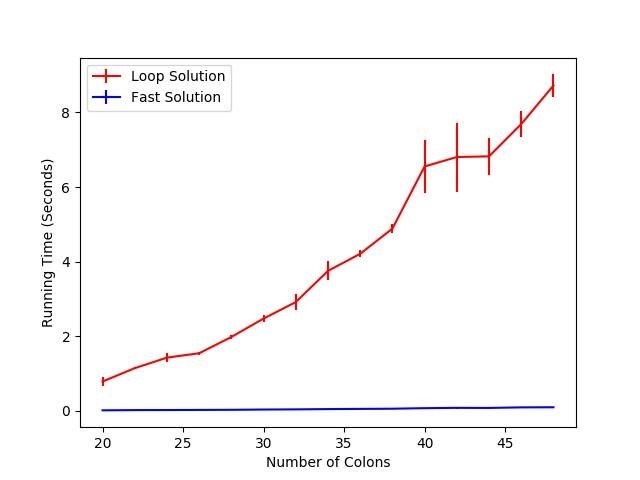
(200;20) (220;22) (240;24) (260;26) (280;28)

(300;30) (320;32) (340;34) (360;36) (380;38)

(400;40) (420;42) (440;44) (460;46) (480;48)

For each test size, there will be generated 10 runs and for each run, the running time for each of the two methods will be stored separately.

Below we can see the results plotted in a graph:



The results are interesting. While the loop solution reflects its polynomial running time, the fast solution is almost linear.

To make sure there is nothing wrong with our code, we zoomed in the graph so we can see the results better for the fast solution.

As we can see, the faster solution outperformed the loop-solution in an impressive way. As we can see, for the fast solution it only took ~0.09Seconds in average to calculate the (480;48) dimension test.

The vertical lines represent the standard deviation for each test. The

spikes are justified by the way how the CPU gives priority and the Random generated test data.

## Conclusion

We claimed the loops are very expensive tool to use in python. We suggested there exist faster ways to perform matrix calculations using some external libraries (Numpy). To support our claims, we built a testing program which measured the performance of both methods. Through this test, we concluded that using built-in libraries to perform matrix mathematical operations is much faster than using loops.