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

$$\mathrm{d}(\mathbf{p},\mathbf{q}) = \mathrm{d}(\mathbf{q},\mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

We square the difference between each respective parameter for both vectors, sum all the squared values and then we get the square root of that sum.
The scalar value obtained is the Euclidian distance between the two vectors.
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 Euclidian Distance between vector i and vector j.

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

```
def euclidian_distance_loop(X):
N = np.zeros([len(X), len(X)])
   xxT=X.dot(X.transpose())

   for i in range(len(N)):
for j in range(i+1, len(N)):
        N[i][j] = pow(xxT[i][i] + xxT[j][j] - 2 * xxT[i][j], 0.5)
        N[j][i] = N[i][j]
   return N
```

On the left, we can see the loopsolution 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 distance between two vectors Xi – Xj is given by the following formula:

$\langle Xi-Xj, Xi-Xj \rangle = Xi^TXi + Xj^TXj - 2*Xi^TXj$

To obtain such matrix to have such values we simply multiply the input matrix X with its transpose. We save this matrix to the xxT matrix. Now every [i,j] cell of that array contains the corresponding value of multiplying Xi^TXj.

Now we simply iterate through all cells of the array the method will return and we assign the respective value to each of them.

The nested **for**-loops iterate through the N array but 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 make only $(length(X))^2/2$ calculations. We implement this in the last two lines of the method.

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

```
def euclidian_distance_fast(X):
    N = np.zeros([len(X), len(X)])

    for row in range(len(N)):
        N[row] = np.sqrt(np.sum(np.power(np.subtract(X, X[row]), 2), 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)

-Square the difference

-Find the sum of every row (returns a vector with size [length(X) x 1])

-Assign the square root of the sum previously found to the "row'th" position in the N array

Measuring and profiling the results

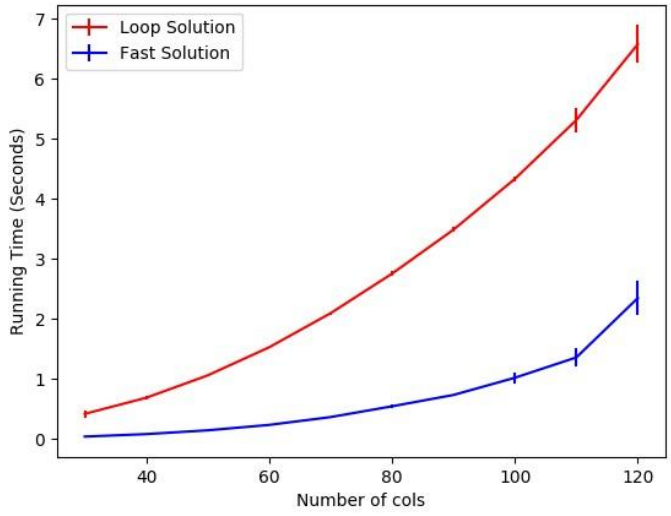
Now that we built the methods, we should prepare a fair test which will show which one of the methods is the fastest.

To compare these two methods, we will be giving them the same input to calculate. We will generate a matrix with random numbers ranging from 0 to 20.

There will be given 10 test matrixes with the following dimensions:

(300;30) (400;40) (500;50) (600;60) (700;70)

(800;80) (900;90) (1000;100) (1100;110) (1200;120)
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 and the results will be plotted.



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 performed much better.

As we can see, for the fast solution it only took ~2 Seconds in average to calculate the (1200;120) 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 properties.

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.