

Autocorrelation - Study of complexity

Ashok Muralidharan

June 27, 2019

Contents

| | |
|--|----------|
| 1 Problem Statement | 1 |
| 2 Classical autocorrelation | 1 |
| 2.1 Python Implementation | 2 |
| 3 Autocorrelation using Fast Fourier Transform(FFT) | 3 |
| 3.1 Python Implementation | 3 |
| 4 Comparison of both methods | 3 |
| 5 Thoughts and suggestions for improvement of algorithm | 4 |
| 6 Computing Periodicity | 4 |
| 7 Conclusion | 4 |

1 Problem Statement

Given a digital signal N=100 samples (signal can be chosen, but shall not be 0-symmetrical and mathematically periodical). The task is to calculate the periodicity by a re-thought, re-worked version of the classical autocorrelation (mean shall be considered). An algorithm shall be designed that is able to complete the calculation by O(N) instead of O(N²). The result shall be normalized between -1 and 1

2 Classical autocorrelation

The autocorrelation can be imagined as the correlation of a signal with a delayed version of it. Let $X(t)$ be the signal at a given time t . The classical formula for autocorrelation is given by

$$\frac{1}{(N-k)\sigma^2} \sum_{t=1}^{N-k} (X(t) - \mu) * (X(t+k) - \mu) \quad (1)$$

where μ and σ are mean and standard deviation of the signals and k goes from 0 to N (correlation for every possible delay).

2.1 Python Implementation

Let us implement this in python and see its complexity. The code can be implemented in much shorter way but the code is made verbose on purpose to study the loops in classical way.

```

1 def autocorrelate(signal):
2     start_time = time.time()
3     n = len(signal)
4     mean = sum(signal)/n
5     var = sum([(s - mean)**2 for s in signal]) / n
6     signal = [s - mean for s in signal]
7
8     auto_correlation = []
9
10    for i in range(n):
11        c = 0
12        for j in range(i, n):
13            c += signal[j-i] * signal[j]
14
15        c = c / (var*(n-i))
16        auto_correlation.append(c)
17    print("—— classical corr - poor implementation: %s seconds ——" % (time.
18    time() - start_time))
19    return auto_correlation

```

Clearly the complexity of the algorithm is $N*N$ since there is a nested for loop. Although the actual number would be $\frac{N(N+1)}{2}$, in computer science world it is still considered $N * N$. The autocorrelation requires inner dot product of two vectors every time and hence it is $N * N$. A cleaner numpy based implementation is shown below

```

1 import numpy as np
2 def autocorr_numpy(signal):
3     start_time = time.time()
4     n = len(signal)
5     variance = signal.var()
6     signal = signal - signal.mean()
7     r = np.correlate(signal, signal, mode = 'full')[-n:]
8     auto_correlation = r/(variance*(np.arange(n, 0, -1)))
9
10    print("—— classical corr - using numpy: %s seconds ——" % (time.time() -
11    start_time))
12    return auto_correlation

```

The numpy implementation is much faster than the original implementations because of the way numpy is implemented. Numpy extensively uses pointers and numpy also has array datatype. Thus it clearly is faster than python lists.

3 Autocorrelation using Fast Fourier Transform(FFT)

A popular alternative for this approach is the autocorrelation using the FFT which is popular for its time complexity of $N\log(N)$. The FFT is a popular algorithm which serves as an alternative for DFT(Discrete Fourier Transform) which has a complexity for $O(N * N)$. We use numpy's fft and ifft functions for computing the forward and inverse fourier transforms. Let us see a snippet of the same.

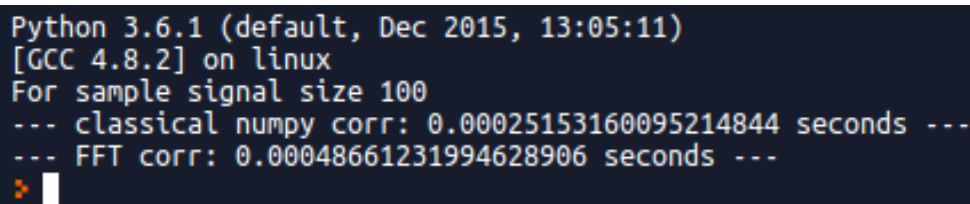
3.1 Python Implementation

```
1 import numpy as np
2
3 def autocorr_fft(x):
4     start_time = time.time()
5     r2=np.fft.ifft(np.abs(np.fft.fft(x))**2).real
6     c=(r2/x.shape-np.mean(x)**2)/np.std(x)**2
7     print("—— FFT corr: %s seconds ——" % (time.time() - start_time))
8     return c[:len(x)//2]
```

The clever idea in FFT is to divide the original signal by a factor to 2 and iteratively dividing and conquering the DFT of the smallest set. The DFT will be sum of the divided results. Thus at each step the number of iterations required is reduced by half and hence finally the complexity becomes $N\log N$.

4 Comparison of both methods

But for smaller sequence the difference between two algorithms is almost negligible. Infact, the classical numpy version performs better than the fft based technique. For a sample size of 100 we can see the results below

A terminal window with a dark background and light-colored text. The text shows the Python version (3.6.1), GCC version (4.8.2), and the sample size (100). It then displays two lines of timing results: 'classical numpy corr: 0.00025153160095214844 seconds' and 'FFT corr: 0.00048661231994628906 seconds'.

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
For sample signal size 100
--- classical numpy corr: 0.00025153160095214844 seconds ---
--- FFT corr: 0.00048661231994628906 seconds ---
>
```

Figure 1: Runtime - Classical correlation vs FFT correlation - 100 samples

As we can see, the classical version using numpy has better run time complexity for 100 samples. But if the sammple set was larger(100000), then we can see that the FFT based method was better than classical numpy based implementation as shown below.

```

Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
For sample signal size 100000
--- classical numpy corr: 89.60324549674988 seconds ---
--- FFT corr: 0.16302490234375 seconds ---
>

```

Figure 2: Runtime - Classical correlation vs FFT correlation - 100000 samples

Since we are given a signal size of 100 and assumed to be small always, we need an alternate though process for identifying potential places for speed up of the algorithm.

5 Thoughts and suggestions for improvement of algorithm

From my current knowledge and research based on general autocorrelation on arbitrary signals, the best performance is defined by $O(N \log N)$ and general classical case is $O(N * N)$. Since every computation in autocorrelation is a unique multiplication, we cannot use any form of saving previous computations for future operations. But to achieve performance close to $O(N)$, I think following compromises/tricks can be adopted.

- Since autocorrelation uses shifted signal for every step, we start appending zeros or neglect the information where we cannot correlate due to the shift. Hence say 100 samples, after shifting 50 steps we are not effectively correlating the 50 elements since we don't have corresponding shifted values. Thus we can efficiently stop at some threshold value (say 60) shifts and use that for studying correlations. This threshold can be efficiently selected if we have a domain knowledge of the signal. Say for example, in case of temperature data for every hour, there will be high correlation between closer hours than say the next day or next month data.

6 Computing Periodicity

With the autocorrelation values, we can check for the best correlated value by finding the absolute max value in the list and find its position in the list which indicates the lag of the signal. It is also better if we can have a threshold of correlation (i.e.) something greater than 0.6.

7 Conclusion

We can achieve a complexity of $N \log(N)$ using FFT based autocorrelation but we are not able to reach complexity of N . But for low values of N , the FFT based technique is not good. Thus if the sample size is low, classical approach implemented using arrays and pointers can

be preferred. This is the best complexity that I can think of for the problem right now from my research.