December, 2023

IA703

Algorithmic Information & Artificial Intelligence

Micro-study

teaching.dessalles.fr/FCI

Name: Edouard DUCLOY

Prime numbers to reduce complexity

Abstract

Several approaches exist in order to reduce the complexity of integers. One of the most famous is the round number complexity, consisting in using round numbers as proxy to reduce its complexity and the ones of its neighbors.
The purpose of the present study is to use prime numbers, and check if it can be used to reduce as well the complexity of integers.
The method described hereafter proposes different ways to use prime numbers for complexity reduction purposes, either with prime decomposition of integers, or by using prime numbers as proxy. And the results obtained are worthwhile as they allow to reduce the complexity of more than 40% of the integers between 0 and 10,000.

Problem

Binary sequences of integers can be very long, which increases considerably the complexity of its information. And more complex information needs more computing resources to process it, which can be very inconvenient when dealing with big amount of data.
In the big data era we live in now, the need to compress data and reduce its complexity is then essential to meet today's challenges. This project aims at proposing solutions to address this issue.

Method

The method presented in this section uses the CompactIntegerCoding.py python code, from J.L. Dessalles course about description complexity of integers. CompactCoding0 and CompactDecoding0 functions are used to return binary coding of integers. And CompactCoding and RoundReferenceCoding are used to assess the performance of our new method.

In order to manipulate primes number, we first need to create some useful functions:

- **primes(N) :** a function which returns all prime numbers from 0 to N, and the next prime number superior to N.

```
>>> primes(8)
[2, 3, 5, 7, 11]
```

- **is_prime_number(N) :** a function which checks if N is a prime number.

```
>>> for i in range(31):
>>>     if is_prime_number(i) == True:
>>>         print(i, is_prime_number(i))
2 True
3 True
5 True
7 True
11 True
13 True
17 True
19 True
23 True
29 True
```

- **first_primes(N) :** a function which returns a list of the N first prime numbers.

```
>>> first_primes(15)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

- **prime_decompose(N) :** a function which performs prime decomposition of N, and returns a list of tuples with prime numbers and associated powers.

```
>>> prime_decompose(399)
[(3, 1), (7, 1), (19, 1)]
```

Then, we propose three different manners to use prime numbers to describe integers, and write the code of their functions. It is to be noted that, in order to enable '0' and '1' coding, the value 2 is summed to each integer. It is also to be noted that, in order to differentiate the different approaches presented below, a two bits heading is added in each case.

- **PrimeDecomposeCoding(N) :** The first approach uses prime decomposition of the integer. The principle is to use CompactCoding0 to code the powers of each prime number from 2 to the highest involved in the decomposition. Even the prime numbers which do not belong to the decomposition shall be coded with a '0' as long as they are lower than the highest one. If not, the function would return a same code for different integers, which would lead to inappropriate results.

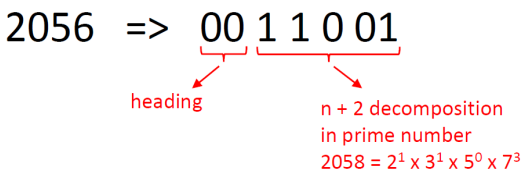

Figure 1: PrimeDecomposeCoding principle

The first '00' is the heading corresponding to PrimeDecomposeCoding approach.
The next sequences correspond to the prime powers descriptions of the prime decomposition.

```
def PrimeDecomposeCoding(N):
    '''
    Code a number N by using prime number decomposition
    Each prime number with 0 power is coded 0
    '''
    N = N + 2
    list_primes = primes(N)
    factors = prime_decompose(N)
    code = ''
    k = 0

    for i,j in (factors):
        while list_primes[k] != i:
            code += ' 0'
            k += 1
        code += ' ' + CompactCoding0(j)
        k += 1

    return '00' + code
```

```
    >>> N = 2056
    >>> print ('Decomposition: ', prime_decompose(N+2))
    Decomposition:  [(2, 1), (3, 1), (7, 3)]
    >>> print ('Primes list: ', primes(prime_decompose(N+2)[-1][0]))
    Primes list:  [2, 3, 5, 7, 11]
    >>> print ('Compact: ', CompactCoding(N), '=>', complexity(
       CompactCoding(N)))
    Compact:  1 00000001010 => 12
    >>> print ('Round: ', RoundReferenceCoding(N), '=>', complexity(
       RoundReferenceCoding(N)))
    Round:  00 00 01 11010 => 11
    >>> print ('PrimeDecompose: ', PrimeDecomposeCoding(N), '=>',
       complexity(PrimeDecomposeCoding(N)
    PrimeDecompose:  00 1 1 0 01 => 7
```

Coding: 2056		
Compact	**Round**	**PrimeDecompose**
1 00000001010	00 00 01 11010	00 1 1 0 01
C = 12	C = 11	C = 7

Figure 2: PrimeDecomposeCoding complexity comparison with CompactCoding and RoundReferenceCoding

We notice in this case that PrimeDecomposeCoding approach allows a significant complexity reduction, with 4 bits less than with RoundReferenceCoding. Nevertheless, the constraint to code each intermediate 0 power can lead to very long sequences, as shown in the example below.

```
>>> N = 197
>>> print ('Decomposition: ', prime_decompose(N+2))
Decomposition:  [(199, 1)]
>>> print ('Primes list: ', primes(prime_decompose(N+2)[-1][0]))
Primes list:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
    53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113,
   127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
   193, 197, 199, 211]
>>> print(PrimeDecomposeCoding(N), '=>', complexity(
   PrimeDecomposeCoding(N)))
PrimeDecompose:  00 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 => 48
```

- **PrimeSkipCoding(N) :** The second approach uses prime decomposition as well, but differs in the way to code the intermediate 0 power. Instead of adding as many '0' as prime number with 0 power, the principle here is to add an extra '0' bit to indicate that the following sequence describes the quantity of prime numbers to skip in the decomposition. This approach can be very useful in case of several consecutive 0 power.

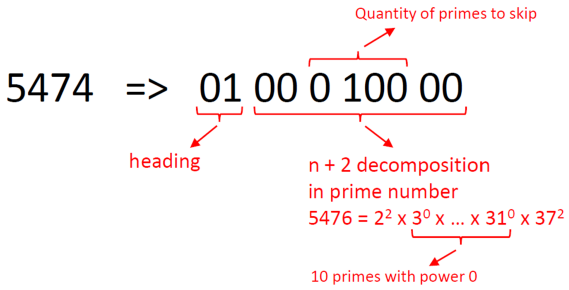

Figure 3: PrimeSkipCoding principle

The first '01' is the heading corresponding to PrimeSkipCoding approach.
The '0' in the middle of the sequence indicates that the following '100' corresponds to the quantity 10 of intermediate prime numbers to skip in the decomposition (from 3 to 31).

```
def PrimeSkipCoding(N):
    '''
    Code a number N by using prime number decomposition
    Quantity of prime number with 0 power in a raw is coded with a heading
        0
    '''
    N = N + 2
    list_primes = primes(N)
    factors = prime_decompose(N)
    code = ''
    k = 0

    for i,j in (factors):
        m = 0
        while list_primes[k] != i:
```

```
            k += 1
            m += 1
        if m == 0:
            code += '␣' + CompactCoding0(j)
        else:
            code += '␣0␣' + CompactCoding0(m) + "␣" + CompactCoding0(j)
        k += 1

    return '01' + code
```

```
>>> N = 5474
>>> print ('Decomposition:␣', prime_decompose(N+2))
Decomposition:  [(2, 2), (37, 2)]
>>> print ('Primes␣list:␣', primes(prime_decompose(N+2)[-1][0]))
Primes list:  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41]
>>> print ('Compact:␣', CompactCoding(N), '=>', complexity(
    CompactCoding(N)))
Compact:  1 010101100100 => 13
>>> print ('Round:␣', RoundReferenceCoding(N), '=>', complexity(
    RoundReferenceCoding(N)))
Round:  01 11001 00 1100 => 13
>>> print ('PrimeSkip:␣', PrimeSkipCoding(N), '=>', complexity(
    PrimeSkipCoding(N)
PrimeSkip:  01 00 0 100 00 => 10
```

Coding: 5474		
Compact	**Round**	**PrimeSkip**
1 010101100100	01 11001 00 1100	01 00 0 100 00
C = 13	C = 13	C = 10

Figure 4: PrimeSkipCoding complexity comparison with CompactCoding and RoundReferenceCoding

- **PrimeProxyCoding(N) :** The third approach consists in using prime numbers as proxy to describe integers. To do so, the integer is described as the position of its closest prime number in the prime numbers lists and the distance which separates them.

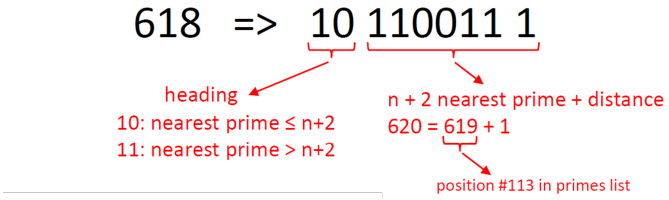

Figure 5: PrimeProxyCoding principle

The first '10' is the heading corresponding to PrimeProxyCoding approach, when the closest prime number is inferior to the integer. if the prime number is superior to the integer, the heading is '11'.
We notice that the description of the integer is composed of three binary sequences, the heading '10', the closest prime number description '110011' (corresponding to the position 113 of the prime number 619 in the prime numbers list) , and the distance to the closest

prime number '1'. It will be always the case, except when the closest prime is the integer itself in which case the distance description is omitted.

```
def PrimeProxyCoding(N):
    '''
    Code a number N by using prime number as proxy
    '''
    N = N + 2
    list_primes = primes(N)
    indice = np.argmin(np.abs(np.array(list_primes)-N))

    if indice == len(list_primes) - 2 and CompactCoding0(N-list_primes[
        indice]) == '0':
        return '10␣' + CompactCoding0(indice)
    elif indice == len(list_primes) - 2 and CompactCoding0(N-list_primes[
        indice]) != '0':
        return '10␣' + CompactCoding0(indice) + '␣' + CompactCoding0(N-
            list_primes[indice])
    elif indice == len(list_primes) - 1:
        return '11␣' + CompactCoding0(indice) + '␣' + CompactCoding0(
            list_primes[indice]-N)
```

```
>>> N = 618
>>> print ('Primes␣list:␣', primes(N+2)[-2:])
Primes list:  [619, 631]
>>> print ('Compact:␣', CompactCoding(N), '=>', complexity(
    CompactCoding(N)))
Compact:  1 001101100 => 10
>>> print ('Round:␣', RoundReferenceCoding(N), '=>', complexity(
    RoundReferenceCoding(N)))
Round:  00 11111 1 010 => 11
>>> print ('PrimeProxy:␣', PrimeProxyCoding(N), '=>', complexity(
    PrimeProxyCoding(N)
PrimeProxy:  10 110011 1 => 9
```

Coding: 618		
Compact	**Round**	**PrimeProxy**
1 001101100	00 11111 1 010	10 110011 1
C = 10	C = 11	C = 9

Figure 6: PrimeProxyCoding complexity comparison with CompactCoding and RoundReference-Coding

Finally, in order to maximize the complexity reduction, we create a last function that takes the best of all three approaches, so the one who returns the shortest binary sequence. This function is called PrimeCoding.

```
def PrimeCoding(N):
    '''
    return the shortest binary sequence between PrimeDecomposeCoding,
        PrimeSkipCoding, and PrimeProxyCoding
    '''
    PDC = PrimeDecomposeCoding(N)
    PSC = PrimeSkipCoding(N)
```

```
    PPC = PrimeProxyCoding(N)
    if min(complexity(PDC), complexity(PSC), complexity(PPC)) == complexity(PDC
        ):
        return PDC
    if min(complexity(PDC), complexity(PSC), complexity(PPC)) == complexity(PSC
        ):
        return PSC
    if min(complexity(PDC), complexity(PSC), complexity(PPC)) == complexity(PPC
        ):
        return PPC
```

We also create the PrimeDecoding function which decodes binary sequences obtained by the PrimeCoding approach. This allows to check the bijectivity of PrimeCoding.

```
def PrimeDecoding(code):
    '''
    Decode a number from its prime number coding
    '''
    exponents = code.strip().split()

    if exponents[0] == '00':
        exponents = exponents[1:]
        exponents = [CompactDecoding0(exp) for exp in exponents]
        primes = first_primes(len(exponents))
        N = 1
        for i, exp in enumerate(exponents):
            N *= primes[i] ** exp

    elif exponents[0] == '01':
        exponents = exponents[1:]
        indice = 0
        N = 1
        for i in range(len(exponents)):
            k = 0
            if exponents[i] == '0':
                k = CompactDecoding0(exponents[i+1])
                indice += k
            elif exponents[i-1] == '0':
                pass
            else:
                N *= first_primes(indice+1)[indice] ** CompactDecoding0(
                    exponents[i])
                indice += 1

    elif exponents[0] == '10':
        exponents = exponents[1:]
        if len(exponents) == 1:
            N = first_primes(CompactDecoding0(exponents[0])+1)[-1]
        else:
            N = first_primes(CompactDecoding0(exponents[0])+1)[-1] +
                CompactDecoding0(exponents[1])

    elif exponents[0] == '11':
        exponents = exponents[1:]
        N = first_primes(CompactDecoding0(exponents[0])+1)[-1] -
            CompactDecoding0(exponents[1])

    return N - 2
```

Results

The figure below shows the complexity of integers described by each of the prime numbers approach presented above. It also allows the comparison with already known CompactCoding and RoundReferenceCoding methods.

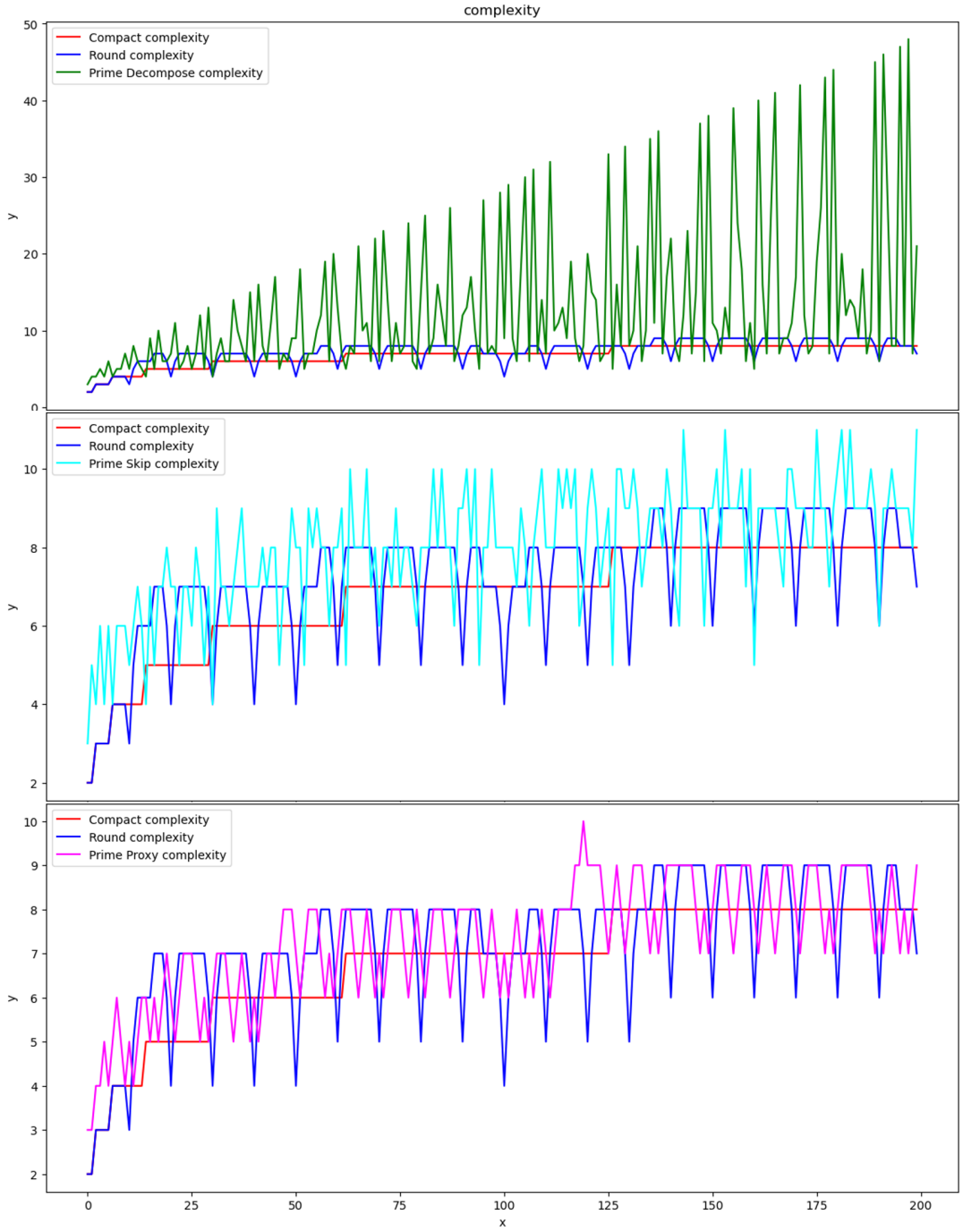

Figure 7: Complexity with the three different approaches

We can notice that each method allows to reduce complexity in some cases. For the PrimeDecomposeCoding approach, the drawback of adding an extra '0' for each intermediate prime number is here obvious.

The figure below shows the integers complexity obtained through the PrimeCoding approach, in comparison with the CompactCoding and RoundReferenceCoding ones.

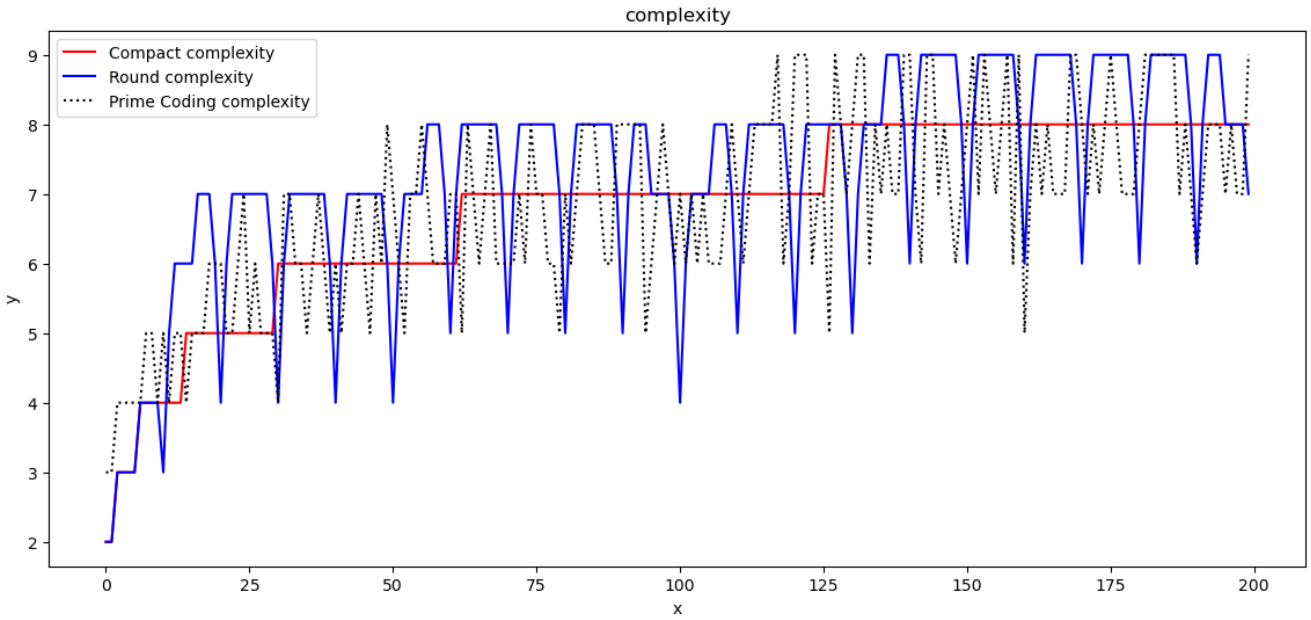

Figure 8: Complexity with PrimeCoding approach

We notice that this approach allows to reduce the integer complexity in several occurrences.

The figure below shows the complexity delta between PrimeCoding and CompactCoding, over the first 10,000 integers. The delta corresponds to the Compact complexity minus the Prime complexity. So a positive delta means that the prime approach allows a complexity reduction.

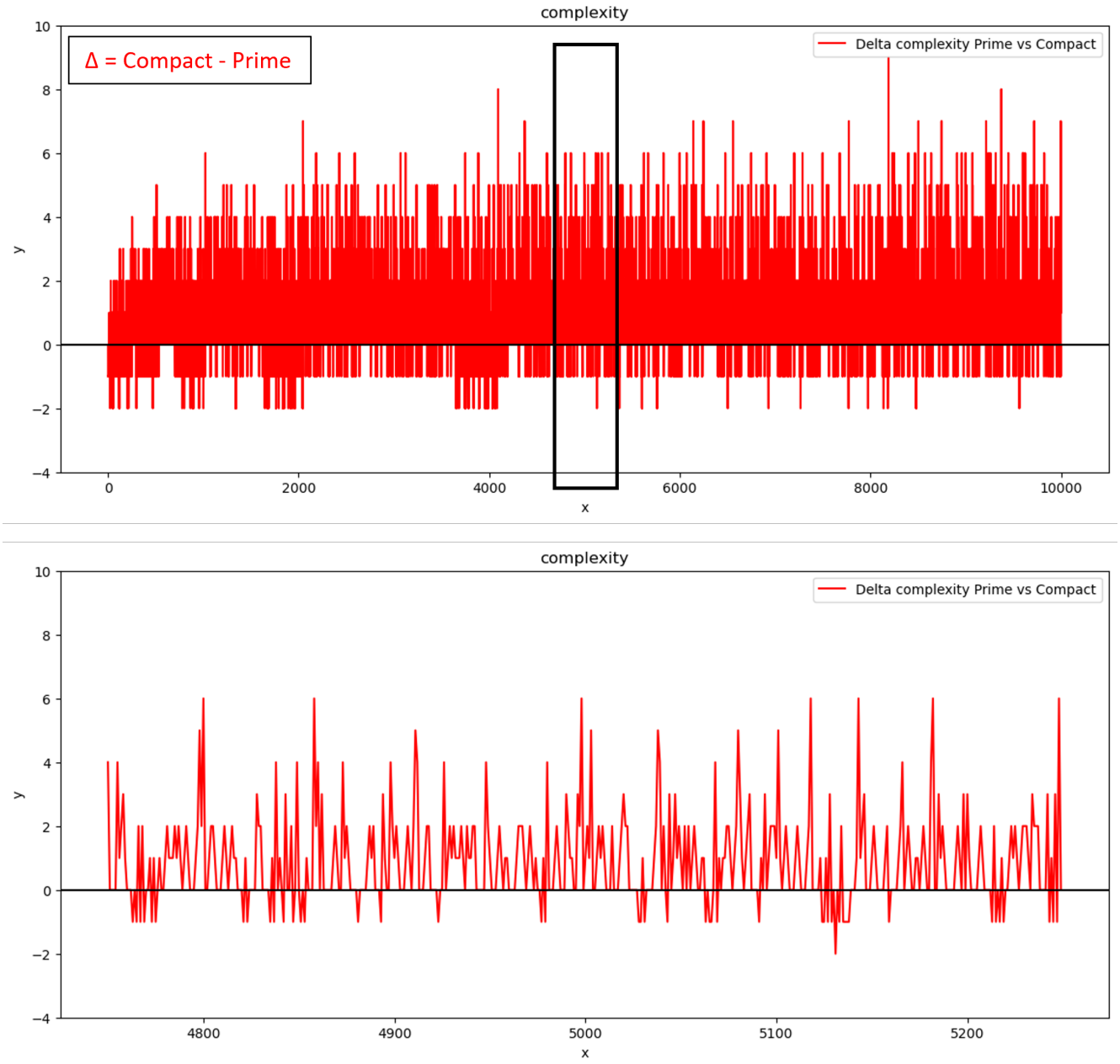

Figure 9: Complexity delta between PrimeCoding and CompactCoding

Over 10,000 integers, PrimeCoding allows a complexity reduction for 4,631 integers (46%), keeps the complexity unchanged for 3,849 integers (38%), and increases the complexity for 1,520 integers (15%). The mean reduction is 0.68.

The figure below shows the complexity delta between PrimeCoding and RoundReferenceCoding, over the first 10,000 integers. The delta corresponds to the Round complexity minus the Prime complexity. So a positive delta means that the prime approach allows a complexity reduction.

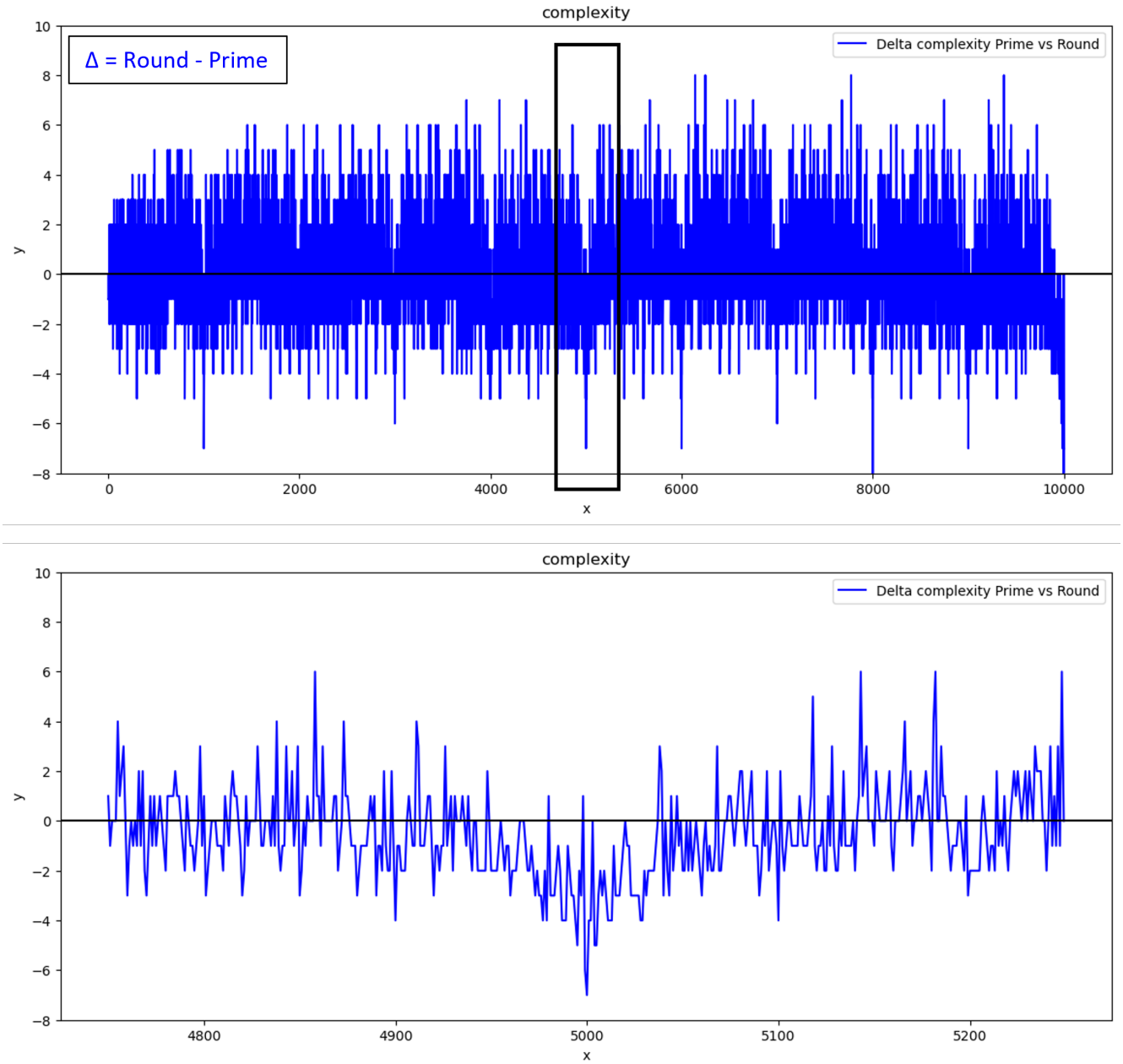

Figure 10: Complexity delta between PrimeCoding and RoundReferenceCoding

Over 10,000 integers, PrimeCoding allows a complexity reduction for 4,117 integers (41%), keeps the complexity unchanged for 2,281 integers (23%), and increases the complexity for 3,602 integers (36%). The mean reduction is 0.12.

Discussion

The results obtained through the use of prime numbers are worthwhile, as they allow complexity reduction in many cases. Nevertheless, the observations we could make do not allow to generalize about the performance of such method, because prime numbers are not uniformly distributed.
The RoundReferenceCoding seems to be more appropriate in more cases as the complexity drop is more significant for integers that seem more relevant (round numbers are more frequently used than other numbers).
This study showed that prime numbers can be good actors in the information compression. But, their usefulness shall be assessed depending on cases we have to deal with.