PyComP

Release 1.0

Jenish Raj Bajracharya

Mar 16, 2023

CONTENTS:

PyComP is a Python library for compressing and decompressing data. It provides a simple and efficient way to reduce the size of data files without losing any information.

Features

PyComP has a range of features that make it a powerful tool for data compression:

- **Supports multiple algorithms:** PyComP supports several compression algorithms, including Huffman, Arithmetic, Range, ABS and ANS, which can be selected based on your specific requirements.
- **Customizable compression level:** PyComP allows you to specify the compression level, which determines the balance between compression ratio and speed. Higher levels result in smaller file sizes but slower processing times.
- **Easy-to-use functions:** PyComP provides a range of convenient functions for compressing and decompressing data and files.

Compression algorithms

Here is a list of algorithms implemented.

- *Huffman codes <https://github.com/JEENB/PyComP/blob/version1.0/PyComP/compressors/huffman.py>*
- *rANS <https://github.com/JEENB/PyComP/blob/version1.0/PyComP/compressors/rANS.py>*
- *sANS <https://github.com/JEENB/PyComP/blob/version1.0/PyComP/compressors/sANS.py>*
- *uABS <https://github.com/JEENB/PyComP/blob/version1.0/PyComP/compressors/uABS.py>*
- *Arithmetic coder <https://github.com/JEENB/PyComP/blob/version1.0/PyComP/compressors/arithmetic.py>*
- *Symmetric Numeral <https://github.com/JEENB/PyComP/blob/version1.0/PyComP/compressors/symmetric_numeral.py>*

Install the `PyComP` package

```
git clone <repo> and cd

pip install -e . #install the package in a editable mode
```

CHAPTER

ONE

PYCOMP

1.1 Core

1.1.1 core.data module

class core.data.**Data**(*symbols: list*, *frequency: list*)

Bases: object

This is the main class. Every compression algorithm inherits this class.

Parameters

- **symbols** – list of symbols, elements can be any format
- **frequency** – frequency list associated with the list

: type freqeuncy: list

Methods

__frequency_distribution()
Computes the frequency distribution and created attributes freq_dist, M(sum of frequency)

__cumul_distribution()
Computes the cumulative distribution of a symbol Creates an attribute cum_dist: dictionary with range

{
s_1:[low, high]
}

Note: __freqency_distribuiton and __cumul_distribution are initialized

Raises

ValueError
If frequency and symbol list do not match.

shannon_entropy(*show_steps=False*) → float

Computes the shanon entroy as sum(p(x)log(p(x)))

Parameters:

show_steps: bool, default = False
Show the steps if bool is true

Returns:
entropy: float the entropy value

```
>>> symbols = ['a', 'b', 'c', 'e']
>>> frequency = [3, 4, 5, 1]
>>> d = Data(symbols, frequency)
>>> print(d.shannon_entropy())
    1.8262452584026092
```

TODO: Implement show steps

1.1.2 Module contents

1.2 Huffman Coding

class huffman.**Huffman**(*symbols: list*, *frequency: list*)

Bases: *Data*

Class for Huffman Encoding. This class inherits the data class.

Attributes:

symbols
[list] list of symbols, elements can be any format

frequency: list
frequency list associated with the list

huffman_table: dict
initialize as an empty dictionary. Datastructure for huffman code

static decode(*encoded_value: str*, *root_node:* Node) → str

Decodes the encoded value into a set of symbols

Parameters:

encoded_value: str
encoded value is a string of binary

root_node: Node
root node of the huffman tree. Traverses through the node to decode.

Returns:

decoded_symbols: str
the symbols after decoding using the huffman tree.

```
>>> h = Huffman(['a', 'b', 'c', 'd', 'e'], [0.3, 0.25, 0.2, 0.15, 0.1])
>>> enc_value, root_node = h.encode(['a', 'b', 'c', 'b', 'c', 'e'])
>>> print(h.decode(enc_value, root_node))
abcbce
```

encode(*msg: list*) → Tuple[str, *Node*]

Using the huffman's table encodes a message

Parameters:

msg: list
a list of symbols

Returns:

encoded_value: str
binary string after encoding the message

root_node: Node
root_node of the huffman tree.

Raises:
Assertion error: If message contains invalid symbol

```
>>> h = Huffman(['a', 'b', 'c', 'd', 'e'], [0.3, 0.25, 0.2, 0.15, 0.1])
>>> enc_value, root_node = h.encode(['a', 'b', 'c', 'b', 'c', 'e'])
>>> print(enc_value)
b0010111011011
```

huffman_tree() → *Node*

Using the recursive huffman's technique creates a huffman tree.

Returns:

root_node: Node
returns the root node. The tree can be constructed from the root node as it internal nodes are childrens.

static print_tree(*root:* Node) → None

Prints a tree in the terminal given a node

Parameters:

root: Node
pass in a object of class Node prints a tree.

```
>>> h = Huffman(['a', 'b', 'c', 'd', 'e'], [0.3, 0.25, 0.2, 0.15, 0.1])
>>> _, root_node = h.encode(['a', 'b', 'c', 'b', 'c', 'e'])
>>> h.print_tree(root_node)
((a(de))(bc))
    /------   ------\
(a(de))            (bc)
/--- ---\       /--- ---\
a    (de)       b       c
       /- -\
       d   e
```

show_table() → None

Prints the huffman encoding table. calls the huffman tree and get_codes function.

```
>>> h = Huffman(['a', 'b', 'c', 'd', 'e'], [0.3, 0.25, 0.2, 0.15, 0.1])
>>> h.show_table()
+---------+-----------+
| Symbols | Codewords |
+---------+-----------+
|    a    |    00     |
|    d    |    010    |
|    e    |    011    |
|    b    |    10     |
|    c    |    11     |
+---------+-----------+
```

class huffman.**Node**(*symbol: str*, *freq: Tuple[int, float, numpy.int32]*, *left=None*, *right=None*)

Bases: object

Class for a node.

Attributes:

symbols: str
symbol of the node

freq: int | float
frequency of the node. For internal nodes calculated as sum of nodes

left: Node, default: None
left child of a node

right: Node, default: None
right child of a node

1.3 Arithmetic Coding

class arithmetic_coding.**ArithmeticCoding**(*symbols: list*, *frequency: list*, *message: list = None*)

Bases: *Data*

Class for arithmetic coding compression with out rescaling. Might not be efficient. Inherits the data class. Allows compression in two ways.

1. **By specifying the symbols and frequency.**
   In this case arg:msg must be provided in the encode step.
2. **By giving the entire message itself.**
   No argument required in the encode step. Computes the probability and cumulative distribution from the message itself.

Attributes:

symbols
[list] list of symbols, elements can be any format

frequency: list
frequency list associated with the list

message: list, default = None
list of message.

decode(*encoded_value: bitarray.bitarray*, *msg_length: int*, *show_steps: bool = False*)

Using the decoding by checking interval, updating interval, and picking new symbol.

Parameters:

encoded_value: bitarray.bitarray
bitarray instance of the encoded value.

msg_length: int
length of the message. needs to be specified to the same number as original msg to get right decoding.

show_steps: bool, default = False
shows decoding steps.

Returns:

decoded_symbols: str
returns the decoded symbols

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [0.8, 0.02, 0.18]
>>> f = ArithmeticCoding(symbols, freq)
>>> encoded_value, msg_len = f.encode('abaca')
>>> decoded_value = f.decode(encoded_value, msg_len)
>>> print(decoded_value)
abaca
```

```
>>> decoded_value = f.decode(encoded_value, msg_len, show_steps=True)
Decoding Process
------------------
+--------------+--------------------------+---------------+----------------------
↪----------------------+-----------+
| Decoded Symb |      Encoded Value       |      Tag      |                ␣
↪Range                     |  Remark   |
+--------------+--------------------------+---------------+----------------------
↪----------------------+-----------+
|      h       | bitarray('00010001011') | 0.06787109375 |                 (0,␣
↪0.2)                   | Pick next |
|      he      | bitarray('00010001011') | 0.06787109375 | (0.04000000000000001,
↪ 0.08000000000000002) | Pick next |
|     hel      | bitarray('00010001011') | 0.06787109375 | (0.05600000000000001,
↪ 0.07200000000000001) | Pick next |
|     hell     | bitarray('00010001011') | 0.06787109375 | (0.06240000000000001,
↪ 0.06880000000000001) | Pick next |
|    hello     | bitarray('00010001011') | 0.06787109375 | (0.06752000000000001,
↪ 0.06880000000000001) | Pick next |
+--------------+--------------------------+---------------+----------------------
↪----------------------+-----------+
Decoded Value = hello
```

encode(*msg: list = None*, *show_steps: bool = False*) → Tuple[bitarray.bitarray, int]

Arithmetic encoding function. Can be used to encdoe in either ways as specifiec before.

Parameters:

msg: list, default = None
message you want to encode

show_steps: bool, default = False
shows the encoding step

Returns:

encoded_value: BITARRAY
binary string of the encoded value.

lenght: int
length of the message. To specify for decoder.

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [0.8, 0.02, 0.18]
>>> f = ArithmeticCoding(symbols, freq)
>>> encoded_value, msg_len = f.encode('abaca')
>>> print(encoded_value, msg_len)
bitarray('10100110110') 5
```

```
>>> f = ArithmeticCoding(symbols = None, frequency= None, message='hello')
>>> encoded_value, msg_len = f.encode(show_steps = True)
Encoding Process
-----------------
+--------+-------------------------------------------+-------------------+
| Symbol |                  Interval                 |      Remark       |
+--------+-------------------------------------------+-------------------+
|   h    |                  (0, 0.2)                 | Pick Next Symbol  |
|   he   | (0.04000000000000001, 0.08000000000000002) | Pick Next Symbol  |
|  hel   | (0.05600000000000001, 0.07200000000000001) | Pick Next Symbol  |
|  hell  | (0.06240000000000001, 0.06880000000000001) | Pick Next Symbol  |
| hello  | (0.06752000000000001, 0.06880000000000001) | Pick Next Symbol  |
|        |            Symbols Encoded = 5            |                   |
|        |         Tag = 0.06816000000000001         |                   |
|        | Compressed Value = bitarray('00010001011') |                   |
+--------+-------------------------------------------+-------------------+
```

msg_prob(*message: list*) → float

Computes the joint probability of message. Requires the class to initianted.
P(x_1, x_2, . . .) = p(x_1).p(x_2). . . .

Parameters:
message: list

Returns:

prob: float
the probability of the entire message

```
>>> f = ArithmeticCoding(['a', 'b', 'c'], [0.8, 0.02, 0.18])
>>> f.msg_prob('aabcca')
0.0003317760000000001
```

class arithmetic_coding.**ArithmeticDecoder**(*symbols: list*, *frequency: list*)

Bases: *Data*

Arithmetic decoder class. Used only for decoing. Assume a communication channel where the receiver has access to the decoding channel only. instantiates the arithmetic coding class and uses the decoding function.

Attributes:

symbols
[list] list of symbols, elements can be any format

frequency: list
frequency list associated with the list

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [0.8, 0.02, 0.18]
>>> f = ArithmeticDecoder(symbols, freq)
>>> f.decode(bitarray.bitarray('10100110110'),5)
abaca
```

decode(*encoded_value: bitarray.bitarray*, *msg_length: int*)

Decodes a bit array using airithmetic coding scheme. Parameters:

encoded_value: bitarray.bitarray
bitarray instance of the encoded value.

msg_length: int
length of the message. needs to be specified to the same number as original msg to get right decoding.

Returns:

decoded_symbols: str
returns the decoded symbols

class arithmetic_coding.**RangeCoding**(*symbols: list*, *frequency: list*, *message: str = None*)

Bases: *ArithmeticCoding*

Class for arithmetic coding compression with rescaling. Might not be efficient. Inherits the arithmetic coding class and changes teh encoding and decodign function. Allows compression in two ways.

1. **By specifying the symbols and frequency.**
   In this case arg:msg must be provided in the encode step.
2. **By giving the entire message itself.**
   No argument required in the encode step. Computes the probability and cumulative distribution from the message itself.

Attributes:

symbols
[list] list of symbols, elements can be any format

frequency: list
frequency list associated with the list

message: list, default = None
list of message.

decode(*encoded_value: bitarray.bitarray*, *msg_length: int*, *show_steps: bool = False*)

Using the decoding by checking interval, rescaling, updating interval, and picking new symbol.

Parameters:

encoded_value: bitarray.bitarray
bitarray instance of the encoded value.

msg_length: int
length of the message. needs to be specified to the same number as original msg to get right decoding.

show_steps: bool, default = False
shows decoding steps.

Returns:

decoded_symbols: str
returns the decoded symbols

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [0.8, 0.02, 0.18]
>>> f = RangeCoding(symbols, freq)
>>> decoded_value = f.decode(bitarray.bitarray('10100011011'), 5)
>>> print(decoded_value)
abaca
```

```
>>> decoded_value = f.decode(encoded_value, msg_len, show_steps=True)
Decoding Process
------------------
+--------------+-----------------------+-------------+--------------------------
↪------------------+---------------+
| Decoded Symb |     Encoded Value     |     Tag     |                    Range ␣
↪                  |    Remark     |
+--------------+-----------------------+-------------+--------------------------
↪------------------+---------------+
|      h       | bitarray('000100011') | 0.068359375 |                   (0, 0.
↪2)                  | Left Scaling  |
|              |                       |             |                          ␣
↪                  |   Remove 0    |
|      h       | bitarray('00100011')  | 0.13671875  |                   (0, 0.
↪4)                  | Left Scaling  |
|              |                       |             |                          ␣
↪                  |   Remove 0    |
|      h       |  bitarray('0100011')  |  0.2734375  |                   (0, 0.
↪8)                  |   Pick next   |
|      he      |  bitarray('0100011')  |  0.2734375  | (0.16000000000000003, 0.
↪32000000000000006) | Left Scaling  |
|              |                       |             |                          ␣
↪                  |   Remove 0    |
|      he      |  bitarray('100011')   |  0.546875   | (0.32000000000000006, 0.
↪6400000000000001)  |   Pick next   |
|     hel      |  bitarray('100011')   |  0.546875   | (0.44800000000000006, 0.
↪5760000000000001)  |   Pick next   |
|     hell     |  bitarray('100011')   |  0.546875   |  (0.4992000000000001, 0.
↪5504000000000001)  |   Pick next   |
```

(continues on next page)

(continued from previous page)

```
|    hello     |  bitarray('100011')   |  0.546875   |  (0.5401600000000001, 0.
↪5504000000000001)  | Right Scaling |
|              |                       |             |                          ␣
↪                  |   Remove 1    |
|    hello     |   bitarray('00011')   |   0.09375   | (0.08032000000000017, 0.
↪10080000000000022) | Left Scaling  |
|              |                       |             |                          ␣
↪                  |   Remove 0    |
|    hello     |   bitarray('0011')    |   0.1875    | (0.16064000000000034, 0.
↪20160000000000045) | Left Scaling  |
|              |                       |             |                          ␣
↪                  |   Remove 0    |
|    hello     |    bitarray('011')    |    0.375    |  (0.3212800000000007, 0.
↪4032000000000009)  | Left Scaling  |
|              |                       |             |                          ␣
↪                  |   Remove 0    |
|    hello     |    bitarray('11')     |    0.75     |  (0.6425600000000014, 0.
↪8064000000000018)  | Right Scaling |
|              |                       |             |                          ␣
↪                  |   Remove 1    |
|    hello     |     bitarray('1')     |     0.5     |  (0.2851200000000027, 0.
↪6128000000000036)  |   Pick next   |
+--------------+-----------------------+-------------+--------------------------
↪------------------+---------------+
Decoded Value = hello
```

encode(*msg: list = None*, *show_steps: bool = False*) → Tuple[bitarray.bitarray, int]

Range encoding function. Can be used to encdoe in either ways as specifiec before.

Parameters:

msg: list, default = None
message you want to encode

show_steps: bool, defalut = False
shows encoding step

Returns:

encoded_value: BITARRAY
binary string of the encoded value.

lenght: int
length of the message. To specify for decoder.

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [0.8, 0.02, 0.18]
>>> f = RangeCoding(symbols, freq)
>>> encoded_value, msg_len = f.encode('abaca')
>>> print(encoded_value, msg_len)
bitarray('1010011011') 5
```

```
>>> f = RangeCoding(symbols = None, frequency= None, message='hello', show_
↪steps=True)
```

(continues on next page)

(continued from previous page)

```
>>> encoded_value, msg_len = f.encode()
Encoding Process
------------------
+--------+-------------------------------------------+-------------------+
| Symbol |                  Interval                 |      Remark       |
+--------+-------------------------------------------+-------------------+
|   h    |                  (0, 0.2)                 |  Left Scaling     |
|        |                                           |    Output = 0     |
|   h    |                  (0, 0.4)                 |  Left Scaling     |
|        |                                           |    Output = 0     |
|   h    |                  (0, 0.8)                 | Pick Next Symbol  |
|   he   | (0.16000000000000003, 0.32000000000000006) |  Left Scaling     |
|        |                                           |    Output = 0     |
|   he   | (0.32000000000000006, 0.6400000000000001)  | Pick Next Symbol  |
|  hel   | (0.44800000000000006, 0.5760000000000001)  | Pick Next Symbol  |
|  hell  |  (0.4992000000000001, 0.5504000000000001)  | Pick Next Symbol  |
| hello  |  (0.5401600000000001, 0.5504000000000001)  |  Right Scaling    |
|        |                                           |    Output = 1     |
| hello  | (0.08032000000000017, 0.10080000000000022) |  Left Scaling     |
|        |                                           |    Output = 0     |
| hello  | (0.16064000000000034, 0.20160000000000045) |  Left Scaling     |
|        |                                           |    Output = 0     |
| hello  |  (0.3212800000000007, 0.4032000000000009)  |  Left Scaling     |
|        |                                           |    Output = 0     |
| hello  |  (0.6425600000000014, 0.8064000000000018)  |  Right Scaling    |
|        |                                           |    Output = 1     |
| hello  |  (0.2851200000000027, 0.6128000000000036)  | Pick Next Symbol  |
|        |            Symbols Encoded = 5            |                   |
|        |  Rescaling Output = bitarray('000100011')  |                   |
|        |                 Tag = 0.5                 |                   |
|        |  Compressed Value = bitarray('000100011')  |                   |
+--------+-------------------------------------------+-------------------+
```

class arithmetic_coding.**RangeDecoder**(*symbols: list*, *frequency: list*)

Bases: `object`

Range decoder class. Used only for decoing. Assume a communication channel where the receiver has access to the decoding channel only. instantiates the range coding class and uses the decoding function.

Attributes:

symbols
: [list] list of symbols, elements can be any format

frequency: list
: frequency list associated with the list

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [0.8, 0.02, 0.18]
>>> f = RangeDecoder(symbols, freq)
>>> f.decode(bitarray.bitarray('1010011011'),5)
abaca
```

decode(*encoded_value: bitarray.bitarray*, *msg_length: int*)

Decodes a bit array using airithmetic coding scheme. Parameters:

encoded_value: bitarray.bitarray
bitarray instance of the encoded value.

msg_length: int
length of the message. needs to be specified to the same number as original msg to get right decoding.

Returns:

decoded_symbols: str
returns the decoded symbols

1.4 Symmetric Numeral

class symmetric_numeral.**SymmetricNumeral**(*base: int*)

Bases: object

Class for symmetric numeral encoding.

Attributes:

base: int
base of numeral system

decode(*encoded_value*)

Decodes the encoded value as per as per the base

Parameters:

encoded_value: base_b
list of digits of base b

Returns:

decoded_symbols: base_b
returns the decoded symbols

```
>>> s = SymmetricNumeral(7)
>>> s.decode(197833)
[1, 4, 5, 2, 5, 2, 6]
```

encode(*message: list*)

Encodes a set of symbols as per the base

Parameters:

message: list
list of digits of base b

Returns:

encoded_value: base_b
encoding of message in base b

```
>>> s = SymmetricNumeral(7)
>>> s.encode([1, 4, 5, 2, 5, 2, 6])
197833
```

shannon_entropy()

Computes the shannon's entorpy for the base

Retuns: float
entropy

```
>>> s = SymmetricNumeral(7)
>>> s.shannon_entropy()
2.807354922057604
```

1.5 ANS

class ANS.**rANS**(*symbols: list*, *frequency: list*)

Bases: *Data*

rANS compressor and decompressor class. Inherits the data class.

Attributes:

symbols: list
list of all possible symbols

frequency: list
list of symbol frequency

decode(*encoded_value: str*, *msg_len*) → list

rANS decode function

Parameters:

encoded_value: int
final state after encoding this function inherits the probability distribuiton of the symbols. This function assumes that the probability distribuiton is know and the class is instantiated

Returns:

symbols: list
the decoded symbols in reverse order

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [5, 5, 2]
>>> a = rANS(symbols, freq)
>>> a.decode(1242,6)
['a', 'b', 'c', 'c', 'a', 'b']
```

encode(*msg: list*, *start_state: int*) → Tuple[str, int]

rANS encode function

Parameters:

data: list
data to be encoded. Has to be a list

Returns:

final_state: int
final encoded value

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [5, 5, 2]
>>> a = rANS(symbols, freq)
>>> a.encode(['a', 'b', 'c', 'c', 'a', 'b'], 0)
1242
```

rANS_decode_step(*x_next: int*) → tuple

Decoding step function

rANS_encode_step(*symbol*, *x_prev: int*) → int

Encoding step function

rANS_encoding_table(*final_state*) → pandas.DataFrame

Returns the rANS encoding table. The format is similar to Dudak's paper. Parameters:

final_state: the final state table should contain

Returns:

table: pd.Dataframe
returns and pandas dataframe and prints the dataframe.

class ANS.**rANSDecoder**(*symbols: list*, *frequency: list*)

Bases: *Data*

rANSDecoder class for decoding given symbols and frequency.

Parmaeters:

symbols: list
a list of symbols

frequency: list
frequency distribuiton list

decode(*encoded_value: str*, *msg_len: int*)

Function to decode, give the correct order Parameters:

encoded_value: int
final state after encoding

Returns:

decoded symbols: list
list of decoded symbols

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [5, 5, 2]
>>> a = rANSDecoder(symbols, freq)
>>> a.decode(1242,6)
['a', 'b', 'c', 'c', 'a', 'b']
```

1.6 uABS

class uABS.**uABS**(*p*)

Bases: object

decode(*final_state*, *msg_len: int*)

encode(*msg: str*, *initial_state=0*)

shannon_entropy()

uABS_decode_step(*x*)

uABS_encode_step(*s: str*, *x_prev: int*)

1.7 Streaming ANS

class sANS.**sANS**(*symbols: list*, *frequency: list*)

Bases: *Data*

sANS compressor and decompressor class. Inherits the data class. Initializes the rANS class. Attributes:

symbols: list
list of all possible symbols

frequency: list
list of symbol frequency

decode(*x: str*, *bit_array: bitarray.bitarray*)

sANS decode function

Parameters:

encoded_value: int
final state after encoding this function inherits the probability distribuiton of the symbols. This function assumes that the probability distribuiton is know and the class is instantiated

bit_array: bitarray.bitarray
the bit output from renormalization

Returns:

symbols: list
the decoded symbols in reverse order

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [5, 5, 2]
>>> a = rANS(symbols, freq)
>>> a.decode(18, bitarray.bitarray('00110011010'))
['a', 'b', 'c', 'c', 'a', 'b']
```

encode(*msg: list*)

sANS encode function

Parameters:

msg: list
data to be encoded. Has to be a list

initial_state: int
initial state must be >= sum of freq

Returns:

final_state: int
final encoded value

bit_output: bitarray.bitarray
bit output from rescaling

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [5, 5, 2]
>>> a = sANS(symbols, freq)
>>> a.encode(['a', 'b', 'c', 'c', 'a', 'b'], 14)
18 bitarray('00110011010')
```

class sANS.**sANSDecoder**(*symbols: list*, *frequency: list*)

Bases: *Data*

rANSDecoder class for decoding given symbols and frequency. initializes the sANS class. Parmaeters:

symbols: list
a list of symbols

frequency: list
frequency distribuiton list

decode(*x: str*, *bit_array: bitarray.bitarray*)

Function to decode, give the correct order Parameters:

encoded_value: int
final state after encoding

Returns:

decoded symbols: list
list of decoded symbols

```
>>> symbols = ['a', 'b', 'c']
>>> freq = [5, 5, 2]
>>> a = sANSDecoder(symbols, freq)
>>> a.decode(18, bitarray.bitarray('00110011010'))
['a', 'b', 'c', 'c', 'a', 'b']
```

1.8 File Compressor

class file_compressor.**FileCompressor**(*file*)

Bases: object

Compresses a file

1.8.1 parameter:

file: str
file location

returns:
final_rANS state

Note: The decode function can be used given the class has been configured with correct prob distn.

`create_aux_file()`

creates an auxiliary file with symbols and their frequency distribution: for decompressor consists of:

symbols, frequency, file_name

using this becomes counter intuituve as the aux_file will have size > original file.

`decode(encoded_value: int)`

`file_encode(compressor)`

Given a compression algorithm compressed a file. Parameters:

compressor: str
compression algorithm either from huffman, airhtmetic, range, rANS, sANS

Returns:

encodec_value: int | str | Tuple
the encoded value can be anything depending on the output of the compression algorithm.

`summary()`

function that gives summary of the file : file_name file_size file_creation_tiem file_modification_time total_symbols unique_symbols frequeency_dist shannon_entrory compressed_size compression_ratio

1.9 Utils

1.9.1 utils.ac_utils module

`utils.ac_utils.decToBinConversion(no: float, precision: int)` → str

Converts a decimal number to binary accepts fraction as well

Parameters:

no: float
decimal number can consist fraction part as well

precision: int
precision required for the fractinal part returns fractional part to that precision level

Returns:

binary: str
returns the binary conversion of a decimal number with user-defined precision

utils.ac_utils.**getBinaryFractionValue**(*binaryFraction*)

Compute the binary fraction value using the formula of: (2^-1) * 1st bit + (2^-2) * 2nd bit + ...

Parameters:

binaryFraction: str
binary string of the fractional part.

Returns:

value: float
returns the fractional part in decimal

1.9.2 utils.bit_array_utils module

utils.bit_array_utils.**bitarray_to_int**(*bit_array: bitarray.bitarray*)

Converts bitarray to int.

Parameters:

bit_array: BITARRAY
bits to be converted

Returns:

dec: int
decimal_equivalent of bit_array

utils.bit_array_utils.**bitarrays_to_float**(*uint_x_bitarray: bitarray.bitarray*, *frac_x_bitarray: bitarray.bitarray*) → float

Converts bitarrays corresponding to integer and fractional part of a floatating point number to a float.

Parameters:

uint_x_bitarray: BitArray
bitarray corresponding to the integer part of x

frac_x_bitarray: BitArray
bitarray corresponding to the fractional part of x

Returns:
x: float, the floating point number

utils.bit_array_utils.**float_to_bitarrays**(*x: float*, *max_precision: int*) → Tuple[bitarray.bitarray, bitarray.bitarray]

Convert floating point number to binary with the given max_precision Utility function to obtain binary representation of the floating point number. We return a tuple of binary representations of the integer part and the fraction part of the floating point number.

Parameters:

x: float
input floating point number

max_precision: int
max binary precision (after the decimal point) to which we should return the bitarray

Returns:
Tuple[BitArray, BitArray]: returns (uint_x_bitarray, frac_x_bitarray)

utils.bit_array_utils.**int_to_bitarray**(*x: int*, *bit_width=None*) → bitarray.bitarray

Converts int to bits.

Parameters:

x: int
integer to be converted

bit_width: int, default = None
length

Returns:

bit: BITARRAY
binary equivalent of x

1.9.3 utils.file_utils module

utils.file_utils.**convert_bytes**(*num*) → str

This function will convert bytes to MB.... GB... etc.

Parameters:

x: float
input floating point number

max_precision: int
max binary precision (after the decimal point) to which we should return the bitarray

Returns:

file_size: str
file size to the nearest MB.... GB...

```
>>> convert_bytes(10580)
    10.3 KB
```

utils.file_utils.**epoch_to_datetime**(*epoch_time*)

This function converts epoch into datetime

Parameters:

epoch_time: int
time in epoch scale

Returns:

date: datetime
standard date and time of the epoch time.

```
>>> epoch_to_datetime(12452687)
    1970-05-25 08:34:47
```

utils.file_utils.**file_creation**(*file_path*)

This function will return the file creation date

Parameters:

file_path: str
path of the file

Returns:

file_creation: Datetime
file creation time in standard format

```
>>> file_creation('/Users/jenish/Library/CloudStorage/Dropbox/crypto/ANS/code/ANS/
↪utils/utils.py')
    2023-03-13 23:11:04.054830
```

utils.file_utils.**file_last_modified**(*file_path*)

This function will return the file modified datetime

Parameters:

file_path: str
path of the file

Returns:

file_moodified_time: Datetime
last modified time in standard format

```
>>> file_last_modified('/Users/jenish/Library/CloudStorage/Dropbox/crypto/ANS/code/
↪ANS/utils/utils.py')
    2023-03-13 23:11:01.540858
```

utils.file_utils.**file_name**(*file_path*)

This function will return the file name.

Parameters:

file_path: str
path of the file

Returns:

file_name: str
name of the file with extension

```
>>> file_name('/Users/jenish/Library/CloudStorage/Dropbox/crypto/ANS/code/ANS/utils/
↪utils.py')
    utils.py
```

utils.file_utils.**file_size**(*file_path*)

This function will return the file size.

Parameters:

file_path: str
path of the file

Returns:

file_size: int
size of file in bits

```
>>> file_size('/Users/jenish/Library/CloudStorage/Dropbox/crypto/ANS/code/ANS/utils/
↪utils.py')
    3305
```

utils.file_utils.**file_summary**(*file_path*)

Compautes the summary of the file. Runs the file util functions.

Parameters:

file_path: str
path of the file

Returns:
name, size, creation, modification: Tuple[str, str, datetime, datetime]

```
>>> file_summary('/Users/jenish/Library/CloudStorage/Dropbox/crypto/ANS/code/ANS/
↪utils/utils.py')
    ('utils.py', 3305, datetime.datetime(2023, 3, 13, 23, 11, 4, 54830), datetime.
↪datetime(2023, 3, 13, 23, 11, 1, 540858))
```

1.9.4 utils.utils module

utils.utils.**convert_list_to_string**(*l: list*) → str

utils.utils.**encode_symbols_to_integer**(*symbols: list*)

Encodes each symbol to a integer starting from 0. Helper function for encoding_table

Parameters:

symbols: list
list of symbols to be encoded

Returns:

encoded_list: dict
dict with the encoded values as keys

utils.utils.**encoding_table**(*table_elements*)

Creates an encoding table given table elements for ANS. This table can be used to determing the symbol spread function. Helper function for ANS.rANS.encoding_table().

Paramaters:

table_elements: Tuple(symbols, x_prev, x_new)
table elements is a list of symbol, x_prev, x_new encoding_table: matrix (A) of size len(symbol) * max(x_new) where A_{symbol,x_new} = x_prev

Returns:

table: pd.Dataframe
the encoding table. Usually for ANS.

utils.utils.**get_symbols**(*symbols: list*, *frequency: list*, *no_symbols: int*) → list

Get arbitrary number of symbols following a particular distribution. Uses inversion sampling to sampling symbols. Used to test compressors.

Parameters:

symbols: list
list of symbols

frequency: list
list of freqency associated with a particular symbol

no_symbols: int
number of symbols you want to sample

Returns:

symbols: list
list of symbols following a particular distribution

```
>>> get_symbols(['a', 'b', 'c', 'd', 'z'], [0.2,0.3,0.1,0.1,0.4], 10)
    ['z', 'a', 'b', 'b', 'z', 'z', 'b', 'b', 'b', 'a']
```

1.9.5 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

I

M

N

P

R

S

U