EE 274 lecture 3

SCL sneak peek

+

Entropy as the fundamental limit for lossless compression

Recap

- Prefix codes & tree representation
- Thumb rule - $l(x) \approx \log_2 \frac{1}{P(x)}$
- Kraft's inequality:
  - Any prefix code => $\sum 2^{-l_i} \leq 1$
  - $\sum 2^{-l_i} \leq 1$ => There exists prefix code with these lengths
    - Simple greedy construction by sorting the lengths
  - In particular, for $l(x) = \left\lceil \log_2 \frac{1}{P(x)} \right\rceil$, above inequality holds, and a code exists!

Decoding for prefix codes…

```python
def decode_block(self, bitarray: BitArray):
    """
    decode the bitarray one symbol at a time using the decode_symbol

    as prefix free codes have specific code for each symbol, and due to the prefix free nature, allow for
    decoding each symbol from the stream, we implement decode_block function as a simple loop over
    decode_symbol function.

    Args:
        bitarray (BitArray): input bitarray with encoding of >=1 integers

    Returns:
        Tuple[DataBlock, Int]: return decoded integers in data block, number of bits read from input
    """
    data_list = []
    num_bits_consumed = 0
    while num_bits_consumed < len(bitarray):
        s, num_bits = self.decode_symbol(bitarray[num_bits_consumed:])
        num_bits_consumed += num_bits
        data_list.append(s)

    return DataBlock(data_list), num_bits_consumed
```

How to implement decode_symbol?

```python
class PrefixFreeTree:
    """
    Class representing a Prefix Free Tree

                        def decode_symbol(self, encoded_bitarray):
                            """
                            Decodes the encoded bitarray stream by decoding symbol by symbol. We parse through the prefix free tree, till
                            we reach a leaf node which gives us the decoded symbol ID using prefix-free property of the tree.

                            - start from the root node
                            - if the next bit is 0, go left, else right
                            - once you reach a leaf node, output the symbol corresponding the node
                            """
                            # initialize num_bits_consumed
                            num_bits_consumed = 0

                            # continue decoding until we reach leaf node
                            curr_node = self.root_node
                            while not curr_node.is_leaf_node:
                                bit = encoded_bitarray[num_bits_consumed]
                                if bit == 0:
                                    curr_node = curr_node.left_child
                                else:
                                    curr_node = curr_node.right_child
                                num_bits_consumed += 1

                            # as we reach the leaf node, the decoded symbol is the id of the node
                            decoded_symbol = curr_node.id
                            return decoded_symbol, num_bits_consumed
```

Why SCL?

- Efficient implementations often hard for a beginner to understand or modify
- Implementations of many basic algorithms hard to find
- Intuitively understanding the algorithm ≠ being able to implement it in practice

Why SCL?

- Provide research implementation of common data compression algorithms
- Provide convenient framework to quickly modify existing compression algorithm and to aid research in the area
- To ourselves understand these algorithms better ☺

SCL at a glance

```
.
├── LICENSE
├── README.md
├── compressors
│   ├── baseline_compressors.py
│   ├── fano_coder.py
│   ├── golomb_coder.py
│   ├── huffman_coder.py
│   ├── prefix_free_compressors.py
│   ├── rANS.py
│   ├── shannon_coder.py
│   ├── shannon_fano_elias_coder.py
│   ├── tANS.py
│   ├── typical_set_coder.py
│   └── universal_uint_coder.py
├── core
│   ├── data_block.py
│   ├── data_encoder_decoder.py
│   ├── data_stream.py
│   ├── encoded_stream.py
│   └── prob_dist.py
├── external_compressors
│   └── zlib_external.py
├── requirements.txt
└── utils
    ├── bitarray_utils.py
    ├── misc_utils.py
    ├── test_utils.py
    └── tree_utils.py
```

Now back to the board…