Problem Set 6

published: 2 June 2022
discussion: 17 June 2022

Data Compression With And Without Deep Probabilistic Models
Prof. Robert Bamler, University of Tuebingen

Course materials available at `https://robamler.github.io/teaching/compress22/`

Problem 6.1: Understanding Information Content

This problem aims to give you a some additional intuition for the concept of "information content" by considering the most trivial compression problem that one could think of. It ties back to the `UniformCoder` that we used in class as a starting point for ANS coding.

The strategy that we use in this problem is something that you might find useful for approaching *any* new topic, not just in this course. Whenever there's something you feel you don't quite understand yet, I strongly encourage you to act like you're debugging some code that doesn't work: reduce the problem to the absolute simplest form, understand that simplified problem, and then gradually build up from that.

Problem Setup. Consider a data source that generates messages $\mathbf{x}$. Different from the problems so far, we don't care what these messages are—they could be sequences of symbols but they don't have to be. All we care about is that the messages come from some finite set $\mathbb{X}$ (which again, could be the product space of some finite alphabet, i.e., $\mathbb{X} = \mathfrak{X}^k$, but it doesn't have to be). Now, we consider the simplest possible probabilistic model: we assume that the messages are uniformly distributed, i.e., $P(\mathbf{X}=\mathbf{x}) = 1/|\mathbb{X}| \; \forall \mathbf{x} \in \mathbb{X}$.

(a) What is the information content of any message $\mathbf{x} \in \mathbb{X}$?

(b) Take a step back from the setup for a moment and consider the binary representation of a positive integer $n \in \mathbb{N}$. How long is this binary representation, i.e., how many bits does it contain? Express your result as a mathematical function of n.

(c) Back to the compression problem: consider the following encoder for messages $\mathbf{x} \in \mathbb{X}$: let f be an arbitrary bijective function from $\mathbb{X}$ to the set of integers $\{0, \ldots, |\mathbb{X}| - 1\}$. To encode a message $\mathbf{x} \in \mathbb{X}$ into a bit string, we simply write out the integer $f(\mathbf{x})$ in binary. To ensure unique decodability, we make all binary representations equally long by padding shorter ones with leading zeros to the length of the longest one (e.g., if the largest number, $|\mathbb{X}| - 1$, is $13 = (1101)_2$ and we want to encode some message $\mathbf{x}$ with $f(\mathbf{x}) = 2$, then we write out $2 = (0010)_2$). Using your result from part (b), calculate the bit rate $R(\mathbf{x}) \; \forall \mathbf{x} \in \mathbb{X}$. Then use part (a) to show that this trivial method achieves the theoretical lower bound on the expected bit rate for lossless compression with less than one bit of overhead.

(d) Now argue—without referring to any fancy information theoretical theorems—why the lossless compression method from part (c) is obviously optimal (up to maybe one bit of overhead).

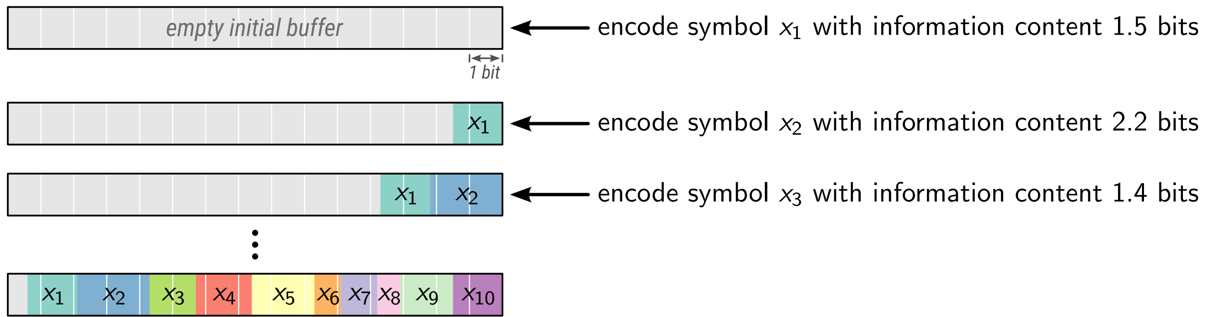

Figure 1: Stream coding with the naive `SlowAnsCoder` from the lecture notes.

Problem 6.2: Streaming ANS in the Style of Piet Mondrian

In the lecture, we first implemented a `SlowAnsCoder` that has near-optimal compression performance but whose run-time cost $O(k^2)$ grows quadratic in the message length k. Figure 1 illustrates how our naive `SlowAnsCoder` represents compressed data if we were to encode the following sequence of symbols:[1]

- a symbol x_1 with information content 1.5 bits;
- a symbol x_2 with information content 2.2 bits;
- a symbol x_3 with information content 1.4 bits;
- a symbol x_4 with information content 1.7 bits;
- a symbol x_5 with information content 1.9 bits;
- a symbol x_6 with information content 0.8 bits;
- a symbol x_7 with information content 1.1 bits;
- a symbol x_8 with information content 0.7 bits;
- a symbol x_9 with information content 1.6 bits; and
- a symbol x_{10} with information content 1.5 bits.

We then reduced the run-time cost to $O(k)$ by splitting the representation of the compressed data into a fixed-sized `head` and a variable-sized `bulk`, resulting in the "streaming ANS" algorithm. We used a setup where `head` can hold up to $2\times$`precision` bits. In streaming ANS, encoding normally only pushes compressed data onto `head` and leaves `bulk` untouched; only if encoding onto `head` would lead to an overflow (i.e., if it would lead to `head` $\geq 2^{2\times\texttt{precision}}$) then one first transfers the `precision` least significant bits from `head` to the end of `bulk` before encoding onto `head`.

[1] We'll gloss over the fact that, in reality, ANS wouldn't be able to represent these *precise* information contents since the corresponding probabilities $2^{-(\text{information content})}$ aren't rational numbers, so they can't be precisely represented with the fixed-point model Q.

Let's visualize how streaming ANS would encode the above sequence of ten symbols. Draw a figure analogous to Figure 1 to sketch what the resulting compressed representation would look like after each step of the process (i.e., which parts of the `bulk` and `head` correspond to which symbol). Assume `precision` = 4 (which would be an unreasonably low precision for real applications but suffices here for demonstration purpose).

You should find that some of the symbols get logically "split up" into two or even three not necessarily neighboring parts. Further, the very first symbol x_1 doesn't get flushed from `head` to `bulk` until the very end. More precisely, after encoding all ten symbols and calling `get_compressed()` you get a compressed representation that is a sequence of four integers, each one being `precision` = 4 bits long, where

- the first integer encodes x_3, x_4, and a part of x_2;
- the second integer encodes x_6, x_7, x_8, and a part of x_5;
- the third integer encodes x_9, x_{10}, another part of x_2, and another part of x_5; and
- the fourth integer encodes x_1 and yet another part of x_2.

Problem 6.3: Range Coding With an Autoregressive Model for English Text

In Problem 3.2 on Problem Set 3, you trained an autoregressive machine learning model (a recurrent neural network) to model the probability distribution of English text. You then used this model as an entropy model for compressing text. Back then, you used a Huffman coder since we hadn't introduced stream codes yet.

In this problem, you'll replace the Huffman coder with a range coder, and you'll evaluate empirically how this affects compression performance (i.e., the bit rate).

(a) Before you start: why is it better to use a range coder here and not an ANS coder?

(b) I won't make you implement the core range coding algorithm because its implementation is a bit involved due to some edge cases and I don't think you'll learn much from it. Instead, we'll use a pre-built range coder provided by the `constriction` library, which was specially developed with research and teaching use cases in mind.[2] Install `constriction` by executing (preferably in a virtual environment):

```
python3 -m pip install constriction~=0.2.4
```

Then try out the first code example from the API documentation of `constriction`'s range coder.[3] The example should execute without errors and print some example message (i.e., a sequence of symbols), encode it, print the compressed representation, and then decode it and print the reconstructed message.

[2] If you run into problems with the `constriction` library, please let me know or report an issue at https://github.com/bamler-lab/constriction/issues

[3] https://bamler-lab.github.io/constriction/apidoc/python/stream/queue.html

Read the code example and make sure you understand what it does. You can ignore anything related to `message_part2`, which shows how to use a model class called `QuantizedGaussian`—we won't need this type of model, only the `Categorical` model that's used for encoding `message_part1` in this example.

(c) Now use your newly acquired range coding skills to replace the Huffman coder in our compression method from Problem 3.2. Don't worry if you haven't completed Problem 3.2, you can always download the proposed solutions[4] from the course website. The PDF document that's part of the solutions also contains instructions for how to set up your python environment and train the model (you'll probably have to reinstall `constriction` in the new python environment using the same command as in part (b)).

Evaluate the compression performance by comparing the bit rate to (i) the results that you get with Huffman coding and (ii) the information content under the model (which is printed to the terminal in the proposed solutions for Problem 3.2).

[4] https://robamler.github.io/teaching/compress22/problem-set-03-solutions.zip