Solutions to Problem Set 3

discussed: 13 May 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 3.1: Kullback-Leibler Divergence

In the lecture, we introduced two probability distributions, p_{data} and p_{model}. Here,

- p_{data} is the true distribution of the data source, which we typically don't know, but we may have a data set of empirical samples from it (e.g., a data set of uncompressed images if we're concerned with image compression); and
- p_{model} is an approximation of p_{data} that we use to construct our lossless compression code; for now, we assume that we can explicitly evaluate $p_{\text{model}}(\mathbf{x})$ for any hypothetical message $\mathbf{x}$.

We showed that, if a lossless compression code C is optimal with respect to p_{model} then its *expected bit rate* on data from p_{data} is given by the cross entropy $H(p_{\text{data}}, p_{\text{model}})$,

$$\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\big[R_C(\mathbf{x})\big] = H(p_{\text{data}}, p_{\text{model}}) + \varepsilon \equiv -\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}\big[\log_2 p_{\text{model}}(\mathbf{x})\big] + \varepsilon. \tag{1}$$

Here, ε is a tiny overhead that is irrelevant for practical purposes (assuming long messages $\mathbf{x}$), the bit rate $R_C(\mathbf{x})$ denotes the total length (in bits) of the compressed representation of a message $\mathbf{x}$, and the notation $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R_C(\mathbf{x})] := \sum_{\mathbf{x}} p_{\text{data}}(\mathbf{x}) R_C(\mathbf{x})$ denotes the (formal) expectation value of $R_C(\mathbf{x})$ under the probability distribution p_{data} (in practice, we can't *evaluate* $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R_C(\mathbf{x})]$ because we don't know $p_{\text{data}}(\mathbf{x})$, but we can *estimate* $\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[R_C(\mathbf{x})]$ by averaging $R_C(\mathbf{x})$ over samples from a finite training set or test set).

Since we can only use p_{model} but not p_{data} to construct our lossless compression algorithm, any deviation between the two probability distributions will degrade compression performance, and the expected bit rate will exceed the fundamental lower bound from Eq. 1. We defined the overhead in expected bit rate due to a mismatch between p_{model} and p_{data} as the Kullback-Leibler divergence,

$$D_{\text{KL}}(p_{\text{data}} \,||\, p_{\text{model}}) := H(p_{\text{data}}, p_{\text{model}}) - H(p_{\text{data}}). \tag{2}$$

(a) Convince yourself that the following two expressions are valid formulations of the Kullback-Leibler divergence:

$$D_{\text{KL}}(p \,||\, q) = \mathbb{E}_{\mathbf{x}\sim p}\big[\log_2 p(\mathbf{x}) - \log_2 q(\mathbf{x})\big] = \mathbb{E}_{\mathbf{x}\sim p}\left[\log_2 \frac{p(\mathbf{x})}{q(\mathbf{x})}\right] \tag{3}$$

(This is a fairly trivial exercise but Eqs. 2 and 3 are both important to remember.)

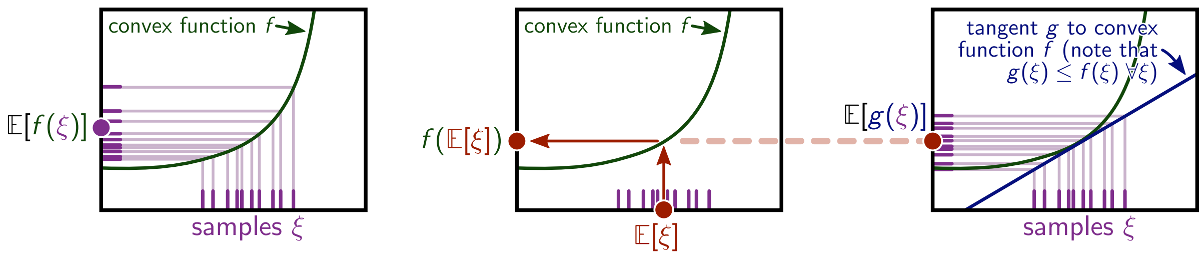

Figure 1: Illustration of Jensen's inequality. Left: $\mathbb{E}[f(\xi)]$ for some convex function f. Center: $f(\mathbb{E}[\xi])$ for the same convex function f. Right: $\mathbb{E}[g(\xi)])$ where g is the affine linear function whose graph (blue) is a tangent to f, touching it at the point $(\mathbb{E}[\xi], f(\mathbb{E}[\xi]))$. Since f is convex, the tangent g to it satisfies $g(\xi) \le f(\xi)\ \forall\xi$ and thus $\mathbb{E}[g(\xi)] \le \mathbb{E}[f(\xi)]$. Further, since g is affine linear, it can be pulled out of the expectation: $\mathbb{E}[g(\xi)] = g(\mathbb{E}[\xi]) = f(\mathbb{E}[\xi])$. Thus, in total, $f(\mathbb{E}[\xi]) \le \mathbb{E}[f(\xi)]$ for any convex function f.

Solution: Both formulations in Eq. 3 follow directly from the definition of D_{KL} in Eq. 2; the definitions of the entropy and the cross entropy, the properties of the logarithm, and the linearity of the expectation value:

$$\begin{aligned} D_{\mathrm{KL}}(p\,||\,q) &= H(p,q) - H(p) \\ &= \mathbb{E}_{\mathbf{x}\sim p}\big[-\log_2 q(\mathbf{x})\big] - \mathbb{E}_{\mathbf{x}\sim p}\big[-\log_2 p(\mathbf{x})\big] \\ &= \mathbb{E}_{\mathbf{x}\sim p}\big[\log_2 p(\mathbf{x}) - \log_2 q(\mathbf{x})\big] \\ &= \mathbb{E}_{\mathbf{x}\sim p}\left[\log_2 \frac{p(\mathbf{x})}{q(\mathbf{x})}\right] \end{aligned}$$

■

(b) Since D_{KL} measures the overhead in expected bit rate over its fundamental lower bound we kind of already know that it cannot be negative. But let's prove this in a more direct way. The proof uses Jensen's inequality (see Figure 1), which states that, for any *convex* function f and any probability distribution p, we have:

$$f\big(\mathbb{E}_{\xi\sim p}[\xi]\big) \le \mathbb{E}_{\xi\sim p}\big[f(\xi)\big] \qquad \text{(for convex } f\text{)}. \tag{4}$$

Prove that $D_{\mathrm{KL}}(p\,||\,q) \ge 0$ using Eq. 3, Jensen's inequality, and the fact that the function $f(\xi) = -\log_2 \xi$ is convex.

Note: Jensen's inequality is often useful to prove bounds in information theory and in approximate Bayesian inference (scheduled for Lecture 7 or 8).

Solution: Let $f : \mathbb{R}_{>0} \to \mathbb{R}$ be the convex function with $f(\xi) := -\log_2 \xi$ (you can see that f is convex by noting that its second derivative, $f''(\xi) = \frac{1}{\xi^2}$, is nonnegative for all ξ in the domain of f). Then start from the last formulation of D_{KL} in Eq. 3

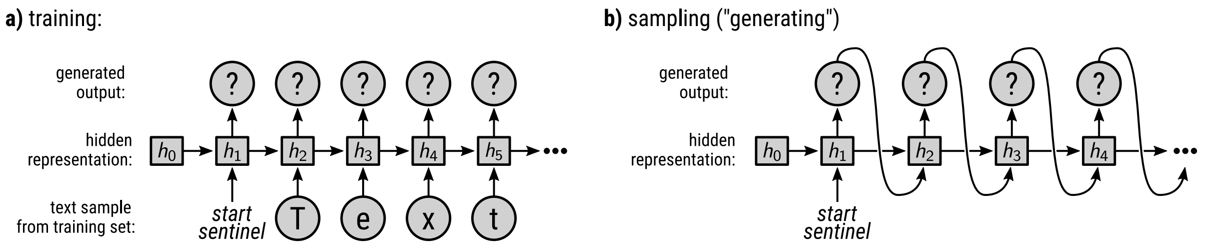

Figure 2: Autoregressive model for character based text generation. a) Training: at training time, the model reads in some English text character by character. After reading character i, it models a probability distribution $p_{\text{model},i+1}$ for the next character x_{i+1}. The training algorithm minimizes the cross entropy between the true data distribution (estimated via samples from a training set) and the modeled probability distribution. b) Sampling, as implemented in the function `generate`: out of the box, the model already comes with a function that generates random text using a trained model. The function draws a random sample $x_1 \sim p_{\text{model},1}$, which it outputs and then also feeds back into the model so that it can then calculate $p_{\text{model},2}$, from which it draws the next character x_2, and so on.

and apply Jensen's inequality:

$$\begin{aligned} D_{\mathrm{KL}}(p\,||\,q) &= \mathbb{E}_{\mathbf{x}\sim p}\left[\log_2 \frac{p(\mathbf{x})}{q(\mathbf{x})}\right] = \mathbb{E}_{\mathbf{x}\sim p}\left[-\log_2 \frac{q(\mathbf{x})}{p(\mathbf{x})}\right] = \mathbb{E}_{\mathbf{x}\sim p}\left[f\left(\frac{q(\mathbf{x})}{p(\mathbf{x})}\right)\right] \\ &\geq f\left(\mathbb{E}_{\mathbf{x}\sim p}\left[\frac{q(\mathbf{x})}{p(\mathbf{x})}\right]\right) = f\left(\sum_{\mathbf{x}} p(\mathbf{x})\,\frac{q(\mathbf{x})}{p(\mathbf{x})}\right) = f\left(\sum_{\mathbf{x}} q(\mathbf{x})\right) = f(1) = 0 \end{aligned}$$

where, on the second line, we explicitly wrote out the expectation as a weighted sum and then used the fact that a normalized probability distribution sums to 1. ■

Problem 3.2: Lossless Compression of Natural Language With Recurrent Neural Networks

This `zip`-file contains code for a simple character-based autoregressive language model. It is a fork of the `char-rnn.pytorch` repository[1] on GitHub. We will talk more about autoregressive models in the next lecture, but Figure 2 should give you enough of an overview to dive into the code. In this problem, you will first train the model on some toy training data. You will then use the trained model to implement your own lossless compression codec for text, which you will evaluate empirically by comparing its bit rate to theoretical bounds and to existing lossless compression methods.

[1] https://github.com/spro/char-rnn.pytorch

Although the compression codec you'll implement in this problem will already be quite effective (considering its simplicity), it will still be suboptimal, and it will also be very slow. We will improve upon it in upcoming problem sets as we learn about better compression techniques.

The code comes as a git bundle. To extract it, run:

```
git clone char-rnn-compression.gitbundle char-rnn-compression
```

You'll also need the python packages PyTorch, `numpy`, `tqdm`, and `unidecode`. You can install them, e.g., as follows (or use your favorite package manager instead):

```
cd char-rnn-compression
python3 -m pip install virtualenv
python3 -m virtualenv -p python3 venv
source venv/bin/activate
python3 -m pip install torch tqdm unidecode numpy
```

The repository contains some toy data set of (historic) English text[2] in the directory `dat`. In order to allow us to compare results quantitatively, the directory also contains a canonical random split into training, validation, and test set.

(a) Train the model on the training set:

```
python3 train.py dat/shakespeare.txt
```

Training this small model doesn't require any fancy hardware; it should only take about 10 to 20 minutes on a regular consumer PC.

The script will use the training set at `dat/shakespeare.train.txt`. Before training and after every tenth training epoch, the script will evaluate the model's performance on the validation set (`dat/shakespeare.val.txt`) and it will print out the cross entropy (to base 2). In regular intervals, the script will also print out some samples from the model (i.e., random generated text). You should be able to observe that the cross entropy decreases (because that's the objective function that the training procedure minimizes), and the generated text should resemble more and more the kind of text you can find in the training set. At the end of training, the cross entropy should oscillate roughly around 2 bits per character.

The trained model will be saved to a file named shakespeare.pt. You can now evaluate it on the validation or test set:

```
python3 evaluate.py shakespeare.pt dat/shakespeare.val.txt
python3 evaluate.py shakespeare.pt dat/shakespeare.test.txt
```

(b) While the model is training, familiarize yourself with the code in `evaluate.py` and in `generate.py` and try to understand what the functions `evaluate` and `generate` do. What does calling `torch.multinomial(output_dist, 1)` in the

[2]Downloaded from `https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt`

method `generate` achieve? (In particular, you should understand that `output_dist` is an unnormalized probability distribution here.)

Note: Both function signatures contain an argument with name `decoder`. The name of this argument is reminiscent of the naming convention in the original code repository, which was not implemented with data compression in mind. Despite its name, this argument is *not* a decoder in the sense of data compression. It is just the trained model.

Solution: The function `generate` takes an initial string of characters `prime_str`, and it then samples text from the model that starts with `prime_str`. It does so by unrolling the model as illustrated in Figure 2 (b). Here, the step from the hidden representation h_i) to the generated character (depicted as a circle with question mark in the figure) deserves special attention. It is the only step in the process that is *stochastic*. By contrast, all other steps in the model are *deterministic*. The hidden representation h_i parameterizes a probability distribution over characters.

In detail, `generate` first obtains a vector of (unnormalized) probabilities for each character in the alphabet and assigns it to the variable `output_dist`. Here, "unnormalized" means that the components of output dist don't necessarily sum up to one. But they are still all nonnegative, and the the probability of the i^{th} character in the alphabet is implicitly defined as `output_dist[i]` $\big/ \sum_j$ `output_dist[j]`. Calling `torch.multinomial(output dist, 1)` draws a single sample from this distribution, taking care of the normalization internally (according to the documentation[3]).

The function `evaluate` estimates the cross entropy $H(p_{\text{data}}, p_{\text{model}})$ by performing an empirical average over $-\log p_{\text{model}}(\mathbf{x})$ on a random sample from the data provided in the argument `text_file`. It normalizes the cross entropy by the length of the sample, i.e., it returns the cross entropy per character. The length of the sample can be controlled by the argument `chunk_len`. By default, `chunk_len` is rather small so that the evaluation doesn't take too much time, but this has the effect that the estimate will be noisy, i.e., the return value of `evaluate` will fluctuate quite a bit across function invocations. Such fluctuations are OK for debugging output, but when you evaluate the model later you should set `chunk_len` to a larger value so as to reduce the variance.

The estimation of the cross entropy also has to take into account that the model only outputs unnormalized probabilities. The model parameterizes probabilities by the `logits` (i.e., the logarithms of unnormalized probabilities). Thus, the negative log probability of character i is given as

$$-\ln p(x_i) = -\ln \frac{\exp\big(\texttt{logit[i]}\big)}{\sum_j \exp\big(\texttt{logit[j]}\big)} = \log\left(\sum_j \exp\big(\texttt{logit[j]}\big)\right) - \texttt{logit[i]}.$$

[3] https://pytorch.org/docs/stable/generated/torch.multinomial.html

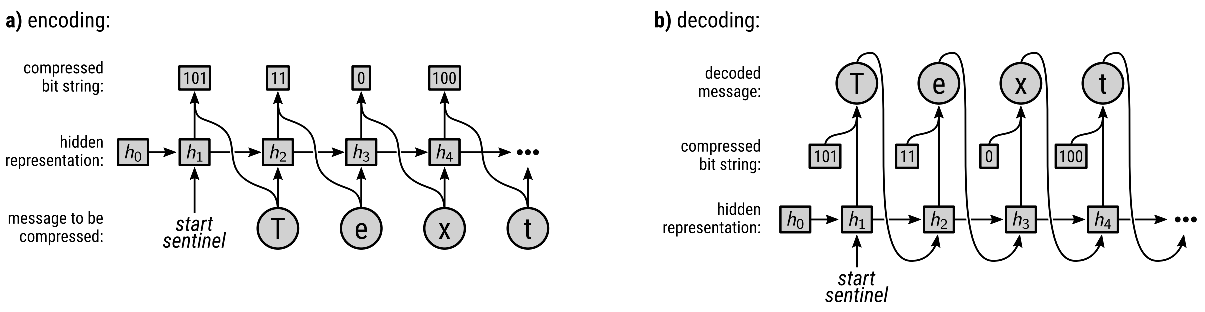

Figure 3: Encoding and decoding with a symbol code that is informed by an autoregressive model. Both encoding and decoding unroll the autoregressive model, which produces a sequence of probability distributions over the alphabet of characters. We use these probability distributions to construct a sequence of Huffman Codes, one Huffman Code per encoded/decoded character. a) at encoding time, we know the entire message, so we can simply unroll the model on the message and use the resulting Huffman Codes to encode each character. b) at decoding time, we start without any knowledge of the message, but we can unroll the autoregressive model up to its first step as this doesn't yet require any input from the message. We can then construct the Huffman Code for the first character, decode the character, and feed it into the autoregressive model in order to transition to the second step. We then repeat this process, consuming a small chunk of the compressed bit string at each step.

Here, a naive evaluation of the first term on the far right-hand-side would be numerically unstable because the exponential function can easily overflow. The function `evaluate` therefore applies the so-called "log-sum-exp trick" to make the calculation numerically stable. The trick is to subtract $\max_k$ `logit[k]` from all logits, observing that such a global shift does not change value of the right-hand side (apart effects due to rounding errors). ■

(c) You should have observed that the function `generate` generates random text. This is possible because the trained model parameterizes a probability distribution p_{model} over character sequences, so one can draw random samples from it. However, in compression, we don't want to generate random text. We want the receiver to be able to deterministically decode the exact same text that the sender encoded. How can you achieve this using the trained probabilistic model. Make a sketch similar to Figure 2 to illustrate your approaches for encoding and decoding. Where do you generate/consume code words and what probability distributions do you use to build the corresponding code books.

Solution: This is precisely the concept of "entropy coding", of which the symbol codes that we've been discussing so far are an example: entropy coding employs a probabilistic model but it still admits deterministic generation. In contrast to the function `generate`, which uses the probabilistic model to draw random samples, we will now use the probabilistic model to construct an optimal symbol code, which we then use to decode a symbol from a bit string.

You can think of this approach as the probabilistic model making a “fuzzy” prediction for the next character. To turn this fuzzy prediction into a precise prediction, we have to inject additional information in the form of a few bits from the compressed bit string. The better the fuzzy prediction was to begin with (i.e., the better the probabilistic model resembles the true data distribution), the less additional information in the form of compressed bits you have to inject. Figure 3 illustrates our approach for encoding and decoding. ■

(d) Create a new file `compression.py` that contains a function `encode_huffman` with the following (or a similar) signature:

```
def encode_huffman(model, message, length_only=False):
```

Here, the argument `model` is a trained model (this is called `decoder` in the the function `generate`) and the argument `message` is a string of English text. The function should return bit string, i.e., the compressed representation of `message`. If the boolean switch `length_only` is set to `True` then the function shouldn't really build up the compressed representation. Instead, it should only simulate the process and return the bit rate, i.e., the length (in bits) of the compressed representation. This is for your convenience, since in the evaluation you'll mostly be interested only in the file size and not in the actual contents of the file.

In order implement `encode_huffman`, you'll need to copy and paste your implementation of the Huffman Coding algorithm from Problem Set 1 (if you didn't solve Problem Set 1, use the solution provided on the course website[4]).

Hint 1: start by copying the body of the function `evaluate`, then adjust it to your needs. You'll have to build up a different Huffman tree for every single character in the message (because the probability distribution is different for every character).

Hint 2: you can apply the Huffman coding algorithm directly to an unnormalized probability distribution (i.e., to `logits.exp()`). This works because the overall scale doesn't affect how the Huffman tree will get constructed.

Solution: See accompanying code.

- To bring the solutions into your code base, cd into your code base and then run

```
git stash
git checkout -b solutions 802c25f
git pull path/to/char-rnn-compression-solutions.gitbundle
```

- If you never cloned the original code repository from the problem set, then run instead:

[4]https://robamler.github.io/teaching/compress22/

```
git clone path/to/char-rnn-compression-solutions.gitbundle \
    char-rnn-compression
cd char-rnn-compression
python3 -m pip install virtualenv
python3 -m virtualenv -p python3 venv
source venv/bin/activate
python3 -m pip install torch tqdm unidecode numpy
```

- If you haven't done so already, train the model with the following command:

  ```
  source venv/bin/activate
  python3 train.py dat/shakespeare.txt
  ```

- Then encode some text file (e.g., the validation set, which is included in the gitbundle at `dat/shakespeare.val.txt`) by running:

  ```
  python3 compression.py shakespeare.pt \
      dat/shakespeare.val.txt encode
  ```

  This prints some statistics to the terminal and it writes the compressed bit string to a file at `dat/shakespeare.val.txt.compressed`. If you're only interested in the statistics and you don't need the compressed data, then add the `--length_only` flag to speed up the process:

  ```
  python3 compression.py shakespeare.pt \
      dat/shakespeare.val.txt encode --length_only
  ```

■

(e) Evaluate the compression performance of your implementation on some sample texts. Try it out on different kinds of texts, ranging from the validation set (which should be very similar to the training set) to more modern English text (e.g., a Wikipedia page) to text in a different language. Compare your codec's bit rate to:

- the information content ($-\sum_{i=1}^{k} \log_2 p_{\text{model},i}(x_i)$ where $p_{\text{model},i}$ is the probability distribution for the i'th character) of the message **x** that was provided to the `encode` function (calculating the information content will be very similar to the implementation of the *evaluate* function);
- the bit rate had you used Shannon coding instead of Huffman Coding (this is $\sum_{i=1}^{k} \lceil -\log_2 p_{\text{model},i}(x_i) \rceil$); and to
- standard lossless compression techniques such as `gzip` or `bzip2` (make sure you use the `--best` switch when running these baselines).

Also, write the compressed output to a binary file (pad to full bytes with trailing zero bits for now) and then compress this file with `gzip` or `bzip2` and record the resulting bit rates. Discuss your results (which compression method works best? How much improvement can you expect at most if you'd use a so-called stream code, i.e., a lossless compression code that is not a symbol code and that can therefore be more effective than Huffman coding?)

Solution: I tested the compression method on the validation and test sets, and on plain-text versions of the Wikipedia articles on Claude Shannon in the English and German language. The Wikipedia articles were preprocessed to ensure that they contain only characters in the alphabet (e.g., by replacing German umlauts with their non-umlaut counterparts). The preprocessed Wikipedia articles are included in the gitbundle at `dat/wikipedia-{en, de}.txt`, and will be referred to as *wikipedia-en* and *wikipedia-de* below, respectively. The following table summarizes the results:

	msg. len	bits per character					
	(chars)	Huffman	Shannon	inf. cont.	`gzip`	`bzip2`	`bzip2'`
validation set	106,864	**2.38**	2.72	2.12	3.43	2.82	2.40
test set	219,561	**2.38**	2.73	2.12	3.33	2.65	**2.38**
wikipedia-en	24,618	4.99	5.67	5.14	3.22	**2.92**	5.14
wikipedia-de	8,426	6.77	7.70	7.19	3.96	**3.76**	7.22

Here, "msg. len" is the length of the uncompressed message $\mathbf{x}$ (number of characters), "inf. cont." is the information content, $-\log_2 p_{\text{model}}(\mathbf{x})$, of the message under our trained autoregressive model, and `bizip2'` is the result of compressing the output of our method (the autoregressive model with Huffman Coding) with `bzip2`. Both `gzip` and `bzip2` were always run with the `--best` switch. We observe that Huffman coding with the trained model outperforms the standard methods `gzip` and `bzip2` on messages that are very similar to the training set, but compression performance degrades the more the message differs in style from the training data. The validation and test set are both very similar to the training set, and the model performs essentially equally well on both (which is to be expected since I never actually used the validation set for hyperparameter tuning). The model performs worse on the English language Wikipedia article and even worse on the Germen language Wikipedia article. This can be explained since modern English language is different from the Shakespeare training text, but still closer to it than German language text.

We further observe that Huffman Coding performs better than Shannon Coding (as expected since both are symbol codes but only the Huffman Code is guaranteed to always be an *optimal* symbol code). Further, both Huffman Coding and Shannon Coding have an overhead over the information content when evaluated on the validation and test set, as expected. Interestingly, however, the bitrate of Huffman Coding on the Wikipedia articles is actually lower than the information content. This is an artefact of symbol codes, as the restriction to integer code word lengths has a regularizing effect: symbol codes have to spend at least one bit for every symbol, even for very probable symbols whose information content is much smaller than one bit; but this also means that, in compensation, symbol codes can assign code words that are considerably shorter than the information content to symbols of very low probability without violating the Kraft inequality. Thus, the code word lengths in a symbol code tend to be more level than the true information contents, and the (*unweighted*) average code word length can be shorter than the (unweighted) average information content.

This regularization effect is typically a poor trade off because slightly longer code words for frequent symbols outweigh the potential benefits even of considerably shorter code words for infrequent symbols. But if the model is evaluated on out-of-distribution data, as we do here, then the probabilities under the model are a poor prediction of true symbol frequencies, and optimizing the lower (unweighted) average code word length of symbol codes can actually become beneficial.

Finally, we observe in the last column of the table that further compressing the already compressed output of our Huffman Coder does not actually reduce the file size. In contrast, it usually even makes things worse, even on the out-of-distribution data where our method performs poorly. This is because `bzip2` compresses its input data by detecting repeated sequences that are aligned with byte boundaries. Since the Huffman Coder produces code words of odd lengths using a different code book for every symbol, there is no reason why it should produce repeated bit strings that are aligned with byte boundaries more often than one would expect in a string of uniformly random distributed bits. As we've learned in the lecture, there's no silver bullet in compression: you always have to make assumptions about the data source—typically in the form of a probabilistic model. In the case of `bizip2'`, the model that the `bzip2` algorithm (implicitly) uses just doesn't match the true characteristics of our Huffman Coder. ■

(f) Implement a decoder and verify empirically that `decode(encode(message)) == message` for some sample `message`.

Note: make sure that `message` is pure ASCII (thus, e.g., no German text with umlauts) because this simple toy model does not support any non-ASCII characters.

Solution: See again accompanying code in the file `compression.py`. You can execute the decoder by running:

```
python3 compression.py shakespeare.pt \
    dat/wikipedia-de.txt encode
python3 compression.py shakespeare.pt \
    dat/wikipedia-de.txt.compressed \
    decode > dat/wikipedia-de.txt.decompressed
sha1sum dat/wikipedia-de.txt dat/wikipedia-de.txt.decompressed
```

The last command should print the same checksum for both files. ■