Solutions to Problem Set 3

discussed:
10 May 2023

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

Course materials available at https://robamler.github.io/teaching/compress23/

Problem 3.1: Kullback-Leibler Divergence

In the lecture, we introduced the Kullback-Leibler (KL) divergence, or relative entropy, D_{KL}. The KL-divergence is an information-theoretical measure of mismatch between two probability distributions. We discussed that $D_{\mathrm{KL}}\big(p_{\mathrm{data}}(\mathbf{x}) \,\big\|\, p_{\mathrm{model}}(\mathbf{x})\big)$ quantifies by how much the expected bit rate of an optimal lossless compression code increases when we use a model p_{model} that does not perfectly match the (unknown) distribution p_{data} of the true data generative process. Thus, the KL-divergence is defined as follows,

$$D_{\mathrm{KL}}\big(p_{\mathrm{data}}(\mathbf{x}) \,\big\|\, p_{\mathrm{model}}(\mathbf{x})\big) := H(p_{\mathrm{data}}, p_{\mathrm{model}}) - H(p_{\mathrm{data}}). \tag{1}$$

The KL-divergence is a useful quantity in more than just data compression. For example, we will see it come up again when we introduce variational inference in Lecture 8.

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

$$D_{\mathrm{KL}}\big(p(\mathbf{x}) \,\big\|\, q(\mathbf{x})\big) = \mathbb{E}_{\mathbf{x}\sim p}\big[\log p(\mathbf{x}) - \log q(\mathbf{x})\big] = \mathbb{E}_{\mathbf{x}\sim p}\left[\log \frac{p(\mathbf{x})}{q(\mathbf{x})}\right] \tag{2}$$

Here, the notation $\mathbb{E}_{\mathbf{x}\sim p}[f(\mathbf{x})] := \sum_{\mathbf{x}} p(\mathbf{x}) f(\mathbf{x})$ denotes the expectation value of some function f under the probability distribution p. (In practice, we sometimes can't evaluate $p(\mathbf{x})$ and therefore can't calculate the weighted sum over all $\mathbf{x}$, but we might be able to *estimate* $\mathbb{E}_{\mathbf{x}\sim p}[f(\mathbf{x})]$ by averaging $f(\mathbf{x})$ over samples from a finite training set or test set, as discussed in the lecture.)

Note: This is a fairly trivial exercise but Eqs. 1 and 2 are both important to remember.

Solution: Both formulations in Eq. 2 follow directly from the definition of D_{KL} in Eq. 1; the definitions of the entropy and the cross entropy, the properties of the logarithm, and the linearity of the expectation value:

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

■

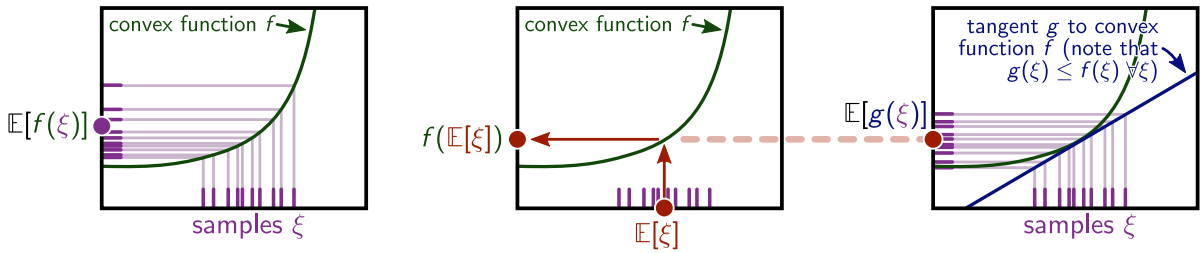


Figure 1: Illustration of Jensen's inequality (Eq. 3). Left: $\mathbb{E}[f(\xi)]$ for some convex function f. Center: $f(\mathbb{E}[\xi])$ for the same convex function f. Right: $\mathbb{E}[g(\xi)]$) where g is the affine linear function whose graph (blue) is a tangent to f, touching it at the point $(\mathbb{E}[\xi], f(\mathbb{E}[\xi]))$. Since f is convex, the tangent g to it satisfies $g(\xi) \leq f(\xi)\ \forall \xi$ and thus $\mathbb{E}[g(\xi)] \leq \mathbb{E}[f(\xi)]$. 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, $f(\mathbb{E}[\xi]) \leq \mathbb{E}[f(\xi)]$ for any convex function f, as claimed in Eq. 3.

(b) Since D_{KL} measures the overhead in expected bit rate over its 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 on the next page), which states that, for any *convex* function f and any probability distribution p, we have:

$$f\big(\mathbb{E}_{\xi\sim p}[\xi]\big) \leq \mathbb{E}_{\xi\sim p}\big[f(\xi)\big] \qquad \text{(for convex } f\text{)}. \tag{3}$$

Prove that $D_{\mathrm{KL}}\big(p(\mathbf{x}) \,\big\|\, q(\mathbf{x})\big) \geq 0$ for all probability distributions p and q by using Eq. 2, Jensen's inequality, and the fact that the function $f(\xi) := -\log \xi$ is convex.

Note: Jensen's inequality (Eq. 3) is a very useful relation that often comes up when proving bounds in information theory and in approximate Bayesian inference (scheduled for Lecture 8).

Solution: Let $f : \mathbb{R}_{>0} \to \mathbb{R}$ be the convex function with $f(\xi) := -\log \xi$ (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. 2 and apply Jensen's inequality:

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

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. ■

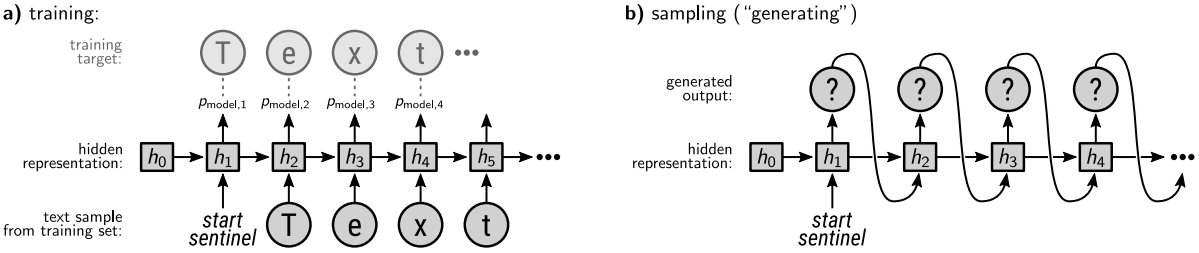


Figure 2: Autoregressive model for character based text generation. a) The training algorithm minimizes the cross entropy between the true distribution of English text (estimated via samples from a training set) and a probabilistic model parameterized by a so-called recurrent neural network. The model reads text character by character (bottom row). After reading character i, the model parameterizes a probability distribution $p_{\text{model},i+1}$ over the next character. The training algorithm then tries to minimize the information content of the correct next character (top row) under $p_{\text{model},i+1}$. b) In addition to a training objective, the model implementation also already comes with a function `generate` that samples from a trained model. The function draws a random first character $x_1 \sim p_{\text{model},1}$, prints it, then feeds it back into the model in order to calculate $p_{\text{model},2}$, from which the function draws the next character x_2, and so on.

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, forked from a GitHub repository[1] by Sean Robertson. You will learn more about autoregressive models in the next lecture, but Figure 2 on the next page should tell you enough to solve this problem.

In this problem, you will train the model and then turn it into a compression method, whose performance you evaluate empirically. Although your resulting compression method will already be quite effective (considering its simplicity), it will still waste some bit rate, and it will also be very slow. You will improve upon it in upcoming problem sets as you learn more about deep probabilistic models and entropy coders.

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

```
git clone path/to/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
```

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

```
python3 -m pip install torch tqdm unidecode numpy
```

Once everything is set up, it's time to get your hands dirty.

(a) The repository contains some toy data set of historic English text[2] in the directory `dat`, together with a canonical random split (by lines) into training, validation, and test set. Train the model on the training set by executing:

```
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 function `generate` achieve, and what quantities does `output_dist` contain?

Note: Both `evaluate` and `generate` take an argument named `decoder`. Despite its name, this argument is *not* a decoder in the sense of data compression. It is just the trained model (the original code base had no relation to data compression).

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 this 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.

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

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 probability of the i^{th} character in the alphabet is implicitly defined as `output_dist[i]` $/ \sum_j$ `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 trained 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

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

Here, a naive evaluation of the first term on the 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 the value of the right-hand side (apart from 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 this distribution. However, in a compression application, 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 and an entropy coder. Make a sketch similar to Figure 2 to illustrate how you would approach encoding and decoding. Where do you generate/consume code

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

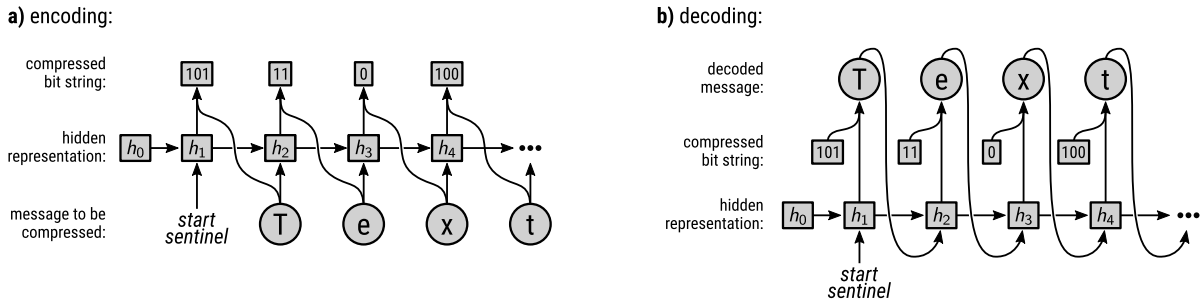


Figure 3: Encoding and decoding with a symbol code that is informed by an autoregressive model. Both encoding and decoding unroll the autoregressive model, which produces a sequence of probability distributions over the alphabet of characters. We use these probability distributions to construct a sequence 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.

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) Let's now implement the encoder. Create a new file `compression.py` and paste your implementations of `HuffmanEncoder` and `HuffmanDecoder` from Problem Sets 1 and 2 into it (if you didn't solve these problem sets, use the solutions from

the course website[4]). Then implement a function `encode_huffman` with signature

```
def encode_huffman(model, message):
```

Here, the argument `model` is a trained model (the same as the confusingly named `decoder` argument in the function `evaluate`), and the argument `message` is a string of English text. The function should compress `message` using Huffman coding, and `yield` the resulting sequence of bits (boolean values). Once the entire message has been processed, the function should print some relevant statistics, such as the total bit rate, the total information content, and the bit rate that you would have obtained had you used Shannon coding instead of Huffman coding.

Your implementation of `encode_huffman` will be similar to the function `evaluate`. Remember that the autoregressive model builds a new probability distribution for each character, so you'll have to build up a new Huffman tree for each character.

Then turn your implementation into an executable script by parsing some appropriate command line arguments (similar to how it is done in `evaluate.py`), reading `message` from a text file, and writing the compressed representation to a new (binary) file. You can use the utilities provided to you in `bitutils.py` for generating a binary file (bring them into scope with `from bitutils import *`).

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

Solution: See accompanying code.

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

```
source venv/bin/activate
git stash
git checkout problem-set-3
git pull path/to/char-rnn-compression-solutions.gitbundle
```

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

```
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:

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

[4] https://robamler.github.io/teaching/compress23/

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

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

  This prints some statistics to the terminal and it writes the compressed bit string to a file at `dat/shakespeare.test.txt.compressed`.

■

(e) **This is the most important part of this problem set:** evaluate the compression performance of your implementation on some sample texts. Try out different kinds of texts, ranging from the test set (which should be very similar to the training set) to more modern English text (e.g., a Wikipedia page), and to text in a different language. Compare your method's bit rate to:

- the information content of the respective texts under the model;
- the bit rate had you used Shannon coding instead of Huffman Coding;
- standard lossless compression techniques such as `gzip` or `bzip2` (make sure you use the `--best` switch when running these baselines); and to
- the file size that you obtain when you take the output of your Huffman coding based compression method and try to compress it further with `gzip` or `bzip2`.

Report your results in *bits per character* so that you can compare compression performance across texts of varying lengths.

Note: This simple toy model only supports ASCII characters, so make sure that your messages don't contain, e.g., German umlauts, fancy quotation marks, etc.

Discuss your results: Which compression method works best? How important is it that the training data resembles the text we end up compressing? Does compressing already compressed data help? 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 are referred to as *wikipedia-en* and *wikipedia-de* below, respectively. Here are 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 sets 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 or early stopping). 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 bit rate 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, conversely, this also means that 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.

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 having a more level distribution of code word lengths 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 byte sequences. But a good compression algorithm should create binary output that is indistinguishable from random data. 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 for some sample text that decoding an encoded message reconstructs the original message. Use the `HuffmanDecoder` class that you implemented on Problem Set 2 and the function `read_bits_from_file` that is provided to you in the file `bitutils.py`.

Hint: the method `decode` on the `HuffmanDecoder` class is a *generator function* that allows you to decode several symbols in sequence, reusing the same `HuffmanDecoder`. However, in this autoregressive model, you have to construct a new `HuffmanDecoder` for each character. Therefore, you'll only want to decode a single character each time. You can use the builtin `next` function to do this:

```
character_index = next(huffman_decoder.decode(bit_iterator))
```

Here, `character_index` is an integer that represents a single character according to conventions that are specific to this machine-learning model. In order to turn `character_index` into an actual printable character, have a look at how this is done in `generate.py`.

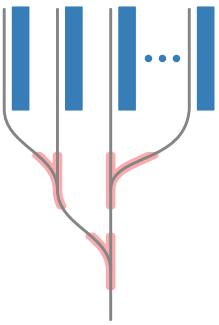
Solution: See again accompanying code in the file `compression.py`. You can execute the decoder by running:

```
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. ■

Congratulations, you have implemented your first machine-learning based compression method! We'll improve upon this method on Problem Set 6.

Problem 3.3: A Different Kind of Trolley Problem



Don't let the title scare you away, we won't be doing any pretentious pseudophilosophy here. But we'll solve a problem that has nothing to do with data compression—or does it?

Imagine you are designing a train station with n platforms labeled $\{1, \dots, n\}$ (blue rectangles in illustration). Each platform serves a single railroad track (gray lines), and all tracks enter the station from the same direction and terminate at the station. You may order the platforms $\{1, \dots, n\}$ arbitrarily, but they must be arranged next to each other and in parallel, as in the illustration.

Here's the difficult part: the train station will serve n different *types* of trains, and each platform $i \in \{1, \dots, n\}$ can serve only trains of type i (think of cargo trains with different kinds of cargo, such as containers, gasoline, lumber, and coal, where each type of cargo requires a designated platform with specialized equipment for loading and unloading). The various types of trains have varying average physical weights, and they arrive at the station with varying frequencies. We denote by $w_i > 0$ the total physical weight (in tons) of trains of type i that we expect to arrive at the station per year.

A single track leaves the train station and is connected to all platforms via some switches (highlighted red in the illustration). Your task is to come up with an arrangement of switches and a convenient order in which to arrange the platforms $\{1, \dots, n\}$. Each switch splits one track into two. Unfortunately, switches are fragile structures: each time a train passes over a switch, the switch takes some damage proportional to the train's weight. How would you come up with a setup that uses as few switches as possible, and that minimizes the total damage on all switches per year for given $(w_i)_{i=1}^n$?

Note: This is problem is obviously contrived. But similar optimization problems are conceivable in sampling problems in statistics, in planning problems (e.g., when a robot explores a sequential decision space), or in routing problems for computer networks.

Solution: For any arrangement that uses as few switches as possible, there can only be one path between each platform and the single track that connects to the outside world. If there were two or more paths, then one of them would be redundant, and cutting it would allow us to remove a switch. Thus, the arrangement of switches forms a tree whose leaf nodes are the platforms, whose non-leaf nodes are the switches, and whose root node is the switch that is connected to the single track that connects to the outside world. The tree is a binary tree since each switch splits one track into two.

All binary trees over n leaves have the same number $(n-1)$ of non-leaf nodes (switches). In order to find the optimal binary tree, we have to minimize the total damage to all switches per year, which is proportional to $W := \sum_{i=1}^n w_i \ell_i$ where ℓ_i is the number of switches that a train of type i has to traverse in order to get to its designated platform (i.e., ℓ_i is the depth of platform i in the tree). We already know an algorithm that optimizes an objective of the form $W := \sum_{i=1}^n w_i \ell_i$ over binary trees: Huffman coding. If we interpret the platforms i as symbols in an alphabet, each weight w_i as the probability

of symbol i, and the tree formed by the railroad tracks as a *trie* that defines a prefix code on the alphabet, then ℓ_i becomes the length of the code word for symbol i, and W is the expected code word length, which Huffman coding minimizes. We can therefore apply Huffman coding to find an optimal tree structure. We can turn the tree into a planar network of railroad tracks (i.e., where no railroad tracks cross each other) by sorting the platforms $\{1, \ldots, n\}$ lexicographically by their Huffman code words.

Note: at a more precise inspection, this analogy is lacking: there is no reason to assume that $\sum_{i=1}^{n} w_i = 1$ (or even that $w_i \leq 1 \, \forall i$). Therefore, we cannot interpret w_i as a probability. Fortunately, the proof of optimality of Huffman coding does not rely on these assumptions, so the theorem actually holds for arbitrary (positive) weights. ■