Solutions to Problem Set 2

Data Compression With Deep Probabilistic Models
Prof. Robert Bamler, University of Tuebingen

Course material available at `https://robamler.github.io/teaching/compress21/`

Problem 2.1: Kraft-McMillan Theorem

In the lecture, we stated and partially proved the Kraft-McMillan Theorem:

Theorem (Kraft-McMillan)**.** *The following two statements are true:*

(a) All B-ary uniquely decodable symbol codes C satisfy the Kraft inequality,

$$\sum_{x\in\mathfrak{X}} \frac{1}{B^{\ell(x)}} \leq 1. \tag{1}$$

Here, $\mathfrak{X}$ is the alphabet of the symbol code and $\ell(x)$ is the length $|C(x)|$ of the code word $C(x)$ (i.e., the number of B-ary bits that make up $C(x)$).

(b) For all functions $\ell : \mathfrak{X} \to \{0, 1, 2, \ldots\}$ that satisfy the Kraft-inequality Eq. 1, there exists a B-ary prefix-free symbol code (i.e., a B-ary prefix code) C with code word lengths ℓ, i.e., $|C(x)| = \ell(x)\ \forall x \in \mathfrak{X}$.

Answer the following questions about the Kraft-McMillan Theorem:

(a) Prove the following statement by combining both parts (a) and (b):

For every uniquely decodable symbol code C, there exists a prefix code $\tilde{C}$ with the same code word lengths, i.e., $|\tilde{C}(x)| = |C(x)|\ \forall x \in \mathfrak{X}$.

Solution: Let C be a uniquely decodable symbol code on an alphabet $\mathfrak{X}$ and let $\ell(x) := |C(x)|$ be the lengths of the code words in C for all $x \in \mathfrak{X}$. Due to part (a) of the theorem, $\ell(x)$ satisfies Eq. 1. Due to part (b) of the theorem, there therefore exists a prefix code $\tilde{C}$ on $\mathfrak{X}$ with $|\tilde{C}(x)| = \ell(x) = |C(x)$ for all $x \in \mathfrak{X}$. (This proof may seem trivial but the insight is important: when dealing with uniquely decodable symbol codes, there is no reason *not* to restrict oneself to prefix codes. Restricting yourself to prefix codes doesn't sacrifice any compression performance and it makes decoding easier because prefix codes can decoded greedily, as we did on the last problem set.) ■

In the lecture, we proved part (a) of the Kraft-McMillan Theorem but we didn't complete the proof of part (b). Let's do this now. Consider Algorithm 1 on the next page, which we introduced in the lecture.

Algorithm 1: Constructive proof of Kraft-McMillan Theorem part (b).

```
Input:  Base $B \in \{2, 3, \ldots\}$, finite alphabet $\mathfrak{X}$,
        function $\ell : \mathfrak{X} \to \{0, 1, 2, \ldots\}$ that satisfies Eq. 1.
Output: Code book $C : \mathfrak{X} \to \{0, \ldots, B-1\}^*$ of a prefix code that
        satisfies $|C(x)| = \ell(x)\ \forall x \in \mathfrak{X}$.
1 Initialize $\xi \leftarrow 1$;
2 for $x$ in $\mathfrak{X}$ in order of nonincreasing $\ell(x)$ do
3     Update $\xi \leftarrow \xi - B^{-\ell(x)}$;
4     Write out $\xi \in [0, 1)$ in its $B$-ary expansion: $\xi = (0.???\ldots)_B$;
5     Set $C(x)$ to the first $\ell(x)$ bits following "0." in the above $B$-ary expansion
        of $\xi$ (pad with trailing zeros to length $\ell(x)$ if necessary);
```

(b) Line 4 of Algorithm 1 claims that $\xi \in [0, 1)$. Why is this always the case at this point in the algorithm?

Solution: The variable ξ is initialized with $\xi \leftarrow 1$ and then never increased during the execution of the algorithm. When we come to line 4, ξ has been decreased by a strictly positive amount ($B^{-\ell(x)} > 0$) at least once, thus ξ is strictly smaller than 1. Further, since ℓ satisfies Eq. 1, the `for`-loop decreases ξ by a total of at most 1, and thus ξ never drops below zero. ■

(c) Denote the value of ξ *after* the update in Line 3 as ξ_x (where x is the iteration variable of the `for` loop). Now consider two symbols $x, x' \in \mathfrak{X}$ with $x \neq x'$ and, without loss of generality, $\xi_{x'} > \xi_x$. Argue that $\xi_{x'} \geq \xi_x + B^{-\ell(x)}$. Then argue that neither can $C(x)$ be a prefix of $C(x')$, nor can $C(x')$ be a prefix of $C(x)$.

Solution: Since each step of the `for`-loop makes ξ smaller and since $\xi_{x'} > \xi_x$, the symbol x' must come *before* the symbol x. Since the `for`-loop iterates in order of nonincreasing $\ell(x)$, this means that $\ell(x') \geq \ell(x)$. Therefore, the only way how $C(x')$ could be a prefix of $C(x)$ is if $\ell(x) = \ell(x')$ and $C(x) = C(x')$, in which case $C(x)$ is also a prefix of $C(x')$. Thus, we only have to prove that $C(x)$ is not a prefix of $C(x')$.

Each step of the algorithm reduces ξ by $B^{-\ell(x)}$. Thus, at the beginning of the iteration for symbol x, the variable ξ had value $\xi_x + B^{-\ell(x)}$, and all $\xi_{x'}$ for symbols x' that come before symbol x satisfy $\xi_{x'} \geq \xi_x + B^{-\ell(x)}$.

Now assume that $C(x)$ (which has length $\ell(x)$) is a prefix of $C(x')$. This means that the fractional parts of the B-ary expansions of ξ_x and of $\xi_{x'}$ agree on the first $\ell(x)$ digits. Thus, they are both in the interval $\big[(0.C(x))_B, (0.C(x))_B + B^{-\ell(x)}\big)$ and thus they differ by strictly less than $B^{-\ell(x)}$, which is a contradiction. ■

(d) *(Advanced:)* Algorithm 1 is limited to a finite alphabet $\mathfrak{X}$ because the `for`-loop would not terminate if $\mathfrak{X}$ was infinite. How could you use the same idea to prove part (b) of the Kraft-McMillan Theorem in the case of a counably infinite alphabet?

Solution: Prooving part (b) of the Kraft-McMillan Theorem doesn't require executing Algorithm 1 for the entire alphabet $\mathfrak{X}$. We only have to find a symbol code $C : \mathfrak{X} \to \{0, \ldots, B-1\}^*$ in which, for all $x, x' \in \mathfrak{X}$ with $x \neq x'$, the code word $C(x)$ is not a prefix of $C(x')$ and $C(x')$ is not a prefix of $C(x)$. Using the same idea as in Algorithm 1, we can define $C(x)$ as the first $\ell(x)$ fractional bits in the B-ary expansion of $1 - \sum_{x' >_\ell x} B^{-\ell(x)}$, where "$>_\ell$" is any total order on $\mathfrak{X}$ that satisfies $\ell(x') > \ell(x)$ for all $x, x' \in \mathfrak{X}$ with $x' >_\ell x$. The rest of the proof goes through as in the case of a finite alphabet. (Another way of thinking about this is that we obtain $C(x)$ by running Algorithm 1 only until the `for`-loop is done with the target symbol x.) ■

Problem 2.2: Entropy

In the lecture, we (re-)introduced the entropy to base B of a probability distribution p:

$$H_B[p] = \mathbb{E}[-\log_B p(x)] = -\sum_{x \in \mathfrak{X}} p(x) \log_B p(x). \tag{2}$$

(a) In the literature, the subscript B will often be dropped. Depending on context, entropies are understood to be either to base 2 (mostly in the compression literature) or to the natural base e (in mathematics, statistics, or machine learning literature). How do $H_2[p]$ and $H_e[p]$ relate to each other?

Solution: Denoting the natural logarithm to base e as ln, we have $\log_B \alpha = \ln \alpha / \ln B$ for all $\alpha \in \mathbb{R}_{>0}$. Therefore, $H_B[p] = H_e[p]/\ln B$. ■

(b) Consider a tuple of $m \in \mathbb{N}$ symbols. The symbols are i.i.d. (statistically independent and identically distributed), i.e., the probability $\tilde{p}$ of a tuple of m symbols is $\tilde{p}\big((x_1, \ldots, x_m)\big) = \prod_{i=1}^m p(x_i)$. Show that

$$H_B[\tilde{p}] = m H_B[p] \qquad \forall m \in \mathbb{N},\ \forall B \in \mathbb{R}_{\geq 0} \tag{3}$$

Solution: Using $\log_B(\alpha\beta) = \log_B \alpha + \log_B \beta$, we find

$$\begin{aligned}
H_B[\tilde{p}] &= -\sum_{\mathbf{x} \in \mathfrak{X}^m} \tilde{p}(\mathbf{x}) \log_B \tilde{p}(\mathbf{x}) = -\sum_{\mathbf{x} \in \mathfrak{X}^m} \tilde{p}(\mathbf{x}) \log_B \left(\prod_{i=1}^m p(x_i) \right) \\
&= -\sum_{\mathbf{x} \in \mathfrak{X}^m} \tilde{p}(\mathbf{x}) \left[\sum_{i=1}^m \log_B p(x_i) \right] = -\sum_{i=1}^m \left[\sum_{\mathbf{x} \in \mathfrak{X}^m} \tilde{p}(\mathbf{x}) \log_B p(x_i) \right] \\
&= -\sum_{i=1}^m \left[\left(\prod_{j \in \{1,\ldots,m\} \setminus \{i\}} \sum_{x_j \in \mathfrak{X}} p(x_j) \right) \left(\sum_{x_i \in \mathfrak{X}} p(x_i) \log_B p(x_i) \right) \right].
\end{aligned}$$

Here, the first factor in the braces drops out since $\sum_{x_j \in \mathfrak{X}} p(x_j) = 1$ for any (properly normalized) probability distribution p. Thus,

$$H_B[\tilde{p}] = -\sum_{i=1}^{m} \sum_{x_i \in \mathfrak{X}} p(x_i) \log_B p(x_i) = \sum_{i=1}^{m} H_B[p] = m H_B[p].$$

■

Problem 2.3: Block Codes and Theoretical Lower Bound for Lossless Compression

In the lecture, we showed that the expected code word length L of a uniquely decodable *symbol code* is lower bounded by the entropy H_B of the symbols. We further showed that this lower bound is nontrivial: for any probability distribution p of symbols, there exists a symbol code (the so-called Shannon Code) that is prefix-free (and therefore uniquely decodable) and for which L comes within less than one bit of this lower bound. Thus, in summary, for the *minimally possible* expected code word length L_{min}, we have

$$H_B[p] \le L_{\text{min}} \le L_{\text{Shannon}} < H_B[p] + 1. \tag{4}$$

So far, these results are limited to symbol codes, i.e., codes for which the encoding $C^*(\mathbf{x})$ of a sequence $\mathbf{x} = (x_i)_{i=1}^{k}$ of symbols x_i is given by simple concatenation of individual context-independent code words $C(x_i)$. In this problem, you will derive a lower bound that holds for *all* lossless B-ary codes, not just for symbol codes. To do so, we introduce the idea of a *Block Code*: Let $m \in \{2, 3, \ldots\}$ and assume that you only care about messages $\mathbf{x} \in \mathfrak{X}^k$ whose length k is a multiple of m, i.e., $k = nm$ for some integer n. You can then group the symbols in the message $\mathbf{x}$ into n blocks of m consecutive symbols each, and construct a symbol code for these blocks.

For example, a message $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5, x_6)$ of length $k = 6$ can be reinterpreted as a message of $n = 2$ blocks of size $m = 3$ each: $\tilde{\mathbf{x}} = \big((x_1, x_2, x_3), (x_4, x_5, x_6)\big)$. In this representation, each block is an element of the product alphabet $\mathfrak{X}^m = \mathfrak{X} \times \mathfrak{X} \times \ldots \times \mathfrak{X}$. One can now construct a code book $\tilde{C}^{(m)} : \mathfrak{X}^m \to \{0, \ldots, B-1\}^*$ for this product alphabet. In particular, we will consider the Shannon Code $\tilde{C}^{(m)}_{\text{Shannon}}$ that one obtains from applying the Shannon Coding algorithm to the product probability distribution $\tilde{p}\big((x_1, \ldots, x_m)\big) = \prod_{i=1}^{m} p(x_i)$.

(a) Use Eqs. 3 and 4 to derive a lower and an upper bound for the expected length of the encoding *per original symbol (from $\mathfrak{X}$)* for $\tilde{C}^{(m)}_{\text{Shannon}}$. You should find that the lower bound does not change compared to Eq. 4, but the upper bound shrinks, i.e., the range narrows.

Solution: Let $L^{(m)}_{\text{Shannon}}$ be the expected code word length of $\tilde{C}^{(m)}_{\text{Shannon}}$. Since $\tilde{C}^{(m)}_{\text{Shannon}}$ is a Shannon code on the alphabet $\mathfrak{X}^m$ for the probability distribution $\tilde{p}$, Eq. 4 applies and we have (using Eq. 3):

$$mH_B[p] \overset{(3)}{=} H_B[\tilde{p}] \le L^{(m)}_{\text{Shannon}} < mH_B[p] + 1. \qquad (\star)$$

Now recall that $\tilde{C}^{(m)}_{\text{Shannon}}$ operates on the alphabet $\mathfrak{X}^m$, and thus $L^{(m)}_{\text{Shannon}}$ is the expected code word length *per block of m original symbols.* Let $L^{(m)'}_{\text{Shannon}}$ be the expected code word length of *per original symbol (from $\mathfrak{X}$).* Thus, $L^{(m)'}_{\text{Shannon}} = L^{(m)}_{\text{Shannon}}/m$. Dividing Eq. ($\star$) by m, we find:

$$H_B[p] \le L^{(m)'}_{\text{Shannon}} < H_B[p] + \frac{1}{m}. \qquad (\star\star)$$

Thus, block codes have the same fundamental lower bound as symbol codes, but the Shannon block code is guaranteed to approach this fundamental lower bound within an overhead of at most $\frac{1}{m}$ bits *per original symbol*, i.e., the overhead becomes smaller as the block size m grows. ■

(b) In the introduction to this problem, we highlighted the restrictions of symbol codes in comparison to arbitrary lossless compression codes: in a symbol code, $C^*(\mathbf{x})$ has to be the concatenation of individual context-independent code words. Now consider a block code for the special case $m = k$. What are its limitations compared to an arbitrary uniquely decodable B-ary lossless compression code that is defined on sequences of k i.i.d. symbols?

Solution: Every uniquely decodable B-ary lossless compression code that is defined on sequences of k i.i.d. symbols can be seen as a block code on the alphabet $\mathfrak{X}$ with block size $m = k$. To be explicit, this holds for any lossless compression code, not just for symbol codes. Thus, the lower and upper bounds in Eq. ($\star\star$) apply to any uniquely decodable B-ary lossless compression code that is defined on sequences of k i.i.d. symbols. Importantly, this means that the lower bound holds for all lossless compression codes, including codes that operate on variable-sized messages (since we could always restrict a code that operates on variable-sized messages to messages of a fixed length).

Note: For simplicity, we have assumed in this problem that we're only interested in compression algorithms that operate on messages $\mathbf{x} \in \mathfrak{X}^k$ of a fixed length k that is already known to the receiver. In many practical applications, only the sender knows the length k of the message but the receiver doesn't. Therefore, one has to consider codes that operate on messages $\mathbf{x} \in \mathfrak{X}^*$ of arbitrary length. In this situation, the lower bound that we derived still holds: any code that operates on $\mathfrak{X}^*$ can be artificially restricted to operate only on $\mathfrak{X}^k$ for some given k. Thus, any lower bound that holds for lossless codes on $\mathfrak{X}^k$ also has to apply to a code that operates on $\mathfrak{X}^*$ when it operates on some $\mathbf{x} \in \mathfrak{X}^k$. However, the upper bound on the optimal code no longer holds since we now have to transmit the length k to the receiver. One way to do this is to use the fact that the length $k \in \mathbb{N}$ comes from a discrete (countably infinite) alphabet $\mathbb{N}$, and we can therefore use Shannon

Coding to encode it into fewer than $H[p_k] + 1$ bits (where p_k is the probability distribution of message lengths k). Prepending the encoding of k to the encoded message adds a small overhead to the length of the compressed representation.

Another way to transmit the message length would be to extend the alphabet $\mathfrak{X}$ by an "end-of-file" sentinel, and to append this sentinel to all messages before encoding them. Then, the decoder knows to stop decoding once it decodes the end-of-file sentinel. However, this second approach doesn't work in our hypothetical setup where we use a block code with block size $m = k$ because, in order to decode messages with this code, the receiver has to know k *before* it starts to decode. ■

(c) What can you say about the Shannon block code $\tilde{C}^{(m)}_{\text{Shannon}}$ with $m = k$ in the (practically relevant) limit of large k? Think about (i) its overhead in expected bits per (original) code word over the theoretical lower bound; and about (ii) the run-time complexity of the Shannon coding algorithm (Algorithm 1) on such a block code as a function of k.

Solution: For long messages, $m = k \gg 1$, the upper bound of $\frac{1}{m}$ on the overhead per symbol becomes negligible small. Thus, the entropy is always a fundamental lower bound on the bit rate of lossless compression but, for long messages, there exists a lossless compression code that essentially achieves this lower bound with virtually no overhead. However, such an optimal lossless code may be hard to find in practice. In particular, our approach via a block Shannon code of block size $m = k$ is infeasible. Since block codes operate on the product alphabet $\mathfrak{X}^m$, the runtime of Algorithm 1 would grow exponentially in m.

In order to turn $\tilde{C}^{(m)}_{\text{Shannon}}$ into a practically useful code, we will have to (i) sum up the terms $B^{-\ell(x)}$ that appear in Algorithm 1 in a more efficient way that doesn't lead to an exponential number of terms, and (ii) limit the numerical precision of the very long B-ary number $\xi = (0.???\ldots)$ to something that can be dealt with in practice without introducing irreparable rounding errors. We will solve both of these problems later in this course, which will lead to Arithmetic Coding and Range Coding, our first examples of lossless compression codes that are not symbol codes but so-called *stream codes.* ■

Problem 2.4: Huffman Coding

In the last tutorial, we introduced the Huffman Coding algorithm. It is restated in a more formal manner in Algorithm Box 2. Like Shannon Coding, the Huffman Coding algorithm takes a probability distribution p over symbols, and it constructs a code book for a prefix code C_{Huffman} whose expected code word length L_{Huffman} satisfies the upper and lower bounds from Eq. 4. Unlike Shannon Coding, a code book constructed by Huffman coding is *optimal* in the sense that there is no uniquely decodable symbol code with expected code word length shorter than L_{Huffman}.

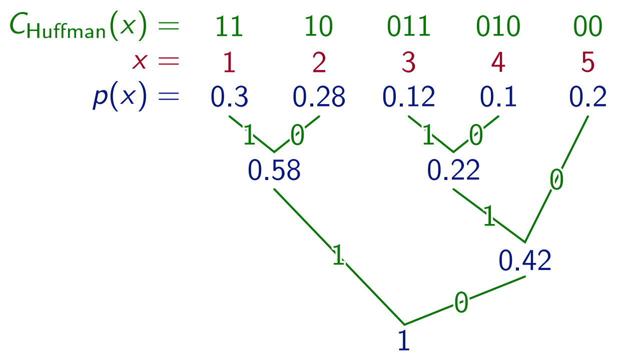
Algorithm 2: Huffman Coding (for base $B = 2$).

```
    Input:  finite alphabet $\mathfrak{X} = \{1, \ldots, |\mathfrak{X}|\}$, probability distribution $p$ on $\mathfrak{X}$.
    Output: Code book $C_{\text{Huffman}} : \mathfrak{X} \to \{0, \ldots, B-1\}^*$ of an optimal prefix code on $\mathfrak{X}$.
 1  Initialize a set of tree roots $R \leftarrow \{(p(x), x) : x \in \mathfrak{X}\}$;
 2  Initialize a forest $(V, E)$ whose vertices are initialized as $V \leftarrow R$, and with a (so
      far) empty set of edges, $E \leftarrow \emptyset$;
 3  Initialize an integer variable $y^* \leftarrow |\mathfrak{X}|$;
 4  while $|R| > 1$ do
 5      Let $(w, y)$, $(w', y')$ be the two smallest elements of $R$, by lexicographic order;
 6      Remove $(w, y)$ and $(w', y')$ from $R$ (but not from $V$);
 7      Update $y^* \leftarrow y^* + 1$;
 8      Add the new node $\gamma := (w + w', y^*)$ to both $R$ and $V$;
 9      Add labeled edges $(\gamma, (w, y), \texttt{label} = 0)$ and $(\gamma, (w', y'), \texttt{label} = 1)$ to $E$;
10  Interpret the resulting tree as a trie of the code book $C_{\text{Huffman}}$: for all $x \in \mathfrak{X}$,
      the code word $C_{\text{Huffman}}(x)$ is obtained by identifying the unique leaf node
      $(w, y) \in V$ with $y = x$, walking along the unique path from the root node to
      said leaf node, and concatenating the labels along the edges of this path.
```

In this problem, we set $B = 2$. (*Note:* to generalize Huffman coding to $B > 2$, you would have to pad the alphabet $\mathfrak{X}$ to a size that is a multiple of $(B-1)$ by "making up" additional symbols with zero probability; we won't concern ourselves with this additional complication here.)

(a) Consider the alphabet $\mathfrak{X} = \{1, 2, 3, 4, 5\}$ and the probabilistic model $p(1) = 0.3$, $p(2) = 0.28$, $p(3) = 0.12$, $p(4) = 0.1$, $p(5) = 0.2$. Execute the Huffman Coding algorithm manually on paper and write down a table for C_{Huffman}. Verify that you obtained a prefix code.

Solution: The following figure shows one possible solution. You may obtain different code words depending on how you label branches with zero and one bits, but your Huffman tree should have the same topology since there are no ties, and you should obtain identical code word lengths. (The Huffman tree in the figure always labels the branch whose child has lower weight with a zero bit, but you could label them arbitrarily as long as you use the same convention on both the encoder and the decoder side.)



$C_{\text{Huffman}}(x) =$ 11 10 011 010 00
$x =$ 1 2 3 4 5
$p(x) =$ 0.3 0.28 0.12 0.1 0.2
1 0 1 0
0.58 0.22
0
1
1
0.42
0
1

■

(b) Extend your table from part (a) with a Shannon Code C_{Shannon} for the same model p. To obtain C_{Shannon}, use Algorithm 1 with $\ell(x) := \lceil -\log_2 p(x) \rceil \; \forall x \in \mathfrak{X}$. Verify again that you obtained a prefix code.

Solution: The table below shows one possible result of Shannon Coding and compares it to our Huffman Code. You may obtain different code words for Shannon Coding if you swap the order of symbols with the same target code word length $\ell_{\text{Shannon}}(x) = \lceil -\log_2 p(x) \rceil$, i.e., if you swap the order of symbols 3 and 4 and/or the order of symbols 1 and 2.

x	$p(x)$	$\ell_{\text{Shannon}}(x) = \lceil -\log_2 p(x) \rceil$	$C_{\text{Shannon}}(x)$	$\tilde{C}_{\text{Shannon}}(x)$	$C_{\text{Huffman}}(x)$
4	0.1	$\lceil 3.32 \rceil = 4$	1111	1111	010
3	0.12	$\lceil 3.06 \rceil = 4$	1110	1110	011
5	0.2	$\lceil 2.32 \rceil = 3$	110	110	00
2	0.28	$\lceil 1.84 \rceil = 2$	10	10	10
1	0.3	$\lceil 1.74 \rceil = 2$	01	0	11
L			2.64	2.34	2.22

As can be seen in the table, the Shannon Code has one obvious inefficiency: the last bit of the code word “01” $= C_{\text{Shannon}}(1)$ can be dropped without violating the conditions of a prefix code. The table contains an extra column for an additional code book $\tilde{C}_{\text{Shannon}}(x)$ which drops this superfluous bit. ■

(c) Evaluate the entropy $H_2[p]$ and the expected code word lengths L_{Huffman} and L_{Shannon}. Verify that both codes satisfy Eq. 4 and that $L_{\text{Huffman}} \leq L_{\text{Shannon}}$ (for this particular example, you should find that L_{Huffman} is strictly smaller than L_{Shannon}, but equality is possible but for other probability distributions).

Solution: The entropy is $H_2[p] \approx 2.20$ and the expected code word lengths are shown in the above table. Both C_{Shannon} and C_{Huffman} have less than one bit per symbol overhead over the the entropy. In expectation, C_{Huffman} saves about 0.42 bits per symbol over C_{Shannon}, and still about 0.12 bits over the improved variant $\tilde{C}_{\text{Shannon}}$. As expected, none of the code books have an expected code word length that is smaller than the entropy. ■

(d) Implement the Huffman Coding algorithm in Python by filling in the blanks in the accompanying Jupyter notebook. You may want to refer back to the code examples for Problem 1.3 on last week's problem set.

Solution: See Jupyter notebook `problem-set-02-solutions.ipynb`. ■

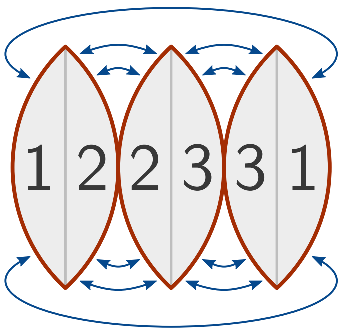
Problem 2.5: RIP, Simplified Game of Monopoly

Starting next week, we will discuss methods from probabilistic machine learning that will allow us to model complex data sources, such as natural language, images, or videos. We will see that many of the concepts that we illustrated so far with the help of toy-ish probabilistic models (like the "Simplified Game of Monopoly") will carry over to these more powerful and more realistic deep probabilistic models.

But before we put the Simplified Game of Monopoly to rest, here's an off-topic brain teaser for you: how would you actually build a 3-sided die as a physical object? What (3-dimensional) shape would it have?

Solution: There's a simple and a more involved solution. The simple solution is to use a regular six-sided die and to identify opposite faces with each other: when you draw the labels on the six sides, draw ⚀, ⚁, and ⚂ as usal. But then, instead of drawing ⚃, draw a second ⚀; instead of ⚄, draw a second ⚁; and instead of ⚅, draw a second ⚂.

Coming up with a shape that really only has three faces is more difficult. In three dimensions, this is only possible if the faces are curved. The figure on the right shows a blueprint for building a three-sided die whose shape roughly resembles that of a pointed pepper (German: "Spitzpaprika"). Cut out the shape along the curved red lines (ignore the straight vertical gray lines), then glue edges together as indicated by the blue arrows. The resulting three-dimensional shape will have three curved faces, three edges (at each edge, two of the three faces meet), and two corners (at both corners, all three faces meet). Each face shows two numbers, which label the *edges* on the two sides of the face.



When you throw this three-sided die, one of the three faces will end up touching the table and facing downwards, and the remaining *two* faces will be visible. The two visible faces share an edge whose label indicates the value of the throw. ■