Data Compression With and Without Deep Probabilistic Models

Lecture 2 (28 April 2022)

Recap from last lecture:

- source coding vs channel coding
- source-channel separation

- symbol codes:

"alphabet" (finite or countably infinite set)

$\hookrightarrow$ message $\underline{x} = (x_1, x_2, \ldots, x_k)$ with $x_i \in \mathfrak{X}$ $\forall i \in \{1, \ldots, k\}$

$\hookrightarrow$ code book $C : \mathfrak{X} \to \{0, 1, \ldots, B\}^*$ (usually: base $B=2$)

$\hookrightarrow$ code $C^* : \mathfrak{X}^* \to \{0, 1, \ldots, B\}^*$ with $C^*(\underline{x}) := C(x_1) \| C(x_2) \| \ldots \| C(x_k)$

concatenation

Recap from tutorial:

probabilistic model of the data source

length of code word $C(x)$ (in bits)

- Def. "expected code word length": $L_C := \mathbb{E}_p[\ell_C(x)] = \sum_{x \in \mathfrak{X}} p(x)\, \ell_C(x)$

- Def. "prefix free symbol code C" (or "prefix code" for short): no code word C(x) is the prefix of another code word C(x')

- Def. "uniquely decodable symbol code C": C* is injective

- prefix free ⇒ unique decodability; but inverse is not necessarily true

- Huffman coding: algorithm that takes a probabilistic model p (on a finite alphabet) and generates a prefix code that is "tailored" for this probabilistic model.

Today: Source Coding Theorem

Two fundamental truths about lossless compression ("good news and bad news"):

- bad news: Consider a data source that produces symbols with probability distribution p. Then, there is a fundamental lower bound H[p], and no uniquely decodable compression code can reach an expected code word length L that is lower than H[p].

- good news: For every data source, there exists a prefix-free (and thus uniqueley decodable) code (the so-called "Shannon code") whose expected code word length approaches the fundamental lower bound H[p] with an overhead of less than 1 bit per symbol.

- bonus: For finite alphabets, the Huffman coding algorithm always produces an optimal symbol code (i.e., a symbol code with lowest possible expected code word length L)

Kraft-McMillan Theorem

(a)

(Interpretation: we can't make code words arbitrarily short. If we shorten one code word by one bit, then we may have to make some other code word(s) longer or else our code can no longer be uniquely decodable)

(b)

Corollary:

⇒ When searching for an optimal symbol code, it suffices to consider only prefix codes.
(Actually, we don't really have to search directly for an optimal symbol code. It suffices to search for an optimal assignment of code word lengths l(x) that satisfy the Kraft inequality. Once we have that, we can construct a prefix code with these code word lengths, see below.)

Lemma:

Proof of the Kraft-McMillan Theorem

Proof of part (a):

Proof of part (b) of the Kraft-McMillan Theorem:

Constructive proof, i.e., we show existence of such a prefix code by providing an explicit algorithm that constructs it for any ℓ.

Algorithm (*):

- Input:

- Output:

- Steps:

Claim: The resulting code book C is prefix free. (Proof: Problem Set 2)

Example: Simplified game of Monopoly

x	$p(x)$					
2	$1/9$					
3	$2/9$					
4	$3/9 = 1/3$					
5	$2/9$					
6	$1/9$					

Check that Kraft inequality holds for ℓ:

Questions: (1) Can we efficiently find the optimal code word lengths l(x) that satisfy the Kraft McMillan inequalit and that lead to the lowest expected code word length L?

(2) Can we estimate the optimal expected code word length L without having to find the whole table of optimal code word lengths l(x)?

To address question (2), we use the following strategy:
(i) We derive a lower bound on L.
(ii) We show that there exists a valid assignment of code word lengths that approaches the lower bound with less than one bit of overhead.

⇒ l* minimizes the expected code word length for under the relaxed constraint. Thus, for any other $\ell : \mathcal{X} \to \mathbb{R}_{\geq 0}$ that satisfies the Kraft inequality, we have:

(ii) How closely can we approach this lower bound (taking into account that l(x) must be integer)?
→ Answer: within an overhead of less than 1 bit per symbol.

Proof:

Note: If we use these code word lengths l(x) and apply Algorithm (*), then the resulting prefix code C is called the "Shannon Code for the probability distribution p".

→ see code C in the "Simplified Game of Monopoly" example above;
more examples on Problem Set 2.

Huffman Coding

While Shannon coding is guaranteed to have less than one bit per symbol of overhead, it can still be suboptimal, i.e., there can still be a different uniquely decodable symbol code with a smaller expected code word length than the Shannon code.

We now want to find the optimal uniquely decodable symbol code for a given probability distribution p, i.e., the one with the lowest expected code word length L.

Reminder:
- The optimal symbol code still satisfies $H[p] \leq L < H[p] + 1$.
- There can be more than one optimal symbol codes for a given p;
- Among all optimal symbol codes for a given p, there is at least one prefix-free code.
  → Why?
- We define "optimality" here as minimizing the expected code word length. This is appropriate for many applications, but there are also use cases of data compression where one should optimize different metrics.
  → Examples:

The Huffman algorithm (for finite alphabets): see Problem Set 1

Claim: the Huffman algorithm constructs an optimal symbol code.

(i.e., no other uniquely decodable symbol code has a shorter expected code word length L than the Huffman code)

Complication: a Huffman code is not necessarily uniquely defined (in case of ties)

Example:

x =	"a"	"b"	"c"	"d"
p(x) =	$1/6$	$1/6$	$1/3$	$1/3$

Theorem 1: Even in presence of a tie in the Huffman algorithm, the expected code word length L of a Huffman code is well defined, i.e., it is independent of how we break the tie.

Proof:
- express L as the sum of the weights of all non-leaf nodes (see above)
- observe that sequence of node weights is independent of how we break ties.

Remark: Encoder & decoder still have to break ties in the same way so that they end up with the same code book. This can lead to subtle bugs due to implicit rounding operations with floating point numbers.

Theorem 2: The Huffman algorithm constructs an optimal symbol code.
More precisely: assume we have

Then:

Reminder: We may assume, without loss of generality, that C is a prefix-free code (due to the corollary to the Kraft-McMillan Theorem).

Lemma 1: Assume again (*), and let C be an optimal (w.r.t. p) prefix code; let's sort the symbols s.t.

We break ties by code word lengths (descendingly):

(if there are still ties after this, break them arbitrarily)

Then:

Proof of Lemma 1:

(i) by contradiction:

(ii) proof by contradiction, building on (i):

Lemma 2: Assume again (*), and let C be an optimal (w.r.t. p) prefix code. Then $\exists x, x' \in \mathcal{X}$ with $x \neq x'$ and:

(i)

(ii)

Proof of Lemma 2: Assume that such a pair does not exist. But, from Lemma 1, we know:

Claim: either C is not optimal because we can drop the last bit of C(x) and we'll still have a prefix free code, or there exists a different pair of symbols that satisfy both (i) and (ii)

Proof:

Recap:

Theorem 2: The Huffman algorithm constructs an optimal symbol code.
More precisely: assume we have

- finite alphabet $\mathcal{X}$ with $|\mathcal{X}| \geq 2$
- probability distribution $p: \mathcal{X} \to [0,1]$ with $p(x) > 0 \; \forall x \in \mathcal{X}$ $\Big\} \; (*)$

Then:

$\forall$ uniq. dec. sym. codes C on $\mathcal{X}$ that minimize the expected code word length L w.r.t. p: $\exists$ a Huffman code C_H with the same code word lengths, i.e., $|C(x)| = |C_H(x)| \; \forall x \in \mathcal{X}$.

Lemma 1: Assume again (*), and let C be an optimal (w.r.t. p) prefix code; let's sort the symbols s.t.

$p(x_1) \leq p(x_2) \leq p(x_3) \leq \dots$

We break ties by code word lengths (descendingly): $\ell(x_i) := |C(x_i)|$

if $p(x_i) = p(x_{i+1})$ then $\ell(x_i) \geq \ell(x_{i+1})$

(if there are still ties after this, break them arbitrarily)

Then: (i) $\ell(x_1) \geq \ell(x_2) \geq \ell(x_3) \geq \dots$

(ii) $\ell(x_1) \overset{\uparrow}{=} \ell(x_2)$

Lemma 2: Assume again (*), and let C be an optimal (w.r.t. p) prefix code. Then $\exists x, x' \in \mathcal{X}$ with $x \neq x'$ and:

(i) $\ell(x) = \ell(x') \geq \ell(\tilde{x}) \; \forall \tilde{x} \in \mathcal{X}$; and

(ii) $C(x)$ & $C(x')$ only differ in last bit

Proof of Theorem 2 (optimality of Huffman coding):

→ by induction over $|\mathcal{X}|$

- base case: $|\mathcal{X}| = 2$

- induction step: $|\mathcal{X}| > 2$ assuming that theorem holds for $\forall \; |\mathcal{X}'| = |\mathcal{X}| - 1$

↳ Claim: $\tilde{C}$ is an optimal prefix code on $\mathcal{X}$ (with respect to $\tilde{p}$)

↓

Proof: if it isn't an optimal prefix code then there exists a better prefix code $\tilde{\tilde{C}}$ on $\tilde{\mathcal{X}}$

Thus, $\tilde{C}$ is indeed an optimal prefix code on $\tilde{\mathcal{X}}$ (which has size $|\tilde{\mathcal{X}}| = |\mathcal{X}|-1$).

↳ Recall that x_1 and x_2 (which are "contracted" in the definition of C) are two symbols with lowest probability.

⇒ Running Huffman algorithm on $\mathcal{X}$also contracts contracts x_1 and x_2 in the first step. The remaining steps of the algorithm then construct a prefix code with the same code word lengths as $\tilde{C}$ on $\tilde{\mathcal{X}}$ by induction assumption.

Remarks and Outlook:

- Huffman coding is still widely used in practice (e.g., in the "deflate" compression method used in zip/gzip and for compressed HTTP streams, in PNG, in most JPEGs, ...)

- However, Huffman coding is only an optimal symbol code. On the new problem set, you will think about the limitations of symbol codes. In Lecture 4 on 12 May 2022, we will start discussing so-called stream codes, which outperform Huffman coding (especially in the regime of low entropy per symbol, which is relevant for modern machine learning based data compression methods).

- On the current problem set (discussed tomorrow), you are guided to an implementation of Huffman coding in real code. You will then use your implementation on next week's problem set. There, you'll combine it with a machine learning model that you'll train yourself, and you will implement a fully functioning (albeit ridiculously slow) deep learning based compression method for English text.