15:853: Algorithms in the Real World
Fall 2019

Lecture 11

October 10 2019

Lecturer: Rashmi Vinayak *Scribe: Pallavi Koppol*

11.1 Recap From Last Class

- **Model** generates probabilities, and **Coder** utilizes them
- **Probabilities** are related to **information**. The more you know, the less information your message will give.
- Increased skew (aided by **context**) in probabilities lowers **entropy**, and the lower the entropy, the better the compression.
- Average length l_a for **optimal prefix code** bounded by $H \leq l_a < H + 1$
  - For example, Huffman codes are optimal prefix codes
  - We can also utilize arithmetic codes to get a lower average length code. These codes are blended, meaning that codeword bits are shared between message.
- Two techniques for exploiting context include:
  - Transforming the data (good for skewing the probabilities – thereby lowering the entropy).
  - Using conditional probabilities
    - This is what we cover in this lecture.
    - At a high level, we use conditional probabilties to determine the likelihood of the next seen character given the context we have so far (based on all previous times we've seen said context). Context is useful, because we tend to see recurring patterns in the data, which we should leverage for effective compression.
- Integer codes involve transforming the data so that the new message is comprised only of integers.
- Run Length Coding involves specifying message value followed by the number of repeated values
- Move to Front Coding involves transforming the message sequence into a sequence of integers and then probability coding

11.2 Overview

In this lecture, we introduced and explained the following techniques and concepts:

- Residual Coding
- PPM: Prediction by Partial Matching
- Lempel-Ziv Algorithms
- Burrows-Wheeler

11.3 Residual Coding

- Typically used for message values that represent some sort of amplitude.
- The gist is to:
  - Guess next value based on current value (i.e. the current value is the context, as we discussed in the recap)
  - Output difference between guess and actual value
  - Use probability code on the output
- This is used in JPEG-Lossless. In particular, the probability code used on the output is a Golomb-code. A Golomb-code consists of two parts parameterized by some M: the quotient and remainder (of the division of input by M). This is similar to a Gamma code which, if you recall from the previous lecture, is a type of integer code that consists of a length and offset pair.
- No proofs or board-work here, but it was mentioned that uploaded supplemental notes would have more detail than the slides.

11.4 PPM: Prediction by Partial Matching

- Use k previous characters as context.
- Base probabilities on counts (e.g. if we see **th** 12 times followed by **e** 7 times, then $p(e|th) = \frac{7}{12}$).

11.4.1 Questions and Discussion

1. **Question.** Each context has its own probability distribution, which keeps changing. Is this a problem?

   Answer. This should *not* be a problem so long as the context precedes the character being coded. This is because all that matters is that the decoder understands the context.

2. **Question.** A dictionary within which to store contexts can grow prohibitively large. What can be done about this?

   Answer. We can keep the dictionary manageable by storing only small contexts k. Typically, we store only $k < 8$. Recall, we need to store contexts of size k and all sizes smaller than k.

3. **Question.** What do we do if we have not seen the context followed by the character before? We can't code 0 probabilities! E.g. we've seen "cod" but not "code". What do we do when "e" appears?

   Answer. This was alluded to in the answer to the previous question. *The key idea of PPM is to reduce context size if previous match has not been seen.* If a character has not been seen before with current context of size 3, try context of size 2 ("ode"), and then context of size 1 ("de"), and then no context ("e"). In other words, if we don't see the context we need, we search for a smaller context, and so on until we find a suitable partial context (hence: prediction by partial matching).

Context	Counts	Context	Counts	Context	Counts
Empty	A = 4	A	C = 3	AC	B = 1
	B = 2		$ = 1		C = 2
	C = 5	B	A = 2		$ = 2
	$ = 3		$ = 1	BA	C = 1
		C	A = 1		$ = 1
			B = 2	CA	C = 1
			C = 2		$ = 1
			$ = 3	CB	A = 2
					$ = 1
				CC	A = 1
					B = 1
					$ = 2

String = ACCBACCACBA k = 2

Figure 11.1. How do we code 'B' next? Table from Slide 16

4. **Question.** For input string "ACCBACCACBA, and $k = 2$, how do we code 'B' next (slide 16, Figure 11.1)?

Answer. We have $k = 2$, so the previous context of relevance is "BA" (e.g. the last two characters of the message so far). However, we do not have any information for a "B" following this context. So, we look for "A" as a context ($k = 1$). Again, we do not have any information to code "B". Finally, we look at B itself ($k = 0$). And find that its count is 2, while the total number of single character counts is 11. Therefore, we find the probability to be $\frac{2}{11}$.

5. **Question.** How do we tell the decoder to use a smaller context?

   Answer. We return an *escape message*. Each escape tells the decoder to reduce the size of the context by 1. The escape can be viewed as special character, but needs to be assigned a probability. Different variants of PPM use different heuristics for the probability. One option that works well in practice is to assign the count as the number of different characters seen (PPMC). We see an example of this on slide 20. Every time a new symbol is encountered for a given context, the count of \$ increases by one. For example, in the character counts column, the value of \$ is 3 (since once an A was newly encountered, once a B, once a C).

Context	Counts	Context	Counts	Context	Counts
Empty	A = 4	A	C = 3	AC	B = 1
	B = 2		\$ = 1		C = 2
	C = 5	B	A = 2		\$ = 2
	\$ = 3		\$ = 1	BA	C = 1
		C	A = 1		\$ = 1
			B = 2	CA	C = 1
			C = 2		\$ = 1
			\$ = 3	CB	A = 2
					\$ = 1
				CC	A = 1
					B = 1
					\$ = 2

String = ACCBACCACBA k = 2

Figure 11.2. How do we code 'A'? Table from Slide 22

6. **Question.** Do we always need multiple escapes when skipping multiple contexts? For string "ACCBACCACBA" and $k = 2$, how do we code 'A' (slide 22, Figure 11.2)?

   Answer. If context has not been seen before, automatically escape. We don't need multiple escapes because the decoder is aware of all smaller contexts. Take for example Slide 22. We see that the context BA gives us no information about A (it returns \$). When we look at the smaller context, A, we see that we have only ever seen a C following an A. However, we know that our new character is also not a C, since otherwise we would have used the information from the BA context. Effectively, we

can "remove" C from our previous contexts, because it's not an option any longer. So only one escape character suffices here!

So, to continue the example on Slide 22, we see that we then look for our individual character counts column. $ here should actually contain a count of 2, because we are discounting all traces of C. Therefore, for A, we find the probability $\frac{4}{4+2+2} = \frac{1}{2}$

7. **Question.** Which probability code to use and why?

   Answer. It is critical to use arithmetic codes, because the probabilities of characters to encode are high; arithmetic codes allow us to use less than one bit on average per character.

11.5 Lempel-Ziv Algorithms

- Still dictionary based, but instead of encoding one character at a time, it codes groups of characters at a time. This allows for utilizing spatial clustering for effective compression, but over the set of previously seen string matches.
- These algorithms work by:
  - Looking for longest match in the preceding text for the string starting at the current position
  - Outputting a code for that string
  - Moving past the match
  - ...and repeating
- In particular, we look at LZ77 (traditionally better than LZ78, but slower)
  - Worth noting that the gzip version of LZ77 is almost as good as LZ78. This is in part because it uses optimizations including huffman coding the positions, lengths, and chars, being non-greedy, and using a hash table to quickly access the dictionary. Also, it uses probability coding as a second pass (which initial algorithms did not)!
  - We briefly touched upon the fact that a sliding window LZ is asymptotically optimal.
  - Main differences between LZ77 and LZ78 include:
    - How the dictionary is stored (LZ78 is a trie)
    - How it is indexed (LZ78 indexes the nodes of the trie)
    - How it is extended (LZ78 only extends an existing entry by 1 character)
    - How items are removed
  - Further information on the details of these can be found in the "Compression notes" linked with the lecture from 10/1

11.5.1 Questions and Discussion

1. **Question.** Given | aac | aacab |, what is the output?

   Answer. Recall that output is of the form (p, l, c) where p is the position of the longest match that starts in the dictionary (relative to the cursor), l is the length of longest match, c is next char in buffer beyond longest match.

   Here, *aac* is in our dictionary, and *aacab* is in our look-ahead buffer. We see that the longest match starting in the dictionary is *aac*, which has position 3 relative to the cursor. Furthermore, we see that the length of the longest match is actually 4, not 3. This is because if the length of the longest match exceeds the matching context from the dictionary, it begins to wrap around. However, we know that b is not seen in our dictionary, and therefore our matching must end there. We can succinctly write this output as $(3, 4, b)$.

2. **Question.** What if $l > p$?

   Answer. This answer was alluded to in the previous question. In this case, we simply copy p from left to write until we reach the end of the match.

11.6 Burrows-Wheeler

- The main idea is to cluster similar characters together in order to increase coding efficacy, while being fully recoverable due to the systematic transform required.
- Transform-based coding technique
- Currently near-best algorithm for text compression
- Breaks file into fixed-size blocks and encodes each block separately by:
  - Creating a full context for each character using wraparound (an example of this is in the answer to Question 3 in the following section).
  - Reverse lexicographic sorting each character by its full context. This is known as the "block-sorting transform"
  - Then, we use move-to-front transform on the sorted characters.

11.6.1 Questions and Discussion

1. **Question.** Why is the output easier to compress?

   Answer. It is easier to compress because there are more repeated characters in a row (e.g. we cluster similar characters together).

2. **Question.** Why not just sort?

   Answer. We cannot sort directly, because there would be no way to recover the original message.

3. **Proof Sketch.** *Theorem:* After sorting, equal valued characters appear in the same order in the output column as in the last column of the sorted context.

   Answer. This follows from the way we sort. We rotate the letters in the message in order to generate our contexts, and the last column of the context is comprised of the last characters of all these rotations. We know that this last column will be sorted depending on everything that precedes it. This is also precisely the case for the output. The example given is:

Context	Output
dedec_3	o_4
coded_1	e_2
decod_5	e_6
odede_2	c_3
ecode_6	d_1
edeco_4	d_5

   We can see here that d_1 is preceded by "code" and d_5 by "deco" in the context and, in the case of the output, d_1 is mapped to the context "ecode_6" and d_5 to "edeco_4". Therefore, whatever sorting is applied to d_1 and d_5 on the context side will hold for d_1 and d_5 on the output side because they are sorted according to the same prefix criteria (with the addition of one extra character which is precisely the same in both cases, and therefore does not affect the lexicographic sort).

   We did not fully review this proof in class, and the main emphasis was on this being a function of the sorting procedure. It was also mentioned that only the second to last column has the ability to invert the code in this way.