Coding Markov Sources

(Chapter 6)

Chapter Outline

After this chapter you should:

- have understood the Burrows-Wheeler Transform (BWT), to the point that:
  - you are able to execute by hand a (compression or decompression) algorithm on a given input, including the forward and inverse BWT.
  - you are able to describe the (compression or decompression) algorithm in your own words, in a reasonably precise manner.
  - you are able to understand and explain, why compression algorithms based on the BWT perform well. Your explanation and understanding should convey the intuition in a reasonably precise manner.

Outline

This chapter is divided into three sections, each roughly taking one lecture.

- Contexts and the BWT.
- Inverting the BWT.
- Coding the output of the BWT and why it works.

Context

- The *context* of a symbol position is the symbols which come before it.
    - Knowing context allows us to make more accurate (higher-probability) guesses of the next symbol $\Rightarrow$ lower entropy, better compression.
    - E.g. `stati?`. Given the context `stati` we can make better guesses of the next symbol.
- Markov sources model "context".
    - LZW/LZ77 can compress output of Markov sources.
- An approach that is more direct in its approach to "context" is based on the *Burrows-Wheeler Transform (BWT)*.
- Context of a symbol position can also be defined to be the symbols that come *after* that position. Equally effective.
    - E.g. `?tatic`. Given the context `tatic` we can make better guesses of the preceding symbol.
    - BWT rearranges symbols, grouping according to their context (this defn).

Lexicographic order

Used to compare two strings (definition assumes both strings are equally long). Let the strings be $X = x_1 \dots x_k$ and $Y = y_1 \dots y_k$.

- $X = Y$ only when all corresponding symbols are equal, i.e. $x_1 = y_1, x_2 = y_2, \dots, x_k = y_k$.
- $X < Y$ only when $X \neq Y$ and if i denotes the leftmost position where X and Y differ, then $x_i < y_i$.
- $X > Y$ otherwise.

Thus, if the alphabet is $\{a, b\}$ and $a < b$ then:
$abbaab < abbbba$

Computing the BWT

Let $S = s_1 s_2 \ldots s_k$ be a string of k symbols. BWT(S) is computed as follows:

1. Create a $k \times k$ matrix A:
   - i-th row of A is the string S 'rotated' by $i - 1$ positions.
2. Sort rows of A in lexicographic order and call the sorted matrix A'. Suppose the original string S is now row number i of A'. **Output** this number i.
3. **Output**, row-by-row, the symbols in the last column of A'.

Next: example on input `good,_jolly_good`

Example: All Rotations of Input

```
g o o d , _ j o l l y _ g o o d <- original string
o o d , _ j o l l y _ g o o d g
o d , _ j o l l y _ g o o d g o <- rotate by 2
d , _ j o l l y _ g o o d g o o
, _ j o l l y _ g o o d g o o d
_ j o l l y _ g o o d g o o d ,
j o l l y _ g o o d g o o d , _
o l l y _ g o o d g o o d , _ j
l l y _ g o o d g o o d , _ j o
l y _ g o o d g o o d , _ j o l <- rotate by 9
y _ g o o d g o o d , _ j o l l
_ g o o d g o o d , _ j o l l y
g o o d g o o d , _ j o l l y _
o o d g o o d , _ j o l l y _ g
o d g o o d , _ j o l l y _ g o
d g o o d , _ j o l l y _ g o o
```

Example: All Sorted Rotations, BWT Output

```
_ g o o d g o o d , _ j o l l y
_ j o l l y _ g o o d g o o d ,
, _ j o l l y _ g o o d g o o d
d , _ j o l l y _ g o o d g o o
d g o o d , _ j o l l y _ g o o
g o o d , _ j o l l y _ g o o d  <- original string in
g o o d g o o d , _ j o l l y _     row number 6
j o l l y _ g o o d g o o d , _
l l y _ g o o d g o o d , _ j o     OUTPUT:
l y _ g o o d g o o d , _ j o l     y,dood__oloojggl (last col)
o d , _ j o l l y _ g o o d g o     and '6' (row index)
o d g o o d , _ j o l l y _ g o
o l l y _ g o o d g o o d , _ j
o o d , _ j o l l y _ g o o d g
o o d g o o d , _ j o l l y _ g
y _ g o o d g o o d , _ j o l l
```

BWT output

- The BWT does not compress, but it does group symbols according to "context".
- All symbols appearing in "context" `o` in the example input

```
g o o d , _ j o l l y _ g o o d
```

- ...are consecutive in the output:

```
F                               L

o d , _ j o l l y _ g o o d g o
o d g o o d , _ j o l l y _ g o
o l l y _ g o o d g o o d , _ j
o o d , _ j o l l y _ g o o d g
o o d g o o d , _ j o l l y _ g
```

- To compress an input s, we will code BWT(s) losslessly. To decompress, we first recover BWT(s), and then *invert* the BWT: given BWT(s), figure out s.

Inverting BWT: Intuition

```
F       L
? ...   y
? ...   ,
? ...   d
? ...   o
? ...   o
? ...   d
? ...   _
? ...   _
? ...   o
? ...   l
? ...   o
? ...   o
? ...   j
? ...   g
? ...   g
? ...   l
```

- We first do inversion “by hand”.
- Output of BWT is last column of rotated matrix.
- Contains same symbols as input string.
-
    -
    -

Inverting BWT: Intuition

F		L
_	...	y
_	...	,
,	...	d
d	...	o
d	...	o
g	...	d
g	...	_
j	...	_
l	...	o
l	...	l
o	...	o
o	...	o
o	...	j
o	...	g
o	...	g
y	...	l

- We first do inversion "by hand".
- Output of BWT is last column of rotated matrix.
- Contains same symbols as input string.
- To get first column of rotated matrix put last column in sorted order.
    - Now we have *pairs* of consecutive input symbols: `_g`, `_j`, . . . , `od`, `od`, `ol`, `oo`, `oo`, `y_`.
    - Symbol in L comes before symbol in F.

Inverting BWT: Intuition

Each pair appears once at the start of a row *in sorted order*.

```
L   F       L        L   F         L        L   F         L
y | _ ... y          y | _ ? ... y          y | _ g ... y
, | _ ... ,          , | _ ? ... ,          , | _ j ... ,
d | , ... d          d | , ? ... d          d | , _ ... d
o | d ... o          o | d ? ... o          o | d , ... o
o | d ... o          o | d ? ... o          o | d g ... o
d | g ... d          d | g ? ... d          d | g o ... d
_ | g ... _          _ | g ? ... _          _ | g o ... _
_ | j ... _          _ | j ? ... _          _ | j o ... _
o | l ... o          o | l ? ... o          o | l l ... o
l | l ... l          l | l ? ... l          l | l y ... l
o | o ... o          o | o d ... o          o | o d ... o
o | o ... o          o | o d ... o          o | o d ... o
j | o ... j          j | o l ... j          j | o l ... j
g | o ... g          g | o o ... g          g | o o ... g
g | o ... g          g | o o ... g          g | o o ... g
l | y ... l          l | y ? ... l          l | y _ ... l
```

Inverting BWT: Inutuition

- From pairs, we get triples, then quadruples etc.
- Continue to get the full matrix of sorted rotations
    - **Note:** from pairs (or triples) you can't get the original string back directly. E.g. the strings *aaabaaabaa* and *aabaaaabaa* have exactly the same set of triples of consecutive symbols. You need to carry this through to the end (get quadruples, quintuples etc.).
- However, this algorithm is way too slow. It takes $O(k^2)$ time, which is completely infeasible for even inputs of size 1MB.
- We now give a faster algorithm, the one used in practice.

Inverting BWT: F2L mapping

We first need a couple of properties of the BWT.

Claim

Let x, y be two strings and let c be a symbol. If $cx < cy$, then $xc < yc$, and vice versa.

- *Example*: Consider `mother` and `mottle` where:

$$c = \mathtt{m}, x = \mathtt{other}, y = \mathtt{ottle}$$

  Since `mother` < `mottle`, `otherm` < `ottlem` as claimed.
- Proof is obvious. If $cx < cy$ then $x < y$. If $x < y$ adding any symbol at the end makes no difference. The proof the other way is the same.

F2L mapping

- From now on we assume that all rotations of the input string are distinct.
  - I.e. we assume that strings don't look like *aaaaaa* (all six rotations are the same) or *ababab* (only two distinct rotations).
  - **Note:** we can always add an extra symbol to ensure this e.g. *aaaaaa*† and *ababab*† each have seven distinct rotations.

We now argue:

Lemma

If all rotations of the input string are distinct, any two equal symbols appear in the same relative order in F as they do in L.

Lemma

If all rotations of the input string are distinct, any two equal symbols appear in the same relative order in L as they do in F.

Proof.

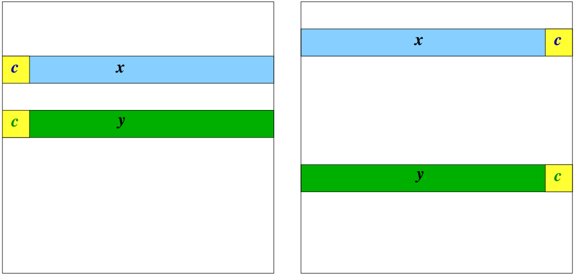
- Suppose that two distinct rows begin with the same symbol c, one say cx and one cy.
- If cx occurs above cy then $cx < cy$.
- Since $cx < cy$, by the previous lemma $xc < yc$.
- Hence, the row containing xc is above the row containing yc.
- Hence, the c that is followed by x is above the c followed by y in both L and F. Other cases similar.

□

Lemma

If all rotations of the input string are distinct, any two equal symbols appear in the same relative order in L as they do in F.

(Proof by picture)



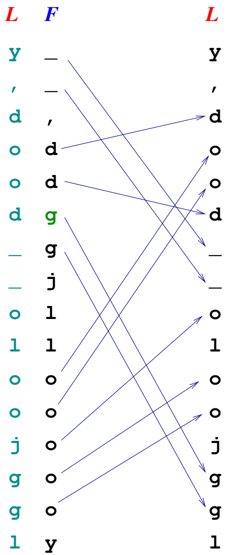
Lemma

If all rotations of the input string are distinct, any two equal symbols appear in the same relative order in L as they do in F.

```
F                             L
_ g o o d g o o d , _ j o l l y
_ j o l l y _ g o o d g o o d ,
, _ j o l l y _ g o o d g o o d
d , _ j o l l y _ g o o d g o o
d g o o d , _ j o l l y _ g o o
g o o d , _ j o l l y _ g o o d
g o o d g o o d , _ j o l l y _
j o l l y _ g o o d g o o d , _
l l y _ g o o d g o o d , _ j o
l y _ g o o d g o o d , _ j o l
o d , _ j o l l y _ g o o d g o
o d g o o d , _ j o l l y _ g o
o l l y _ g o o d g o o d , _ j
o o d , _ j o l l y _ g o o d g
o o d g o o d , _ j o l l y _ g
y _ g o o d g o o d , _ j o l l
```

[PROOF BY EXAMPLE]

F2L mapping (2)



- Lemma means: for any c, the i-th c in F is the same as the i-th c in L.
- Create an array F2L such that F2L[i] is the position where the i'th symbol in F can be found in L. (E.g. F2L[6] $=$ 14)
- Decode as follows:

```
1. Let i be the row containing
   the original string.
2. do k times
     begin
       output F[i]
       i := F2L[i]
     end
```

Outline

- BWT and contexts (review).
- Effect of BWT
- MTF and coding the output of BWT.
- Practical considerations.

Context Revisited

Context of a symbol position can also be defined to be the symbols that come *after* that position.

- E.g. ?`tatic`. Given the context `tatic` we can make better guesses of the preceding symbol.

BWT organises text into parts with similar contexts.

Effect of BWT

L	F
y	_
,	_
d	,
o	d
o	d
d	g
_	g
_	j
o	l
l	l
o	o
o	o
j	o
g	o
g	o
l	y

L	F	
y	_	g
,	_	j
d	,	_
o	d	,
o	d	g
d	g	o
_	g	o
_	j	o
o	l	l
l	l	y
o	o	d
o	o	d
j	o	l
g	o	o
g	o	o
l	y	_

- BWT groups symbols in input according to their context.
- Symbols with context _ are adjacent in BWT:
  `good,_jolly_good`
- Those with context `od` also adjacent in BWT:
  `good,_jolly_good`
- For any given context, all symbols in that context are adjacent in BWT.

Effect of BWT: Large Input

```
L|F
          ⋮
t|hat acts like this: it alloca...
t|hat buffer to the constructor...
t|hat corrupted the heap, or wo...
W|hat goes up must come down, s...
t|hat happens, it isn't likely ...
w|hat if you want to dynamicall...
t|hat indicates an error, allow...
t|hat looks like this: a large ...
t|hat looks something like this...
t|hat looks something like this...
t|hat once I detect the mangled...
          ⋮
```

Note: In each context, a small set of symbols [Nelson, *DDJ* '96].

Move-to-front (MTF) Coding

Aim: To give small numbers to symbols that appear frequently in a context.

- PROBLEM: We don't really know exactly where a "context" begins or ends.

MTF algorithm

- Initialise list L to contain all symbols in the input (in no particular order). Then:
  1. Read the next symbol s.
  2. Search for s in L, say s is the j-th symbol from the front of L.
  3. **Output** $j - 1$, and **move** s to the front of L.

MTF coder: Example

Input: `y,dood␣␣oloojggl`.
Intial L: (‘␣’, ‘,’, ‘d’, ‘g’, ‘j’, ‘l’, ‘o’, ‘y’).

- Read `y`

  Output 7 as `y` is the eighth symbol in the list.

  $L \leftarrow$ (‘y’, ‘␣’, ‘,’, ‘d’, ‘g’, ‘j’, ‘l’, ‘o’).

- Read `,`

  Output 2 as `,` is the third symbol in the list.

  $L \leftarrow$ (‘,’, ‘y’, ‘␣’, ‘d’, ‘g’, ‘j’, ‘l’, ‘o’).

Proceeding like this, the output is: $7, 2, 3, 7, 0, 1, 4, 0, 2, 7, 1, 0, 7, 7, 0, 3$.

Properties of MTF

- Popular symbols in current context at front of L.
- Code a symbol by its position in L.

▷ popular symbols in current context get small integers.

- Each context has its own set of popular symbols.
- When that context is reached, it's own popular symbols get small integers.
  - Output of MTF after BWT has *very* strong bias towards small integers.
  - Does not need to "decide" "when" a new context begins: adaptation to change of context is smooth.

Finishing Up

- Overview of `bzip`:

$$s \to \mathbf{BWT}(s) \to \mathbf{MTF}(\mathbf{BWT}(s)) \xrightarrow{\text{Huffman}} \textbf{output.}$$

- It is lossless, since we know how to decode Huffman and MTF (why?)

`bzip` Implementation Notes

▷ Sorting all rotations of s is slow: $O(n \lg n)$ on average, but inverting BWT is in fact $O(n)$ time!

▷ Coding slower than decoding.

▷ bzip divides file into blocks of 64KB and compresses each block separately.

E-lecture

The e-lecture contains the following:

- Comparisons among compression algorithms.
- Applications of LZW/LZ77.

This ends PART I of the course (coding symbolic data).