Compressed Sparse Row (CSR) Format

The Compressed Sparse Row (CSR) format is used to reduce storage requirements. For example, the training data for Reuters Corpus Version 1 (RCV1) consists of 20,242 documents (rows) with 47,236 distinct words (columns); but relatively few words appear in any given document. Out of the 956,151,112 (20,242 x 47,236) cells that might hold non-zero values indicating the presence of a word in a document, only 1,498,952 cells [0.16% (less than 1%)] actually hold non-zero values.

Suppose we have the following data:

$$X = \begin{bmatrix} 0 & 66 & 0 & 0 & 44 \\ 0 & 0 & 55 & 0 & 0 \\ 0 & 0 & 0 & 0 & 99 \\ 44 & 88 & 0 & 0 & 0 \\ 0 & 0 & 0 & 11 & 0 \end{bmatrix} \qquad y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

This is a text file containing a sparse format version of this data:

```
C:\Temp> type sparse.dat
0 2:66 5:44
0 3:55
1 5:99
1 1:44 2:88
0 4:11
C:\Temp>
```

Here's the data read by library(e1071)'s read.matrix.csr() method:

```
> library(e1071)
> input = read.matrix.csr("C:/Temp/sparse.dat")
> input
$x
An object of class "matrix.csr"
Slot "ra":
[1] 66 44 55 99 44 88 11

Slot "ja":
[1] 2 5 3 5 1 2 4

Slot "ia":
[1] 1 3 4 5 7 8

Slot "dimension":
[1] 5 5


$y
[1] 0 0 1 1 0
Levels: 0 1
```

There is ...

- one entry per non-zero cell value for the input$x@ra slot, containing actual cell values
- one entry per non-zero cell value for the input$x@ja slot, containing column index values
- one entry per row (plus one) for the input$x@ia slot, containing indices for the "ra" and "ja" slots [note: the difference between an element of "ia" and the next element of "ia" indicates how many non-zero cells occur on the corresponding row]

Rather than storing 25 numbers for the input$x matrix, we only store 20 numbers for the slots: 7 for input$x@ra, 7 for input$x@ja, and 6 for input$x@ia. For this trivial example, storage space requirements have been reduced by 20% [(20 - 25) / 25].

```
> library(SparseM)

Attaching package: 'SparseM'

The following object is masked from 'package:base':

    backsolve

> as.matrix(input$x)
     [,1] [,2] [,3] [,4] [,5]
[1,]    0   66    0    0   44
[2,]    0    0   55    0    0
[3,]    0    0    0    0   99
[4,]   44   88    0    0    0
[5,]    0    0    0   11    0
```