LaF

A package for processing large ASCII files

D.J. van der Laan

2011-11-06

1 Introduction

`LaF` is a R package for reading large ASCII files. It offers some functionality that is missing from the regular R routines for processing ASCII files. First of all, it is optimised for speed. Especially reading fixed width files is very slow with the regular R routine `read.fwf`. However, since it is optimised for speed some of the flexibility of the regular routines is lost. Seconly, it offers random access: only those rows and columns are read that are needed. With the regular routines one always has to read all columns.

The problem with big files is that they do not fit into memory. One could consider this even to be the definition of 'big'. To comfortably work with data in R the data set needs to fit multiple ($\sim$3) times into memory. There are roughly two methods for working with data that doesn't fit into memory. The first is to read the data in blocks that do fit into memory process each of these block and merge the results. More on this can be found in section 3. The second is to read only that part of the data into memory which is needed for the calculation at hand hoping that that subset does fit into memory. For example to crosstabulate two variabeles one only needs these two variables. As most datasets contain dozens of variables this can easily reduce the memory needed for the operation by a factor of ten. More on this in section 4.

Why ASCII? Why not use a binary format like `ff` and similar packages do? True, binary storage allows for much faster access since the conversion from ASCII to binary format is not needed and data can often be stored much more compact. The main reason is portability. Almost every program designed for data precessing can read ASCII files. And even if one wants to use a package like `ff`, the source files are often ASCII files and first need to be converted to `ff` format. `LaF` can also speed up this last process.

2 Opening a file

2.1 Column types

LaF currently supports the following column types

double Fields containing floating point numbers. Scientific notation (e.g. 1.9E-16) is not supported. The character used for the decimal mark can be specified using the `dec` option of the functions used to open files.

integer Fields containing positive or negative integer numbers (e.g. 42, -100)

categorical Categorical fields are treated as character fields except that a table is built mapping all observed values to integers. A factor vector is returned in R when this type is used. The levels can be read and set using the `levels` method.

string Character fields such as postcodes, identification numbers.

As of version 0.5 of `LaF` it is also possible to set levels of non categorical columns using the `levels` method. For more information see paragraph 5.

2.2 Fixed width files

In fixed width files columns are defined by character positions in the files. For example, the first seven characters of each line belong to the first column, the next two characters belong to the second, etc. Each line therefore has the same number of characters. This is also a disadvantage of the format. If there is a column with variable string lenghts, the column has to be wide enough to accomodate the widest field. The main advantage of the format is that reading in large files (and especially random access) can be very efficient as the positions of rows and columns can be calculated.

Fixed width files can be openen using the function `laf_open_fwf`. In order to open a file the following options can be specified:

filename name of the file to be opened.

column_types Character vector containing the types of data in each of the columns. Valid types are: double, integer, categorical and string.

column_widths Numeric vector containing the width in number of character of each of the columns.

column_names (optional) Optional character vector containing the names of the columns. The default names are 'V1', 'V2', etc.

dec (optional) Optional character specifying the decimal mark. The default value is '.'.

trim (optional) Optional logical value specifying whether of not character strings should be trimmed left and right from white space. This applies to both columns of type 'string' as 'categorical'. For fixed width files the default is true (trim white space).

Suppose the following data is stored in the file 'file.fwf' in the current working directory (showing only the first five lines):

```
 54346F1030RR 66  539.17
210239M9034NP 35 1227.04
 24952M6029NS 73 2329.88
652946M9230VC 78  206.66
867177M5699CO 37   40.84
```

Then this file can be openened using the following command:

```
> dat <- laf_open_fwf(filename="file.fwf",
+     column_types=c("integer", "categorical",
+         "string", "integer", "double"),
+     column_names=c("id", "gender", "postcode", "age", "income"),
+     column_widths=c(6, 1, 6, 3, 8))
```

`dat` is now a `laf` object. Data can be extracted from this object using the commands described in sections 3 and 4. For example, to read all data in the file one can use the following command:

```
> alldata <- dat[ , ]
```

2.3 Comma separated files

In comma seperated files each line contains a row of data, the columns are seperated using a seperator character which is usually a comma although other seperator characters are also used (e.g. the ';' is often used in Europe where the comma is often used as the decimal seperator). It is a often used format. The disadvantage compared to the fixed width format is that the positions of columns and rows in the file can not be calculated. Therefore, a program reading a comma seperated file has to scan through the entire file to find a certain row or column making random access much slower than with fixed width files.

A comma seperated file for the `LaF` package has to observe the following rules someof which slightly deviate from the 'official' rules of comma seperated files:

- The first row can not contain the column names. These should be specified using the option `column_names` of the function `laf_open_csv`. The first line in the file is treated as the first data row and the columns in this line should be of the correct type.
- Quotes are treated slightly different from the way they are normally treated in csv files. Only double quotes are accepted. Everything inside double quotes is considered part of the field except newline characters and double quotes. Double quotes in text fields are therefore not possible. Below are a few examples of how quotes are interpreted:
  - `12345` = `12345`
  - `"12345"` = `12345`
  - `"123"45` = `12345`
  - `"123""45"` = `123"45"`
  - `"123\n45"` = ERROR
  - `12"345"` = `12"345"`
- Each line in the file should contain exactly one row of data. Normally line breaks should be possible inside quoted columns. In order to keep the code as fast as possible, this is not the case in the `LaF` package.

Comma seperated files can be opened using the function `laf_open_csv`. This function accepts the following arguments:

filename name of the file to be opened.

column_types Character vector containing the types of data in each of the columns. Valid types are: double, integer, categorical and string.

column_names (optional) Optional character vector containing the names of the columns. The default names are 'V1', 'V2', etc.

sep (optional) Optional character specifying seperator mark used between the columns. The default value is ','.

dec (optional) Optional character specifying the decimal mark. The default value is '.'.

trim (optional) Optional logical value specifying whether of not character strings should be trimmed left and right from white space. This applies to both columns of type 'string' as 'categorical'. For comma separated files files the default is false (do not trim white space).

skip (optional) Optional numeric value specifying the number of lines at the beginning of the file that should be skipped before starting to read data. This can be used, for example, to skip the header as headers are not supported: the user is required to specify the types and names of the columns.

As of version 0.5 of the `LaF` package, there is also the `detect_dm_csv` routine, which can automatically detect column types. See paragraph 2.4 for more information on how to use data models to open files.

Suppose the following data is stored in the file 'file.csv' in the current working directory (showing only the first five lines):

```
54346,F,1030RR,66,539.170000
210239,M,9034NP,35,1227.040000
24952,M,6029NS,73,2329.880000
652946,M,9230VC,78,206.660000
867177,M,5699CO,37,40.840000
```

Then this file can be openened using the following command:

```
> dat <- laf_open_csv(filename="file.csv",
+     column_types=c("integer", "categorical",
+         "string", "integer", "double"),
+     column_names=c("id", "gender", "postcode", "age", "income"))
```

`dat` is now a `laf` object. Data can be extracted from this object using the commands described in sections 3 and 4. For example, to read all data in the file one can use the following command:

```
> alldata <- dat[ , ]
```

2.4 Opening using data models

As of version 0.5 `LaF` has the ability to store all of the arguments needed by `laf_open_fwf` and `laf_open_csv` in so called data models. These data models can be written to and read from files using the functions `write_dm` and `read_dm` respectively. `write_dm` accepts either a data model or a `laf` object as its input. To write the data model of the data set from the previous section to file:

```
> write_dm(dat, "model.yaml")
```

The data model is written in the well documented and readable YAML format:

```
type: csv
filename: file.csv
```

```
sep: ','
dec: .
skip: 0
trim: no
columns:
- name: id
  type: integer
- name: gender
  type: categorical
- name: postcode
  type: string
- name: age
  type: integer
- name: income
  type: double
```

The format probably speaks for itself. It is also probable to manually write these files and read them using `read_dm`. To open a file using a data model the function `laf_open` can be used:

```
> dat <- laf_open(read_dm("model.yaml"))
```

Data models can also be generated from CSV-files and Blaise data models using the routines `detect_dm_csv` and `read_dm_blaise`. See the documentation of these routines for more information.

3 Blockwise processing

Blockwise processing of a file usually has the following structure:

1. Go to the beginning of the file
2. Read a block of data
3. Perform calculations on this block perhaps using results from the previous block.
4. Store results
5. Repeat 2–4 until all data has been processed.
6. If necessary combine the results of all the blocks.

In order to go to a specific position in the file `LaF` offers two methods: `begin` and `goto`. The first method simply goes to the beginning of the file while the second goes to the specified line. Assume, a `laf` object named `dat` has been created (see section 2 for this). The only argument needed by `begin` is the `laf` object:

```
> begin(dat)
```

For `goto` also the line number needs to be specified. The following command sets the filepointer at the beginning of line 1000. The next call to `next_block` (see below) will therefore return as first row the data belonging in line 1000 of the file.

```
> goto(dat, 1000)
```

Blocks of data can be read using `next_block`. The first argument needs to be the reference to the file (the `laf` object); other arguments are optional. By default all columns and 5000 lines are read:

```
> d <- next_block(dat)
> nrow(d)

[1] 5000
```

The number of lines can be specified using the `nrows` argument and the columns that should be read can be specified using the `columns` argument. The following command reads 100 lines and the first and third column.

```
> d <- next_block(dat, columns=c(1,3), nrows=100)
> dim(d)

[1] 100   2
```

If possible the use of the `columns` argument is advised. This can significantly speed up the processing of the file. First of all, the amount of data that needs to be transfered to R is much smaller. Second, the strings in the unread columns do not need to be converted to numerical values.

When the end of the file is reached `next_block` returns a `data.frame` with zero rows. This can be used to detect the end of file. The following example shows how `begin` and can be used to calculate the number of elements equal to 2 in the second column.

```
> n <- 0
> begin(dat)
> while (TRUE) {
+     d <- next_block(dat, 2)
+     n <- n + sum(d$gender == 'M')
+     if (nrow(d) == 0) break;
+ }
> print(n)

[1] 4987
```

Since processing a file in this way is such a common task, the method `process_blocks` has been defined that automates this and is faster. This method accepts as its first argument a `laf` object. The second argument should be the function that should be applied to each of the blocks. This function should accept as its first argument the data blocks. The last time the function is called it receives a `data.frame` with zero rows. This can be used to do some end calculations. The second argument of the function is the result of the previous function call. The first time the function is called the second argument had the value `NULL`. This can be used to perform initialisation. Additional arguments to `process_blocks` are passed on to the function. The previous example can be translated into:

```
> count <- function(d, prev) {
+   if (is.null(prev)) prev <- 0
+   return(prev + sum(d$gender == 'M'))
+ }
> (n <- process_blocks(dat, count))

[1] 4987
```

Using `process_blocks` is faster than using `next_block` repeatedly since the `data.frame` containing the data that is read in, is destroyed and created every iteration, while in `process_blocks` this `data.frame` is created only once.

Below is an example that calculates the average of the third column of the file and illustrates initialisation and finilisation (note this is not how you will want to calculate the average over a column in a large file). Since only the third column of the file is needed for this calculation, the `columns` option is used which makes the calculation much faster.

```
> ave <- function(d, prev) {
+   # initialisation
+   if (is.null(prev)) {
+     prev <- c(sum=0, n=0)
+   }
+   # finilisation
+   if (nrow(d) == 0) {
+     return(as.numeric(prev[1]/prev[2]))
+   }
+   result <- prev + c(sum(d$income), nrow(d))
+   return(result)
+ }
> (n <- process_blocks(dat, ave, columns=5))

[1] 1016.662
```

4 Selecting subsets

An other common way of handling large files is to only read in the data that is needed for the operation at hand. This is feasible when such a subset of the data does fit into memory. For this, selections can be performed on a `laf` object using the same methods one would use for a regular `data.frame`. The code below shows several different examples:

```
> # select the first 10 rows
> result <- dat[1:10, ]
> # select the second column
> result <- dat[ , 2]
> # select the first 10 rows and the second column
> result <- dat[1:10, 2]
```

Indexing a `laf` object always results in a `data.frame`. For example, the second and last example would have resulted in a vector when applied to a `data.frame`, while in the examples above a `data.frame` with one column is returned.

Using the `$` and [[operators columns can be selected from the `laf` object. The result is an object of type `laf_column` which is a subclass of `laf`. It is a `laf` object with a field containing the column number. To get the data inside these columns indexing can be used as is shown in the following examples. In the first example the records are selected from the file for which the age is higher than 65:

```
> result <- dat[dat$age[] > 65, ]
```

The same can be done using the column number

```
> result <- dat[dat[[4]][] > 65, ]
```

or

```
> result <- dat[dat[ , 4] > 65, ]
```

or

```
> result <- dat[dat[ , "age"] > 65, ]
```

5 Setting levels of columns

It is possible to set the levels of categorical and non-categorical columns. Since the file is read only, it is not possible to renumber the columns as would happen if we would change a column in a `data.frame` to `factor`. Therefore, we need to specify both the levels and the corresponding labels as a `data.frame`. For example, to change the 'age' column to a factor:

```
> levels(dat)[["age"]] <- data.frame(levels=0:100, labels=paste(0:100, "years"))
> dat$age[1:10]

 [1] 66 years 35 years 73 years 78 years 37 years
 [6] 19 years 79 years 78 years 72 years 7 years
101 Levels: 0 years 1 years 2 years ... 100 years
```

These levels are also written to file when writing a data model to file using `write_dm` and read in by `read_dm`. You can therefore also specify the levels of a column in the data model.

```
type: csv
filename: file.csv
sep: ','
dec: .
skip: 0
trim: no
columns:
- name: id
  type: integer
- name: gender
  type: categorical
- name: postcode
  type: string
- name: age
  type: integer
  labels:
  - level: 0
    label: 0 years
  - level: 1
    label: 1 years
  - level: 2
    label: 2 years
  - level: 3
    label: 3 years
  - level: 4
    label: 4 years
  - level: 5
    label: 5 years
  - level: 6
...
```

6 Calculating column statistics

Using `process_blocks` one can calculate all kinds of summary statistics for columns. However, as some summary statistics are very common, these have been implemented in the package. The available methods are:

`colsum`	Calculate column sums
`colmean`	Calculate column means
`colfreq`	Calculate frequency tables of columns
`colrange`	Calculate the maximum and minimum value of columns
`colnmissing`	Calculate the number of missing values in columns

All methods accept as first argument either a `laf` or `laf_column` object. In case of a `laf` object the second argument should be a vector of column numbers for which the statistics should be calculated. For a `laf_column` this is not necessary. For example, to calculate the average age the following options are available:

```
> (m1 <- colmean(dat, columns=4))

    age
55.0967

> (m1 <- colmean(dat$age))

    age
55.0967
```

Most methods also accept an `na.rm` argument, which ignores, as one would expect, missing values when calculating the statistics. The method `colnmissing` does not have this argument which would be meaningless. `colfreq` has the argument `useNA` which can take one the values 'ifany', 'always' or 'no'.