Mineração de Dados Aplicada

awk

Loïc Cerf
September, 4th 2019
DCC – ICEx – UFMG
Outline for the next sessions

Today  

awk’s basics;

23/09  Oral presentation of your data and awk’s array;

25/09  A few words about efficiency.
Outline for the next sessions

Today  awk’s basics;
23/09  Oral presentation of your data and awk’s array;
25/09  A few words about efficiency.
Outline for the next sessions

Today  awk’s basics;
23/09 Oral presentation of your data and awk’s array;
25/09 A few words about efficiency.

Date for the exam on the POSIX text-processing commands?
Outline for this session

1. Correction of last week exercise
2. Computational capabilities in the text-processing flow
3. Associative arrays and string processing
Correction of last week exercise

Outline

1. Correction of last week exercise
2. Computational capabilities in the text-processing flow
3. Associative arrays and string processing
Correction of last week exercise

Downloading the distribution pages

Listing the distributions using `ls */*`

```
ls */* | cut -d / -f 2 | grep -v INDEX
| sort -uo distros
```
Correction of last week exercise

Downloading the distribution pages

Listing the distributions using `ls */*`

`ls */* | cut -d / -f 2 | grep -v INDEX | sort -uo distros`

Listing the distributions using `ls */`

`ls */ | grep -ve INDEX -e / -e ’^$’ | sort -uo distros`
Correction of last week exercise

Downloading the distribution pages

Listing the distributions using `ls */*

```
ls */* | cut -d / -f 2 | grep -v INDEX
| sort -uo distros
```

Listing the distributions using `ls */

```
ls */ | grep -ve INDEX -e / -e ’^$’
| sort -uo distros
```

Downloading the distribution pages

```
wget -B http://distrowatch.com/ -i distros -P
../dw_pages
```
The origins of the distributions

---

**With sed**

```
sed -n '/origin=/ s/.*origin=\((.*)\)#.*/1/p' ../dw_pages/* | paste distros -
```

---

Loïc Cerf
Correction of last week exercise

The origins of the distributions

**With sed**

```
sed -n 's/.*/origin=\(.\)*#/.*//1/p' ../dw_pages/* | paste distros -
```

**With sed**

```
sed -n '/origin=/ s/.*/origin=\(.\)*#/.*//1/p' ../dw_pages/* | paste distros -
```
Correction of last week exercise

The origins of the distributions

**With sed (1’00s)**

```bash
sed -n 's/.*origin=\((.*)\)#.*/\1/p' ../dw_pages/* | paste distros -
```

**With sed (0.24s)**

```bash
sed -n '/origin=/ s/.*origin=\((.*)\)#.*/\1/p' ../dw_pages/* | paste distros -
```
Correction of last week exercise

The origins of the distributions

**With sed (1'00s)**

```bash
sed -n 's/.*origin=\((.*\)\)#.*/\1/p' ../dw_pages/* | paste distros -
```

**With sed (0.24s)**

```bash
sed -n '/origin=/ s/.*origin=\((.*\)\)#.*/\1/p' ../dw_pages/* | paste distros -
```

**Without sed (0.06s)**

```bash
grep -o 'origin=[^#]*' ../dw_pages/* | cut -d = -f 2 | paste distros -
```
Correction of last week exercise

The origins of the distributions

**With sed (1'00s)**

```
sed -n 's/.*/origin=\((.*)\)#.*#/1/p' ../dw_pages/*
| paste distros -
```

**With sed (0.24s)**

```
sed -n '/origin=/ s/.*/origin=\((.*)\)#.*#/1/p'
../dw_pages/* | paste distros -
```

**Without sed (0.04s)**

```
grep -m 1 -o 'origin=[^#]*' ../dw_pages/*
| cut -d = -f 2 | paste distros -
```
Outline

1. Correction of last week exercise

2. Computational capabilities in the text-processing flow

3. Associative arrays and string processing
sed is suited to select lines and transform them. To actually consider these lines as *records with fields* and do some *numerical computation* on them, awk is much more appropriate.
Computational capabilities in the text-processing flow

Adding computational capabilities

**sed** is suited to select lines and transform them. To actually consider these lines as *records with fields* and do some *numerical computation* on them, **awk** is much more appropriate.

**awk** is Turing complete and is closer than **sed** to general purpose programming languages (*variables, associative arrays, Boolean operators, functions, conditional statements, loops, etc.*).
Computational capabilities in the text-processing flow

Adding computational capabilities

**sed** is suited to select lines and transform them. To actually consider these lines as *records with fields* and do some *numerical computation* on them, **awk** is much more appropriate.

**awk** is Turing complete and is closer than **sed** to general purpose programming languages (variables, associative arrays, Boolean operators, functions, conditional statements, loops, etc.).

**awk** has inspired **perl**, which is general purpose and very expressive... but also messy, slow and harder to learn.
Computational capabilities in the text-processing flow

Learning awk

1. The following slides;
2. Tutorials on the Web;
3. 'info awk';
Computational capabilities in the text-processing flow

**Learning awk**

1. The following slides;
2. Tutorials on the Web;
3. `info awk`;

And cheat sheets on the Web...
awk, which is specifically designed to process *structured* text, is a mix of sed, the Shell and the C programming language.

- **sed**: Condition (to select records) followed by an action (on these records) and sed-like substitutions.
- **Shell**: The fields of a record are $1$, $2$, etc. and awk’s actions can use pipes and redirections.
- **C**: Boolean operators, conditional statements, loops, numerical and string-processing functions à la C.
A few options

The most useful options of `awk` are `-f` (like with `sed`, it introduces a script), `-F` followed by the `regexp` separating the fields (one or more spaces-tabs is the default) and `-v` followed by the definition of a variable (very useful when calling `awk` in a Shell script).
Computational capabilities in the text-processing flow

awk program structure

Letting aside the possibility to define functions (with the function keyword), an awk program is a sequence of condition-action statements. A space separates the condition from the action.
Computational capabilities in the text-processing flow

**awk program structure**

Letting aside the possibility to define functions (with the `function` keyword), an *awk* program is a sequence of condition-action statements. A space separates the condition from the action.

Like in *sed*, the condition can be a regexp between `'/` and intervals are possible. Nevertheless, any test that could be in an action can be, as well, a condition. Two “special” conditions are particularly useful: `BEGIN` matches the beginning of the input (before the first record); `END` its end (after the last record). By default (no condition), every record is matched.
Letting aside the possibility to define functions (with the `function` keyword), an `awk` program is a sequence of condition-action statements. A space separates the condition from the action.

Like in `sed`, the condition can be a regexp between `'/` and intervals are possible. Nevertheless, any test that could be in an action can be, as well, a condition. Two “special” conditions are particularly useful: BEGIN matches the beginning of the input (before the first record); END its end (after the last record). By default (no condition), every record is matched.

The action is between braces. It is executed on the records satisfying the condition. By default (a condition but no action), every record satisfying the condition is printed.
A simple example


```java
{ sex = "m" }

# Female names ending in a/e/i/y.
/^.*[aeiy]/ { sex = "f" }

# Allison (and variations)
/^All?[iy](ss?)|z)on$/ { sex = "f" }

# Pete, Serge, Shane, ...
/^[^BG](e[rst]|ha)[^il]*e$/ { sex = "m" }

# Angel, Gail, Isabel, ...
/^ADF[GI][^r]*([bg]e[lr]|il|wn)$/ { sex = "f" }

```

(a total of 41 regexps)
Computational capabilities in the text-processing flow

Some predefined variables

By default records are lines and fields are space-separated strings. Redefining the RS/FS (resp. ORS/OFS) variables changes the record/field separators at input (resp. output). Input separators can be any regexp. When printing, ',' inserts the output field separator. A space concatenates.
Computational capabilities in the text-processing flow

Some predefined variables

By default records are lines and fields are space-separated strings. Redefining the $RS/FS$ (resp. $ORS/OFS$) variables changes the record/field separators at input (resp. output). Input separators can be any regexp. When printing, ',' inserts the output field separator. A space concatenates.

ARGC is the number of files in arguments, ARGV their names, FILENAME that of the currently processed file, NR the current record number (and FNR is reset for each file). That record is $0$. It has $NF$ fields: $1$, $2$, $\ldots$, $NF$ (literally, i.e., the last field can be processed without knowing its number).
Computational capabilities in the text-processing flow

Instructions à la C

**Numerical computation** Numerical constants/variables can be compared, operated, incremented, decremented and more complex functions (exponentiation, logarithm, trigonometry, etc.) are available as well.

**Boolean operations** &&, ||, !, etc.

**Control statements** if, for, while, break, continue, etc.
Computational capabilities in the text-processing flow

Instructions à la C

Numerical computation Numerical constants/variables can be compared, operated, incremented, decremented and more complex functions (exponentiation, logarithm, trigonometry, etc.) are available as well.

Boolean operations &&, ||, !, etc.

Control statements if, for, while, break, continue, etc.

Like in C, ; separates the instructions (not required if on different lines), blocks are between { and }, strings are between "s, = attributes a value (on the right) to a variable (on the left), etc.
Computational capabilities in the text-processing flow

**Instructions à la C**

**Numerical computation**  Numerical constants/variables can be compared, operated, incremented, decremented and more complex functions (exponentiation, logarithm, trigonometry, etc.) are available as well.

**Boolean operations**  &&, ||, !, etc.

**Control statements**  if, for, while, break, continue, etc.

Like in C, ; separates the instructions (not required if on different lines), blocks are between { and }, strings are between "s, = attributes a value (on the right) to a variable (on the left), etc.

Unlike in C, variables need not be declared and are dynamically initialized at "" for a string or 0 for a number.
Computational capabilities in the text-processing flow

Getting the data

```
$ wget dcc.ufmg.br/~lcerf/data.tar.xz -O - | tar -xJ
```
Exercises

From brasileirão/retweets.txt:

1. Print the fields in the opposite order (user first) and transform every number of retweets $t$ into $\frac{1}{1+e^{0.2(20-t)}}$.

2. In these logs, how many times was milton_neves retweeted?

3. The log starts at the end of 2010 and ends at the beginning of 2011. In 2010, every week number was moved back by one week in the past. Correct this error.

4. List the distinct users who were retweeted at least 20 times when writing about a specific team during a specific week (you are allowed to process awk’s output with sort).
Summary of awk’s basics

An awk script:

- is a sequence of condition-action statements;
- processes records having fields;
- can use variables that are implicitly initialized;
- can do numerical computation;
- can use control statements with a C-like syntax.
Outline

1. Correction of last week exercise

2. Computational capabilities in the text-processing flow

3. Associative arrays and string processing
Array [key] is the value that array associates with key. There is no simpler collection in awk.

An array can be sorted w.r.t. the keys (resp. the values) with the built-in function `asort` (resp. `asorti`). The information which is not sorted is lost and the keys become the sequential integers, starting at 1. The length is returned.
Associative arrays and string processing

**Associative arrays**

`array[key]` is the value that array associates with key. There is no simpler collection in `awk`.

`variable in array` checks whether `variable` is a key in `array`, except within a `for` statement (i.e., `for (variable in array)`) where `variable` successively is every key in `array`.
Associative arrays

array[key] is the value that array associates with key. There is no simpler collection in awk.

variable in array checks whether variable is a key in array, except within a for statement (i.e., for (variable in array)) where variable successively is every key in array.

An array can be sorted w.r.t. the keys (resp. the values) with the built-in function asorti (resp. asort). The information which is not sorted is lost and the keys become the sequential integers, starting at 1. The length is returned.
Some simple built-in string functions

Some classics from C: `printf` and `sprintf`.
Some simple built-in string functions

Some classics from C: printf and sprintf.

Some simple predefined functions: length (of $0$ if no argument is given), tolower, toupper, index (which starts at 1; 0 if not found) of the first occurrence of a substring, substr (using positions, which start at 1), etc.
Some simple built-in string functions

Some classics from C: printf and sprintf.

Some simple predefined functions: length (of $0$ if no argument is given), tolower, toupper, index (which starts at 1; 0 if not found) of the first occurrence of a substring, substr (using positions, which start at 1), etc.

Some more complex ones: split (from a string to an array of strings with any regexp as a delimiter), match (returns where a regexp is matched and optionally instantiates an array with the strings matching parts between parentheses). The operator ~ (resp. !~) checks whether part of the string (on the left) verifies (resp. does not verify) the regexp (on the right, between '/').
sed’s substitution power!

- `gensub(/regexp/, replacement, flag[, str])` performs like `sed ’s/regexp/replacement/flag’` on `str` (by default `$0`).
- `sub(/regexp/, replacement[, str])` performs like `gensub(/regexp/, replacement, 1[, str])`.
- `gsub(/regexp/, replacement[, str])` performs like `gensub(/regexp/, replacement, "g"[, str])`.

The parentheses (to reuse part of the match in the replacement) are not “backslashed”. The numbered reused parts need a double backslash. `gensub` does not alter `str` and returns the modified string. `sub` and `gsub` modify (or not) `str` and return 1 (or 0).
Exercises

From brasileirão/retweets.txt:

1. List the distinct users who were retweeted at least 20 times when writing about a specific team during a specific week (you are not allowed to use sort!).

2. Compute the total numbers of retweets per team.
Exercises

Given a list of keywords and a text, use awk to count the occurrences of each keyword in the text:

1. from the output of `tr -s [:space:][:punct:] \n < text | grep -ixf keywords;`

2. from scratch and to be piped into `grep -wf keywords;`

3. only using awk.
Exercises for next Wednesday

The teams have very different total numbers of retweets. A normalization w.r.t. the “popularity” is wished. Let $\mu_{\text{team}}$ the total number of retweets for a specific team and $\mu$ their arithmetic mean over all teams. Multiply every number of retweets by $\frac{\mu}{\mu_{\text{team}}}$, where team is the team the number of retweets relates to. You can proceed in two steps:

1. From retweets.txt, list every team id (200, 201, ..., 219) along with the coefficient $\frac{\mu}{\mu_{\text{team}}}$ in a file named coeff.

2. Write an awk program that takes both coeff and retweets.txt as arguments and applies the coefficients.

Hint: understanding the solutions to the exercises on the two previous slides helps.
The TapTempo C++ program measures the BPM of a music whose beat is tapped on a key. The \( k \text{th} \) time it is tapped:

- nothing is output if \( k = 1 \);
- “\( x \text{ bpm} \)” is output.

\( x \) is the integer part of the BPM computed from:

- the first and the \( k \text{th} \) times the key was tapped, if \( k \leq 5 \);
- the \((k - 4)\text{th}\) and the \( k \text{th} \) times the key was tapped, otherwise.
Challenge

TapTempo with a 131-character Perl command

perl -M'Time::HiRes qw/gettimeofday/' -ne
'@$t[0]+=gettimeofday; shift @t if @t>5; printf
"%3.0f %bpm",60*$#t/($t[-1]-$t[0]) if $#t'
Associative arrays and string processing

Challenge

**TapTempo with a 131-character Perl command**

```
perl -M'Time::HiRes qw/gettimeofday/' -ne
'@t+=gettimeofday; shift @t if @t>5; printf
"%3.0f %bpm",60*$#t/($t[-1]-$t[0]) if $#t'
```

**Challenge (updated)**

A student found a 104-character solution, with the exact same output: awk '{system("date +\%s\%N")}' | awk '...'. Will you do better? A 101-character solution exists.
Reward

The first student sending me the shortest correct solution I receive until September 29th at 23h59 earns:

- 1 bonus point if the whole solution has 107 characters or more;
- 2 bonus point if the whole solution has 106 characters;
- 3 bonus point if the whole solution has 105 characters or less.
© 2012–2019 Loïc Cerf

These slides are licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.