The Joy of Text

Andrew Robinson

CEBRA / School of Mathematics & Statistics
University of Melbourne

February 19, 2016

Centre of Excellence for
Biosecurity Risk Analysis

WOMBAT

"Making Data Analysis Easier"

~~“Making Data Analysis Easier”~~

Outline

Red Letters, and Where They Are Going

CEBRA 1301A1 — Spatial Analysis of Intercepted Mail

International mail is monitored by DDU, X-ray, and manual inspection in Gateway Facilities.

- Delivery address is recorded for all articles intercepted with BRM.
- Addresses can be geolocated to census region.

CEBRA is using data-mining tools to identify patterns.

- Spatial analysis — spatial patterns in intercepted goods?
- Statistical analysis — any correlation with census-measured characteristics at the ABS statistical unit level?

But Addresses are Hand Coded.

. . . and they are ugly . . .

```
addresses <- read.csv("../sources/sampleAddresses.csv")

as.character(addresses[1:10, "rawAddress"])

##  [1] "115 STANHOPE ROAD"   "P O BOX 1232"        "PO BOX 1232"
##  [4] "10 ADAMS RD"         "19/83A LINCOLN ROAD" "P.O. BOX 1232"
##  [7] "P.O. BOX  1232"      "115  STANHOPE ROAD"  "10 ADAMS ROAD"
## [10] "115 STANHOPE RD"

grep("1232", addresses$rawAddress, value = TRUE)

## [1] "P O BOX 1232"   "PO BOX 1232"    "P.O. BOX 1232"  "P.O. BOX  1232"

grep("stanhope", addresses$rawAddress, ignore.case = TRUE, value = TRUE)

## [1] "115 STANHOPE ROAD"  "115  STANHOPE ROAD" "115 STANHOPE RD"
## [4] "115  STANHOPE RD"
```

What to do?

The Pleasure of the Text

An Instructive Example from Forestry

```
str(ugly)

## 'data.frame': 5 obs. of  3 variables:
##  $ Plot.ID: Factor w/ 3 levels "1_A","1_B","2_A": 1 1 2 3 3
##  $ Species: Factor w/ 4 levels "F","GF","GF var. Bupkiss",..: 2 4 1 2 3
##  $ Dbh    : Factor w/ 5 levels "-","18.8","20.0",..: 2 5 3 4 1
```

In order to make the names easier to work with and easier to read, within the bounds of taste, we write

```
(names(ugly) <- tolower(names(ugly)))

## [1] "plot.id" "species" "dbh"
```

Notice that `names` is being used to both get (RHS) and set (LHS) the names of the object, and that parentheses print the object.

Also, note that `toupper` plays an intuitively obvious role.

Missing Value Flags

The data have more than one missing flag.

```
is.na(ugly$dbh[ugly$dbh %in% c("NA","-")]) <- TRUE
ugly$dbh <- as.numeric(as.character(ugly$dbh))
ugly$dbh

## [1] 18.8   NA 20.0 25.8   NA
```

Note the glorious many-to-many match provided by `%in%`.

NB: the help file for `factor` points out that
`as.numeric(levels(f))[f]`
. . . is slightly more efficient than . . .
`as.numeric(as.character(f))`

Grep: for the Finding of Things

Next, we may be interested in locating the fir trees in the dataset.

```
grep("F", ugly$species) # ... or ...

## [1] 1 3 4 5

table(grep("F", ugly$species, value = TRUE))

##
##                F               GF GF var. Bupkiss
##                1                2               1
```

We may have some data entry problems: probably the `F` is meant to be a `GF`. We now make that call, explicitly documented in the code, so that it can be audited.

We use `sub` and `gsub` to replace one character string with another. But first ...

REGular EXpressions

Regular expressions (regex) are a family of mark-up dialects that provide a convenient and flexible language for expressing a pattern to use to match character strings.[1]

Several R functions accept regular expressions as arguments.

Regular expressions use familiar symbols in a specific way to unambiguously describe text that has specific properties. For example,

[1] `regexbuddy` etc. can help composition; thanks to Klaus Ackermann.

REGular EXPressions: FOr EXAmple

To get strings that start with `F`, prepend `^`.

```r
grep("^F", c("F","FG","GF","FF"), value = TRUE)

## [1] "F"  "FG" "FF"
```

To get only those strings that end with `F`, append `$`.

```r
grep("F$", c("F","FG","GF","FF"), value = TRUE)

## [1] "F"  "GF" "FF"
```

Use both for strings that start and end with the same `F`.

```r
grep("^F$", c("F","FG","GF","FF"), value = TRUE)

## [1] "F"
```

Process

Now, let's fix our little `F` problem in a considered way. We (i) make a rule, (ii) check the rule, (iii) apply the rule, (iv) audit the rule.

```
F.to.GF <- grep("^F$", ugly$species)
sort(table(ugly$species[F.to.GF]))


##
##                GF  GF var. Bupkiss                WS                 F
##                 0                0                 0                 1

ugly$species[F.to.GF] <- "GF"
ugly$species <- factor(ugly$species)
table(ugly$species)


##
##                GF  GF var. Bupkiss                WS
##                 3                1                 1
```

Ok, ok, in this case we could also just have done this:

```
ugly$species[ugly$species == "F"] <- "GF"
```

Wildcards

We use `.` to denote any character, and the following to denote counts:
`*` denotes zero or more,
`+` denotes one or more,
`?` denotes zero or one, and
`{n}` denotes n (can also do a range).

Here are all the strings that begin and end with distinct `F`.

```
grep("^F.*F$", c("F","FG","GF","FF","FaFa","FaaF","Fa aF"), value = TRUE)

## [1] "FF"    "FaaF"  "Fa aF"
```

NB: `.*` means zero or more characters that match the `.`, rather than one or more repeats of a character that matches the `.`

What if we want to be less flexible?

A choice between collections of characters is denoted by `or`: `|`.

```
grep("gray|grey", c("gray","grey","groy","red"), value = TRUE)

## [1] "gray" "grey"
```

Square brackets denote a set from which a single character must be selected.

```
grep("gr[ae]y", c("gray","grey","groy","red"), value = TRUE)

## [1] "gray" "grey"
```

The square brackets also admit a range.

```
grep("gr[a-z]y", c("gray","grey","groy","groovy"), value = TRUE)

## [1] "gray" "grey" "groy"

grep("gr[A-Z]y", c("gray","grey","groy","groovy"), value = TRUE)

## character(0)

grep("gr[A-z]y", c("gray","grey","groy","groovy"), value = TRUE)

## [1] "gray" "grey" "groy"

grep("gr[1-9]y", c("gray","grey","groy","groovy"), value = TRUE)

## character(0)

grep("gr[a-z]*y", c("gray","grey","groy","groovy"), value = TRUE)

## [1] "gray"   "grey"   "groy"   "groovy"
```

Tools of Greater Delicacy

More specialized markups are available.

`\b` flags the start of a word. (NB: double the escape for R.)

```
grep("road", c("broadway","broad road"), value = TRUE)

## [1] "broadway"   "broad road"

grep("\\b(road)", c("broadway","broad road"), value = TRUE)

## [1] "broad road"
```

`\s` is multiple spaces
`\n` is newline
`^` in a list indicates negation

`[[:alpha:]]` is any alphabet character, where supported. [2]

[2] NB: `[A-z]` may fail for non-English alphabets; thanks for this tip, Thomas Lumley.

Back-casting

We can refer back to groups, denoted by parentheses.

```
varieties.regex <- "(^[A-Z]+) +(var|sensu)(.*$)"
```

Our regex has three portions, each of which can be referred to.

```
sort(table(grep(varieties.regex, ugly$species, value = TRUE)))

## GF var. Bupkiss
##               1
```

```
(ugly$species <- gsub(varieties.regex, "\\1", ugly$species))

## [1] "GF" "WS" "GF" "GF" "GF"
```

NB: works within expressions. Here are pairs of letters.

```
grep("[a-z]*([a-z])\\1[a-z]*", c("broom", "bromo"), value = TRUE)

## [1] "broom"
```

Efficient Conversion

Run the regex across the levels instead of the variable.

```
(absurdly.large <- factor(c("A","B","B","see","D")))

## [1] A   B   B   see D
## Levels: A B D see

levels(absurdly.large) <- gsub("see", "C", levels(absurdly.large))
absurdly.large

## [1] A B B C D
## Levels: A B D C
```

Surgery

Finally, the plot and subplot identifiers have been combined into a single character string. We would like to separate them.

```
(ugly$plot <- substr(ugly$plot.id, 1, 1))

## [1] "1" "1" "1" "2" "2"

(ugly$subplot <- substr(ugly$plot.id, 3, 3))

## [1] "A" "A" "B" "A" "A"
```

Surgery

But sometimes the labels are not the same length.

```
pieces <- strsplit(x = as.character(ugly$plot.id), split = "_")

(ugly$plot <- sapply(pieces, function(x) x[1]))

## [1] "1" "1" "1" "2" "2"

(ugly$subplot <- sapply(pieces, function(x) x[2]))

## [1] "A" "A" "B" "A" "A"
```

Escape the wild cards in order to split on them.

```
strsplit(x = "my.test", split = "\\.")

## [[1]]
## [1] "my"   "test"
```

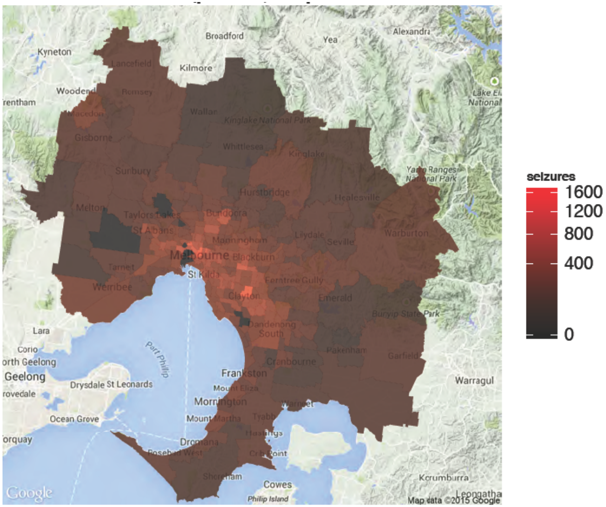
Cleaning in Action

DAWR wrote `cleanAddress`,
a function that takes care of business.

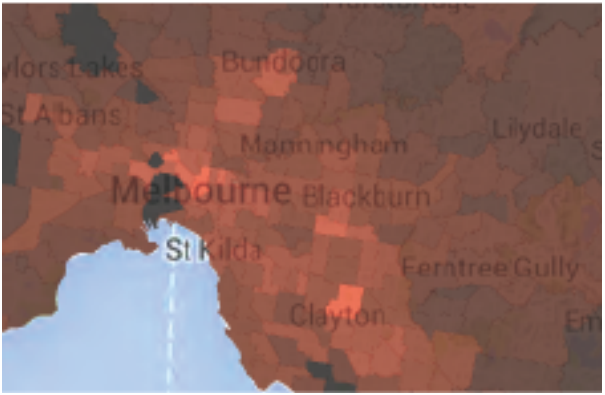
```
cbind(as.character(addresses[,"rawAddress"]),
      cleanAddress(addresses[,"rawAddress"]))[1:10,]


##       [,1]                    [,2]
##  [1,] "115 STANHOPE ROAD"   "115 STANHOPE RD"
##  [2,] "P O BOX 1232"        "PO BOX 1232"
##  [3,] "PO BOX 1232"         "PO BOX 1232"
##  [4,] "10 ADAMS RD"         "10 ADAMS RD"
##  [5,] "19/83A LINCOLN ROAD" "19/83A LINCOLN RD"
##  [6,] "P.O. BOX 1232"       "PO BOX 1232"
##  [7,] "P.O. BOX  1232"      "PO BOX 1232"
##  [8,] "115  STANHOPE ROAD"  "115 STANHOPE RD"
##  [9,] "10 ADAMS ROAD"       "10 ADAMS RD"
## [10,] "115 STANHOPE RD"     "115 STANHOPE RD"
```

Clean addresses can be geocoded.

2008 Seizures per 100,000 people (SA2)



Zoom



Bundoora
St Albans
Manningham
Lilydale
Melbourne
Blackburn
St Kilda
Ferntree Gully
Clayton

Distance in Text-Space: `adist`

Biodiversity Measured as Number of Species

Victorian State Forest Inventory

- About 300
  - large (0.04 ha) plots
  - small (0.01 ha) plots
  - sets of quadrats ($12 \times 1\text{m}^2$)
- 30,000 biota
- 1309 unique species (before cleaning!)

Resources: a list of about 10,000 species names — a dictionary.

Generalized Levenshtein Distance

Defined as: the smallest number of additions / substitutions / subtractions that it takes to get from A to B.

```
adist("Tursday", "Tuesday")

##      [,1]
## [1,]    1

adist("Tursday", "Thursday")

##      [,1]
## [1,]    1
```

Available in pattern matching via `agrep`.

```
agrep("Turkday", c("Tuesday","Thursday"), max.distance = 1, value = TRUE)

## character(0)

agrep("Turkday", c("Tuesday","Thursday"), max.distance = 2, value = TRUE)

## [1] "Tuesday"  "Thursday"
```

Speedy Resolution

Strategy

- Work with 1309 unique values.
- Convert to all lower case.
- Identify absolute matches using `%in%` (see previously).
- Remove cruft (informalities, formalities, etc.)
- Loop: for each of several distances (low to high),
  - `agrep` to identify contenders within the given distance, increase distance until at least one match is found.
  - Use `adist` to find the closest match.
  - Review.
  - Hand edit as needed.

Speedy resolution

Outcome:

- 924 unique species.
- Entire cleaning system scripted, documented, and auditable.
- Very happy collaborator.

Pre-Cleaning: SED

History of Horticultural Exports

Many plant products are inspected by DAWR before export.

But: single consignments can contain different products.

What combinations predominate? Are they region-specific?

Drop into `system`. NB: you may need extra software tools.

(Assume a `data.folder` and a `target.folder`.)

```
system(paste( "ls -lh", data.folder, "| cut -f4- -d ' '"),
       intern = TRUE)

## [1] ""
## [2] "andrewpr  staff    26M Jan 30 12:30 HortExports1.csv"
## [3] "andrewpr  staff    26M Jan 30 12:30 HortExports2.csv"
## [4] "andrewpr  staff    26M Jan 30 12:30 HortExports3.csv"
## [5] "andrewpr  staff    30M Jan 30 12:30 HortExports4.csv"
```

Exploring the Challenge

How many Mb total?

```
sum(as.numeric(system(paste( "ls -lh",
                             data.folder, "| cut -c 34-35 "),
                      intern = TRUE))[-1])
```

```
## [1] 108
```

How about a quick line count?

```
(system(paste( "cd ", data.folder, "; ls | xargs wc -l"),
        intern = TRUE))
```

```
## [1] "  300001 HortExports1.csv" "  300001 HortExports2.csv"
## [3] "  300001 HortExports3.csv" "  358964 HortExports4.csv"
## [5] " 1258967 total"
```

The files are csv, but `read.csv` fails because of infelicities.

SED: the Stream EDitor

A simple invocation:

```
$ cat inFile | sed 'pattern' > outFile
```

We focus on SED's very useful substitution tool:

```
s/target/replacement/options
```

- SED is fast.
- SED is light.
- SED speaks REGEX.
- SED won't overwrite.

Practise Safe SED

Print all the matches before changing them.

```
sed -n '/match/ p'
```

E.g.,

```
strsplit(system("cat ~/Desktop/test.csv | sed -n '/ORANGES,NAVAL/ p'",
                intern = TRUE), ",")

## [[1]]
##  [1] "8675309"                 "XX"
##  [3] "2014-05-28 00:00:00.000" "2014-06-05 00:00:00.000"
##  [5] "HOHOHO"                  "NULL"
##  [7] "ORA"                     "ORANGES"
##  [9] "NAVAL"                   "23940.000"
## [11] "KGM"
```

Using SED from R

```r
hort.files <- list.files(data.folder, full.names = FALSE,
                             pattern = "\\.csv$")
sed.file <- function(fileName) {
    sed.string <-
        paste("cat ", data.folder, "/", fileName,
              " | sed 's/ALFALFA,SNOW/ALFALFA SNOW/g'",
              " | sed 's/SULTANAS,RAISINS/SULTANAS RAISINS/g'",
              " | sed 's/FLAXSEED, SAFFLOWER/FLAXSEED SAFFLOWER/g'",
              " > ", target.folder, "/", fileName, sep = "")
    system(sed.string)
    return(TRUE)
}

system.time(sapply(hort.files, sed.file))
```

```
##    user  system elapsed
##   5.540   0.531   2.394
```

To chain sed commands — semi-colon, or the -e flag.

But . . .

"Some people, when confronted with a Unix problem, think 'I know, I'll use sed.' Now they have two problems." — Jamie Zawinski 1992.

Thanks!