Wide-to-tall Data Reshaping Using Regular Expressions and the nc Package
by Toby Dylan Hocking

Abstract Regular expressions are powerful tools for extracting tables from non-tabular text data. Capturing regular expressions that describe information to extract from column names can be especially useful when reshaping a data table from wide (few rows with many regularly named columns) to tall (fewer columns with more rows). We present the R package nc (short for named capture), which provides functions for wide-to-tall data reshaping using regular expressions. We describe the main new ideas of nc, and provide detailed comparisons with related R packages (stats, utils, data.table, tidyr, tidyfast, tidyfst, reshape2, cdata).

Introduction
Regular expressions are powerful tools for text processing that are available in many programming languages, including R. A regular expression pattern or regex defines a set of matches in a subject string. For some example subjects, consider the column names of the famous iris data set in R: Species, Sepal.Length, Petal.Width, etc. Some example patterns: a dot between square brackets [. ] matches a period, a dot by itself . matches any non-newline character, and a dot followed by a star . * matches zero or more non-newline characters. Therefore the pattern . * [. ] . * matches zero or more non-newline characters, followed by a period, followed by zero or more non-newline characters. It would match Sepal.Length and Petal.Width but it would not match Species. For a more detailed discussion of regular expressions, we refer the reader to help(regex) in R or the book of Friedl (2002).

The focus of this article is patterns with capture groups, which are typically defined using parentheses. For example, the pattern (. *) [. ] (. *) results in the same matches as the pattern in the previous paragraph, and it additionally allows the user to capture and extract the substrings by group index (e.g., group 1 matches Sepal, group 2 matches Length).

Named capture groups allow extracting the substring by name rather than by index. Using names rather than indices is preferable in order to create more readable regular expressions (names document the purpose of each sub-pattern), and to create more readable R code (it is easier to understand the intent of named references than numbered references). For example, the pattern (?<part>. *) [. ] (?<dimension>. *) documents that the flower part appears before the measurement dimension; the part group matches Sepal and the dimension group matches Length.

Recently, Hocking (2019a) proposed a new syntax for defining named capture groups in R code. Using this new syntax, named capture groups are specified using named arguments in R, which results in code that is easier to read and modify than capture groups defined in string literals. For example, the pattern (?<part>. *) [. ] (?<dimension>. *) documents that the flower part appears before the measurement dimension; the part group matches Sepal and the dimension group matches Length.

The main thesis of this article is that regular expressions can greatly simplify the code required to specify wide-to-tall data reshaping operations (when the input columns adhere to a regular naming convention). For one such operation the input is a “wide” table with many columns, and the desired output is a “tall” table with more rows, and some of the input columns converted into a smaller number of output columns (Figure 1). To clarify the discussion we first define three terms that we will use to refer to the different types of columns involved in this conversion:

Reshape columns contain the data which is present in the same amount but in different shapes in the input and output. There are equivalent terms used in different R packages: varying in utils::reshape, measure vars in melt (data.table, reshape2), etc.

Copy columns contain data in the input which are each copied to multiple rows in the output (id. vars in melt).

Capture columns are only present in the output, and contain data which come from matching a capturing regex pattern to the input reshape column names.

For example the wide iris data (W in Figure 1) have four numeric columns to reshape: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width. For some purposes (e.g., displaying a histogram of each reshape input column using facets in ggplot2) the desired reshaping operation results in a table with a single reshape output column (S in Figure 1), two copied columns, and two columns captured from the names of the reshaped input columns. For other purposes (e.g., scatterplot to compare sepal and
Convert four input reshape columns to single output reshape column and two output capture columns

Figure 1: Two rows of the iris data set (W, black) are considered as the input to a wide-to-tall reshape operation. Four input reshape columns are converted to either a single output reshape column (S, blue) or multiple (2) output reshape columns (M1, M2, red). Other output columns are either copied from the non-reshaped input data, or captured from the names of the reshaped input columns.

petal sizes) the desired reshaping operation results in a table with multiple reshape output columns (M1 with Sepal and Petal columns in Figure 1), two copied columns, and one column captured from the names of the reshaped input columns.

In this article our original contribution is the R package nc which provides a new implementation of the previously proposed named capture regex syntax of Hocking (2019a), in addition to several new functions that perform wide-to-tall data reshaping using regular expressions. The main new idea is to use a single named capture regular expression for defining both (1) the subset of reshape input columns to convert, and (2) the additional capture output columns. We will show that this results in a simple, powerful, non-repetitive syntax for wide-to-tall data reshaping. A secondary contribution of this article is a detailed comparison of current R functions for wide-to-tall data reshaping, in terms of syntax, computation times, and functionality (Table 1). Note that in this article we do not discuss tall-to-wide data reshaping, because regular expressions are not useful in that case.

The organization of this article is as follows. The rest of this introduction provides an overview of current R packages for regular expressions and data reshaping. The second section describes the proposed functions of the nc package, then the third section provides detailed comparisons with other R packages. The article concludes with a summary and discussion of possible future work.

Related work

There are many R functions which can extract tables from non-tabular text using regular expressions. Recommended R package functions include base::regexpr and base::gregexpr as well as utils::strcapture. CRAN packages which provide various functions for text processing using regular expressions include namedCapture (Hocking, 2019b), rematch2 (Csárdi, 2017), rex (Ushey et al., 2017), stringr (Wickham, 2018), stringi (Gagolewski, 2018), tidyr (Wickham and Henry, 2018), and re2r (Wenfeng, 2017). We refer the reader to our previous research paper for a detailed comparison of these packages (Hocking, 2019a).

For reshaping data from wide (one row with many columns) to tall (one column with many rows), there are several different R functions that provide similar functionality. Each function supports a different set of features (Table 1); each feature/column is explained in detail below:
Table 1: Reshaping functions in R support various features: “single” for converting input columns into a single output column; “multiple” for converting input columns (either “if sorted” in a regular order, or “yes” for any order) into multiple output columns of possibly different types; “regex” for regular expressions to “match” input column names or to “capture” and create new output column names; “na.rm” for removal of missing values; “types” for converting input column names to non-character output columns; “list” for output of list columns.

<table>
<thead>
<tr>
<th>pkg::function</th>
<th>single</th>
<th>multiple</th>
<th>regex</th>
<th>na.rm</th>
<th>types</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>nc::capture_melt_multiple</td>
<td>no</td>
<td>yes</td>
<td>capture</td>
<td>yes</td>
<td>any</td>
<td>yes</td>
</tr>
<tr>
<td>nc::capture_melt_single</td>
<td>yes</td>
<td>no</td>
<td>capture</td>
<td>yes</td>
<td>any</td>
<td>yes</td>
</tr>
<tr>
<td>tidyr::pivot_longer</td>
<td>yes</td>
<td>yes</td>
<td>capture</td>
<td>yes</td>
<td>any</td>
<td>yes</td>
</tr>
<tr>
<td>stats::reshape</td>
<td>yes</td>
<td>if sorted</td>
<td>capture</td>
<td>no</td>
<td>some</td>
<td>no</td>
</tr>
<tr>
<td>data.table::melt, patterns</td>
<td>yes</td>
<td>if sorted</td>
<td>match</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>tidyfst::longer_dt</td>
<td>yes</td>
<td>no</td>
<td>match</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>tidyr::gather</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>some</td>
<td>yes</td>
</tr>
<tr>
<td>tidyfast::dt_pivot_longer</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>cdata::rowrecs_to_blocks</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>cdata::unpivot_to_blocks</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>reshape2::melt</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>utils::stack</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**single** refers to support for converting input reshape columns of the same type to a single reshape output column.

**multiple** refers to support for converting input reshape columns of possibly different types to multiple output reshape columns; “if sorted” means that conversion works correctly only if the input reshape columns are sorted in a regular order, e.g., Sepal.Length, Sepal.Width, Petal.Length, Petal.Width; “yes” means that conversion works correctly even if the they are not sorted, e.g., Sepal.Length, Sepal.Width, Petal.Width, Petal.Length.

**regex** refers to support for regular expressions; “match” means a pattern is used to match the input column names; “capture” means that the specified pattern is used to create new output capture columns — this is especially useful when the names consist of several distinct pieces of information, e.g., Sepal.Length; “no” means that regular expressions are not directly supported (although base::grep can always be used).

**na.rm** refers to support for removing missing values.

**types** refers to support for converting captured text to numeric output columns.

**list** refers to support for output of list columns.

Recommended R package functions include stats::reshape and utils::stack for reshaping data from wide to tall. Of the features listed in Table 1, utils::stack only supports output with a single reshape column, whereas stats::reshape supports the following features. For data with regular input column names (output column, separator, time value), regular expressions can be used to specify the separator (e.g., in Sepal.Length, Sepal is output column, dot is separator, Length is time value). Multiple output columns are supported, but incorrect output may be computed if input columns are not sorted in a regular order. The time value is output to a capture column named time by default. Automatic type conversion is performed on time values when possible, but custom type conversion functions are not supported. There is neither support for removing missing value removal nor list column output.

The tidyr package provides two functions for reshaping data from wide to tall format: gather and pivot_longer. The older gather function only supports converting input reshape columns to a single output reshape column (not multiple). The input reshape columns to convert may not be directly specified using regular expressions; instead R expressions such as x:y can be used to indicate all columns starting from x and ending with y. It does support limited type conversion; if the convert = TRUE argument is specified, the utils::type.convert function is used to convert the input column names to numeric, integer, or logical. In contrast the newer pivot_longer also supports multiple output reshape columns (even if input reshape columns are unsorted), and regular expressions for specifying output capture columns (but to specify input reshape columns with a regex, grep must be used). Arbitrary type conversion is also supported in pivot_longer, via the names_transform argument, which should be a named list of conversion functions. Both functions support list columns and removing missing values, although different arguments are used (na.rm for gather, values_drop_na for pivot_longer).
The reshape2 and data.table packages each provide a melt function for converting data from wide to tall (Wickham, 2007; Dowle and Srinivasan, 2019). The older reshape2 version only supports converting input reshape columns to a single output reshape column, whereas the newer data.table version also supports multiple output reshape columns. Regular expressions are not supported in reshape2, but can be used with data.table::patterns to match input column names to convert (although the output can be incorrect if columns are not sorted in a regular order). Neither function supports type conversion, and both functions support removing missing values from the output using the na.rm argument. List column output is supported in data.table but not reshape2. The tidyfast (Barrett, 2020) and tidyfst (Huang and Zhao, 2020) packages provide reshaping functions that use data.table::melt internally (but do not support multiple output reshape columns).

The cdata package provides several functions for data reshaping, including rowrecs_to_blocks and unpivot_to_blocks which can convert data from wide to tall (Mount and Zumel, 2019). The simpler of the two functions is unpivot_to_blocks, which supports a single output reshape column (interface similar to reshape2::melt/tidyr::gather). The user of rowrecs_to_blocks must provide a control table that describes how the input should be reshaped into the output. It therefore supports multiple output reshape columns, for possibly unsorted input columns. Both functions support list column output, but other features from Table 1 are not supported (regular expressions, missing value removal, type conversion).

**Basic features for wide-to-tall data reshaping using regular expressions**

The nc package provides new regular expression functionality based on the syntax recently proposed by Hocking (2019a). During the rest of the article we give only a brief overview of this syntax; for a more detailed review please read the nc package vignettes. In this section we show how new nc functions can be used to reshape wide data (with many columns) to tall data (with fewer columns, and more rows). We begin by considering the two data visualization problems which were mentioned in the introduction, and which involve the familiar iris data set.

**Single reshape output column**

First, suppose we would like to visualize the univariate distribution of each numeric variable. One way would be to use a histogram of each numeric variable, with row facets for flower part and column facets for measurement dimension. Our desired output therefore needs a single column with all of the reshaped numeric data to plot (Figure 1, W → S).

We can perform this operation using nc::capture_melt_single, which inputs a data frame and a pattern which should match the names of the input columns to reshape. Any input columns with names that do not match the pattern are considered copy columns; the output also contains a capture column for each group specified in the pattern:

```r
> (iris.tall.single <- nc::capture_melt_single(
+   iris, part = ".*", ".", dim = ".*", value.name = "cm"))
```

```
Species part dim cm
1: setosa Sepal Length 5.1
2: setosa Sepal Length 4.9
3: setosa Sepal Length 4.7
4: setosa Sepal Length 4.6
5: setosa Sepal Length 5.0
```

The output above is a data table (a data frame subclass with special methods that have reference semantics) because data.table::melt is used internally for the reshape operation. The output data
The table consists of one copy column (Species), two capture columns (part, dim), and a single reshape column (cm). These data can be used to create the desired histogram with `ggplot2` via:

```r
> library(ggplot2)
> ggplot(iris.tall.single) + facet_grid(part ~ dim) +
+  theme_bw() + theme(panel.spacing = grid::unit(0, "lines")) +
+  geom_histogram(aes(cm, fill = Species), color = "black", bins = 40)
```

For comparison, we show how the same reshape operation can be accomplished with the `data.table` package,

```r
> iris.pattern <- "(\[.]*\[.]*\)"
> iris.wide <- data.table::as.data.table(iris)
> iris.tall <- data.table::melt(
  +  iris.wide, measure = patterns(iris.pattern), value.name = "cm")
> iris.tall[, 
  +  variable := sub(iris.pattern, \1, variable),
  +  dim := sub(iris.pattern, \2, variable)])
Species  variable cm part dim
1:  setosa  Sepal.Length  5.1 Sepal Length
2:  setosa  Sepal.Length  4.9 Sepal Length
3:  setosa  Sepal.Length  4.7 Sepal Length
4:  setosa  Sepal.Length  4.6 Sepal Length
5:  setosa  Sepal.Length  5.0 Sepal Length
---
596:  virginica  Petal.Width  2.3 Petal Width
597:  virginica  Petal.Width  1.9 Petal Width
598:  virginica  Petal.Width  2.0 Petal Width
599:  virginica  Petal.Width  2.3 Petal Width
600:  virginica  Petal.Width  1.8 Petal Width
```

The code above uses `data.table::melt` with `patterns` which takes a regex used to specify the four columns to reshape. The part and dim capture columns must be created during a post-processing step. In this case the `nc` code is substantially simpler because the named capture regular expression was used to specify both the input columns to reshape, and the capture columns to output.

Finally we show how the same reshape operation could be done using the `tidyr` package,

```r
> tidyr::pivot_longer(iris, matches(iris.pattern), values_to = "cm",
+  names_to=c("part", "dim"), names_pattern=iris.pattern)
# A tibble: 600 x 4
Species part dim cm
<fct> <chr> <chr> <dbl>
1 setosa  Sepal.Length  5.1
2 setosa  Sepal.Width  3.5
3 setosa  Petal.Length  1.4
4 setosa  Petal.Width  0.2
5 setosa  Sepal.Length  4.9
6 setosa  Sepal.Width  3
7 setosa  Petal.Length  1.4
8 setosa  Petal.Width  0.2
```
The code above is almost as simple as the corresponding `nc` code, but with one key difference. The output capture column names are defined in the `names_to` argument, which is far away from the definition of the groups in `iris.pattern`. In this simple example with two groups in the regex this separation of related concepts is not a huge problem, but the `nc` syntax should be preferred for more complex patterns (with more groups) in order to keep the group names and sub-patterns closer and easier to maintain/read in the code.

Multiple reshape output columns

For the second data reshaping task, suppose we want to determine whether or not sepals are larger than petals, for each measurement dimension and species. We could use a scatterplot of sepal versus petal, with a facet for measurement dimension. We therefore need a data table with two reshape output columns: a Sepal column to plot against a Petal column (Figure 1, W→M1). We can perform this operation using another function, `nc::capture_melt_multiple`, which inputs a data frame and a pattern which must contain the special column group and at least one other named group:

```r
> (iris.parts <- nc::capture_melt_multiple(iris, column = ".*", ".", dim = ".*"))
```

<table>
<thead>
<tr>
<th>Species</th>
<th>dim</th>
<th>Petal</th>
<th>Sepal</th>
</tr>
</thead>
<tbody>
<tr>
<td>setosa</td>
<td>Length 1.4</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>setosa</td>
<td>Length 1.4</td>
<td>4.9</td>
<td></td>
</tr>
<tr>
<td>setosa</td>
<td>Length 1.3</td>
<td>4.7</td>
<td></td>
</tr>
<tr>
<td>setosa</td>
<td>Length 1.5</td>
<td>4.6</td>
<td></td>
</tr>
<tr>
<td>setosa</td>
<td>Length 1.4</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>virginica</td>
<td>Width 2.3</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>virginica</td>
<td>Width 1.9</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>virginica</td>
<td>Width 2.0</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>virginica</td>
<td>Width 2.3</td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>virginica</td>
<td>Width 1.8</td>
<td>3.0</td>
<td></td>
</tr>
</tbody>
</table>

Again, any input columns with names that do not match the pattern are considered copy columns (Species in the example above). Each unique value captured in the special column group becomes the name of an output reshape column (Petal, Sepal); other groups are used to create output capture columns (dim). These data can be used to create the scatterplot using `ggplot2` via:

```r
> ggplot(iris.parts) + facet_grid(. ~ dim) +
  + theme_bw() + theme(panel.spacing = grid::unit(0, "lines")) +
  + coord_equal() + geom_abline(slope = 1, intercept = 0, color = "grey") +
  + geom_point(aes(Petal, Sepal, color = Species), shape = 1)
```

For comparison, we show how to output a data table with multiple reshape output columns using the `data.table` and `tidyr` packages,

```r
> iris.multiple <- data.table::melt(iris.wide, measure = patterns(Petal = "Petal", Sepal = "Sepal"))
> iris.multiple[, dim := c("Length", "Width") [variable] ]
```
Species variable Petal Sepal dim
1: setosa 1 1.4 5.1 Length
2: setosa 1 1.4 4.9 Length
3: setosa 1 1.3 4.7 Length
4: setosa 1 1.5 4.6 Length
5: setosa 1 1.4 5.0 Length
---
296: virginica 2 2.3 3.0 Width
297: virginica 2 1.9 2.5 Width
298: virginica 2 2.0 3.0 Width
299: virginica 2 2.3 3.4 Width
300: virginica 2 1.8 3.0 Width

> tidyr::pivot_longer(iris, matches(iris.pattern), values_to = "cm",
+ names_to=c(".value", "dim"), names_pattern=iris.pattern)

# A tibble: 300 x 4
Species dim Sepal Petal
<fct> <chr> <dbl> <dbl>
1 setosa Length 5.1 1.4
2 setosa Width 3.5 0.2
3 setosa Length 4.9 1.4
4 setosa Width 3 0.2
5 setosa Length 4.7 1.3
6 setosa Width 3.2 0.2
7 setosa Length 4.6 1.5
8 setosa Width 3.1 0.2
9 setosa Length 5 1.4
10 setosa Width 3.6 0.2
#... with 290 more rows

The code above computes equivalent results, but suffers from the same drawbacks as discussed in the previous section (repetition, separation of pattern and group names).

To conclude this section, nc provides two new functions for data reshaping using regular expressions. Both functions input a data frame to reshape, and a pattern to match to the column names. For nc::capture_melt_single, all matching input columns are reshaped in the output to a single column which is named using the value.name argument. For nc::capture_melt_multiple the output is multiple reshape columns with names defined by the values captured in the special column group. Values from other groups are stored in capture columns in the output. Both functions support output of numeric capture columns via user-specified type conversion functions, as we will see in the next section.

Comparisons which highlight differences with other packages

In this section we compare the new data reshaping functions in the nc package with similar functions in other packages. We aim to demonstrate that the new nc syntax is often more convenient and less repetitive, without sacrificing speed.

Building a complex pattern from smaller sub-patterns

In terms of functionality for wide-to-tall data reshaping, the most similar package to nc is tidyr (Table 1). One advantage of nc is that complex patterns may be defined in terms of simpler sub-patterns, which can include group names and type conversion functions. Integrating these three pieces results in a syntax that is easy to read as well; it is more difficult to build and read complex patterns using tidyr syntax, which requires specifying regex pattern strings, group names, and types as separate arguments. For example, consider a data set from the World Health Organization (WHO):

> data(who, package = "tidyr")
> set.seed(1);sample(names(who), 10)
[1] "newrel_f3544"  "year"    "new_ep_m65"  "country"  "new_ep_m1524"
[6] "new_sn_m4554" "new_ep_f3544" "new_sp_f2534" "new_sp_f65"  "newrel_m4554"
Each reshape column name starts with new and has three distinct pieces of information: diagnosis type (e.g., ep, rel), gender (m or f), and age range (e.g., 1524, 4554). We extract all three pieces of information below, and include a function for converting gender to a factor with levels in a specific (non-default) order:

```r
> nc.who.sub.pattern <- list(
+ "new_?", diagnosis = ".*", ",",
+ gender = ".", function(mf)factor(mf, c("m", "f")))
> nc.who.ages <- nc::capture_melt_single(who, nc.who.sub.pattern, ages = ".*")
> print(nc.who.ages[1:2], class = TRUE)

country iso2 iso3 year diagnosis gender ages value
<chr> <chr> <chr> <int> <chr> <fctr> <chr> <int>
1: Afghanistan AF AFG 1997 sp m 014 0
2: Afghanistan AF AFG 1998 sp m 014 30
```

First note that nc.who.sub.pattern is a sub-pattern list variable that we have used as the first part of the pattern in the call to nc::capture_melt_single above (and we will use that sub-pattern again below). Sub-pattern lists may contain regex character strings (patterns to match), functions (for converting the previous capture group), or other sub-pattern lists. The reshaped output is a data table with gender converted to a factor — this can also be done using tidyr::pivot_longer:

```r
> tidyr.who.sub.names <- c("diagnosis", "gender") #L0
> tidyr.who.sub.pattern <- "new_?(.*)_(.)" #L1
> tidyr.who.pattern <- paste0(tidyr.who.sub.pattern, "(.*)") #L2
> tidyr::pivot_longer(#L3
+ who, cols = matches(tidyr.who.pattern), #L4
+ names_to = c(tidyr.who.sub.names, "ages"), #L5
+ names_ptypes = list(gender = factor(levels = c("m", "f"))), #L6
+ names_pattern = tidyr.who.pattern)[1:2,] #L7

# A tibble: 2 x 8
country iso2 iso3 year diagnosis gender ages value
<chr> <chr> <chr> <int> <chr> <fctr> <chr> <int>
1 Afghanistan AF AFG 1980 sp m 014 NA
2 Afghanistan AF AFG 1980 sp m 1524 NA
```

In the code above we first define a sub-pattern variable for the diagnosis and gender capture groups, as we did using nc. One difference is that the tidy sub-pattern variable is a string with un-named capture groups, whereas the nc sub-pattern variable is a list which includes capture group names as well as a type conversion function. These three parameters are specified as three separate arguments in tidy, which results in some separation (e.g., group names defined on L0 and L5 but corresponding sub-patterns defined on L1 and L2) and repetition (e.g., gender appears on L0 and L6) in the code. The pattern also must be repeated: first in the cols argument (L4) to specify the set of input reshape columns, second in the names_pattern argument (L7) to specify the conversion from input reshape column names to output capture column values.

Now suppose we want to extract two numeric columns from ages, for example to use as interval-censored outputs in a survival regression. Using nc we can use the previously defined sub-pattern (including the previously defined group names and type conversion function) as the first part of a larger pattern:

```r
> who.typed <- nc::capture_melt_single(who, nc.who.sub.pattern, ages = list(
+ ymin = "0|[0-9]{2}", as.numeric,
+ ymax = "[0-9]{0,2}", function(x)ifelse(x == ",", Inf, as.numeric(x))))
> who.typed[1:2]

country iso2 iso3 year diagnosis gender ages ymin ymax value
<chr> <chr> <chr> <int> <chr> <fctr> <chr> <chr> <int>
1 Afghanistan AF AFG 1997 sp m 014 0 14 0
2 Afghanistan AF AFG 1998 sp m 014 0 14 30
```

In the code above we first define a sub-pattern variable for the diagnosis and gender capture groups, as we did using nc. One difference is that the tidy sub-pattern variable is a string with un-named capture groups, whereas the nc sub-pattern variable is a list which includes capture group names as well as a type conversion function. These three parameters are specified as three separate arguments in tidy, which results in some separation (e.g., group names defined on L0 and L5 but corresponding sub-patterns defined on L1 and L2) and repetition (e.g., gender appears on L0 and L6) in the code. The pattern also must be repeated: first in the cols argument (L4) to specify the set of input reshape columns, second in the names_pattern argument (L7) to specify the conversion from input reshape column names to output capture column values.

Now suppose we want to extract two numeric columns from ages, for example to use as interval-censored outputs in a survival regression. Using nc we can use the previously defined sub-pattern (including the previously defined group names and type conversion function) as the first part of a larger pattern:
Note in the code above that each group name, regex pattern string, and corresponding type conversion function appears on the same line — this syntax keeps these three related pieces of information close together, which makes complex patterns easier to read and build from smaller pieces. Also, note how an anonymous function is used to convert the values captured in the `ymax` group to numeric (and it maps the empty string to `Inf`). Such custom type conversion functions are supported by `tidyr` since version 1.1.0 (early 2020), so we can do:

```r
> tidyr.who.range.pattern <- paste0(tidyr.who.sub.pattern, "((0|\[0-9]{2})([0-9]{0,2}))")
> tidyr::pivot_longer(
+ who, cols = matches(tidyr.who.range.pattern),
+ names_to = c(tidyr.who.sub.names, "ages", "ymin", "ymax"),
+ names_transform = list(
+ gender = function(x)factor(x, levels = c("m", "f")),
+ ymin = as.numeric,
+ ymax = function(x)ifelse(x == "", Inf, as.numeric(x))),
+ names_pattern = tidyr.who.range.pattern)[1:7,]
# A tibble: 7 x 10
  country iso2 iso3 year diagnosis gender ages ymin ymax value
  <chr> <chr> <chr> <int> <chr> <fct> <chr> <dbl> <dbl> <int>
1 Afghanistan AF AFG 1980 sp m 014 0 14 NA
2 Afghanistan AF AFG 1980 sp m 1524 15 24 NA
3 Afghanistan AF AFG 1980 sp m 2534 25 34 NA
4 Afghanistan AF AFG 1980 sp m 3544 35 44 NA
5 Afghanistan AF AFG 1980 sp m 4554 45 54 NA
6 Afghanistan AF AFG 1980 sp m 5564 55 64 NA
7 Afghanistan AF AFG 1980 sp m 65 65 Inf NA
```

The code above uses the `names_transform` argument to define type conversion functions, which requires some repetition (e.g., `ymax` and `ymin` each appear twice).

To conclude this comparison, we have seen that `nc` syntax makes it easy to read and write complex patterns, because it keeps group-specific names and type conversion functions near the corresponding sub-patterns. We have also shown that repetition is often necessary with `tidyr` (e.g., pattern, group names), whereas such repetition can be avoided by using `nc`.

**Comparison with other packages which support multiple reshape output columns**

In this section we demonstrate the advantages of using `nc` over several alternatives which support multiple reshape output columns. A major advantage is that `nc` directly supports regular expressions for defining the input reshape columns and output capture columns. Another advantage is that `nc` always returns a correct output data set with multiple reshape columns, even when the input columns are not sorted in a regular order. For example, consider the following simple data set in which the columns are not in regular order:

```r
> (TC <- data.table::data.table(
+ treatment.age = 13,
+ control.gender = "M",
+ treatment.gender = "F",
+ control.age = 25))
```

<table>
<thead>
<tr>
<th>treatment.age</th>
<th>control.gender</th>
<th>treatment.gender</th>
<th>control.age</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>M</td>
<td>F</td>
<td>25</td>
</tr>
</tbody>
</table>

It is clear from the table above that the treatment group consists of a teenage female, whereas the control group consists of a male aged 25 (not the best experimental design, but easy to remember for the demonstration in this section). Assume we need an output data table with two reshape columns (age and gender) as well as a capture column (group). The `nc` syntax we would use is:

```r
> nc::capture_melt_multiple(TC, group = ".*", "\.[.]", column = ".*"
```
group age gender
1: control 25 M
2: treatment 13 F

The correct result is computed above because nc reshapes based on the input column names (the order of the input columns is not relevant). A naïve user may attempt to perform this reshape using data.table::patterns:

```r
> data.table::melt(TC, measure.vars = patterns(age = "age", gender = "gender"))
```

variable age gender
1: 1 13 M
2: 2 25 F

First, note that the syntax above requires repetition of age and gender (in names and in pattern strings). Also it is clear that the result is incorrect! Actually, the patterns function is working as documented; it “returns the matching indices” of the provided regex. However, since the input columns are not sorted in regular order, melt returns an incorrect result (this is an incorrect use of these functions, not a bug). To get a correct result, we can provide a list of index vectors:

```r
> data.table::melt(TC, measure.vars = list(age = c(1,4), gender = c(3,2)))
```

variable age gender
1: 1 13 F
2: 2 25 M

This is what nc does internally; it also converts the variable output column to a more interpretable/useful capture column (e.g., group above).

The stats::reshape function suffers from the same issue as the patterns usage above. Another issue with this function is that it assumes the output reshape column names are the first part of the input column names (e.g., Figure 1, W→M1). When input column names have a different structure (e.g., Figure 1, W→M2), they must be renamed, putting the desired output reshape column names first:

```r
> TC.renamed <- structure(TC, names = sub("(.*)\[(.*)\]", \"\2.\1\", names(TC)))
> stats::reshape(TC.renamed, 1:4, direction = "long", timevar = "group")
```

However the result above still contains incorrect results in the gender column. The correct result can be obtained by sorting the input column names:

```r
> TC.sorted <- data.frame(TC.renamed)[, sort(names(TC.renamed))]
> stats::reshape(TC.sorted, 1:4, direction = "long", timevar = "group")
```

After renaming and sorting the input columns, the correct result is obtained using stats::reshape. Another way to obtain a correct result is with the cdata package:

```r
> cdata::rowrecs_to_blocks(TC, controlTable = data.frame(
+ group = c("treatment", "control"),
+ age = c("treatment.age", "control.age"),
+ gender = c("treatment.gender", "control.gender"),
+ stringsAsFactors = FALSE))
```

The cdata package is very powerful, and can handle many more types of data reshaping operations than nc. However, it requires a very explicit definition of the desired conversion in terms of a control table, which results in rather verbose code. In contrast, the terse regular expression syntax of nc is a more implicit approach, which assumes the input columns to reshape have regular names.
To conclude this section, we have discussed some advantages of nc relative to other R packages. Input columns with regular names do not need to be renamed/sorted for nc functions, whereas renaming/sorting may be necessary using stats::reshape. Verbose/explicit control table code is always necessary with cdata, whereas a terse/implicit regular expression syntax is used with nc to simplify the definition of reshape operations.

Comparing computation times of functions for wide-to-tall data reshaping

In previous sections we have showed that the nc package provides a convenient syntax for defining wide-to-tall reshape operations. In this section we investigate whether this convenience comes at the cost of increased computation time. We aim to demonstrate that computation time required for the proposed nc package is comparable with other packages for data reshaping. In particular, since nc is implemented using data.table, we expect that nc should be slightly slower than data.table (by only the amount of time required for regex matching). In our result figures we show the median and quartiles over 10 timings using the microbenchmark package on an Intel Core i7-8700 3.20GHz processor. Note that these timings include both the regex matching (which should be relatively fast) and the data reshaping operation (which should be relatively slow). We varied the number of rows/columns in each experiment by copying/duplicating the rows/columns in each source data set.

First, we performed timings on variants of the iris data with a variable number of rows, and twice the original number of reshape columns (8). The input reshape column names were of the form day1.Sepal.Length, day2.Sepal.Length, day1.Sepal.Width, etc. Since the desired output has two reshape columns (Sepal and Petal), we considered packages which support multiple output columns (cdata, stats, tidyr, nc, data.table). As expected, we observed that all algorithms have similar asymptotic time complexity (Figure 2). We observed that nc is slightly slower than data.table (by constant factors), slightly faster than the other packages (cdata, stats), and about the same speed as tidyr.

Second, we performed similar timings on variants of the iris data with a variable number of columns, and the original number of rows (150). As in the previous experiment, we expected that all functions would have similar slopes, indicating linear asymptotic time complexity. Surprisingly, we observed on the log-log plot (Figure 3) that cdata has a larger asymptotic slope than the other packages, which suggests its time complexity may be super-linear in the number of columns to reshape. The other packages differed by constant factors, with data.table being fastest, followed by tidyr, nc, cdata, and finally the slowest stats. All packages except stats performed the operation in less than 1 second for 1,000 or fewer columns. This comparison confirms the expectation that nc speed is comparable to other packages.

Third, we performed timings on versions of the WHO data with a variable number of duplicated rows, and the original number of columns (56). We ran reshaping functions from several additional packages (utils, reshape2, tidyfast) that can compute the desired output table with a single reshape output column. We computed the amount of time it takes to create zero or four capture output columns (with additional post-processing steps for tidyfast::dt_pivot_longer, reshape2::melt, tidyr::gather, cdata::unpivot_to_blocks). We expected that functions which require additional post-processing steps should be slower by constant factors. As we expected, all functions appear to have similar asymptotic time complexity, and differ only in terms of constant factors. For zero capture
output columns the slowest functions were \texttt{stats::reshape} and \texttt{cdata::unpivot_to_blocks}, which were the only ones to take more than one second for 10,000 input rows. The fastest functions were \texttt{data.table::melt} and \texttt{tidyfast::dt_pivot_longer} (about 10ms for 10,000 input rows). As expected, for four capture output columns the functions which require post-processing were slower, and the fastest functions were \texttt{data.table::melt} and \texttt{tidyfast::dt_pivot_longer}.

Finally, we performed similar timings on variants of the WHO data with a variable number of columns, and a fixed number of rows (11). The desired output again has a single reshape output column, and we again tried computing either zero or four capture output columns. We observed timings (Figure 3) with similar asymptotic trends as in the previous comparisons. In particular, timings for most packages appear to be linear in the number of input reshape columns, and timings for \texttt{cdata} appear to be super-linear for a large number of columns. These data indicate that \texttt{nc} speed is similar to comparable R packages.

Discussion and conclusions

In this paper we described the \texttt{nc} package and its new functions for regular expressions and data reshaping. The \texttt{nc} package allows a user to define a regular expression in R code, along with capture group names and corresponding type conversion functions. We showed how this syntax makes it easy to define complex regular expressions in terms of simpler sub-patterns, while providing a uniform interface to three regex engines (ICU, PCRE, RE2). We showed several examples of how \texttt{nc} can be used for parsing non-tabular text data, and for wide-to-tall data reshaping. We provided a detailed comparison with other data reshaping functions in terms of syntax, functionality, and computation time.

In all of our speed comparisons, we observed that the speed of \texttt{nc} is similar to other R functions for wide-to-tall data reshaping. We expected that all R functions would have linear asymptotic timings, and differ only in constant factors. We were surprised to observe in our empirical timings that the \texttt{cdata} package appears to have asymptotic time complexity that is super-linear in the number of columns to reshape. This result suggests that the speed of \texttt{cdata} could be improved by adopting one of the linear time reshaping algorithms used in the other packages.

The \texttt{tidyr::pivot_longer} function provides a feature set which is most similar to \texttt{nc} data reshaping functions. We showed that both packages can perform the same data reshaping operations, but \texttt{nc} provides a syntax that reduces repetition in user code. Another advantage is that \texttt{nc} R code allows sub-pattern lists which contain group names, regex patterns, and type conversion functions, whereas in \texttt{tidyr} these three related pieces of information must be defined in separate arguments. Therefore \texttt{nc} syntax may be preferable in order to ease definition of complex patterns, and to avoid repetition in user code.

In \texttt{nc} there are two different functions for wide-to-tall data reshaping: \texttt{nc::capture_melt_single} computes a single output reshape column, and \texttt{nc::capture_melt_multiple} computes multiple output reshape columns. In contrast, other functions that support multiple output reshape columns also support a single output reshape column (Table 1). It is natural to ask whether these two \texttt{nc} functions could be combined into a single function that could handle both kinds of output. Of course it is possible, but we prefer to keep the two functions separate in order to provide more specific/informative
Figure 4: Timings for computing a tall output table with a single reshape column from a wide input table with 56 reshape columns and a variable number of rows (x axis). Left panel shows time to compute output data table with no capture columns; Right panel shows time to compute output data table with four capture columns (typically slower as post-processing steps may be necessary).

Figure 5: Timings for computing a tall output table with a single reshape column from a wide input table with 11 rows and a variable number of columns to reshape (x axis). Left panel shows time to compute output data table with no capture columns; Right panel shows time to compute output data table with four capture columns (typically slower as post-processing steps may be necessary).

Future work

For future work, we will be interested to explore other operations and R packages/functions which could be simplified using regular expressions. For example, the `tidyr::pivot_longer` function requires some repetition of the pattern (in `names_pattern` and `cols` arguments); it could be simplified by changing the behavior when `names_pattern` is specified and `cols` is not (currently an error, could instead set `cols` to the set of columns which match `names_pattern`).

Another example where there is room for improvement is `data.table::melt` which we have shown requires some post-processing steps to output capture columns. As a result of this research we have proposed changes to `data.table::melt`¹ that allow efficient specification and output of capture columns. Since `nc` uses `data.table` internally, we plan to eventually use these changes for speedups of `nc` functions.

Reproducible research statement. The source code for this article can be freely downloaded from https://github.com/tdhock/nc-article

¹https://github.com/Rdatatable/data.table/pull/4731
Bibliography


Q. Wenfeng. re2r: RE2 Regular Expression, 2017. URL https://CRAN.R-project.org/package=re2r. R package version 0.2.0. [p2]


Toby Dylan Hocking
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, Arizona
USA
toby.hocking@nau.edu
ORCID 0000-0002-3146-0865