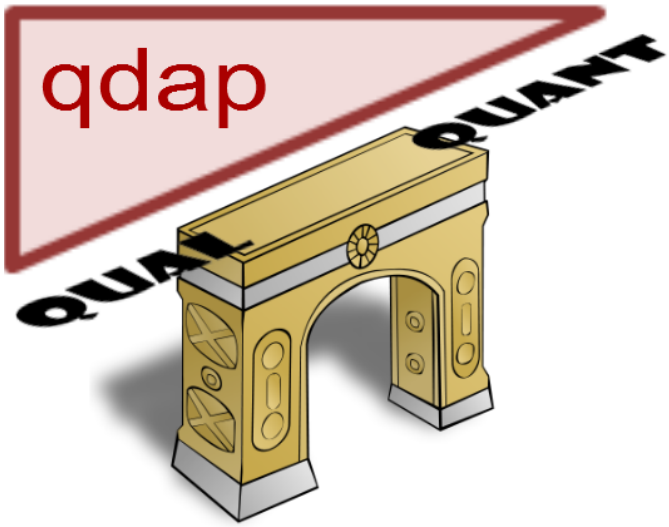
Cleaning Text & Debugging

Tyler W. Rinker

October 8, 2015



The **qdap** package (Rinker, 2013) contains many functions that assume that the text strings supplied are cleaned and in the expected form. Failing to prepare data may result in errors, warnings, and incorrect results. This vignette will outline the checking and prepping of text as well as how to isolate and identify errors caused by unprepared text.

Contents

1 qdap Text Assumptions **3**

2 Cleaning and Debugging Procedures **3**

3 Checking Text **4**

3.1 `check_text` Introduced 4

3.2 `check_text` Output Explained 5

3.3 `check_text` Example 6

4 Cleaning **12**

5 Debugging **13**

5.1 Halving Example 13

6 Reporting a Potential Bug **15**

1 qdap Text Assumptions

Many of the analysis/scoring functions in **qdap** make the following assumptions about your data:

1. Each row contains a single sentence
2. Each sentence contains a **qdap** end-mark ("?", ".", "!", "|")
3. Each sentence contains only one punctuation end-mark
4. Commas are followed by a space
5. Numbers are unimportant (they are ignored unless converted to text equivalent)
6. Symbols (non-text other than apostrophes and end-marks) are ignored
7. Text elements contain alphabetic characters or are `NA`
8. Text words are spelled correctly
9. Text contains only ASCII characters
10. Text contains no escape characters

If these assumptions are not met the integrity of **qdap**'s functions may be undermined.

2 Cleaning and Debugging Procedures

The follow procedure will help the user to avoid problems caused by poorly formatted text and cope with errors:

1. *Check text* for potential problems with `check_text`
2. Perform `check_text`'s recommended *cleaning procedures*
3. *Recheck text* for potential problems with `check_text`
4. Run *analysis*
5. *Debugging* (via halving) to isolate and identify errors when they occur

3 Checking Text

3.1 `check_text` Introduced

The `check_text` function is designed to check text for the following potential sources of errors, warnings, and incorrect results:

- **non_character** – Text that is of `factor` class.
- **missing_ending_punctuation** – Text with no end-mark at the end of the string.
- **empty** – Text that contains an empty element (i.e., `""`).
- **double_punctuation** - Text that contains two **qdap** punctuation marks in the same string.
- **non_space_after_comma** – Text that contains commas with no space after them.
- **no_alpha** – Text that contains string elements with no alphabetic characters.
- **non_ascii** – Text that contains non-ASCII characters.
- **missing_value** – Text that contains missing values (i.e., `NA`).
- **containing_escaped** – Text that contains escaped (see `?Quotes`).
- **containing_digits** – Text that contains digits.
- **indicating_incomplete** – Text that contains end-marks that indicate incomplete/trailing sentences.
- **potentially_misspelled** – Text that contains potentially misspelled words.

The user simply supplies a text variable and `check_text` will output a list. The list prints to the console (or can be saved to an external file) as a prettified summary of potential text problems .

3.2 `check_text` Output Explained

Here is a sample section for potential misspellings. Notice that there are typically 4 elements within a section: (A) a section header, (B) the index within the vector for where the potential problems are located, (C) the actual text strings that raised the alert, and (D) a suggested fix for the problem.

```
======================
POTENTIALLY MISSPELLED
======================

The following observations were potentially misspelled:

2, 11, 13

The following text is potentially misspelled:

2: i want. <<thet>> them .
11: I like <<goud>> eggs!
13: <<tgreat>>

*Suggestion: Consider running `check_spelling_interactive`
```

3.3 `check_text` Example

In this section you will see an instance of the use and output of `check_text`. Here is the data we'll use:

```
x <- c("i like", "i want. thet them .", "I am ! that|", "", NA,
    "they,were there", ".", "   ", "?", "3;", "I like goud eggs!",
    "i 4like...", "\\tgreat",  "She said \"yes\"")
x
```

```
##  [1] "i like"              "i want. thet them ." "I am ! that|"
##  [4] ""                    NA                    "they,were there"
##  [7] "."                   "   "                 "?"
## [10] "3;"                  "I like goud eggs!"   "i 4like..."
## [13] "\\tgreat"            "She said \"yes\""
```

Output from `check_text`:

```
check_text(x)
```

```
## 5:  NA

##
## =============
## NON CHARACTER
## =============
##
##   --IS CHARACTER--
##
##
## ============================
## MISSING ENDING PUNCTUATION
## ============================
##
## The following observations were missing ending punctuation:
##
## 1, 4, 6, 8, 10, 13, 14
##
## The following text is missing ending punctuation:
```

```
##
## 1: i like
## 4:
## 6: they,were there
## 8:
## 10: 3;
## 13: \tgreat
## 14: She said "yes"
##
## *Suggestion: Consider cleaning the raw text or running `add_incomplete`
##
##
## =====
## EMPTY
## =====
##
## The following observations were empty:
##
## 4
##
## The following text is empty:
##
## 4:
##
## *Suggestion: Consider running `blank2NA`
##
##
## ==================
## DOUBLE PUNCTUATION
## ==================
##
## The following observations were double punctuation:
##
## 2, 3, 12
##
## The following text is double punctuation:
##
```

```
## 2: i want. thet them .
## 3: I am ! that|
## 12: i 4like...
##
## *Suggestion: Consider running `sentSplit`
##
##
## =====================
## NON SPACE AFTER COMMA
## =====================
##
## The following observations were non space after comma:
##
## 6
##
## The following text is non space after comma:
##
## 6: they,were there
##
## *Suggestion: Consider running `comma_spacer`
##
##
## ========
## NO ALPHA
## ========
##
## The following observations were no alpha:
##
## 4, 7, 8, 9, 10
##
## The following text is no alpha:
##
## 4:
## 7: .
## 8:
## 9: ?
## 10: 3;
```

```
##
## *Suggestion: Consider cleaning the raw text
##
##
## =========
## NON ASCII
## =========
##
## The following observations were non ascii:
##
##   --NONE FOUND--
##
##
## =============
## MISSING VALUE
## =============
##
## The following observations were missing value:
##
## 5
##
##
## ==================
## CONTAINING ESCAPED
## ==================
##
## The following observations were containing escaped:
##
## 13
##
## The following text is containing escaped:
##
## 13: \tgreat
##
## *Suggestion: Consider using `clean` or `scrubber`
##
##
```

```
## =================
## CONTAINING DIGITS
## =================
##
## The following observations were containing digits:
##
## 10, 12
##
## The following text is containing digits:
##
## 10: 3;
## 12: i 4like...
##
## *Suggestion: Consider using `replace_number`
##
##
## ======================
## INDICATING INCOMPLETE
## ======================
##
## The following observations were indicating incomplete:
##
## 12
##
## The following text is indicating incomplete:
##
## 12: i 4like...
##
## *Suggestion: Consider using `incomplete_replace`
##
##
## ======================
## POTENTIALLY MISSPELLED
## ======================
##
## The following observations were potentially misspelled:
##
```

```
## 2, 11, 13
##
## The following text is potentially misspelled:
##
## 2: i want <<thet>> them
## 11: i like <<goud>> eggs
## 13: great
##
## *Suggestion: Consider running `check_spelling_interactive`
```

4 Cleaning

qdap has a number of functions that can be used to clean and format the text into the form expected by many **qdap** functions. `check_text` suggests the use of some of these cleaning functions. The user should consider cleaning the text as recommended or have legitimate rationale for not doing so. Failure to clean data as recomended may lead to errors, warnings, and incorrect results.

The following is a list of **qdap** cleaning & formatting functions and their designed uses:

- **clean** – Remove escaped characters
- **scrubber** – General text cleaning function that removes extra white spaces other textual anomalies that may cause errors
- **strip** – Strip text of unwanted characters
- **replace_number** – Replace numerically represented numbers with words
- **blank2NA** – Replace blank (empty) cells
- **comma_spacer** – Add a space after a comma
- **sentSplit** – Split multi-sentence strings into individual sentences
- **incomplete_replace** – Replace incomplete sentence end marks (e.g., "...") with "|"
- **check_spelling_interactive** – Interactively check for misspelled words
- **Encoding** – Set the declared encodings for a character vector
- **replace_abbreviation** – Replace abbreviations with long form
- **multigsub** – Wrapper for gsub that takes a vector of search terms and a vector or single value of replacements
- **sub_holder** – Hold the place for particular character values, manipulate the string(s), and then revert place holders back
- **replace_symbol** – Replace select symbols with word equivalents
- **replace_contraction** – Replace contractions with multi-word form
- **Trim** – Remove leading/trailing white spaces
- **bracketX** – Remove bracketed text
- **genX** – Remove text between markers

While an exhaustive discussion of these functions is beyond the scope of this vignette the reader should be aware of these functions and their uses to clean and format text for analysis.

5 Debugging

While the `check_text` is useful for catching many text problems it can not detect all problems a user may encounter. Some functions may provide useful error/warning message that help the user isolate the source of the problem. At other times the source of the error is unknown.

When a user encounters such an error it is important to isolate and identify the source of the error. Often text data sets (corpora) are larger and thus a reasonable strategy can be employed to isolate the source of the bug more quickly. The use of *halving* is one such approach.

In halving, after the user experiences an error, she recursively divides the data set into halves until the source of the error is found. With each pass the non-offending portion is excluded while half of the offending portion of the data is used to test smaller and smaller chunks of the data for the location of the error.[1]

5.1 Halving Example

Here is an example of the entire halving process. First we'll create an error generating function to use with the built in **qdap** data set `DATA`:

```
fake_fun <- function(x) {
    stopifnot(!any(grepl("talk", x)))
    x
}
```

```
with(DATA, fake_fun(state))
```

```
Error: !any(grepl("talk", x)) is not TRUE
```

An initial unknown error is raised. Cut the data in half:

```
with(DATA[1:5, ] , fake_fun(state))

## [1] "Computer is fun. Not too fun." "No it's not, it's dumb."
## [3] "What should we do?"            "You liar, it stinks!"
## [5] "I am telling the truth!"
```

No error is raised with this half. Grab the other half of the data and confirm the error is in that portion of the data:

[1] Note that the halving technique can be used with code testing as well.

```
with(DATA[5:11, ] , fake_fun(state))
```

```
Error: !any(grepl("talk", x)) is not TRUE
```

Confirmed! So take half of the second half (the offending data):

```
with(DATA[5:8, ] , fake_fun(state))

## [1] "I am telling the truth!" "How can we be certain?"  "There is no way."
## [4] "I distrust you."
```

No error is raised with the first half of the original second half. Grab the other half (last 1/4 of the original data set) of the data and confirm the error is in that portion of the data:

```
with(DATA[9:11, ] , fake_fun(state))
```

```
Error: !any(grepl("talk", x)) is not TRUE
```

Confirmed again! At this point there are three rows left to test so we test them one at a time:

```
with(DATA[9, ] , fake_fun(state))
```

```
Error: !any(grepl("talk", x)) is not TRUE
```

Source of the error identified!

Now that you have identified the location (row 9) identify the source. Generally a visual inspection will turn up the source. Other times it may elude you (for example an `Encoding` issue may be masked). If you are eluded now you can begin the halving technique on the individual string. Here we use `substring` to accomplish this task:

```
with(DATA[9, ], fake_fun(substring(state, 1:20)))
```

```
Error: !any(grepl("talk", x)) is not TRUE
```

Found it! Now let's test the other half:

```
with(DATA[9, ], fake_fun(substring(state, 21)))

## [1] " about?"
```

All clear! We continue the halving technique until we've identified the source of the error, in this case the use of the word *talk*.

At this point you may have either identified (a) improperly cleaned data or (b) a potential bug in **qdap**. It is important that you can replicate the problem (either bad data or the bug). If you have located (where) and identified the source of an error (what) you should be able to recreate the error with new data that contains the same type of error causing string(s). If you cannot replicate the error you have not identified the source.

6 Reporting a Potential Bug

If you believe that there is a bug in **qdap** you can:

1. Fork the repository, correct the code, run a CRAN check, and send a pull request.
2. Report the bug via **qdap**'s GitHub issues page. Be sure that you:
   (a) Describe the bug (be specific; in other words, show work on your part to locate and identify the source of the error)
   (b) Provide a minimally reproducible example, including a small data set and code to produce the error

The former is preferred if you are experienced with R programming, the latter if not.

References

Rinker TW (2013). *qdap: Quantitative Discourse Analysis Package*. University at Buffalo/SUNY, Buffalo, New York. Version 2.1.0, URL `http://github.com/trinker/qdap`.