Declutter your R workflow with tidy tools

Zev Ross *
ZevRoss Spatial Analysis
and
Hadley Wickham
RStudio
and
David Robinson
Stack Overflow

Abstract

The *R* language has withstood the test of time. Forty years after it was initially developed (in the form of the S language) *R* is being used by millions of programmers on workflows the inventors of the language could never have imagined. Although base R packages perform well in most settings, workflows can be made more efficient by developing packages with more consistent arguments, inputs and outputs and emphasizing constantly improving code over historical code consistency. The universe of *R* packages known as the tidyverse, including `dplyr`, `tidyr` and others, aim to improve workflows and make data analysis as smooth as possible by applying a set of core programming principles in package development.

Keywords: tidy tools, tidyverse, dplyr, tidyr, tidytext, ggplot2, readr, workflow, pipe, piping, R, base R

*Contact - ZevRoss Spatial Analysis, 120 N. Aurora St, Suite 3A; Ithaca, NY; zev@zevross.com. The authors gratefully acknowledge the constructive feed back from reviewers Jenny Bryan, Amelia McNamara and Stefan Holst Milton Bache.

Introduction

> The process of preparing programs for a digital computer . . . can be an aesthetic experience much like composing poetry or music. — Knuth (1973)

For more than 15 years base R has provided a strong, stable coding foundation. This stability has huge benefits: you can write R code with confidence, knowing that others, now and in the future, can understand and execute the code.

But there is also a cost associated with this stability: base R is weighed down with a number of historical inefficiencies and idiosyncrasies. These are design decisions that made sense at the time, but are not well suited to today's computation environment, today's R user, or today's data. In the last 20 years, computers have grown dramatically more powerful, the users of R have become substantially more diverse, and the types and amounts of data have expanded.

To meet these new demands, we need new tools. It's hard to fundamentally change R without breaking a huge amount of code. That means that most innovation in the data analysis process now occurs in the package ecosystem. The goal of this paper is to show off one part of that ecosystem known as the tidyverse.

The tidyverse

The tidyverse refers to a set of packages that share interfaces and data structures. These commonalities make the packages easier to learn (because there are fewer special cases to memorize), and allow data analyses to flow naturally from one task to the next (because you don't change the "shape" of your data). The philosophy of the tidyverse is similar to and inspired by the "unix philosophy" (Raymond 2003), a set of loose principles that ensure most command line tools play well together.

The packages of the tidyverse are built on and in base R. The differences between the two are subtle, in fact for every idiom in the tidyverse, there's likely a function in base R that is similar. There are two main differences in philosophy:

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

1. Less emphasis on historical code consistency. While adjustments to base R tend to be relatively small and incremental to ensure backward compatibility and overall stability, packages in the tidyverse aim for perfection – even if this means "breaking" links with earlier code.

2. Shared vision for package development with an emphasis on uniformity of function syntax, inputs and outputs. Since the inception of R more than 10,000 R packages have been developed by thousands of R coders. Packages have tended to be developed in isolation which can lead to important innovations. At the same time, the inconsistency in arguments, syntax, object types and others can make stringing together operations inneficient and occasionally painful.

These two factors have led to a small number of consistent principles that are used again and again throughout the tidyverse:

1. Use consistent data structures so the output from one function can easily be fed into the next. The data structure used most commonly in the tidyverse is tidy data (Wickham et al. 2014): a rectangular structure where the columns are variables and the rows are cases.

2. Each function should solve one small and well-defined class of problems. To solve more complex problems, you combine simple pieces in a standard way.

3. Rely on function composition to simplify data science workflows by using, as an example, the `magrittr` pipe thus enhacing readability and avoiding the need to name interim objects.

All in all, there are few things that you can do with the tidyverse that you cannot do with base R. The big difference is the level of friction.

A case study

To illustrate the value of the tidyverse in a modern context we work through an example using a database of Shakespeare's word usage available through Google's BigQuery database

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

(Google 2017). The data was originally retrieved from BigQuery using the `bigrquery` package, and cached locally.

Import with `readr`

Base R has workhorse import and export functions that have served the R coding community for more than a decade (e.g., `read.table()`, `write.table()`, `read.csv()` and `write.csv()`). These functions perform well under a majority of settings but have limitations that have become a source of frustration for modern R users. Most notably, by default, these functions automatically convert strings to factors. This conversion made sense at the time (when the relative efficiency of factors versus strings often mattered). Today, the combination of relatively powerful computers, inexpensive digital storage and fewer computing tasks that use factors make the default conversion of strings to factors unnecessary and cumbersome.

The `readr` functions such as `read_csv` and `read_delim` improve upon legacy functions by keeping input types as is (no conversion to factor), creating a `tibble` rather than a `data.frame` by default (thus tidying console printing) and providing a dramatic improvement in speed – reading text files 4-5 times faster than the original text reading functions.

```
library(readr)
shakespeare <- read_csv("data/shakespeare.csv")
#> Parsed with column specification:
#> cols(
#>   word = col_character(),
#>   word_count = col_integer(),
#>   work = col_character(),
#>   work_date = col_integer()
#> )
shakespeare
#> # A tibble: 164,656 x 4
#>          word word_count              work work_date
```


PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```
#>          <chr>      <int>           <chr>      <int>
#>  1        hive          1 loverscomplaint       1609
#>  2  plaintful           1 loverscomplaint       1609
#>  3         Are          1 loverscomplaint       1609
#>  4        Than          1 loverscomplaint       1609
#>  5    attended          1 loverscomplaint       1609
#>  6        That          7 loverscomplaint       1609
#>  7    moisture          1 loverscomplaint       1609
#>  8     praised          1 loverscomplaint       1609
#>  9 particular           1 loverscomplaint       1609
#> 10        tend          1 loverscomplaint       1609
#> # ... with 164,646 more rows
```

Wrangle and reshape with `dplyr` and `tidyr`

The core data wrangling functions in the tidyverse were developed by Hadley Wickham in 2014 and included in the packages `dplyr` (Wickham & Francois 2016) and `tidyr` (Wickham 2017). The `dplyr` package contains verbs that correspond to the most common data manipulation tasks and act as replacements for base R data manipulation functions like `aggregate()`, `subset()`, `sort/order()` and `merge()` (Wickham & Grolemund 2017). The `tidyr` package can be used to reshape data from wide to long or from long to wide, providing a tidyverse alternative to the base R function `reshape()`. Both packages were designed with ease-of-use and clarity in mind with consistent inputs and outputs – the goal is that even those unfamiliar with R could recognize what code is doing.

In addition to simplifying common workflow tasks such as subsetting and arranging data the `dplyr` package also provides a powerful new way of performing analysis on groups. The base solutions for group-level computations often tend to involve chopping data into pieces, working on the parts and gluing them back together. Instead, with `dplyr` we can use `group_by` combined with `mutate` or `do`, for example, as a powerful and streamlined approach to group-level computations.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

We illustrate the use of these packages in our example by summarizing the raw Shakespeare data into word counts by year and then reshaping into wide format. We take advantage of the so-called "pipe" operator, `%>%`, from the `magrittr` package (Bache & Wickham 2014), to eliminate the need to create and name interim objects. The pipe, combined with the consistency of function inputs and outputs (the first argument is always a data frame and the output is likewise a data frame) also allows us to avoid typing an object name more than once. For example, in the following code examples using hypothetical data you can see that without the pipe we have to name two objects and we also need to type `grp` twice – the pipe simplifies the code even in this small example.

```
# Without the pipe
grp <- group_by(data, variable1)
fin <- summarize(grp, avg = mean(variable2))
```

```
# With the pipe
fin <- group_by(data, variable1) %>%
  summarize(avg = mean(variable2))
```

Wrangle with `dplyr`

In the toy example above the extra work is minimal but in a real-world setting the additional naming and typing could be significant. To illustrate the advantages of using `dplyr` + the pipe, we begin our analysis of the Shakespeare data.

In this block, we convert the words to lowercase and then count the number of times a word occurs within each work (e.g., how many times does "question" occur in Hamlet).

```
library(dplyr)


shakespeare <- shakespeare %>%
  mutate(word = str_to_lower(word)) %>%
  group_by(word, work) %>%
  summarize(word_count = sum(word_count))
```

Now we can compute the total number of times a word occurs across all the works of

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

Shakespeare and the number of distinct works each word occurs in.

```
words <- shakespeare %>%
  group_by(word) %>%
  summarize(n = sum(word_count), works = n_distinct(work)) %>%
  arrange(desc(n))

words
#> # A tibble: 26,928 x 3
#>     word     n works
#>    <chr> <int> <int>
#>  1   the 29801    42
#>  2   and 27529    42
#>  3     i 21029    42
#>  4    to 20957    42
#>  5    of 18514    42
#>  6     a 15370    42
#>  7   you 14010    42
#>  8    my 12936    42
#>  9    in 11722    42
#> 10  that 11519    42
#> # ... with 26,918 more rows
```

As you can see, the most common words are not a good representation of the breadth of Shakespeare's considerable vocabulary so we will eliminate less interesting words like "the" and "to" using a list of stop words from the `tidytext` package (discussed later). We will also focus on words that don't appear in every work and contain at least four letters.

As part of the `filter()` example below we use the useful `anti_join()` function which is essentially the opposite of the `inner_join()` function – it identifies and returns all records in the left-side table that do **not** occur in the right-side table based on a common ID (in this case the `word` variable).

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```r
words <- words %>%
  anti_join(tidytext::stop_words, by = "word") %>%
  filter(works < 42, nchar(word) > 4) %>%
  arrange(desc(n))
words
#> # A tibble: 23,890 x 3
#>      word     n works
#>     <chr> <int> <int>
#>  1  enter  2406    39
#>  2  henry  1311    13
#>  3  speak  1194    40
#>  4 exeunt  1061    37
#>  5  queen  1005    35
#>  6  night   933    41
#>  7  death   933    41
#>  8 father   868    40
#>  9  scene   825    38
#> 10 master   803    39
#> # ... with 23,880 more rows
```

This sort of filtering and minor transformation is a vital early part of every data analysis. Because every function in `dplyr` inputs and outputs data in the same basic format, early exploration can be performed in fast and fluent ways. The pipe frees us from naming unimportant intermediates.

Reshape with `tidyr`

The results from the code above provide a good summary of the relative frequency of words across the entire works of Shakespeare. If we want to dive deeper into the usage of, say, the five most common words we could join the top five records from `words` with our `shakespeare` dataset to get the counts by individual work.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```r
# Our counts by work
head(shakespeare)
#> Source: local data frame [6 x 3]
#> Groups: word [1]
#>
#> # A tibble: 6 x 3
#>    word                  work word_count
#>   <chr>                 <chr>      <int>
#> 1     '          1kinghenryiv         33
#> 2     '          1kinghenryvi         14
#> 3     '          2kinghenryiv         38
#> 4     '          2kinghenryvi         22
#> 5     '          3kinghenryvi         26
#> 6     ' allswellthatendswell         23
```

```r
# Our total counts
head(words)
#> # A tibble: 6 x 3
#>     word     n works
#>    <chr> <int> <int>
#> 1  enter  2406    39
#> 2  henry  1311    13
#> 3  speak  1194    40
#> 4 exeunt  1061    37
#> 5  queen  1005    35
#> 6  night   933    41
```

To do this join we use another novel join function from `dplyr`, `semi_join()`. The `semi_join()` function performs identically to `inner_join()` except that does not keep any columns from the table on the right hand side.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```
shakespeare %>%
  semi_join(head(words, 5), by = "word")
#> Source: local data frame [164 x 3]
#> Groups: word [?]
#>
#> # A tibble: 164 x 3
#>     word                  work word_count
#>    <chr>                 <chr>      <int>
#>  1 enter          1kinghenryiv         63
#>  2 enter          1kinghenryvi         83
#>  3 enter          2kinghenryiv         65
#>  4 enter          2kinghenryvi         84
#>  5 enter          3kinghenryvi         78
#>  6 enter allswellthatendswell         56
#>  7 enter   antonyandcleopatra        112
#>  8 enter           asyoulikeit         54
#>  9 enter        comedyoferrors         40
#> 10 enter            coriolanus         94
#> # ... with 154 more rows
```

This table provides us with the pieces we need to compare occurrences of words by work but this table is visually difficult to interpret. Many data science-related tasks such as this require us to change the shape of our date (long to wide or wide to long). Our data, for example, might be more easily understood if each word were a column – `tidyr` provides tools like `spread()` and `gather()` to perform the reshaping.

In this example, we still perform the the join as above, but we add one more step. We use `spread()` from `tidyr` to create a much more readable "wide" table that makes it easier to visually inspect the relative frequency of words by work. This table makes it easy to see, for example, that the word "henry" occurs frequency in the King Henry works but the word "queen" is much less common.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```r
library(tidyr)

shakespeare %>%
  semi_join(head(words, 5), by = "word") %>%
  spread(word, word_count, fill = 0)
#> # A tibble: 41 x 6
#>                     work enter exeunt henry queen speak
#>  *                 <chr> <dbl>  <dbl> <dbl> <dbl> <dbl>
#>  1          1kinghenryiv    63     28   255     3    29
#>  2          1kinghenryvi    83     36   103    10    18
#>  3          2kinghenryiv    65     32   133     1    40
#>  4          2kinghenryvi    84     40   162   103    22
#>  5          3kinghenryvi    78     34   176   131    45
#>  6 allswellthatendswell    56     26     0     3    42
#>  7   antonyandcleopatra   112     55     0    47    38
#>  8          asyoulikeit    54     27     0     1    28
#>  9       comedyoferrors    40     14     0     0    14
#> 10           coriolanus    94     45     0     1    55
#> # ... with 31 more rows
```

Visualize with ggplot2

No matter how much reshaping and filtering you do, the huge number of different words in the works of Shakespeare does not lend itself well to tabular representation. Instead we can use data visualization to help us make sense of the data.

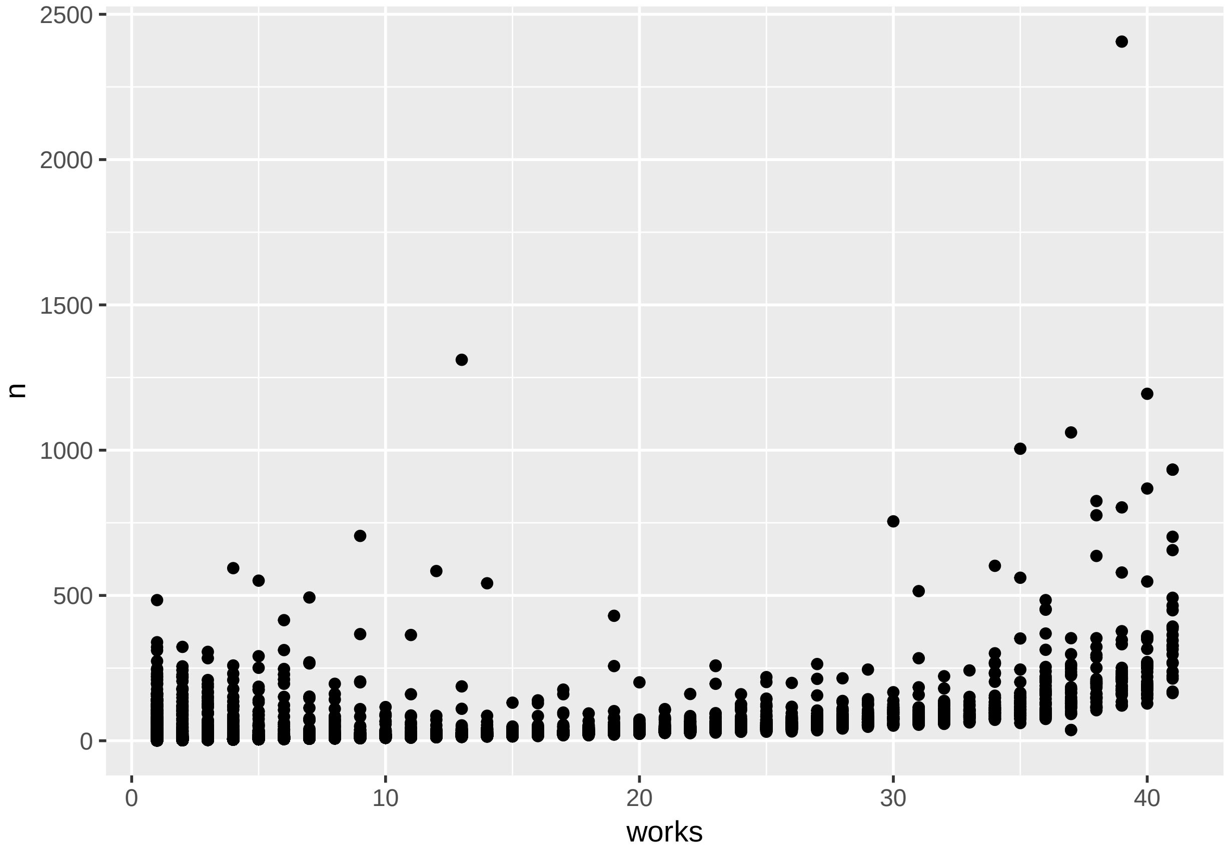
The package `ggplot2` (Wickham 2009), developed in 2007, was designed to efficiently generate complex multi-layered graphics. As the oldest member of the tidyverse, `ggplot2` was developed before the core principles of the tidyverse were well established and thus does not follow all the principles. For example, `ggplot2` uses addition instead of function composition and piping. Addition is a nice metaphor, but does not span the full range of

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

activities that function composition does.

We can explore the relationship between the number of word occurrences across all of Shakespeare's works and the number of works they appear in. We can see from the plot below that, as one would expect, there is a relationship between the two, though this simple plot makes it hard to identify the patterns.

```
ggplot(words, aes(works, n)) +
  geom_point()
```

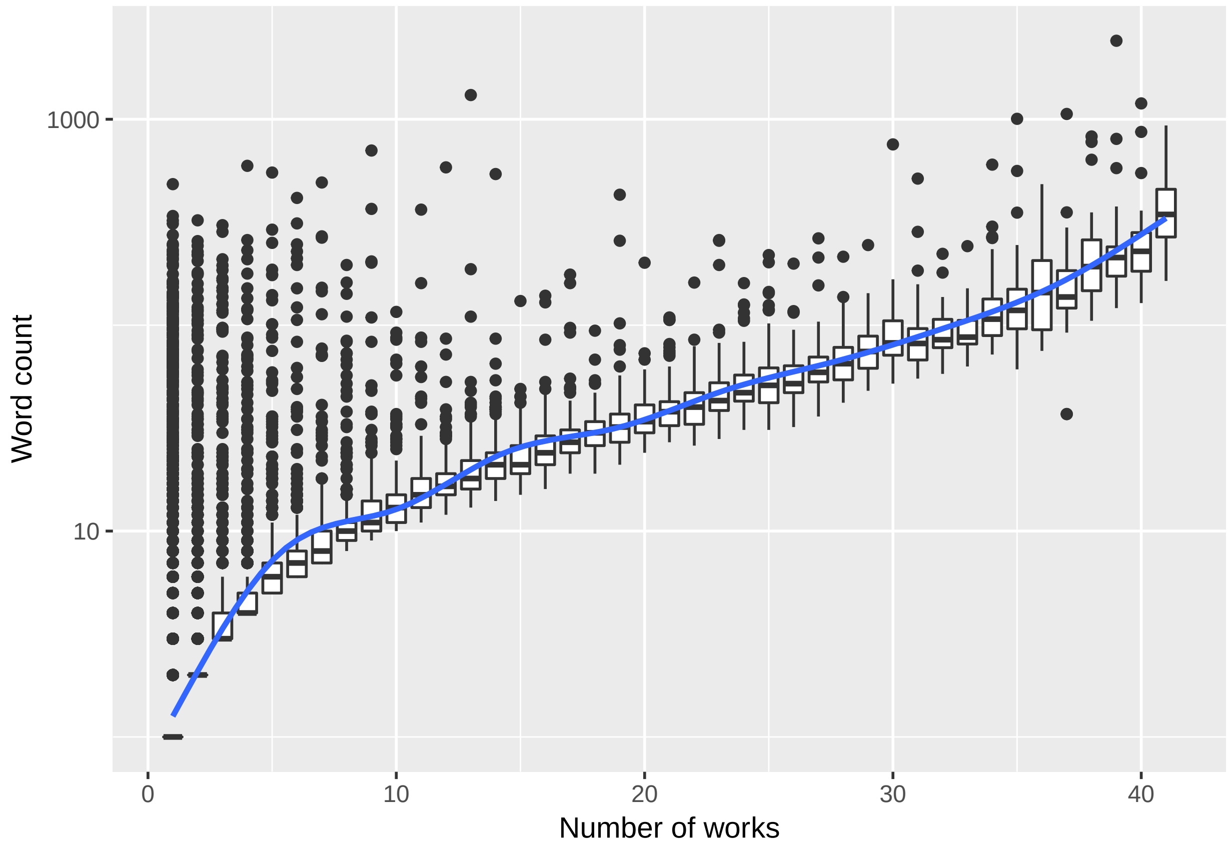


Exploratory visualizations often start out simple, but build in complexity over time. For example, we can improve the previous visualization by log transforming the y axis, using boxplots to summarize the distribution at each unique x, and improving the labels. `ggplot2` makes the process of including multiple layers simpler than base R plotting.

```
ggplot(words, aes(works, n)) +
  geom_boxplot(aes(group = works)) +
  geom_smooth(se = FALSE) +
```


PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```
  scale_y_log10() +
  labs(
    x = "Number of works",
    y = "Word count"
  )
```



Our plot clearly shows that a significant number of words are used only once. We can use dplyr to list them (sorting by string length to show the most interesting).

```
words %>%
  filter(n == 1) %>%
  arrange(desc(str_length(word)))
#> # A tibble: 9,815 x 3
#>                               word     n works
#>                              <chr> <int> <int>
#>  1 honorificabilitudinitatibus     1     1
```

```
#>  2          indistinguishable     1     1
#>  3          anthropophaginian     1     1
#>  4           northumberland's     1     1
#>  5           northamptonshire     1     1
#>  6           interchangeably'     1     1
#>  7           incomprehensible     1     1
#>  8            unreconciliable     1     1
#>  9            uncomprehensive     1     1
#> 10            uncompassionate     1     1
#> # ... with 9,805 more rows
```

Using `tidytext` to analyze text

The core tidyverse includes packages like `readr`, `tidyr`, `dplyr`, and `ggplot2` that facilitate tasks that are required in almost every analysis. But the tidyverse is also a broader platform that others can build on to provide more specialized tools. Developing these specialized tools using the core tidyverse principles means that the tools can more easily be adopted into workflows by a wider range of analysts. An example, well-suited to this sample analysis, is `tidytext` (Silge & Robinson 2016). Created by Julia Silge and David Robinson, `tidytext` provides a set of "tidy" tools for handling text data.

In the tables above, we can glean a general sense of word counts and we could see that some words occur often in many different works and there are some words that seem to occur mostly in a single work. In order to quantify this pattern we can take advantage of a statistic known as "term frequency-inverse document frequency" (TF-IDF). This statistic, which essentially shows how important a word is to a particular document, can be computed within the context of the tidyverse using the `bind_tf_idf()` function from `tidytext`.

```
library(tidytext)

shakespeare_tf_idf <- shakespeare %>%
  bind_tf_idf(word, work, word_count) %>%
```


PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```
  arrange(desc(tf_idf))

shakespeare_tf_idf
#> Source: local data frame [147,219 x 6]
#> Groups: word [26,928]
#>
#> # A tibble: 147,219 x 6
#>          word           work word_count         tf      idf     tf_idf
#>         <chr>          <chr>      <int>      <dbl>    <dbl>      <dbl>
#>  1    macbeth        macbeth        311 0.01686642 3.737670 0.06304112
#>  2     hamlet         hamlet        484 0.01491709 3.737670 0.05575517
#>  3   syracuse comedyoferrors        232 0.01418006 3.737670 0.05300039
#>  4     dromio comedyoferrors        221 0.01350773 3.737670 0.05048744
#>  5 antipholus comedyoferrors        219 0.01338549 3.737670 0.05003054
#>  6      timon  timonofathens        322 0.01622902 3.044522 0.04940962
#>  7       iago        othello        361 0.01285796 3.737670 0.04805880
#>  8      romeo romeoandjuliet        322 0.01229383 3.737670 0.04595028
#>  9    othello        othello        339 0.01207437 3.737670 0.04513000
#> 10   rosalind    asyoulikeit        274 0.01186815 3.737670 0.04435923
#> # ... with 147,209 more rows
```

The result provides the term frequency (`tf`) which is the frequency **within a work** and the inverse document frequency (`idf`) which is the inverse of the frequency **across works**. The `tf_idf` is just `tf * idf`.

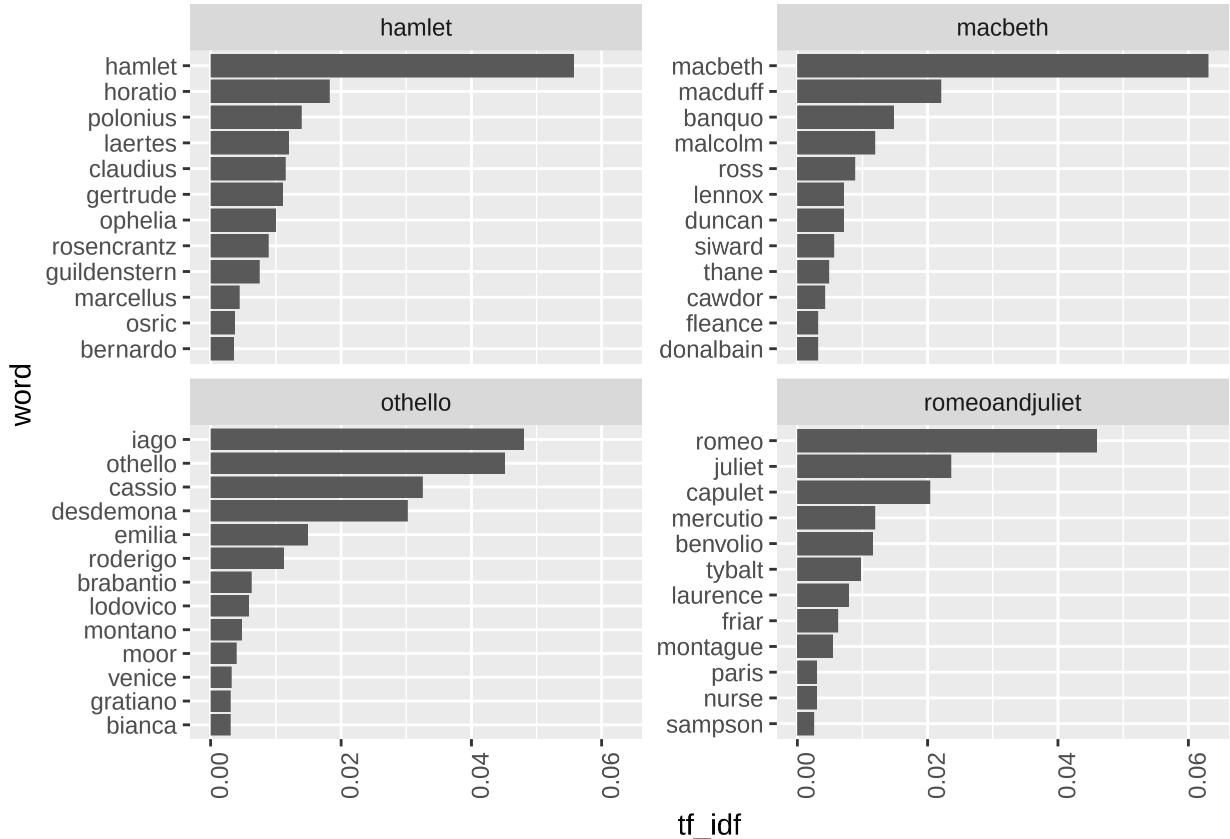
We can see from our results that high TF-IDF words tend to be words unique to and common within a particular document, such as the names of protagonists. By keeping the word data in a tidy form, the data can be further manipulated using `dplyr` to find the highest TF-IDF words within particular works, and visualized using `ggplot2`.

```
top_tf_idf_words <- shakespeare_tf_idf %>%
  filter(work %in% c("macbeth", "hamlet", "romeoandjuliet", "othello")) %>%
```


PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```r
  group_by(work) %>%
  top_n(12, tf_idf) %>%
  mutate(word = reorder(word, tf_idf))

ggplot(top_tf_idf_words, aes(word, tf_idf)) +
  geom_bar(stat = "identity") +
    coord_flip() +
  facet_wrap(~ work, scales = "free_y") +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```



The `tidytext` package also provides datasets for sentiment analysis in the form of tidy data frames. For example, the `sentiments` dataset provided by `tidytext` includes word sentiment scores and categorizations in three lexicons. Here we use the lexicon of Finn Arup Nielsen (AFINN) to measure sentiment, making extensive use of the tidyverse tools in the process.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```r
# Limit to one lexicon
AFINN <- sentiments %>%
  filter(lexicon == "AFINN") %>%
  select(word, score) %>%
  arrange(desc(score))

# The most "positive" words in the lexicon
AFINN
#> # A tibble: 2,476 x 2
#>            word score
#>           <chr> <int>
#>  1 breathtaking     5
#>  2       hurrah     5
#>  3  outstanding     5
#>  4       superb     5
#>  5     thrilled     5
#>  6      amazing     4
#>  7      awesome     4
#>  8    brilliant     4
#>  9     ecstatic     4
#> 10     euphoric     4
#> # ... with 2,466 more rows
```

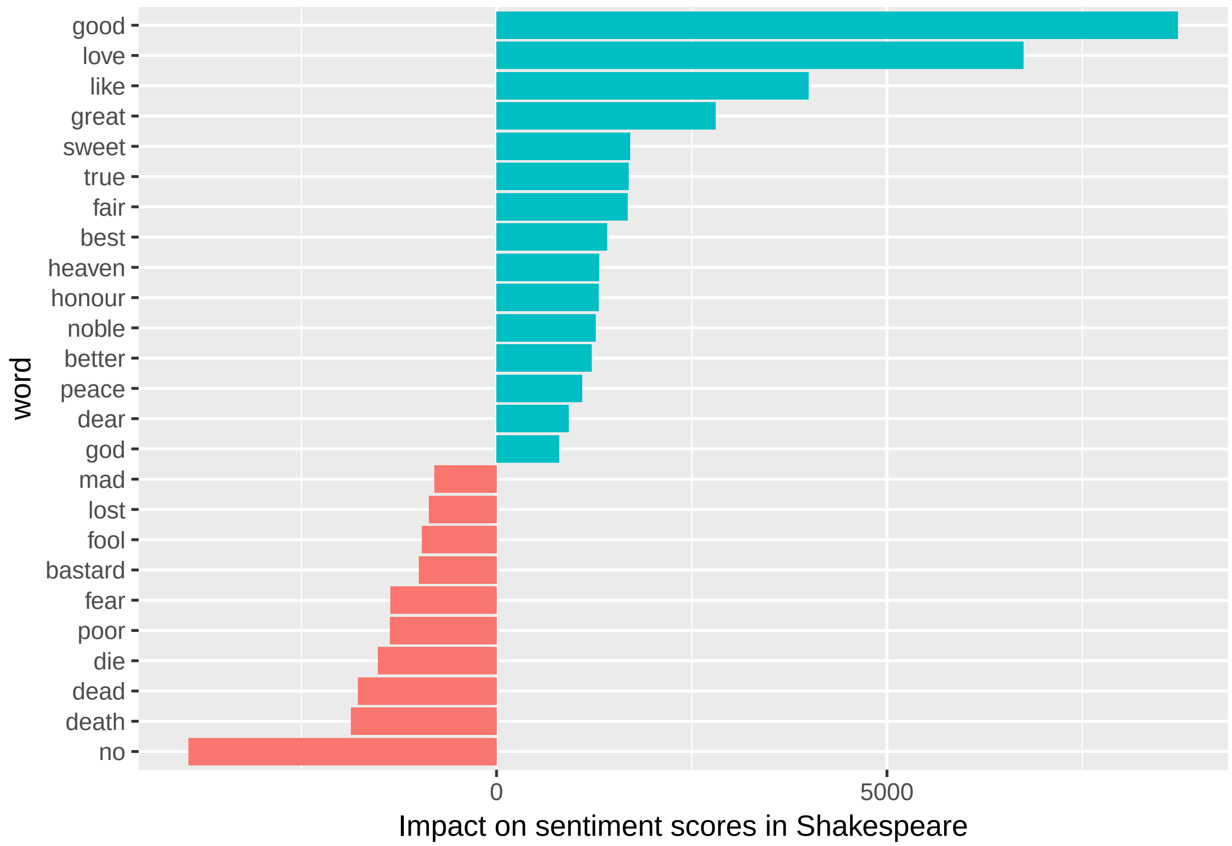
We can then join sentiment scores to the words in Shakespeare and see which words had the greatest influence on sentiment, postive or negative, in the work.

```r
shakespeare_sentiment <- shakespeare %>%
  inner_join(AFINN, by = "word")

shakespeare_sentiment %>%
  group_by(word) %>%
```


PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

```
summarize(impact = sum(word_count * score)) %>%
top_n(25, abs(impact)) %>%
mutate(word = reorder(word, impact)) %>%
ggplot(aes(word, impact, fill = impact > 0)) +
geom_bar(stat = "identity", show.legend = FALSE) +
coord_flip() +
ylab("Impact on sentiment scores in Shakespeare")
```



Conclusion

R is an incredible tool for data analytics that has withstood the test of time. Developed (initially as S) more than 40 years ago at Bell Labs to help researchers tackle statistical challenges, R is now used by millions worldwide in ways the initial developers, John Chambers, Ross Ihaka and Robert Gentleman, could not have imagined. R is being used to

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

analyze Facebook and Twitter posts, trends in Airbnb bookings, perform image processing and conduct spatial analysis.

Much of the language has remained essentially unchanged in the nearly 20 years since the initial version of R was released to the public but there is a need for a smoother, more efficient and more readable pipeline for modern R workflows.

Tidy tools, those that do one thing well, accept inputs and produce outputs with a full workflow in mind and take advantage of "glue" to move outputs from one task to another, simplify the process of solving complex tasks. Packages such as `dplyr`, `tidyr`, `ggplot2` and `tidytext` and other members of the tidyverse were developed to fulfill these needs.

References

Bache, S. M. & Wickham, H. (2014), *magrittr: A Forward-Pipe Operator for R.* R package version 1.5.
URL: *https://CRAN.R-project.org/package=magrittr*

Google (2017), 'Google cloud platform sample tables: Shakespeare'.
URL: *https://cloud.google.com/bigquery/sample-tables*

Knuth, D. E. (1973), *The art of computer programming. Volume 1, Fundamental algorithms*, Addison-Wesley.

Raymond, E. S. (2003), *The art of Unix programming*, Addison-Wesley Professional.

Silge, J. & Robinson, D. (2016), 'tidytext: Text mining and analysis using tidy data principles in r', *JOSS* **1**(3).
URL: *http://dx.doi.org/10.21105/joss.00037*

Wickham, H. (2009), *ggplot2: Elegant Graphics for Data Analysis*, Springer-Verlag New York.
URL: *http://ggplot2.org*

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017

Wickham, H. (2017), *tidyr: Easily Tidy Data with spread and gather Functions.* R package version 0.6.1.
URL: *https://CRAN.R-project.org/package=tidyr*

Wickham, H. & Francois, R. (2016), *dplyr: A Grammar of Data Manipulation.* R package version 0.5.0.
URL: *https://CRAN.R-project.org/package=dplyr*

Wickham, H. & Grolemund, G. (2017), *R for Data Science*, O'Reilly.

Wickham, H. et al. (2014), 'Tidy data', *Journal of Statistical Software* **59**(10), 1–23.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3180v1 | CC BY 4.0 Open Access | rec: 23 Aug 2017, publ: 23 Aug 2017