broom: an R package for turning messy model outputs into a tidy format

by David Robinson

Abstract The concept of "tidy data" offers a powerful framework for structuring data to ease manipulation, modeling and visualization. However, most R functions, both those built-in and those found in third-party packages, produce output that is not tidy, and that is therefore difficult to reshape, recombine, and otherwise manipulate. Here I introduce the **broom** package, which turns the output of model objects into tidy data frames that are suited to further analysis, manipulation, and visualization with input-tidy tools. **broom** defines the `tidy`, `augment`, and `glance` generics, which arrange a model into three levels of tidy output respectively: the component level, the observation level, and the model level. I provide examples to demonstrate how these generics work with tidy tools to allow analysis and modeling of data that is divided into subsets, to perform simulations that investigate the effect of replications and input parameter variation, and to recombine results from bootstrap replicates.

Introduction

A common observation is that more of the data scientist's time is occupied with data cleaning, manipulation, and "munging" than it is with actual statistical modeling (Rahm and Do, 2000; Dasu and Johnson, 2003). Thus, the development of tools for manipulating and transforming data is necessary for efficient and effective data analysis. One important choice for a data scientist working in R is how data should be structured, particularly the choice of dividing observations across rows, columns, and multiple tables.

The concept of "tidy data," introduced by Wickham (2014a), offers a set of guidelines for organizing data in order to facilitate statistical analysis and visualization. In short, data can be described as "tidy" if it is represented in a table following three rules:

- Each variable forms one column
- Each observation forms one row
- Each type of observational unit forms one table

This framework makes it easy for analysts to reshape, combine, group and otherwise manipulate data. Packages such as **ggplot2**, **dplyr**, and many built-in R modeling and plotting functions require the input to be in a tidy form, so keeping the data in this form allows multiple tools to be used in sequence in a seamless analysis pipeline (Wickham, 2009; Wickham and Francois, 2014).

Tools are classified as "messy-output" if their output does not fit into this framework. Unfortunately, the majority of R modeling tools, both from the built-in **stats** package and those in common third party packages, are messy-output. This means the data analyst must tidy not only the original data, but the results at each intermediate stage of an analysis. Wickham (2014a) describes this problem succinctly:

> While model inputs usually require tidy inputs, such attention to detail doesn't carry over to model outputs. Outputs such as predictions and estimated coefficients aren't always tidy. For example, in R, the default representation of model coefficients is not tidy because it does not have an explicit variable that records the variable name for each estimate, they are instead recorded as row names. In R, row names must be unique, so combining coefficients from many models (e.g., from bootstrap resamples, or subgroups) requires workarounds to avoid losing important information. This knocks you out of the flow of analysis and makes it harder to combine the results from multiple models. I'm not currently aware of any packages that resolve this problem.

The **broom** package is an attempt to solve this issue, by bridging the gap from untidy outputs of predictions and estimations to create tidy data that is easy to manipulate with standard tools. It centers around three S3 methods, `tidy`, `augment`, and `glance`, that each take an object produced by R statistical functions (such as `lm`, `t.test`, and `nls`) or by popular third-party packages (such as **glmnet**, **survival**, **lme4**, and **multcomp**) and convert it into a tidy data frame without rownames (Friedman et al., 2010; Therneau, 2014; Bates et al., 2014; Hothorn et al., 2008). These outputs can then be used with input-tidy tools such as **dplyr** or **ggplot2**, or downstream statistical tests.

broom should be distinguished from packages such as **reshape2** and **tidyr**, which rearrange and reshape data frames into different forms Wickham (2007b, 2014b). Those packages perform essential tasks in tidy data analysis but focus on manipulating data frames in one specific format into another.

In contrast, **broom** is designed to take data that is not in a data frame (sometimes not anywhere close) and convert it to a tidy data frame.

How broom Works

Common Attributes of Messy Outputs

The process of tidying a model output must be tailored to each object, since each model object is messy in its own way. However, one can recognize some common features of messy-output models. Many are the same issues described in Wickham (2014a) as features of messy datasets, such as having variables stored in column names. The following tendencies, however, are more specific to model outputs:

- *Relevant information is stored in rownames.* Examples include the coefficient names in a regression coefficient matrix or ANOVA table. Since R does not allow row names to be duplicated, this prevents one from combining the results of multiple analyses or bootstrap replications.
- *Column names are inconsistent and inconvenient.* For instance, p-values for each coefficient produced by the `summary.lm` function are stored in a column named `Pr(>|t|)`. Besides being incomparable to other model outputs that use `p.value`, `pvalue`, or just `p`, this column name is difficult to work with due to the use of punctuation: for instance, it cannot easily be extracted using '`$`'.
- *Information is computed by downstream functions.* Many models need to be run through additional processing steps, often the `summary` generic, to produce the statistical information desired. The steps required can be inconsistent even between similar models. For example, the `anova` function produces an ANOVA table immediately, while an object from `aov` must be run through `summary.aov` to produce the table.
- *Information is printed rather than returned.* Some packages and functions place relevant calculations into the `print` method, where they are displayed using `cat`. An example is the calculation of a p-value from an F-statistic in `print.summary.lm`. This requires either examining the source of the `print` method to extract the code of interest or capturing and parsing the printed output.
- *Vectors are stored separately rather than combined in a table.* For example, residuals and fitted values are both returned by many model outputs, but are accessed with the `residuals` and `fitted` generics. In other cases multiple vectors of the same length are included as separate elements in a named list. This requires recombining these vectors into a data frame to use them with input-tidy tools.

Each of these obstacles can be individually overcome by the knowledgeable programmer, but in combination they serve as a massive inconvenience. Time spent reforming these model outputs into the desired structure breaks the natural flow of analysis and takes attention away from examining and questioning the data. They further invite inconsistency, where different analysts will approach the same task in very different ways, which makes it time-consuming to understand and adapt shared code. Defining a standard "tidy form" of any model output, and collecting tools for constructing such a form from an R object, makes analyses easy, efficient, and consistent.

Three ways of tidying models

Statistical models are complex objects: they often contain information about multiple levels of an analysis. For instance, consider the object produced by a regression of n observations with p predictors. Wickham 2007a notes that such a model contains computed statistics at the following levels:

- **coefficient level**: estimate, standard error, T-statistic, p-value: p values per model
- **residual level**: fitted value, residual, Cook's standard deviation: n values per model
- **model level**: R^2, adjusted R^2, F-statistic and p-value, residual standard error: 1 value per model

As a simple demonstration of these three levels, consider a linear regression on the built-in `mtcars` dataset, predicting the fuel efficiency of cars (`mpg`, measured in miles-per-gallon) based on the weight of the cars (`wt`, measured in thousands of pounds) and the speed/acceleration (`qsec`, the time in seconds to drive a quarter of a mile).

```
fit <- lm(mpg ~ wt + qsec, data = mtcars)
summary(fit)
```

```
##
## Call:
## lm(formula = mpg ~ wt + qsec, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.3962 -2.1431 -0.2129  1.4915  5.7486
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  19.7462     5.2521   3.760 0.000765 ***
## wt           -5.0480     0.4840 -10.430 2.52e-11 ***
## qsec          0.9292     0.2650   3.506 0.001500 **
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
##
## Residual standard error: 2.596 on 29 degrees of freedom
## Multiple R-squared:  0.8264,Adjusted R-squared:  0.8144
## F-statistic: 69.03 on 2 and 29 DF,  p-value: 9.395e-12
```

The summary of this regression shows that it contains coefficient-level information, including the estimate, standard error, and p-value, about each of the intercept, `wt`, and `qsec` terms. The `fit` object also contains observation-level information, such as the residuals and fitted values. Finally, it contains model-level information in the form of R^2, adjusted R^2, an F statistic, and a p-value for the whole dataset.

Values computed at each of these three levels have different dimensionalities and observations: there is no natural way to combine a calculation of R^2 with the estimates of coefficient values, or with a vector of residuals, in a single data frame. In the tidy data terminology, each level forms a separate "observational unit" and therefore deserves its own table. To generate these three separate tidy data frames, **broom** provides three S3 methods that do three distinct kinds of tidying.

- **tidy** constructs a data frame that summarizes the model's statistical components, such as coefficients and p-values for each term in a regression. While Wickham (2007a) calls this the "coefficient level" when applied to regressions, in other models it could have a different interpretation, such as per-cluster information in clustering applications, or per-test information for multiple comparison functions. I therefore refer to this as the **component level**.
- **augment** add columns to the original data that was modeled, thus working at the **observation level**. This includes predictions, residuals, and cluster assignments. By convention, each column that augment adds starts with '.' to ensure it does not conflict with existing columns.
- **glance** constructs a concise one-row summary of the **model level** values. This typically contains values such as R^2, adjusted R^2, residual standard error, deviance, or cross validation accuracy that are computed once for the entire model.

For this regression, these three methods would give the outputs:

```
library(broom)
tidy(fit)

##          term  estimate std.error  statistic      p.value
## 1 (Intercept) 19.746223 5.2520617   3.759709 7.650466e-04
## 2          wt -5.047982 0.4839974 -10.429771 2.518948e-11
## 3        qsec  0.929198 0.2650173   3.506179 1.499883e-03

head(augment(fit))

##            .rownames  mpg    wt  qsec       .hat   .sigma     .resid
## 1        AMC Javelin 15.2 3.435 17.30 0.03524115 2.565582 -3.2815295
## 2 Cadillac Fleetwood 10.4 5.250 17.98 0.17681412 2.640475  0.4487031
## 3         Camaro Z28 13.3 3.840 15.41 0.09660408 2.627824 -1.3809126
## 4  Chrysler Imperial 14.7 5.345 17.42 0.18444635 2.352379  5.7486123
## 5         Datsun 710 22.8 2.320 18.61 0.06072636 2.595763 -2.5272788
## 6   Dodge Challenger 15.5 3.520 16.87 0.04244725 2.609209 -2.1528959
##     .fitted   .se.fit     .cooksd .std.resid
```

```
## 1 18.481530 0.4873703 0.020163967 -1.2868649
## 2  9.951297 1.0916727 0.002598066  0.1904917
## 3 14.680913 0.8069223 0.011163038 -0.5596199
## 4  8.951388 1.1149850 0.453212444  2.4519001
## 5 25.327279 0.6397681 0.021742528 -1.0044379
## 6 17.652896 0.5348830 0.010611604 -0.8474375

glance(fit)

##   r.squared adj.r.squared    sigma statistic      p.value df    logLik
## 1 0.8264161     0.8144448 2.596175  69.03311 9.394765e-12  3 -74.36025
##        AIC      BIC deviance df.residual
## 1 156.7205 162.5834 195.4636          29
```

These three methods appear across many analyses, though some model objects may have only one or two of these methods defined. (For example, there is no sense in which a Student's T test or correlation test generates information about each observation, and therefore no `augment` method exists). Indeed, that these three levels must be combined into a single S3 object is a common reason that model outputs are not tidy.

Conventions

In order to maintain consistency, we attempt to follow some conventions regarding the structure of returned data. This lets users know what to expect from tidy output and makes it easy to recombine results across different kinds of analyses.

- The output of the `tidy`, `augment` and `glance` functions is always a data frame.
- The output never has rownames. This ensures that you can combine it with other tidy outputs without fear of losing information (since rownames in R cannot contain duplicates).
- Some column names are kept consistent, so that they can be combined across different models and so the user knows what to expect. The examples below are not all the possible column names, nor will all tidy output contain all or even any of these columns.

`tidy` methods are the most flexible of the generics. Each row in a `tidy` output typically represents a well-defined component of the model, such as one coefficient in a regression, one statistical test in a multiple comparisons analysis, or one cluster or class in a clustering or classification algorithm. This meaning varies across model objects but is usually self-evident from a simple examination of the object to be tidied. The one thing each row cannot represent is a row in the initial data (for that, use the `augment` method).

Common column names of `tidy` outputs include:

term the term in a regression or model that is being estimated

p.value this spelling was chosen (over common alternatives such as `pvalue`, `PValue`, or `pval`) to be consistent with functions in R's `stats` package

statistic a test statistic, usually the one used to compute the p-value. Combining these across many sub-groups is a reliable way to perform bootstrap or permutation testing

estimate estimate of an effect size, slope, or other value

std.error standard error of the estimate

conf.low the low end of a confidence interval on the `estimate`

conf.high the high end of a confidence interval on the `estimate`

df degrees of freedom

The `augment` generic adds columns to the original data, which means each row in an `augment` output matches the corresponding row in the data. `augment` typically takes two arguments: `x` (the object) and `data` (the original data). If the `data` argument is missing, `augment` attempts to reconstruct the data from the model, which may or may not be possible. Newly added column names begin with '`.`' to avoid overwriting columns in the original data, and if the original data contained rownames, `augment` turns them into a column called `.rownames`.

Common column names of `augment` output include:

.fitted the predicted values, typically calculated from the model with `fitted`.

.resid prediction residuals, typically calculated from the model with `residuals`

.cluster cluster assignments

Finally, the output from `glance` is always a one-row data frame. The only exception is that `glance(NULL)` returns an empty data frame (this is useful in combination with **dplyr**'s `failwith`, so that models that raise errors can be discarded from the output). Common column names include:

r.squared the fraction of variance explained by the model

adj.r.squared R^2 adjusted based on the degrees of freedom

sigma the square root of the estimated variance of the residuals

df.residual number of residual degrees of freedom

logLik the log-likelihood of the data given the model

AIC Akaike's Information Criterion, an assessment of a model fit typically calculated by `AIC`

BIC Bayesian Information Criterion, an assessment of a model fit typically calculated by `BIC`

Case Studies

Here I show how **broom** can be widely useful across data science applications. I give four examples: regressions on each gene in a molecular biology dataset, a simulation of k-means clustering, a demonstration of bootstrapping, and an analysis using the **survival** package. Each example highlights some of the advantages of keeping model outputs tidy.

These examples assume familiarity with the **dplyr**, **tidyr**, and **ggplot2** packages, since they are powerful implementations of tidy tools (though it is possible to take advantage of **broom** without these packages). Note that in contrast to these packages, **broom** functions take up only a small part of the code, which is by design. **broom** is meant to serve as a simple bridge between the modeling tools provided by R and the downstream analyses enabled by tidy tools, requiring minimal experience with or configuration of the package.

Case study: analyzing gene expression per gene with brooom and dplyr

As an illustrative example I demonstrate an analysis on the gene expression dataset of Brauer et al. (2008). This dataset contains measurements of mRNA abundance levels of each of the $\sim$ 5200 genes in *Sacchromyces cerevisiae*, or baker's yeast, in varying conditions to examine the effect of growth rate on each gene's expression. The yeast were grown using one of six compounds as the limiting nutrient, each of which is coded by a letter: glucose (G), leucine (L), ammonia (N), phosphate (P), sulfate (S), and uracil (U). For each of these nutrients, the growth rate was varied between a dilution rate of 0.05 (slow) to .3 (fast).

```
brauer <- read.delim("input/DataSet1.tds")
head(brauer, 3)

##         GID       YORF
## 1 GENE1331X A_06_P5820
## 2 GENE4924X A_06_P5866
## 3 GENE4690X A_06_P1834
##                                                                                                  NAME
## 1           SFB2       || ER to Golgi transport || molecular function unknown || YNL049C || 1082129
## 2                 || biological process unknown || molecular function unknown || YNL095C || 1086222
## 3 QRI7       || proteolysis and peptidolysis || metalloendopeptidase activity || YDL104C || 1085955
##   GWEIGHT G0.05  G0.1 G0.15  G0.2 G0.25  G0.3 N0.05 N0.1 N0.15  N0.2 N0.25
## 1       1 -0.24 -0.13 -0.21 -0.15 -0.05 -0.05  0.20 0.24 -0.20 -0.42 -0.14
## 2       1  0.28  0.13 -0.40 -0.48 -0.11  0.17  0.31 0.00 -0.63 -0.44 -0.26
## 3       1 -0.02 -0.27 -0.27 -0.02  0.24  0.25  0.23 0.06 -0.66 -0.40 -0.46
##    N0.3 P0.05  P0.1 P0.15  P0.2 P0.25 P0.3 S0.05  S0.1 S0.15  S0.2 S0.25
## 1  0.09 -0.26 -0.20 -0.22 -0.31  0.04 0.34 -0.51 -0.12  0.09  0.09  0.20
## 2  0.21 -0.09 -0.04 -0.10  0.15  0.20 0.63  0.53  0.15 -0.01  0.12 -0.15
## 3 -0.43  0.18  0.22  0.33  0.34  0.13 0.44  1.29 -0.32 -0.47 -0.50 -0.42
##    S0.3 L0.05 L0.1 L0.15  L0.2 L0.25  L0.3 U0.05  U0.1 U0.15  U0.2 U0.25
## 1  0.08  0.18 0.18  0.13  0.20  0.17  0.11 -0.06 -0.26 -0.05 -0.28 -0.19
## 2  0.32  0.16 0.09  0.02  0.04  0.03  0.01 -1.02 -0.91 -0.59 -0.61 -0.17
```

```
## 3 -0.33 -0.30 0.02 -0.07 -0.05 -0.13 -0.04 -0.91 -0.94 -0.42 -0.36 -0.49
##    U0.3
## 1  0.09
## 2  0.18
## 3 -0.47
```

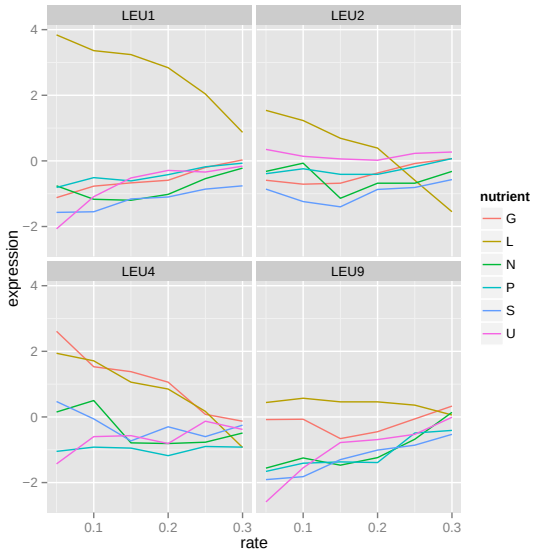
The framework of tidy data requires that each combination of a gene and sample has its own row, and that each attribute of a gene has its own column. As with most "naturally occuring" datasets, the supplementary data provided by Brauer et al 2008 does not follow these standards. However, it can be easily tidied using the **tidyr** and **dplyr** packages.

```
library(dplyr)
library(tidyr)
brauer_tidied <- brauer %>% na.omit() %>%
    separate(NAME, c("name", "BP", "MF", "gene", "ID"), sep = " \\|\\| ") %>%
    filter(gene != "") %>%
    gather(condition, expression, G0.05:U0.3) %>%
    separate(condition, c("nutrient", "rate"), 1, convert = TRUE)

head(brauer_tidied, 3)
```

```
##         GID       YORF       name                            BP
## 1 GENE1331X A_06_P5820 SFB2              ER to Golgi transport
## 2 GENE4924X A_06_P5866               biological process unknown
## 3 GENE4690X A_06_P1834 QRI7       proteolysis and peptidolysis
##                              MF    gene      ID GWEIGHT nutrient rate
## 1    molecular function unknown YNL049C 1082129       1        G 0.05
## 2    molecular function unknown YNL095C 1086222       1        G 0.05
## 3 metalloendopeptidase activity YDL104C 1085955       1        G 0.05
##   expression
## 1      -0.24
## 2       0.28
## 3      -0.02
```

Following the guidelines for tidy data, we first split up the YORF column, which contains multiple attributes of each gene, into multiple columns. Since the column names of the form `G0.05` are in fact storing observations rather than variable names, we gather them into a column, which we split into `nutrient` and `rate`. This produces a useful tidy output that makes visualization and modeling straightforward. For example, we can filter for the genes in a single biological process (using the `BP` column) and graph its expression as a function of the growth rate.

```
library(ggplot2)
brauer_tidied %>% filter(BP == "leucine biosynthesis") %>%
    ggplot(aes(rate, expression, color = nutrient)) + geom_line() +
    facet_wrap(~ name)
```



Up until this stage we have succeeded in working entirely with tidy data tools. However, suppose we wish to perform a linear regression within each combination of gene and nutrient. This regression would produce values for each gene, including coefficients, standard errors, and t-statistics, and we would like to recombine them in a way that includes information from each gene in a large tidy dataset. The `tidy.lm` method, combined with `do` from **dplyr**, makes this easy.

```
library(broom)
regressions <- brauer_tidied %>% filter(gene %in% gene[1:250]) %>%
    group_by(gene, nutrient) %>%
    do(tidy(lm(expression ~ rate, .)))
head(regressions)

## Source: local data frame [6 x 7]
## Groups: gene, nutrient
##
##    gene nutrient        term   estimate std.error  statistic     p.value
## 1 Q0045        G (Intercept) -0.8513333 0.1678563 -5.0717977 0.007121455
## 2 Q0045        G        rate  4.8457143 0.8620305  5.6212793 0.004924029
## 3 Q0045        L (Intercept) -0.4440000 0.2961343 -1.4993197 0.208167251
## 4 Q0045        L        rate  0.1942857 1.5208054  0.1277519 0.904510494
## 5 Q0045        N (Intercept) -1.5060000 0.3136267 -4.8018869 0.008636256
## 6 Q0045        N        rate  5.1771429 1.6106381  3.2143427 0.032454564
```

Having the analyses, and not just the original data, in this tidied form allows us to construct downstream plots and analyses using standard tidy tools. We can use **dplyr**'s data manipulation operations to filter out the intercept term and perform Benjamini-Hochberg FDR control within each set of p-values (Benjamini and Hochberg, 1995):

```
coefs <- coefs %>%
    filter(term != "(Intercept)") %>% select(-term) %>%
    group_by(nutrient) %>%
    mutate(p.adjusted = p.adjust(p.value, method = "BH"))
```

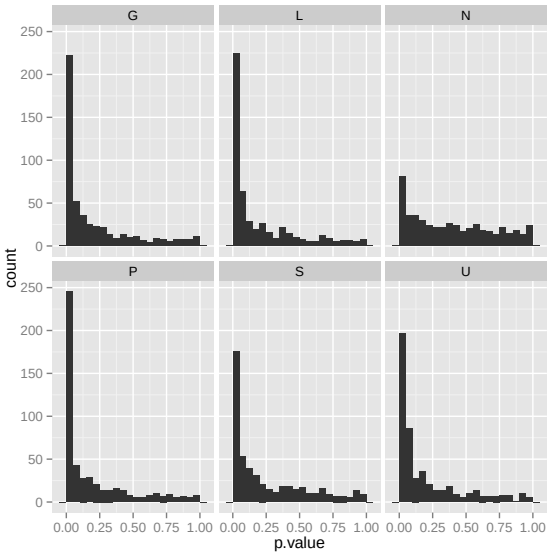
After this processing, we can perform a summary to show the number of genes found to have a statistically significant trend within each limiting nutrient:

```
coefs %>% summarize(significant = sum(p.adjusted < .05)) %>% kable()
```

nutrient	significant
G	144
L	102
N	9
P	168
S	38
U	3

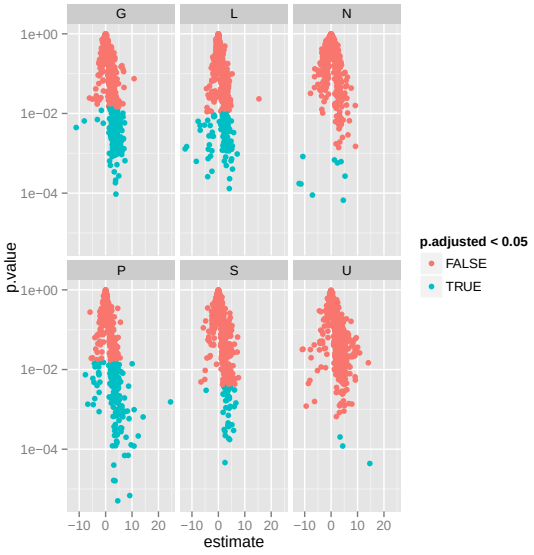
This tidied model output is friendly to many visualizations. For instance, we can construct a p-value histogram, a useful tool for characterizing behavior across many hypothesis tests (see e.g. Storey and Tibshirani 2003).

```
ggplot(coefs, aes(p.value)) + geom_histogram(binwidth = .05) +
    facet_wrap(~ nutrient)
```



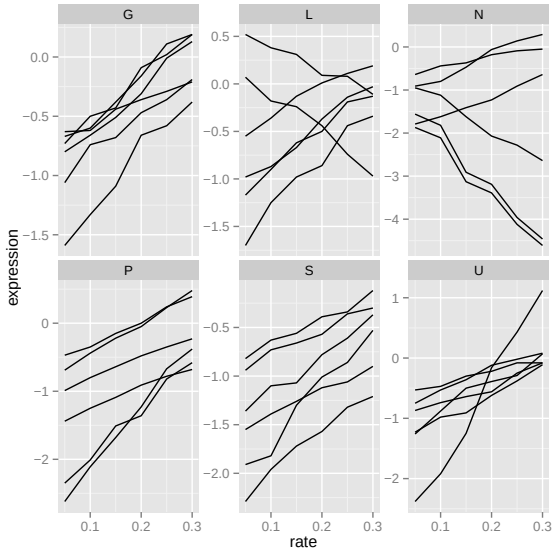
We could also create a volcano plot, which compares the estimated effect size (slope) of each hypothesis to its significance (p-value) (Allison et al., 2006).

```
ggplot(coefs, aes(estimate, p.value, color = p.adjusted < .05)) +
    geom_point() + scale_y_log10() + facet_wrap(~ nutrient)
```



For a more complicated visualization that examines the details of specific genes, one could extract the 6 most statistically significant genes of each condition, and merge it with the original tidied data to produce a plot of those genes and their rate-dependent expression:

```
topgenes <- coefs %>% arrange(p.value) %>% slice(1:6)
head(topgenes, 10)

## Source: local data frame [10 x 7]
## Groups: nutrient
##
##       gene nutrient  estimate std.error statistic      p.value p.adjusted
## 1  YOR266W        G  3.885714 0.2464441  15.76712 9.453313e-05 0.02228524
## 2  YPL103C        G  3.714286 0.2783821  13.34240 1.824416e-04 0.02228524
## 3  YGL097W        G  3.800000 0.2960051  12.83761 2.122496e-04 0.02228524
## 4  YIL093C        G  4.988571 0.4135396  12.06311 2.708169e-04 0.02228524
## 5  YBR220C        G  3.257143 0.2867600  11.35843 3.425810e-04 0.02228524
## 6  YNL268W        G  1.811429 0.1755690  10.31747 4.978948e-04 0.02228524
## 7  YPR138C        L  4.137143 0.2846265  14.53534 1.302775e-04 0.02549124
## 8  YGR084C        L  4.257143 0.3385443  12.57485 2.301700e-04 0.02549124
## 9  YLR090W        L -4.045714 0.3315927 -12.20085 2.590519e-04 0.02549124
## 10 YPR145W        L -2.440000 0.2166410 -11.26287 3.540532e-04 0.02549124

merged <- topgenes %>% inner_join(brauer_tidied, by = c("gene", "nutrient"))
ggplot(merged, aes(rate, expression, group = gene)) + geom_line() +
    facet_wrap(~ nutrient, scales = "free_y")
```



These downstream analyses would have been equally straightforward even if the per-gene model were more complicated, such as a generalized linear model, an analysis of variance (ANOVA), a spline regression, or a nonlinear least squares fit, since **broom** provides tidying methods for each of these classes. Similarly, this output could be incorporated into other input-tidy tools, such as R's built-in plotting, or **ggvis** for interactive visualizations (RStudio Inc., 2014).

Case study: a simulation to examine k-means clustering

Because the **broom** package makes it possible to recombine many analyses, it is also well suited to simulation for the purpose of exploring a model's behavior and robustness. Tidied model outputs can easily be recombined across varying input parameters, or across simulation replications, and lend themselves to downstream analysis and visualization. In this simulation we examine a k-means clustering problem, and observe how the results are affected by the choice of the number of clusters and by the within-cluster variation in the data.

K-means clustering divides a set of points in an n-dimensional space into k centers by assigning each point to the cluster with the nearest of k designated centers. This is well suited for clustering

problems where the data is assumed to be approximately multivariate Gaussian distributed around each cluster. The **stats** package provides an implementation of k-means clustering (based, by default, on the algorithm of Hartigan and Wong (1979)), which requires the choice of k to be made beforehand.

We start by exploring the effect of the choice of k on the behavior of the clustering algorithm. We first generate random 2-dimensional data with three centers, around which points are distributed according to a standard multivariate Gaussian distribution:

```
library(dplyr)

set.seed(2014)
centers <- data.frame(oracle = factor(1:3),
                      size = c(100, 150, 50),
                      x1 = c(5, 0, -3),
                      x2 = c(-1, 1, -2))

kdat <- centers %>%
    group_by(oracle) %>%
    do(data.frame(x1 = rnorm(.$size[1], .$x1[1]),
                  x2 = rnorm(.$size[1], .$x2[1])))
```

The `inflate` function, which **broom** provides, expands a dataset such that it repeats its original data once for each factorial combination of parameters.

```
d <- data.frame(a = 1:3, b = 8:10)
d %>% inflate(x = c("apple", "orange"), y = c("car", "boat"))

## Source: local data frame [12 x 4]
## Groups: x, y
##
##         x    y a  b
## 1   apple boat 1  8
## 2   apple boat 2  9
## 3   apple boat 3 10
## 4   apple  car 1  8
## 5   apple  car 2  9
## 6   apple  car 3 10
## 7  orange boat 1  8
## 8  orange boat 2  9
## 9  orange boat 3 10
## 10 orange  car 1  8
## 11 orange  car 2  9
## 12 orange  car 3 10
```

This is useful for performing tidy data modeling with varying input parameters. In this case, we perform clustering on the same data (`kdat`) with each value of k in `1:9`. Note that to reduce the role of randomness in the clustering process, we set `nstart = 5` to the `kmeans` function.

```
kclusts <- kdat %>% inflate(k = 1:9) %>% group_by(k) %>%
    do(clust = kmeans(select(., x1, x2), .$k[1], nstart = 5))
kclusts

## Source: local data frame [9 x 2]
## Groups: <by row>
##
##   k       clust
## 1 1 <S3:kmeans>
## 2 2 <S3:kmeans>
## 3 3 <S3:kmeans>
## 4 4 <S3:kmeans>
## 5 5 <S3:kmeans>
## 6 6 <S3:kmeans>
## 7 7 <S3:kmeans>
## 8 8 <S3:kmeans>
## 9 9 <S3:kmeans>
```

There are three levels at which we can examine a k-means clustering. As is true of regressions and other statistical models, each of these levels describes a separate observational unit, and therefore is extracted by a different **broom** generic.

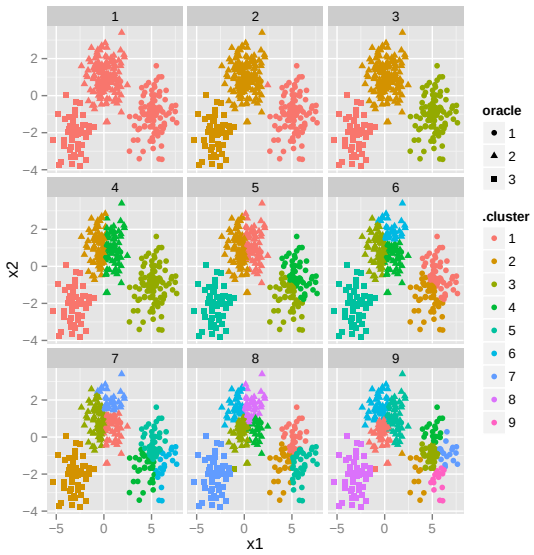
- **Component level**: The centers, size, and within sum-of-squares for each cluster; computed by `tidy`
- **Observation level**: The assignment of each point to a cluster; computed by `augment`
- **Model level**: The total within sum-of-squares and between sum-of-squares; computed by `glance`

We extract tidied versions of these three levels from each of the 9 clustering objects produced in the simulation, then recombine each level into a single large table containing the results for all values of k.

```
kc <- kclusts %>% group_by(k)
clusters <- kc %>% do(tidy(.$clust[[1]]))
assignments <- kc %>% do(augment(.$clust[[1]], kdat))
clusterings <- kc %>% do(glance(.$clust[[1]]))
```

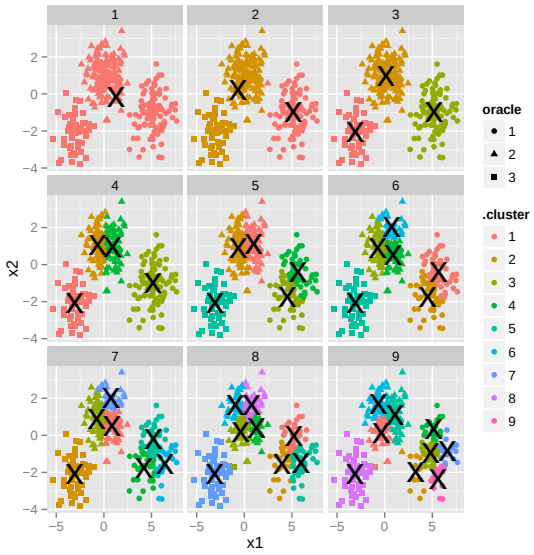
The `assignments` object, generated using the `augment` method on each clustering, combines the cluster assignments across all values of k, thus allowing for simple visualization using faceting.

```
head(assignments)

## Source: local data frame [6 x 5]
## Groups: k
##
##   k oracle       x1         x2 .cluster
## 1 1      1 4.434320  0.5416470        1
## 2 1      1 5.321046 -0.9412882        1
## 3 1      1 5.125271 -1.5802282        1
## 4 1      1 6.353225 -1.6040549        1
## 5 1      1 3.712270 -3.4079344        1
## 6 1      1 5.322555 -0.7716317        1

p1 <- ggplot(assignments, aes(x1, x2)) +
    geom_point(aes(color = .cluster, shape = oracle)) +
    facet_wrap(~ k)
p1
```



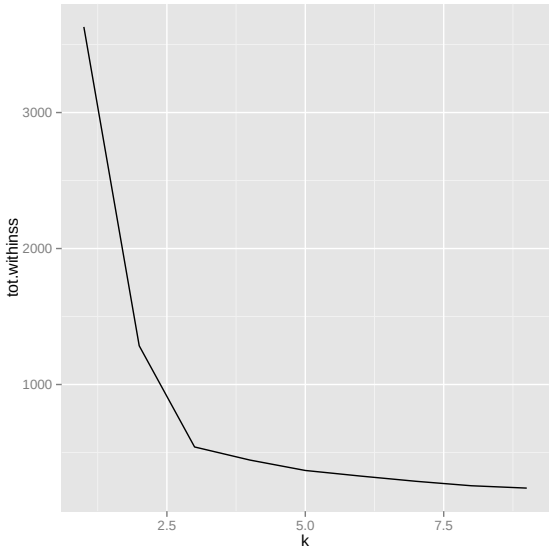
We can use the results from the `tidy` outputs, which provide per-cluster information, to add the estimated center of each cluster to the plot.

```
p2 <- p1 + geom_point(data = clusters, size = 10, shape = "x")
p2
```



Finally, we can examine the per-model data using the `glance` outputs. Of particular interest is the total within cluster sum-of-squares. The decrease in this within-cluster variation as clusters are added to the model can serve as a criterion for choosing k.

```
ggplot(clusterings, aes(k, tot.withinss)) + geom_line()
```



The variance within each estimated cluster decreases as k increases, but one can notice a bend (or "elbow") around the true value of $k = 3$. This bend indicates that additional clusters beyond the third have little value in terms of explaining the variation in the data. Tibshirani et al. (2002) provides a more mathematically rigorous interpretation and implementation of a method to choose k based on this profile. Note that all three methods of tidying data provided by **broom** play a separate role in visualization and analysis of these simulations.

While a simulation varying k is useful, we may also be interested in how robust the algorithm is when the original data is altered. For example, the points around each center are generated from a multivariate normal distribution with standard deviation $\sigma = 1$. If this standard deviation increases, making the cloud of points around each center more disperse, we would expect k-means clustering to be less accurate. We also wish to know the role of random variation, and therefore will perform

50 replicates of each simulation. These goals can be achieved with some small modifications to the previous simulation. We first generate the data using the same centers, but with multiple values of σ and multiple independent replications:

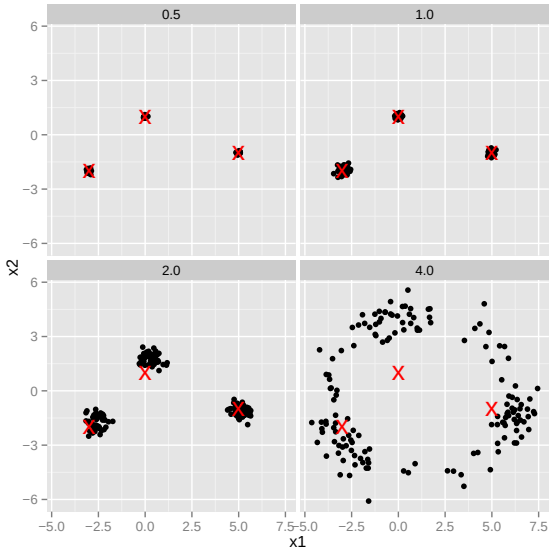
```
set.seed(2014)

kdat_sd <- centers %>%
    inflate(sd = c(.5, 1, 2, 4), replication = 1:50) %>%
    group_by(oracle, sd, replication) %>%
    do(data.frame(x1 = rnorm(.$size[1], .$x1[1], .$sd[1]),
                  x2 = rnorm(.$size[1], .$x2[1], .$sd[1])))
```

We then perform the k-means clustering 9 times, choosing a different value of k each time. We then extract the tidied, augmented, and glanced forms, which in this case are combined across all factorial combinations of `k`, `sd`, and `replication`. One could easily extend the simulation to alter other parameters such as the number and distribution of true cluster centers, or the `nstart` parameter.

```
kclusts_sd <- kdat_sd %>% inflate(k = 1:9) %>% group_by(k, sd, replication) %>%
    do(dat = (.), clust = kmeans(select(., x1, x2), .$k[1], nstart = 5))

ksd <- kclusts_sd %>% group_by(k, sd, replication)
clusters_sd <- ksd %>% do(tidy(.$clust[[1]]))
assignments_sd <- ksd %>% do(augment(.$clust[[1]], .$dat[[1]]))
glances_sd <- ksd %>% do(glance(.$clust[[1]]))
```

One interesting question is whether the cluster centers were estimated accurately in each simulation. The estimated centers are included in the recombined `tidy` output, which can be visualized alongside the true centers (red X's) separately for each value of σ:

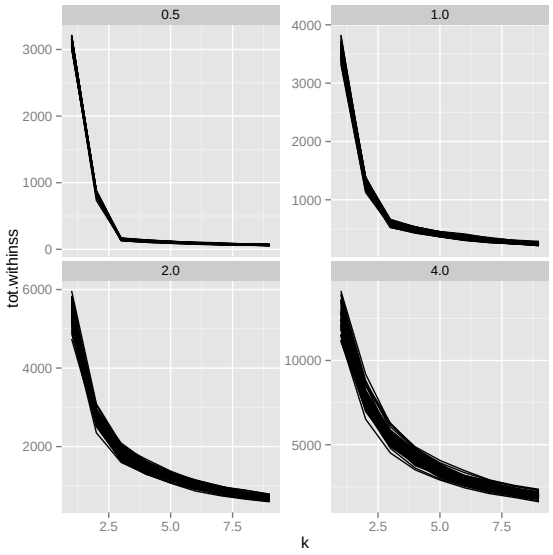
```
clusters_sd %>% filter(k == 3) %>%
    ggplot(aes(x1, x2)) + geom_point() +
    geom_point(data = centers, size = 7, color = "red", shape = "x") +
    facet_wrap(~ sd)
```



We can see that the centers are estimated very accurately for $\sigma = .5, 1$, less accurately for $\sigma = 2$, and rather inaccurately for $\sigma = 4$. Also notably, the center estimates for $\sigma = 4$ are systematically biased "outward" for two of the three clusters, as an artifact of the greater variation.

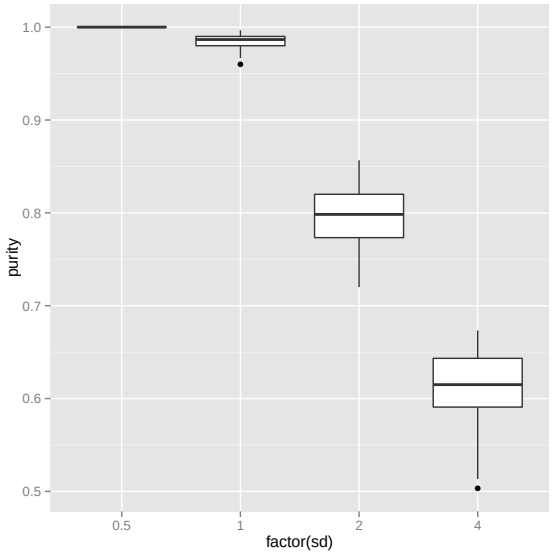
We can also see how the total within-sum-of-squares graph appears across all replications, and how it varies when changing the value of σ.

```
ggplot(glances_sd, aes(k, tot.withinss, group = replication)) +
    geom_line() + facet_wrap(~ sd, scales = "free_y")
```



We can observe from this that the choice of k based on the total within sum-of-squares profile becomes more difficult as σ increases, since the bend at $k = 3$ becomes less distinct. We may also want to measure the cluster purity, and see how the accuracy depends on σ, focusing on the cases where we (correctly) set $k = 3$. This requires some processing but can be done entirely in **dplyr** operations.

```
accuracies <- assignments_sd %>% filter(k == 3) %>%
    count(replication, sd, oracle, .cluster) %>%
    group_by(replication, sd, .cluster) %>%
    summarize(correct = max(n), total = sum(n)) %>%
    group_by(replication, sd) %>%
    summarize(purity = sum(correct) / sum(total))

ggplot(accuracies, aes(factor(sd), purity)) + geom_boxplot()
```



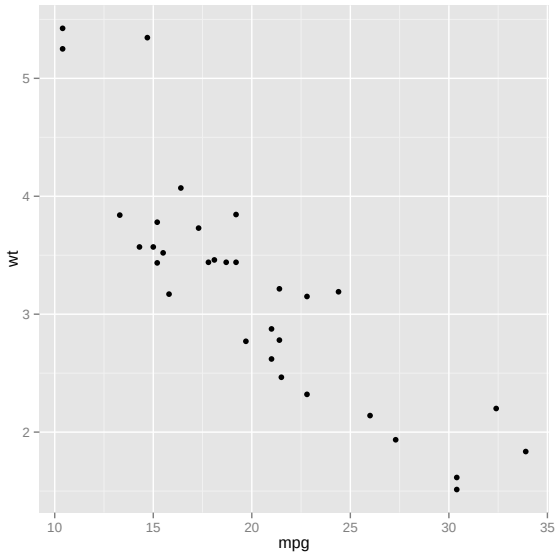
We can see that, as one might expect, the classification accuracy decreases on average as the residual standard deviation increases and the clusters get more disperse. We can see from these examples that combining the models from simulations into a tidied form thus lends itself well to many kinds of exploratory analyses and experiments.

Case study: bootstrapped confidence and prediction intervals

One advantage of working with tidy data is that it can easily be recombined across replicates or analyses, which makes it well suited to bootstrapping and permutation tests. Bootstrapping consists of randomly sampling observations from a dataset with replacement, then performing the same analysis individually on each bootstrapped replicate. The variation in the resulting value is then a reasonable approximation of the standard error of the estimate (Efron and Tibshirani, 1994).

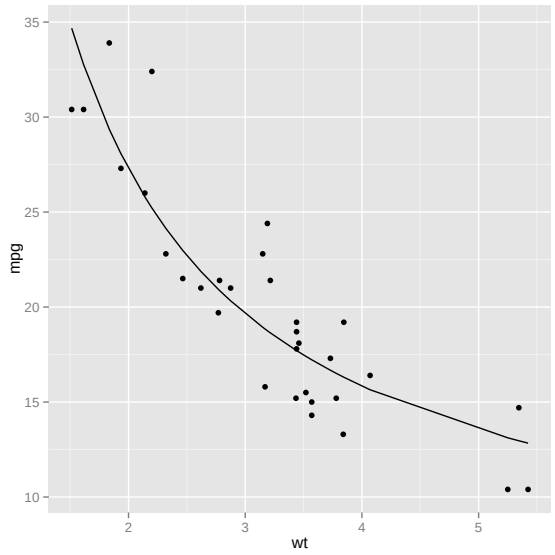
Suppose we wish to fit a nonlinear model to the weight/mileage relationship in the `mtcars` dataset, which comes built-in to R.

```
library(ggplot2)
data(mtcars)
ggplot(mtcars, aes(mpg, wt)) + geom_point()
```



We might use the method of nonlinear least squares (the built-in `nls` function) to fit a model.

```
nlsfit <- nls(mpg ~ k / wt + b, mtcars, start=list(k=1, b=0))
summary(nlsfit)

##
## Formula: mpg ~ k/wt + b
##
## Parameters:
##   Estimate Std. Error t value Pr(>|t|)
## k   45.829      4.249  10.786 7.64e-12 ***
## b    4.386      1.536   2.855  0.00774 **
## ---
## Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
##
## Residual standard error: 2.774 on 30 degrees of freedom
##
## Number of iterations to convergence: 1
## Achieved convergence tolerance: 2.877e-08

ggplot(mtcars, aes(wt, mpg)) + geom_point() + geom_line(aes(y=predict(nlsfit)))
```



While this does provide a p-value and confidence intervals for the parameters, these are based on model assumptions that may not hold in real data. Bootstrapping is a popular method for providing confidence intervals and predictions that are more robust to the nature of the data.

The **broom** package provides a **bootstrap** function that in combination with **dplyr** and a tidying method makes bootstrapping straightforward. `bootstrap(.data,B)` wraps a data table so that the next do step occurs B times, each time operating on the data resampled with replacement. Within each of these do applications, we construct a tidied version of a nonlinear least squares fit.

```
set.seed(2014)
bootnls <- mtcars %>% bootstrap(500) %>%
    do(tidy(nls(mpg ~ k / wt + b, ., start=list(k=1, b=0))))
```

This produces a summary of the coefficients from each replication, combined into one data frame:

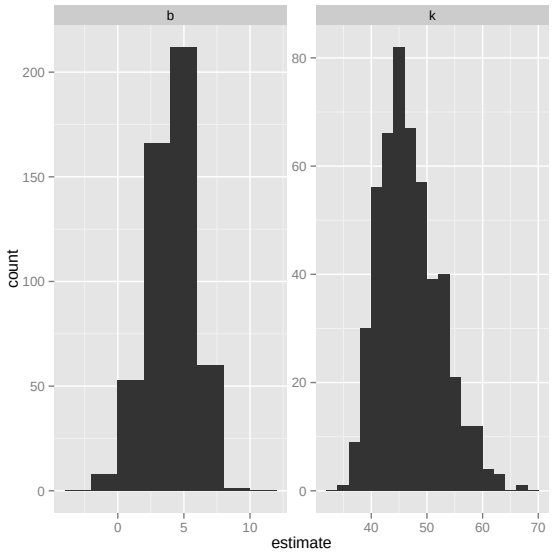
```
head(bootnls)

## Source: local data frame [6 x 6]
## Groups: replicate
##
##   replicate term  estimate std.error  statistic      p.value
## 1         1    k 46.632502  4.026016 11.5827898 1.343891e-12
## 2         1    b  4.360637  1.538267  2.8347730 8.129583e-03
## 3         2    k 54.182476  4.964976 10.9129374 5.756829e-12
## 4         2    b  1.004868  1.897196  0.5296599 6.002460e-01
## 5         3    k 43.257212  3.564860 12.1343382 4.222953e-13
## 6         3    b  4.833510  1.297909  3.7240748 8.101899e-04
```

We can then calculate confidence intervals by considering the quantiles of the bootstrapped estimates. This is often referred to as the percentile method of bootstrapping, though it is not the only way to construct a confidence interval from bootstrap replicates.

```
alpha = .05
bootnls %>% group_by(term) %>%
    summarize(conf.low = quantile(estimate, alpha / 2),
              conf.high = quantile(estimate, 1 - alpha / 2))

## Source: local data frame [2 x 3]
##
##   term  conf.low conf.high
## 1    b  0.214338  6.952325
## 2    k 38.492700 59.024352
```

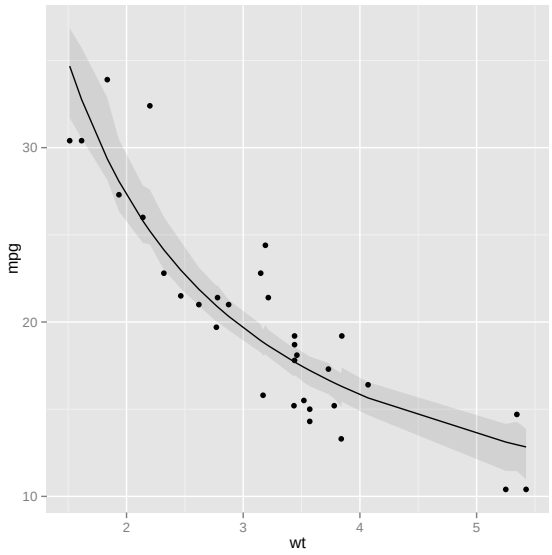
We can instead use histograms to get a more detailed idea of the uncertainty in each estimate:

```
library(ggplot2)

ggplot(bootnls, aes(estimate)) + geom_histogram(binwidth = 2) +
    facet_wrap(~ term, scales="free")
```



Finally, we can visualize the uncertainty in the actual curve using `augment` on each replication. We can then summarize the quantiles within each time point to produce bootstrap confidence intervals at each point.

```
set.seed(2014)

bootnls <- mtcars %>% bootstrap(500) %>%
    do(augment(nls(mpg ~ k / wt + b, ., start=list(k=1, b=0)), .))

alpha = .05
bootnls_bytime <- bootnls %>% group_by(wt) %>%
    summarize(conf.low = quantile(.fitted, alpha / 2),
              conf.high = quantile(.fitted, 1 - alpha / 2),
              median = median(.fitted))
head(bootnls_bytime)

## Source: local data frame [6 x 4]
##
##      wt conf.low conf.high   median
## 1 1.513 31.69528  36.84963 33.63675
## 2 1.615 30.53525  35.73523 32.38457
## 3 1.835 28.14591  32.87095 29.93906
## 4 1.935 26.35532  30.45919 28.14018
## 5 2.140 24.53684  27.83956 25.84737
## 6 2.200 24.44731  27.59607 25.75246

ggplot(mtcars, aes(wt)) + geom_point(aes(y = mpg)) +
    geom_line(aes(y = .fitted), data = augment(nlsfit)) +
    geom_ribbon(aes(ymin = conf.low, ymax = conf.high), data = bootnls_bytime, alpha = .1)
```

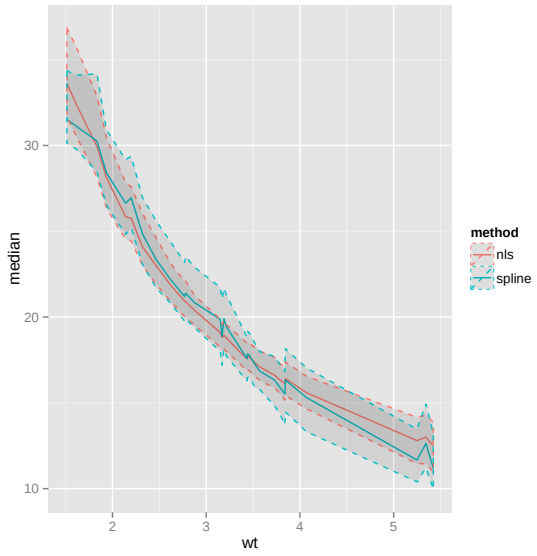


This bootstrapping approach could be applied to any prediction function for which an `augment` method is defined. For example, we could use the built-in **splines** package to predict the curve using a natural cubic spline basis.

```
library(splines)
bootspline <- mtcars %>% bootstrap(500) %>%
    do(augment(lm(mpg ~ ns(wt, 4), .), .))
```

Since the bootstrap results are in the same format as the NLS fit, it is easy to combine them and then compare them on the same axis.

```
bootnls$method <- "nls"
bootspline$method <- "spline"
allboot <- rbind_all(list(bootnls, bootspline))

allboot_bytime <- allboot %>% group_by(wt, method) %>%
    summarize(conf.low = quantile(.fitted, alpha / 2),
              conf.high = quantile(.fitted, 1 - alpha / 2),
              median = median(.fitted))

ggplot(allboot_bytime, aes(wt, median, color = method)) +
    geom_line() +
    geom_ribbon(aes(ymin = conf.low, ymax = conf.high), lty = 2, alpha = .1)
```



It is worth noting that the tidy approach to bootstrapping may not necessarily be the most computationally efficient option for all modeling approaches. Some bootstrapping problems can be simplified to matrix operations that allow more very efficient processing and storage. Framing bootstrapping as a problem of recombining many tidy outputs is useful, however, for its simplicity and universality, as these same idioms can be applied to boostrap any model with a `tidy` method.

Case study: visualizing, comparing and bootstrapping survival curves

The **broom** package also provides tidying methods for several popular third party packages. One example of this is the **survival** package, which provides methods for survival models. Here we show how the approaches for visualization, simulation, and boostrapping discussed in previous sections can be applied to enrich a survival analysis.

The **survival** package provides the `lung` dataset, which records the survival of patients with advanced lung cancer over time. We use the `coxph` function to fit a Cox proportional hazards regression model to model the likelihood of survival based on the demographic features of age and sex, and use `survfit` to construct a survival curve based on these hazards (Therneau and Grambsch, 2000).

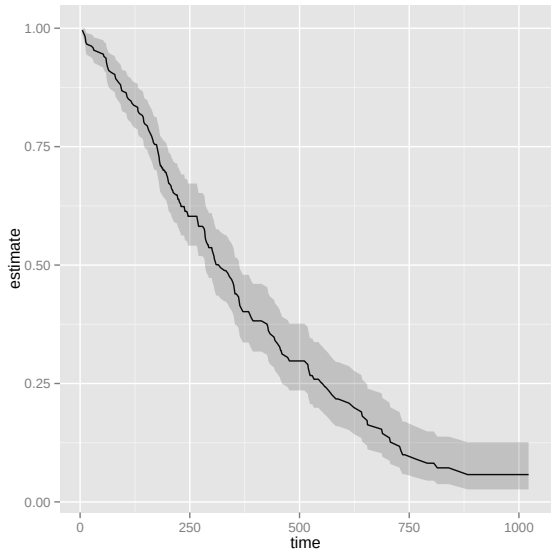
```
library(survival)
coxfit <- coxph(Surv(time, status) ~ age + sex, lung)
sfit <- survfit(coxfit)
```

broom contains tidying methods for both the proportional hazards model and the resulting survival curve fit. The tidied version of the proportional hazards model contains the estimates and significance of each term in the regression, while the tidied survival curve constructs one row for each time point in the model and estimates the survival rate at that point.

```
tidy(coxfit)

##   term    estimate   std.error statistic     p.value     conf.low
## 1  age  0.01704533 0.009223273  1.848078 0.064591012 -0.001031952
## 2  sex -0.51321852 0.167457962 -3.064760 0.002178445 -0.841430092
##     conf.high
## 1  0.03512262
## 2 -0.18500694

head(tidy(sfit))

##   time n.risk n.event n.censor  estimate   std.error conf.high  conf.low
## 1    5    228       1        0 0.9958207 0.004189316 1.0000000 0.9876776
## 2   11    227       3        0 0.9832688 0.008446584 0.9996823 0.9671247
## 3   12    224       1        0 0.9790670 0.009474977 0.9974188 0.9610529
## 4   13    223       2        0 0.9706383 0.011286679 0.9923495 0.9494021
## 5   15    221       1        0 0.9664127 0.012106494 0.9896182 0.9437513
## 6   26    220       1        0 0.9621802 0.012883740 0.9867862 0.9381877
```

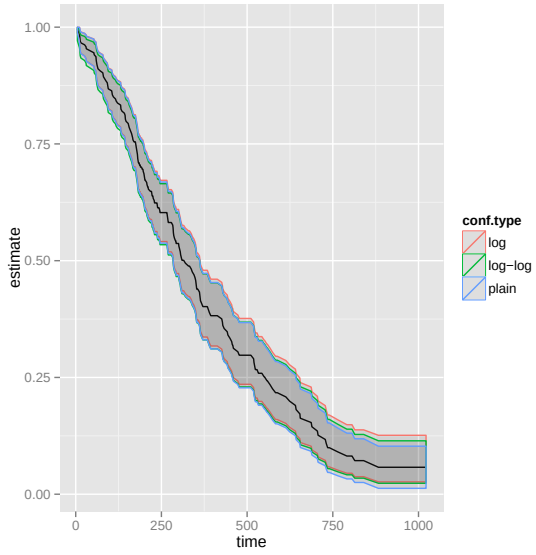
While the **survival** package provides a `plot.survfit` method, **broom**'s tidying method is well suited for producing a data frame that can be plotted in **ggplot2**.

```
library(ggplot2)
ggplot(tidy(sfit), aes(time, estimate)) + geom_line() +
    geom_ribbon(aes(ymin = conf.low, ymax = conf.high), alpha = .2)
```



Using **ggplot2** on tidied version rather than the built-in plotting method lets us make decisions about the values that are calculated and displayed, and to take advantage of **ggplot2**'s ability to combine, layer and facet plots intuitively. For example, `survfit` offers three ways to compute confidence intervals. Using the `inflate` function introduced in Case study: a simulation to examine k-means clustering, we can combine tidied versions of those three methods and compare them visually.

```
survfits <- lung %>%
    inflate(conf.type = c("plain", "log", "log-log")) %>%
    do(tidy(survfit(coxph(Surv(time, status) ~ age + sex, .),
                    conf.type = .$conf.type[1])))

ggplot(survfits, aes(time, estimate)) + geom_line() +
    geom_ribbon(aes(ymin = conf.low, ymax = conf.high, color = conf.type),
                alpha = .1)
```

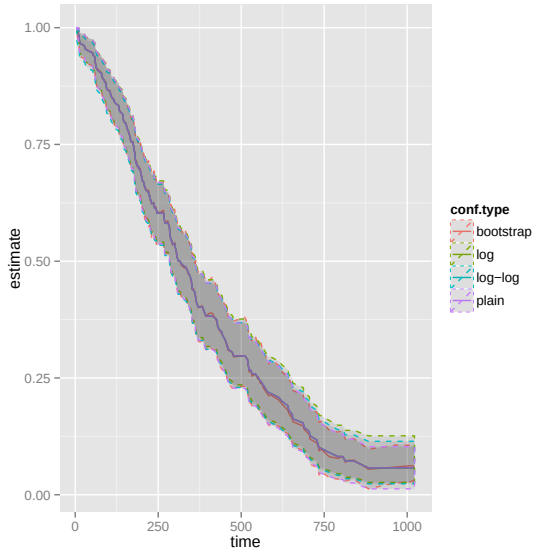


Alternatively, we may prefer to compute confidence intervals using bootstrapping rather than a parametric approach. Similar to the approach in Case study: bootstrapped confidence and prediction intervals, we generate 500 curves from bootstrap resamplings of the data, then summarize to find the 2.5% and 97.5% quantile points at each time point.

```
bootstraps <- lung %>% bootstrap(500) %>%
    do(tidy(survfit(coxph(Surv(time, status) ~ age + sex, .))))

alpha = .05
bootstraps_bytime <- bootstraps %>% group_by(time) %>%
    summarize(conf.low = quantile(estimate, alpha / 2),
              conf.high = quantile(estimate, 1 - alpha / 2),
              estimate = median(estimate))
head(bootstraps_bytime)

## Source: local data frame [6 x 4]
##
##   time  conf.low conf.high  estimate
## 1    5 0.9870881 0.9960334 0.9957560
## 2   11 0.9649739 0.9958697 0.9837897
## 3   12 0.9541879 0.9919727 0.9789447
## 4   13 0.9472856 0.9878533 0.9709267
## 5   15 0.9380506 0.9871614 0.9664878
## 6   26 0.9352451 0.9831075 0.9619900
```

We can then combine it with the parametrically computed confidence intervals and compare them in the same plot.

```
bootstraps_bytime$conf.type = "bootstrap"
survfits <- rbind_list(survfits, bootstraps_bytime)

ggplot(survfits, aes(time, estimate, color = conf.type)) +
    geom_line() +
    geom_ribbon(aes(ymin = conf.low, ymax = conf.high), lty = 2, alpha = .1)
```



By tidying the survival model into a consistent format, **broom** thus allows the user to construct novel visualizations, compare across models or settings, and construct confidence intervals using bootstrapping. A similar approach could be used with many of the other analysis methods that **broom** can tidy.

Discussion

In this paper I introduce the **broom** package, define the `tidy`, `augment`, and `glance` generics, and describe standards for their behavior. I then provide case studies that illustrate how the tidied outputs that **broom** creates are useful in a variety of analyses and simulation. The examples listed here are by

no means meant to be exhaustive, and indeed make use of only a minority of the tidying methods implemented by **broom**. Rather, they are meant to suggest the diversity of visualizations and analyses made possible by tidied model outputs.

While **broom** provides implementations of these generics for many popular R objects, there is no reason that such implementations should be confined to this package. In the future, packages that wish to be output-tidy while retaining the structure of their output object could provide their own `tidy`, `augment`, and `glance` implementations. This would take advantage of each developer's familiarity with his or her software and its goals while offering users a standard tidying language for working with model outputs.

Tidying model outputs is not an exact science, and it is based on a judgment of the kinds of values a data scientist typically wants out of a tidy analysis (for instance, estimates, test statistics, and p-values). Any implementation may lose some of the information that a user wants or keep more information than one needs. It is my hope that data scientists will propose and contribute their own features to expand the functionality of **broom** and to advance the entire available suite of tidy tools.

Acknowledgments

I thank Hadley Wickham for essential comments early in the package's development and for contributing code from **ggplot2** to the **broom** package. I thank Matthieu Gomez and Boris Demeshev for early contributions to the software.

Bibliography

D. B. Allison, X. Cui, G. P. Page, and M. Sabripour. Microarray data analysis: from disarray to consolidation and consensus. *Nature Reviews Genetics*, 7:55–65, 2006. [p8]

D. Bates, M. Maechler, B. Bolker, and S. Walker. *lme4: Linear mixed-effects models using Eigen and S4.*, 2014. URL http://CRAN.R-project.org/package=lme4. R package version 1.1-7. [p1]

Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995. [p7]

M. J. Brauer, C. Huttenhower, E. M. Airoldi, R. Rosenstein, J. C. Matese, D. Gresham, V. M. Boer, O. G. Troyanskaya, and D. Botstein. Coordination of growth rate, cell cycle, stress response, and metabolic activity in yeast. *Molecular Biology of the Cell*, 19(1):352–367, Jan. 2008. [p5]

T. Dasu and T. Johnson. Exploratory data mining and data cleaning. 2003. [p1]

B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*, volume 57. CRC press, 1994. [p15]

J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 2010. [p1]

J. Hartigan and M. Wong. A K-means clustering algorithm. *Applied Statistics*, 28:100–108, 1979. [p10]

T. Hothorn, F. Bretz, and P. Westfall. Simultaneous inference in general parametric models. *Biometrical Journal*, 50(3):346–363, 2008. [p1]

E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng Bull*, 2000. [p1]

RStudio Inc. *ggvis: Interactive grammar of graphics*, 2014. URL http://CRAN.R-project.org/package=ggvis. R package version 0.4. [p9]

J. Storey and R. Tibshirani. Statistical significance for genomewide studies. *Proc. Natl. Acad. Sci.*, 100: 9440–9445, 2003. [p8]

T. Therneau. *A Package for Survival Analysis in S*, 2014. URL URL:http://CRAN.R-project.org/package=survival. R package version 2.37-7. [p1]

T. M. Therneau and P. M. Grambsch. *Modeling survival data: extending the Cox model*. Springer, New York, 2000. [p19]

R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423, Jan. 2002. [p12]

H. Wickham. Meifly: Models explored interactively, 2007a. URL http://had.co.nz/model-vis/2007-jsm.pdf. [p2, 3]

H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(12):1–20, 2007b. URL http://www.jstatsoft.org/v21/i12/. [p1]

H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book. [p1]

H. Wickham. Tidy Data. 59, Aug. 2014a. URL http://www.jstatsoft.org/v59/i10. [p1, 2]

H. Wickham. *tidyr: Easily tidy data with spread and gather functions.*, 2014b. URL https://github.com/hadley/tidyr. R package version 0.1.0.9000. [p1]

H. Wickham and R. Francois. *dplyr: A Grammar of Data Manipulation*, 2014. URL http://CRAN.R-project.org/package=dplyr. R package version 0.3.0.2. [p1]

David Robinson
Princeton University
Carl Icahn Laboratory, Washington Road, Princeton, NJ 08544
United States admiral.david@gmail.com