broom: An R Package for Converting Statistical Analysis Objects Into Tidy Data Frames

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 recombine results from bootstrap replicates, and to perform simulations that investigate the effect of varying input parameters.

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.

Three methods of tidying objects

As a simple demonstration of tidying a statistical object, 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 (accessed by `residuals(fit)`) and fitted values (accessed by `fitted(fit)`). 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. As previously observed by Wickham (2007a), 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`, `augment` and `glance`.

`tidy` constructs a data frame that summarizes the model's statistical components, which we refer to as the *component level*. In a regression such as the above it may refer to coefficient estimates, p-values, and standard errors for each term in a regression. The `tidy` generic is flexible- in other models it could represent per-cluster information in clustering applications, or per-test information for multiple comparison functions.

```
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
```

augment add columns to the original data that was modeled, thus working at the *observation level*. This includes predictions, residuals and prediction standard errors in a regression, and can represent cluster assignments or classifications in other applications. By convention, each new column starts with '.' to ensure it does not conflict with existing columns. To ensure that the output is tidy and can be recombined, rownames in the original data, if present, are added as a column called .rownames.

```
head(augment(fit))
```

```
##            .rownames  mpg    wt  qsec  .fitted   .se.fit      .resid
## 1          Mazda RX4 21.0 2.620 16.46 21.81511 0.6832424 -0.81510855
## 2      Mazda RX4 Wag 21.0 2.875 17.02 21.04822 0.5468271 -0.04822401
## 3         Datsun 710 22.8 2.320 18.61 25.32728 0.6397681 -2.52727880
## 4     Hornet 4 Drive 21.4 3.215 19.44 21.58057 0.6231436 -0.18056924
## 5 Hornet Sportabout 18.7 3.440 17.02 18.19611 0.5120709  0.50388581
## 6            Valiant 18.1 3.460 20.22 21.06859 0.8032106 -2.96858808
##         .hat   .sigma      .cooksd  .std.resid
## 1 0.06925986 2.637300 2.627038e-03 -0.32543724
## 2 0.04436414 2.642112 5.587076e-06 -0.01900129
## 3 0.06072636 2.595763 2.174253e-02 -1.00443793
## 4 0.05761138 2.641895 1.046036e-04 -0.07164647
## 5 0.03890382 2.640343 5.288512e-04  0.19797699
## 6 0.09571739 2.575422 5.101445e-02 -1.20244126
```

Finally, glance constructs a concise one-row summary of the *model level* values. In a regression this typically contains values such as R^2, adjusted R^2, residual standard error, Akaike Information Criterion (AIC), or deviance. In other applications it can include calculations such as cross validation accuracy or prediction error that are computed once for the entire model.

```
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; indeed, the fact that these three levels must be combined into a single S3 object is a common reason that model outputs are not tidy. Importantly, 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). Table 1 shows the tidying methods that are implemented for each statistical object that **broom** can tidy.

Common attributes of messy outputs

Extracting these three levels of tidied contents from a statistical object is not a trivial process, since they are often stored in a messy format that is not conducive to further analysis. The process of tidying a model output must be tailored to each object, since each model object is messy in its own way, but 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 '$' or passed to aes in a **ggplot2** call.
- *Information is computed by downstream functions.* Many models need to be run through additional processing steps, often the summary method, 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.

package	class	tidy	augment	glance
base	data.frame	X		X
	table	X		
stats	anova, aov, density, ftable, manova, pairwise.htest, spec, ts, TukeyHSD	X		
	kmeans, lm, nls	X	X	X
	smooth.spline		X	X
	Arima, htest	X		X
	glm			X
glmnet	cv.glmnet, glmnet	X		X
lfe	felm	X	X	X
lme4	mer, merMod	X	X	X
maps	map	X		
MASS	ridgelm	X		X
multcomp	cld, confint.glht, glht, summary.glht	X		
sp	Line, Lines, Polygon, Polygons, SpatialLinesDataFrame, SpatialPolygons, SpatialPolygonsDataFrame	X		
survival	aareg, cch, pyears, survexp, survfit	X		X
	coxph, survreg	X	X	X
zoo	zoo	X		

Table 1: The statistical objects for which `tidy`, `augment`, and `glance` methods are implemented in **broom** (version 0.3.4).

- *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 objects 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.

Case studies

Here I show how **broom** can be widely useful across data science applications. I give three examples: combining regressions performed on subgroups of data, a demonstration of bootstrapping, and a simulation of k-means clustering. Each example highlights some of the advantages of keeping model outputs tidy.

These examples assume familiarity with the **dplyr** and **ggplot2** packages, since they are powerful implementations of tidy tools (though it is certainly 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 these examples, 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.

Split-apply-combine using broom and dplyr

The "split-apply-combine" pattern, where a dataset is broken down into subgroups for some analysis to be performed before being recombined, is a common task in data analysis (Wickham, 2011). **broom** eases the application of this pattern to a wide range of problems, since it converts many analysis outputs to a consistently-structured data frame that can be recombined. Here we demonstrate examples of using **broom** to perform the split-apply-combine pattern, first in a simple regression on the `mtcars` dataset, then in a more complex analysis of baseball statistics.

We earlier showed an example of tidying a regression on the `mtcars` dataset.

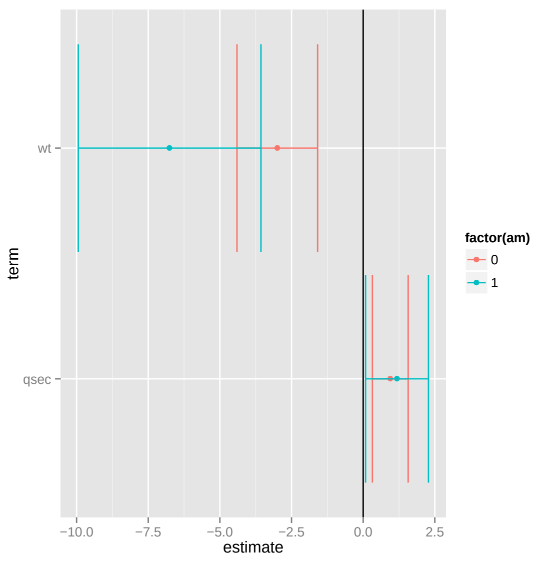
```
regression <- lm(mpg ~ wt + qsec, data = mtcars)

tidy(regression, conf.int = TRUE)

##          term  estimate std.error  statistic      p.value   conf.low
## 1 (Intercept) 19.746223 5.2520617   3.759709 7.650466e-04  9.0045503
## 2          wt -5.047982 0.4839974 -10.429771 2.518948e-11 -6.0378678
## 3        qsec  0.929198 0.2650173   3.506179 1.499883e-03  0.3871768
##   conf.high
## 1 30.487895
## 2 -4.058096
## 3  1.471219
```

Suppose we wished to perform this analysis separately for each subset of the data: specifically, once for cars with automatic transmissions and once for cars with manual transmissions. **dplyr**'s group_by and do functions provide a straightforward way to perform an analysis on each subgroup, but only if the output of each individual analysis takes the form of a data frame. **broom** accomplishes this goal by converting each model object into a tidy format.

```
library(dplyr)

regressions <- mtcars %>% group_by(am) %>%
    do(tidy(lm(mpg ~ wt + qsec, data = .), conf.int = TRUE))

regressions

## Source: local data frame [6 x 8]
## Groups: am
##
##   am        term   estimate  std.error statistic      p.value    conf.low
## 1  0 (Intercept) 11.2489412  6.7148019  1.675245 0.1133158633 -2.98580299
## 2  0          wt -2.9962762  0.6635548 -4.515491 0.0003520832 -4.40294960
## 3  0        qsec  0.9454396  0.2945500  3.209776 0.0054642663  0.32102149
## 4  1 (Intercept) 20.1753989 11.1990599  1.801526 0.1017988425 -4.77766163
## 5  1          wt -6.7543597  1.4305934 -4.721369 0.0008147494 -9.94192037
## 6  1        qsec  1.1809718  0.4924515  2.398148 0.0374338987  0.08372136
## Variables not shown: conf.high (dbl)
```

This shows the estimated coefficients of regressions within automatic and manual cars, recombined. This makes it easy to perform downstream visualizations and analyses on these regression results. For example, we could use **ggplot2** to create a coefficient plot that distinguished between the two types of transmission:

```
library(ggplot2)

regressions %>% filter(term != "(Intercept)") %>%
    ggplot(aes(x = estimate, y = term, color = factor(am), group = am)) +
    geom_point() + geom_errorbarh(aes(xmin = conf.low, xmax = conf.high)) +
    geom_vline()
```



The **do** function also allows one to construct regressions once and save them as a column in a rowwise table, so that we can perform multiple operations on them.

```
regressions <- mtcars %>% group_by(am) %>%
    do(mod = lm(mpg ~ wt + qsec, .))

regressions

## Source: local data frame [2 x 2]
## Groups: <by row>
##
##   am      mod
## 1  0 <S3:lm>
## 2  1 <S3:lm>
```

This is useful if we want to tidy, augment and glance each regression separately. Indeed, **broom** includes a shortcut for this purpose, where `%>% tidy('column')` on a rowwise table will perform a tidying operation on each object in that column:

```
regressions %>% tidy(mod, conf.int = TRUE)

## Source: local data frame [6 x 8]
## Groups: am
##
##   am        term   estimate  std.error statistic      p.value    conf.low
## 1  0 (Intercept) 11.2489412  6.7148019  1.675245 0.1133158633 -2.98580299
## 2  0          wt -2.9962762  0.6635548 -4.515491 0.0003520832 -4.40294960
## 3  0        qsec  0.9454396  0.2945500  3.209776 0.0054642663  0.32102149
## 4  1 (Intercept) 20.1753989 11.1990599  1.801526 0.1017988425 -4.77766163
## 5  1          wt -6.7543597  1.4305934 -4.721369 0.0008147494 -9.94192037
## 6  1        qsec  1.1809718  0.4924515  2.398148 0.0374338987  0.08372136
## Variables not shown: conf.high (dbl)


regressions %>% glance(mod)

## Source: local data frame [2 x 12]
## Groups: am
##
##   am r.squared adj.r.squared    sigma statistic      p.value df    logLik
## 1  0 0.7501667     0.7189375 2.032590  24.02135 1.517760e-05  3 -38.80416
## 2  1 0.8896117     0.8675340 2.244353  40.29465 1.639138e-05  3 -27.25026
## Variables not shown: AIC (dbl), BIC (dbl), deviance (dbl), df.residual
##   (int)
```

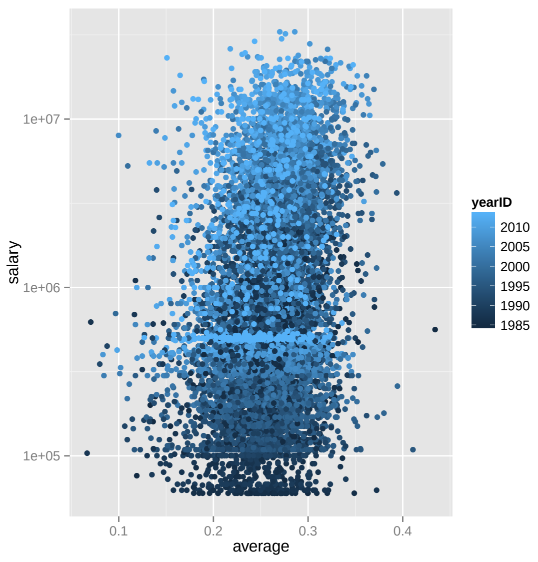
This split-apply-combine pattern appears in a variety of contexts. One might perform analyses for each gene in an organism, within each county or region in a country, or for each stock in a financial dataset. Here I consider another example, where we investigate how baseball players' batting averages are reflected in their salaries, using data from the **Lahman** package (Friendly, 2014).

```
library(Lahman)

merged <- Batting %>% tbl_df() %>%
    inner_join(Salaries) %>%
    mutate(average = H / AB) %>%
    filter(salary > 0, AB >= 50, !(playerID %in% Pitching$playerID))

merged

## Source: local data frame [10,576 x 26]
##
##     playerID yearID stint teamID lgID   G G_batting  AB  R   H X2B X3B HR
## 1  abbotje01   1998     1    CHA   AL  89        89 244 33  68  14   1 12
## 2  abbotje01   1999     1    CHA   AL  17        17  57  5   9   0   0  2
## 3  abbotje01   2000     1    CHA   AL  80        80 215 31  59  15   1  3
## 4  abbotku01   1993     1    OAK   AL  20        20  61 11  15   1   0  3
## 5  abbotku01   1994     1    FLO   NL 101       101 345 41  86  17   3  9
## 6  abbotku01   1995     1    FLO   NL 120       120 420 60 107  18   7 17
## 7  abbotku01   1996     1    FLO   NL 109       109 320 37  81  18   7  8
## 8  abbotku01   1997     1    FLO   NL  94        94 252 35  69  18   2  6
## 9  abbotku01   1998     1    OAK   AL  35        35 123 17  33   7   1  2
## 10 abbotku01   1999     1    COL   NL  96        96 286 41  78  17   2  8
## ..       ...    ...   ...    ...  ... ...       ... ... .. ... ... ... ..
## Variables not shown: RBI (int), SB (int), CS (int), BB (int), SO (int),
##   IBB (int), HBP (int), SH (int), SF (int), GIDP (int), G_old (int),
##   salary (int), average (dbl)
```

Here I've used **dplyr** to perform a few filtering and processing operations to set up the analysis. I merged Lahman's `Batting` dataset, which contains statistics on batting performance per player per season, with player salary information. I computed each player's batting average in each season: the fraction of times at bat ("AB") that led to a hit ("H"). I filtered for cases where the player had a nonzero salary, was at bat at least 50 times, and was not a pitcher (pitchers in baseball are typically not paid based on their batting ability).

Suppose we wish to investigate how players' salaries are related to their batting averages. We could start by performing a single linear regression predicting the log of salary based on batting average, including the year as a second predictor. A scatterplot shows that this relationship is plausible:

```
library(ggplot2)

ggplot(merged, aes(average, salary, color = yearID)) + geom_point() +
    scale_y_log10()
```



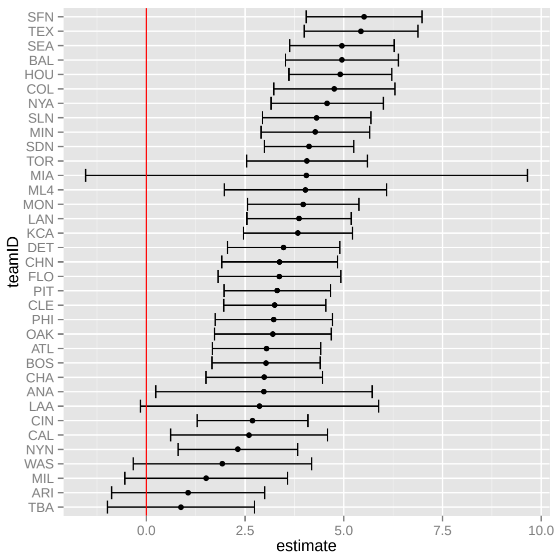
```
salary_fit <- lm(log10(salary) ~ average + yearID, merged)

summary(salary_fit)

##
## Call:
## lm(formula = log10(salary) ~ average + yearID, data = merged)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.32065 -0.48538 -0.01891  0.44176  1.44202
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6.055e+01  1.255e+00  -48.23   <2e-16 ***
## average      3.743e+00  1.328e-01   28.18   <2e-16 ***
## yearID       3.277e-02  6.278e-04   52.19   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5252 on 10573 degrees of freedom
## Multiple R-squared:  0.2513,	Adjusted R-squared:  0.2512
## F-statistic:  1775 on 2 and 10573 DF,  p-value: < 2.2e-16
```

This finds both a player's batting average and the year to be positively related to salary. What if we hypothesize that different teams place different amounts of emphasis on batting average for determining salary? We may be interested in performing the regression within each team. This can be done using group_by and do.

```
team_regressions <- merged %>% group_by(teamID) %>%
    do(tidy(lm(log10(salary) ~ average + yearID, .), conf.int = TRUE))

team_regressions

## Source: local data frame [105 x 8]
## Groups: teamID
##
##    teamID        term      estimate    std.error  statistic      p.value
## 1     ANA (Intercept) -113.85204105 46.013954092  -2.474294 1.493615e-02
## 2     ANA     average    2.97881540  1.382006026   2.155429 3.339096e-02
## 3     ANA      yearID    0.05951026  0.023018726   2.585298 1.108800e-02
## 4     ARI (Intercept)  -49.17939069 15.715043429  -3.129447 2.015982e-03
## 5     ARI     average    1.05909227  0.982962692   1.077449 2.825906e-01
## 6     ARI      yearID    0.02741524  0.007820337   3.505634 5.633804e-04
## 7     ATL (Intercept)  -44.42790245  6.730631459   -6.600852 1.411298e-10
## 8     ATL     average    3.04433694  0.697931557    4.361942 1.671522e-05
## 9     ATL      yearID    0.02482295  0.003374796    7.355394 1.225674e-12
## 10    BAL (Intercept)  -66.54304889  6.374462314  -10.439006 1.249081e-22
## ..    ...         ...           ...          ...         ...          ...
## Variables not shown: conf.low (dbl), conf.high (dbl)
```

With the regressions in this tidied format, we can examine and visualize the effect size estimates and confidence interval across all teams:

```
coefs <- team_regressions %>% ungroup() %>%
    filter(term == "average") %>%
    mutate(teamID = reorder(teamID, estimate))

ggplot(coefs, aes(x = estimate, y = teamID)) + geom_point() +
    geom_errorbarh(aes(xmin = conf.low, xmax = conf.high)) +
    geom_vline(color = "red")
```



Any analysis that has a tidying method can easily be performed within each team. For example, rather than performing a linear regression, we could find the Spearman correlation between salary and batting average within each team:

```
cors <- merged %>% group_by(teamID) %>%
    do(tidy(cor.test(.$salary, .$average, method = "spearman"))) %>%
    ungroup %>% arrange(estimate)

tail(cors, 3)

## Source: local data frame [3 x 4]
##
##   teamID  estimate statistic      p.value
## 1    SFN 0.3455152   5615313 7.188991e-12
## 2    SLN 0.3480923   5196985 8.869388e-12
## 3    TEX 0.3641821   4821506 1.227396e-12
```

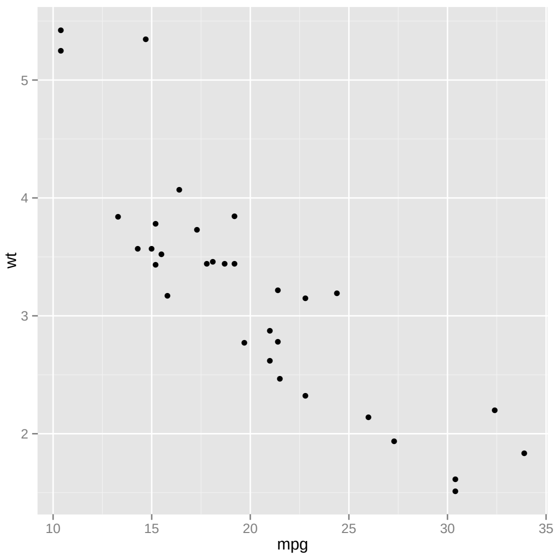
One could just as easily perform ANOVA, time series analyses, or nonlinear regressions within each subgroup. Similarly, one could perform analyses while grouping by league, by year, or by player, rather than by team. These analyses of subgroups are made possible by **broom**'s tidying methods, since they allow the model outputs to be recombined into a new tidy output.

Bootstrapped confidence and prediction intervals

Since tidy outputs can be recombined across replicates, it is also 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.

```
ggplot(mtcars, aes(mpg, wt)) + geom_point()
```

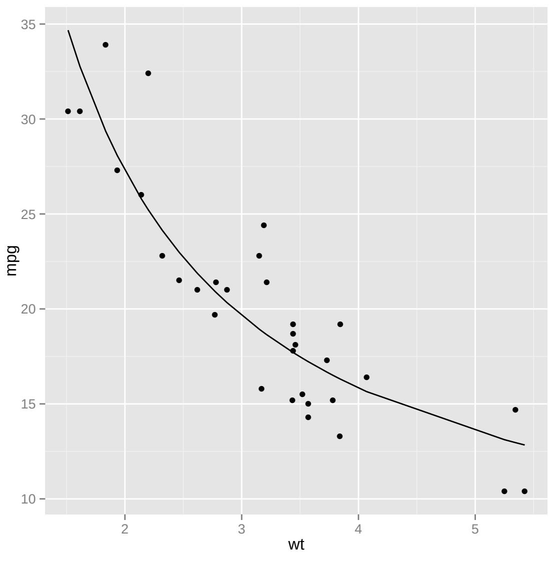


We might use the method of nonlinear least squares (the built-in `nls` function) to fit a model.

```
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
```

```
confint(nlsfit)

##        2.5%     97.5%
## k 37.151556 54.507419
## b  1.248471  7.524038
```

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



While this does provide p-values 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:

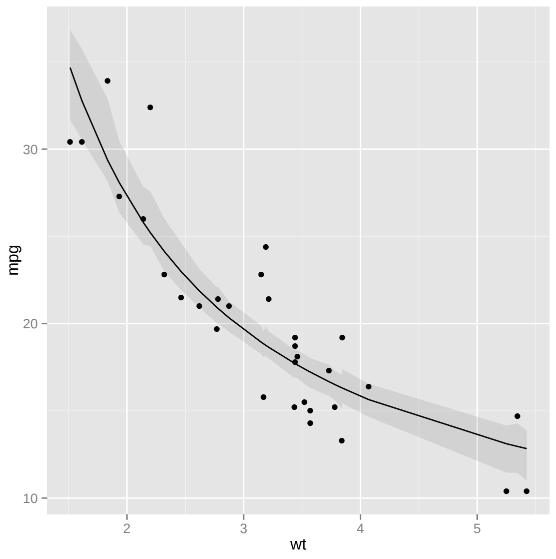
```
head(bootnls)

## 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
```

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)
```



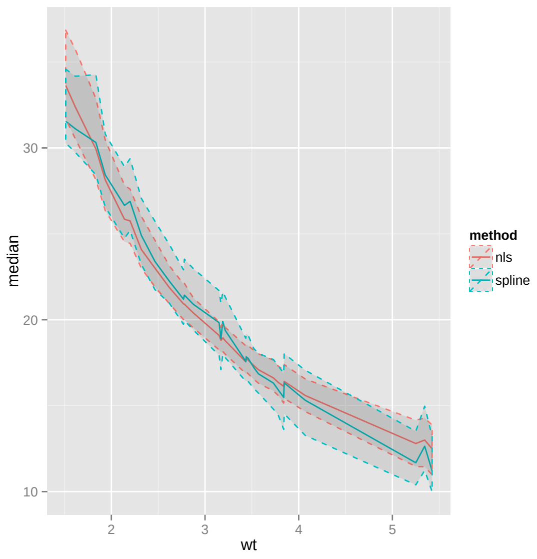
This bootstrapping approach could be applied to any prediction function for which an augment method is defined. For example, we could use the built-in **splines** package to predict the curve using a natural cubic spline basis.

```
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_list(bootnls, bootspline)

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

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



Bootstrapping can be applied in this way to any object that **broom** can tidy, such as survival models from the **survival** package. Here we consider 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). Just like we did on the mtcars dataset, we perform this analysis within each of 500 bootstrap resamplings of the dataset.

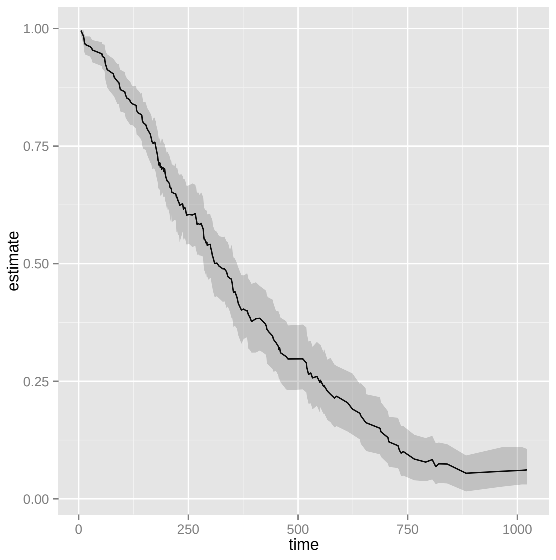
```
library(survival)

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

head(bootstraps_survival)

## Source: local data frame [6 x 9]
## Groups: replicate
##
##   replicate time n.risk n.event n.censor  estimate   std.error conf.high
## 1         1   11    228       2        0 0.9916307 0.005946493 1.0000000
## 2         1   12    226       1        0 0.9874253 0.007312389 1.0000000
## 3         1   13    225       3        0 0.9747625 0.010452461 0.9949378
## 4         1   15    222       2        0 0.9662883 0.012150010 0.9895752
## 5         1   26    220       1        0 0.9620413 0.012929476 0.9867322
## 6         1   30    219       1        0 0.9577853 0.013674363 0.9838023
## Variables not shown: conf.low (dbl)
```

We can then construct and visualize a confidence interval for the fraction of survival at each time point, just as we did for the NLS and spline fits.

```
alpha = .05
```

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

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



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 bootstrap any model with `tidy` or `augment` methods.

Simulation of 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 method'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:

```
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])))
```

Now suppose we would like to perform k-means clustering on this data, while varying the number of clusters k. The `inflate` function, provided by **broom**, is useful for this purpose: it expands a dataset to repeat 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
```

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.

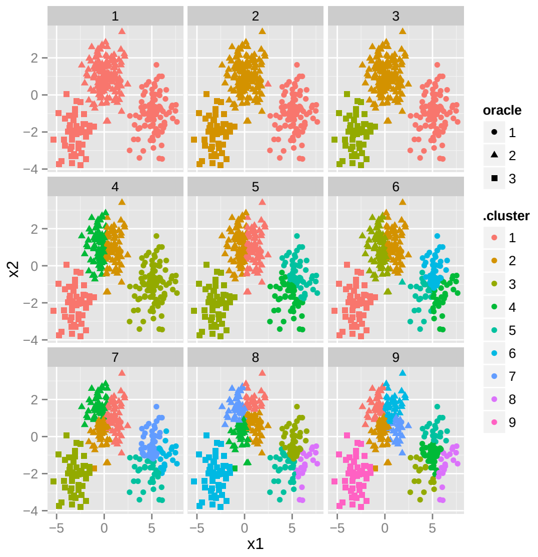
- **Component level**: The centers, size, and within sum-of-squares for each cluster; computed by `tidy`
- **Observation level**: The assignment of each point to a cluster; computed by `augment`
- **Model level**: The total within sum-of-squares and between sum-of-squares; computed by `glance`

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. We use the shortcut for performing tidying functions on a rowwise table first demonstrated in Split-apply-combine using **broom** and **dplyr**.

```
clusters <- kclusts %>% tidy(clust)

assignments <- kclusts %>% augment(clust, kdat)

clusterings <- kclusts %>% glance(clust)
```

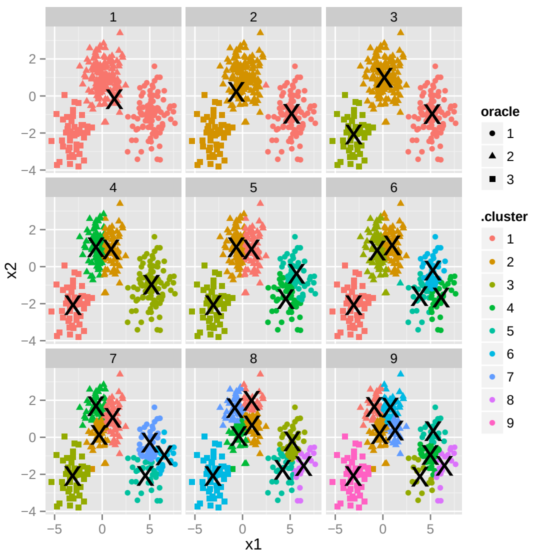
The `assignments` object, generated using the `augment` method on each clustering, combines the cluster assignments across all values of k, thus allowing for simple visualization using faceting.

```
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
```



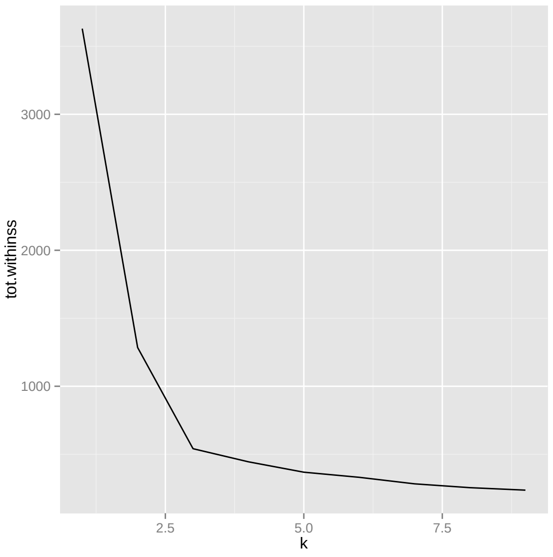
We can use the results from the `tidy` outputs, which provide per-cluster information, to add the estimated center of each cluster to the plot.

```
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 correct 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)
```

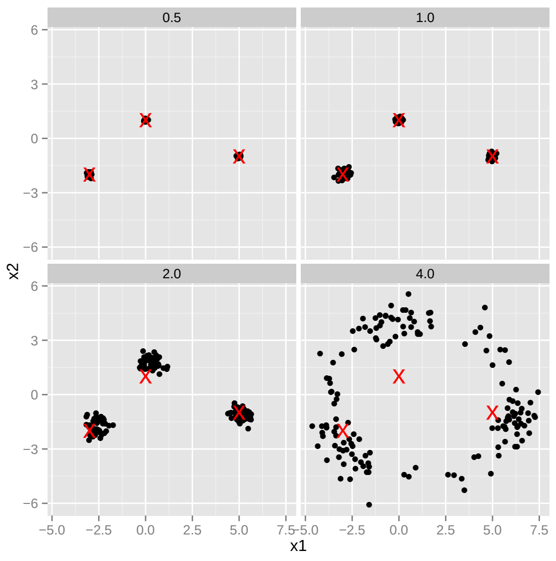
```
kdat_sd <- centers %>%
    inflate(sd = c(.5, 1, 2, 4), replication = 1:50) %>%
    group_by(oracle, sd, replication) %>%
    do(data.frame(x1 = rnorm(.$size[1], .$x1[1], .$sd[1]),
                  x2 = rnorm(.$size[1], .$x2[1], .$sd[1])))
```

We perform k-means clustering 9 times on each dataset, 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))

clusters_sd <- kclusts_sd %>% tidy(clust)

glances_sd <- kclusts_sd %>% glance(clust)

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

(Note that since the `augment` operation requires the original data, which changes in each replication, we needed a different approach to perform `augment` on each row). 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:

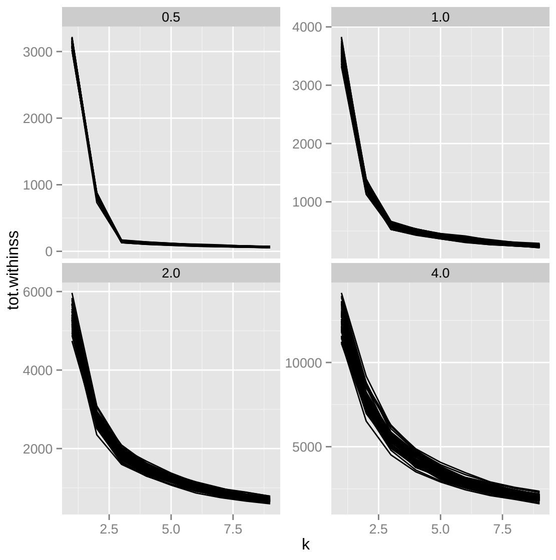
```
clusters_sd %>% filter(k == 3) %>%
    ggplot(aes(x1, x2)) + geom_point() +
    geom_point(data = centers, size = 7, color = "red", shape = "x") +
    facet_wrap(~ sd)
```



We can see that the centers are estimated very accurately for $\sigma = .5$ and 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.

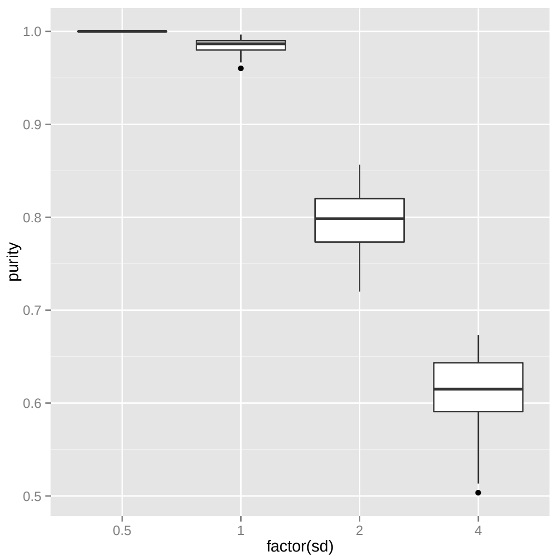
We can also see how the total within-sum-of-squares graph appears across all replications, and how it varies when changing the value of σ.

```
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 could even measure the cluster purity, and see how the classification 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. These examples demonstrate that combining the models from simulations into a tidied form thus lends itself well to many kinds of exploratory analyses and experiments.

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 and improve the functionality of **broom** and to advance the entire available suite of tidy tools.

Acknowledgments

I thank Andy Bass, Dmitry Gorenshteyn, and Bob Muenchen for helpful comments and discussions on this manuscript. I thank Hadley Wickham for comments early in **broom**'s development and for contributing code from **ggplot2** to the package, and thank Matthieu Gomez and Boris Demeshev for contributions to the software.

This work was supported in part by NIH (R01 HG002913).

Bibliography

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. [p]

T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003. [p]

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

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

M. Friendly. *Lahman: Sean Lahman's Baseball Database*, 2014. URL `http://CRAN.R-project.org/package=Lahman`. R package version 3.0-1. [p]

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

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

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

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. [p]

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

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. [p]

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

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

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. [p]

H. Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40, Apr. 2011. URL http://www.jstatsoft.org/v40/i01/. [p]

H. Wickham. Tidy Data. *Journal of Statistical Software*, 59, Aug. 2014a. URL http://www.jstatsoft.org/v59/i10. [p]

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. [p]

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. [p]

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