Introduction to Data Science

Session 6: Model fitting ~~and simulation~~

Simon Munzert
Hertie School | GRAD-C11/E1339

Table of contents

The modeling workflow

`Credit` David Hood

The modeling workflow

Why modeling?

- Modeling is at the heart of the data science workflow.
- We use models to explore, test, infer, predict based on data.
- The art and science of statistical modeling is vast.
- Today, we will focus on key steps of the workflow which are common in most modeling endeavors. We won't touch on theoretical/statistical backgrounds though and ignore workflow issues in particular areas, such as simulation-based Bayesian inference.

The modeling workflow

Why modeling?

- Modeling is at the heart of the data science workflow.
- We use models to explore, test, infer, predict based on data.
- The art and science of statistical modeling is vast.
- Today, we will focus on key steps of the workflow which are common in most modeling endeavors. We won't touch on theoretical/statistical backgrounds though and ignore workflow issues in particular areas, such as simulation-based Bayesian inference.

Steps of the workflow

1. **Choose** a modeling strategy
2. **Specify** the model (structural components / parameters)
3. **Run/implement** the model (estimation)
4. **Evaluate** the model output
5. **Present** the results

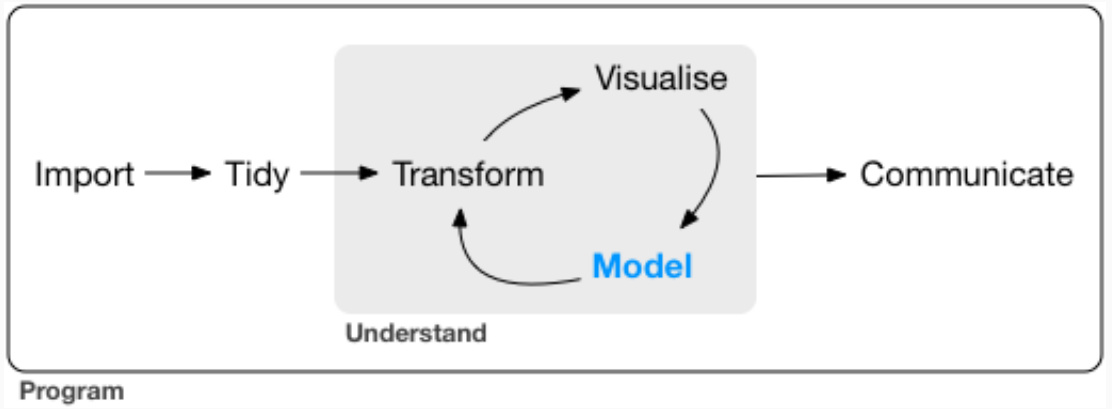
The modeling workflow

Why modeling?

- Modeling is at the heart of the data science workflow.
- We use models to explore, test, infer, predict based on data.
- The art and science of statistical modeling is vast.
- Today, we will focus on key steps of the workflow which are common in most modeling endeavors. We won't touch on theoretical/statistical backgrounds though and ignore workflow issues in particular areas, such as simulation-based Bayesian inference.

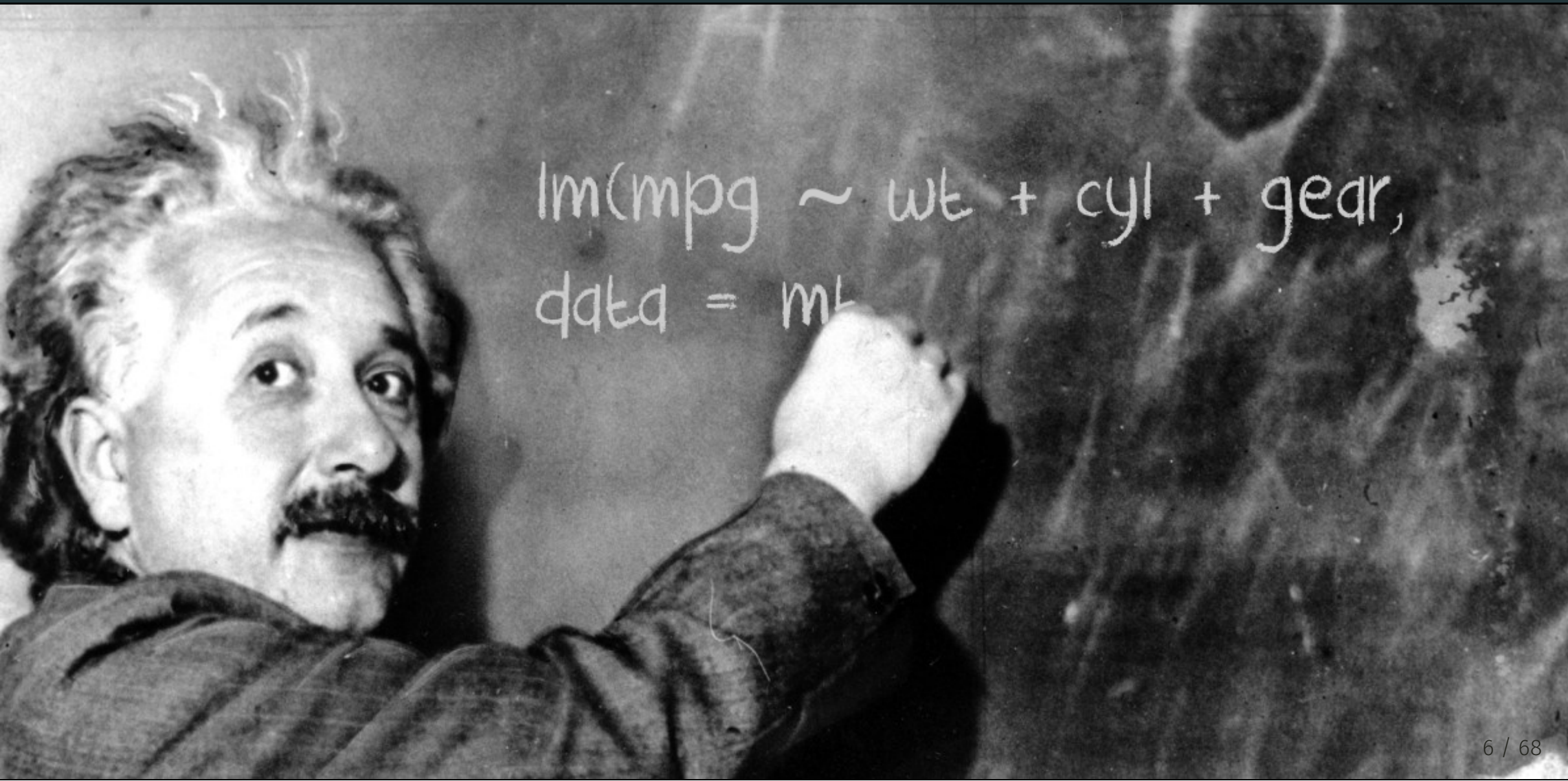
Steps of the workflow

1. **Choose** a modeling strategy
2. **Specify** the model (structural components / parameters)
3. **Run/implement** the model (estimation)
4. **Evaluate** the model output
5. **Present** the results



Crafting formulas

Crafting formulas



Model building

Systematic + stochastic components of models

- When we try to model data, we often start by assuming a data-generating process that looks like

$$Y = f(x) + \epsilon$$

- In doing so, we decompose a model (or data-generating process) into a random or stochastic part (here: ϵ) and a systematic/structural/deterministic part (here: $f(x)$).
- (We might go on to impose further assumptions about the stochastic component, e.g., $\epsilon \sim N(0, 1)$.)
- In many cases, we want to learn how certain variables systematically relate to each other. To that end, we specify the systematic component of a model and then "let the data speak" to estimate parameters associate with elements of the systematic component.
- As an example, we might specify

$$f(x) = \beta_0 + \beta_1 x$$

- But how can we express our belief about the model structure in R?

Model formulas in R

- In R, there's a standardized way to specify models like this: working with the `formula` class.
- In many cases you can still think of the model formula as just a string specifying the structural part of your model (there are exceptions).
- But `formula` class objects also allow you to do more useful things with formula.
- The basic structure of a formula is the tilde symbol (~) and at least one independent (righthand) variable. In most (but not all) situations, a single dependent (lefthand) variable is also needed. Thus we can construct a formula quite simply by just typing:

```
R> y ~ x
```

- Spaces in formulas are not important, but I recommend using them to make the formulas more readable.
- Running a model with a formula is straightforward. Note that we don't even have to put the formula in parentheses - it is automatically interpreted as one formula expression when provided as the first argument:

```
R> lm(arr_delay ~ distance + origin, data = flights)
```

Model formulas in R (cont.)

Storing formulas

- A more explicit way is to write the formula as string and then use `as.formula()` to turn it into a formula object. This implies that we can store formulas as an R object and check its class.

```
R> fmla ← as.formula("arr_delay ~ distance + origin")
R> class(fmla)
```

```
[1] "formula"
```

- Next, we'd pass on the `formula` object to the model function, e.g.:

```
R> lm(fmla, data = flights)
```

```
Call:
lm(formula = fmla, data = flights)

Coefficients:
(Intercept)     distance    originJFK    originLGA
  13.414049    -0.004045    -2.704255    -4.456169
```

Formula syntax: basics

- We can use multiple independent variables by simply separating them with the plus (+) symbol:

```
R> y ~ x1 + x2
```

- If we use a minus (-) symbol, objects in the formula are ignored in an analysis:

```
R> y ~ x1 - x2
```

- We can also use this to drop the intercept:

```
R> y ~ x1 - x2 - 1
```

- The `.` operator refers to all other variables in the matrix/data frame not yet included in the model. This is useful when you plan to run a regression on all variables in a matrix/data frame:

```
R> y ~ .
```

Formula syntax: interactions

In a regression modeling context, we often need to specify interaction terms. There are two ways to do this. If we want to include two variables and their interaction, we use the star/asterisk (`*`) symbol:

```
R> y ~ x1 * x2
```

That's equivalent to

```
R> y ~ x1 + x2 + x1*x2
```

If you only want their interaction, but not the variables themselves as main effects (which you probably don't want), use the colon symbol:

```
R> y ~ x1:x2
```

Formula syntax: variable transformations

One trick to formulas is that they don't evaluate their contents. So, for example, if we wanted to include x and x^2 in our model, we might be tempted to type:

```
R> y ~ x + x^2
```

This won't work though. We therefore have to either calculate and store all of the variables we want to include in the model in advance, or we need to use the `I()` "as-is" operator, short for "Inhibit Interpretation/Conversion of Objects". In a formula function, `I()` is used to inhibit the interpretation of operators such as `+`, `-`, `*` and `^` as formula operators, so they are used as arithmetical operators. To obtain our desired two-term formula, we type:

```
R> y ~ x + I(x^2)
```

Again, the alternative would have been something like:

```
R> data$x2 ← (data$x)^2
R> y ~ x + x2
```

Specifying multiple models

When one model is not enough

- Often we want to specify not one but multiple different models.
- Such models can differ in terms of model family, modelled outcome, covariate/feature set, transformations of input variables, and data being modelled.

Generating model formulas at scale

- If outcomes/features vary across models, so does the model formula.
- Regarding formulas as character strings, it's straightforward to generate them in a programmatic fashion.

Specifying multiple models

When one model is not enough

- Often we want to specify not one but multiple different models.
- Such models can differ in terms of model family, modelled outcome, covariate/feature set, transformations of input variables, and data being modelled.

Generating model formulas at scale

- If outcomes/features vary across models, so does the model formula.
- Regarding formulas as character strings, it's straightforward to generate them in a programmatic fashion.

Example:

```
R> xvars ← paste0("x", 1:20)
R> fmla ← as.formula(paste("y ~ ", paste(xvars, collapse= "+")))
R> fmla

  y ~ x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 +
      x12 + x13 + x14 + x15 + x16 + x17 + x18 + x19 + x20
```

Specifying multiple models (cont.)

- Another example of multiple model specification is extreme bounds analysis (EBA).
- Here, the idea is to compute all possible estimates given a set of allowed coefficients to answer questions like:
  - Which determinants are robustly associated with the dependent variable across a large number of possible regression models?
  - Is a particular determinant robustly associated with the dependent variable?
- In its basic form, EBA just estimates models with all possible combinations of variables and then looks into the distribution (or range → extreme bounds) of effects across all models.
- There are R packages to do this for us (e.g., `ExtremeBounds` by Marek Hlavac) but we can also run the basics on our own.

Specifying multiple models (cont.)

- Another example of multiple model specification is extreme bounds analysis (EBA).
- Here, the idea is to compute all possible estimates given a set of allowed coefficients to answer questions like:
  - Which determinants are robustly associated with the dependent variable across a large number of possible regression models?
  - Is a particular determinant robustly associated with the dependent variable?
- In its basic form, EBA just estimates models with all possible combinations of variables and then looks into the distribution (or range → extreme bounds) of effects across all models.
- There are R packages to do this for us (e.g., `ExtremeBounds` by Marek Hlavac) but we can also run the basics on our own.

Example

Step 1: Define dependent variable and covariate set

```
R> depvar ← "arr_delay"
R> covars ← c("dep_delay", "carrier", "origin", "air_time", "distance", "hour")
```

Specifying multiple models (cont.)

Step 1a (just for fun): Compute the number of unique combinations of all these covariates

```
R> combinations ←
+   map(1:6, function(x) {combn(1:6, x)}) %>% # create all possible combinations (draw 1 to 6 out of 6)
+   map(ncol) %>% # extract number of combinations
+   unlist() %>% # pull out of list structure
+   sum() # compute sum
```

Specifying multiple models (cont.)

Step 2: Build function to run lm models across set of all possible variable combinations

```
R> combn_models ← function(depvar, covars, data)
+ {
+   combn_list ← list()
+   # generate list of covariate combinations
+   for (i in seq_along(covars)) {
+     combn_list[[i]] ← combn(covars, i, simplify = FALSE)
+   }
+   combn_list ← unlist(combn_list, recursive = FALSE)
+   # function to generate formulas
+   gen_formula ← function(covars, depvar) {
+     form ← as.formula(paste0(depvar, " ~ ", paste0(covars, collapse = "+")))
+     form
+   }
+   # generate formulas
+   formulas_list ← purrr::map(combn_list, gen_formula, depvar = depvar)
+   # run models
+   models_list ← purrr::map(formulas_list, lm, data = data)
+   models_list
+ }
```

Specifying multiple models (cont.)

Step 3: Run models (careful, this'll generate a quite heavy list)

```
R> models_list ← combn_models(depvar = depvar,  covars = covars, data = flights)
```

How many models did we fit?

```
R> length(models_list)
```

```
  [1] 63
```

And what did we get? A glimpse at the first list element:

```
R> models_list[[1]]
```

```
  Call:
  .f(formula = .x[[i]], data = ..1)

  Coefficients:
  (Intercept)    dep_delay
       -5.864        1.019
```

Specifying multiple models (cont.)

Step 3: Run models (careful, this'll generate a quite heavy list)

```
R> models_list ← combn_models(depvar = depvar,  covars = covars, data = flights)
```

How many models did we fit?

```
R> length(models_list)
```

```
  [1] 63
```

And what did we get? A glimpse at the first list element:

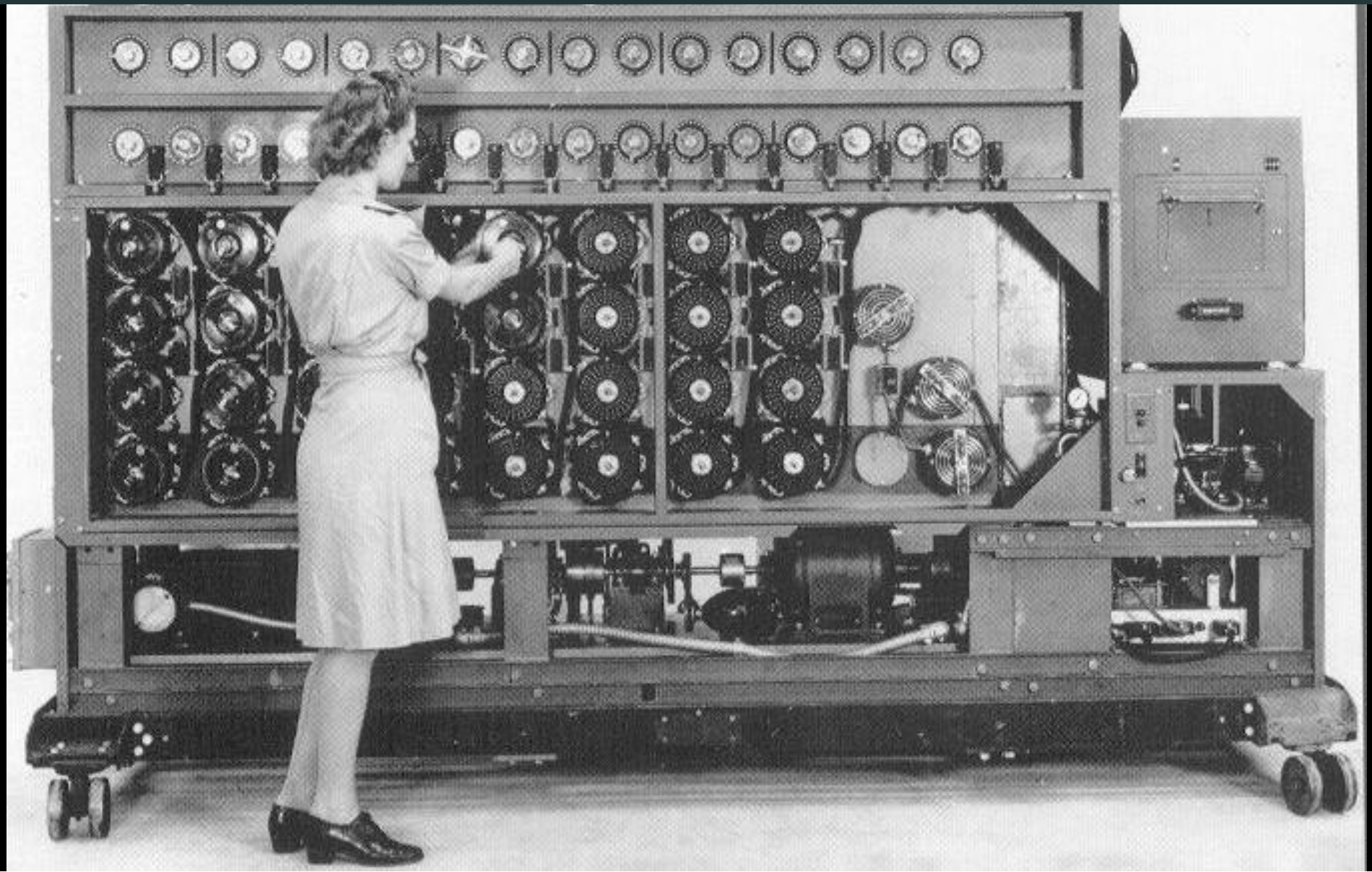
```
R> models_list[[1]]
```

```
  Call:
  .f(formula = .x[[i]], data = ..1)

  Coefficients:
  (Intercept)    dep_delay
       -5.864        1.019
```

Running models

Running models



Model families

Today we'll mostly focus on linear models as an example. However, there is a multitude of families of statistical models that allow extend linear models. Examples are

- **Generalized linear models** [`stats::glm()`], which extend linear models to include non-continuous responses (e.g., binary or categorical data, counts)
- **Generalized additive models** [`mgcv::gam()`], which extend generalized linear models to incorporate arbitrary smooth functions
- **Penalized linear models** [`glmnet::glmnet()`], which introduce terms that penalize complex models to make models that generalize better to new datasets

Model families

Today we'll mostly focus on linear models as an example. However, there is a multitude of families of statistical models that allow extend linear models. Examples are

- **Generalized linear models** [`stats::glm()`], which extend linear models to include non-continuous responses (e.g., binary or categorical data, counts)
- **Generalized additive models** [`mgcv::gam()`], which extend generalized linear models to incorporate arbitrary smooth functions
- **Penalized linear models** [`glmnet::glmnet()`], which introduce terms that penalize complex models to make models that generalize better to new datasets

Also there is so much more to learn in terms of modeling/machine learning/AI. There are many (MANY!) models for measurement, (un-)supervised learning, clustering, dimensionality reduction, ... R is uniquely flexible for implementing these models. To get a quick glance at the universe from afar, check out the CRAN Task Views, a curated online directory of topics and the R packages relevant for tasks related to these.

Running models: examples

Luckily, these models work all similarly from a programming experience - once you've mastered how to run linear models, you will find it easy to implement others. Understanding and applying them wisely is a different matter though.

Logistic regression

```
R> logit_out ← stats::glm(am ~ cyl + hp + wt, data = mtcars, family = binomial)
```

Generalized additive model regression

```
R> gam_out ← mgcv::gam(mpg ~ s(disp) + s(wt), data = mtcars)
```

Penalized (here: lasso) regression

```
R> lasso_out ← glmnet(as.matrix(mtcars[-1]), mtcars[,1], standardize = TRUE, alpha  = 1)
```

Multilevel model with random intercepts

```
R> library(lme4)
R> ml_out ← lmer(arr_delay ~ distance + origin + (1|carrier) + (1|tailnum), data = flights)
```

Decisions in the modeling workflow

Big data and the need for models

- In the early days of the big data hype, people were overly enthusiastic about its implications for modeling (see quote on the right).
- This is falling for the inception that we can simply "let the data speak".
- However, the data science workflow is a sequence of subjective decisions from start to finish, with lots of researcher degrees of freedom.
- Think of all the weakly justified decisions regarding:
  - data collection / selection
  - measurement
  - model choice
  - model specification
  - reporting

> "Scientists are trained to recognize that correlation is not causation, that no conclusions should be drawn simply on the basis of correlation between X and Y. (...) Once you have a model, you can connect the data sets with confidence. Data without a model is just noise. (...) There is now a better way. Petabytes allow us to say: "Correlation is enough." We can stop looking for models. We can analyze the data without hypotheses about what it might show."

Chris Anderson, "The End of Theory: The Data Deluge makes the Scientific Method Obsolete (2008, *Wired*)"

Decisions in the modeling workflow

Big data and the need for models

- In the early days of the big data hype, people were overly enthusiastic about its implications for modeling (see quote on the right).
- This is falling for the inception that we can simply "let the data speak".
- However, the data science workflow is a sequence of subjective decisions from start to finish, with lots of researcher degrees of freedom.
- Think of all the weakly justified decisions regarding:
  - data collection / selection
  - measurement
  - model choice
  - model specification
  - reporting

Question: How do researchers usually deal with this?

> "Scientists are trained to recognize that correlation is not causation, that no conclusions should be drawn simply on the basis of correlation between X and Y. (...) Once you have a model, you can connect the data sets with confidence. Data without a model is just noise. (...) There is now a better way. Petabytes allow us to say: "Correlation is enough." We can stop looking for models. We can analyze the data without hypotheses about what it might show."

Chris Anderson, "The End of Theory: The Data Deluge makes the Scientific Method Obsolete (2008, *Wired*)"

Nested models

TABLE 3 Change in Pork Earmark Spending by Zip Code from Pre- to Post-Redistricting Years

	Dependent Variable: Change in Logged Earmark Spending per Capita, per Year				
	Model (1)	**Model (2)**	**Model (3)**	**Model (4)**	**Model (5)**
Proposition 1(c):	−1.23*	−1.50**	−1.50**	−1.46**	−1.61**
Δ District Diversity Index ($D_{POST} - D_{PRE}$)	(0.55)	(0.55)	(0.55)	(0.55)	(0.57)
Per Capita Income ($1,000s)	−0.0072**	−0.0063*	−0.0057*	−0.0056*	−0.0056
	(0.0027)	(0.0028)	(0.0029)	(0.0029)	(0.0029)
Poverty Rate	−0.73	−0.78	−0.74	−0.73	−0.66
	(0.42)	(0.42)	(0.42)	(0.42)	(0.43)
Racial Minority	0.36**	0.34**	0.46*	0.52**	0.49**
	(0.12)	(0.12)	(0.19)	(0.18)	(0.19)
Population Density	0.0073	0.21	0.54	0.66	0.62
(1,000,000s/Sq. Mi.)	(1.38)	(1.38)	(1.44)	(1.43)	(1.43)
Democrat (Pre-Redistricting) to Republican (Post-Redistricting) Senator	—	0.39***	0.40***	0.40***	0.49***
		(0.099)	(0.10)	(0.099)	(0.11)
Republican (Pre-Redistricting) to Democrat (Post-Redistricting) Senator	—	−0.24	−0.25	−0.25	−0.27
		(0.18)	(0.18)	(0.18)	(0.18)
2000 Gore Vote Share	—	—	−0.30	—	—
			(0.36)		
|2000 Gore Vote Share −0.50|	—	—	—	−0.57	−0.41
				(0.44)	(0.44)
Post-Redistricting Senator is More Junior	—	—	—	—	−0.039
					(0.079)
Post-Redistricting Senator is More Senior	—	—	—	—	−0.17
					(0.087)
Constant	0.078	0.046	0.16	0.047	0.090
	(0.098)	(0.099)	(0.18)	(0.099)	(0.11)
N	1,599	1,599	1,599	1,599	1,599

***p < .001; **p < .01; *p < .05; (two-tailed); standard errors in parentheses.
The dependent variable is measured as $\log(Y_Z^{POST}/Population_Z + 1) - \log(Y_Z^{PRE}/Population_Z + 1)$, where Y_Z^{POST} represents per-year pork spending in zip code Z during 2003–2004, and Y_Z^{PRE} is the same measurement for years 1998–2002.

`Credit` Chen 2010, The Effect of Electoral Geography on Pork Barreling in Bicameral Legislatures

TABLE 3. HIERARCHICAL REGRESSION ANALYSIS OF PREDICTORS OF CIVIC PARTICIPATION

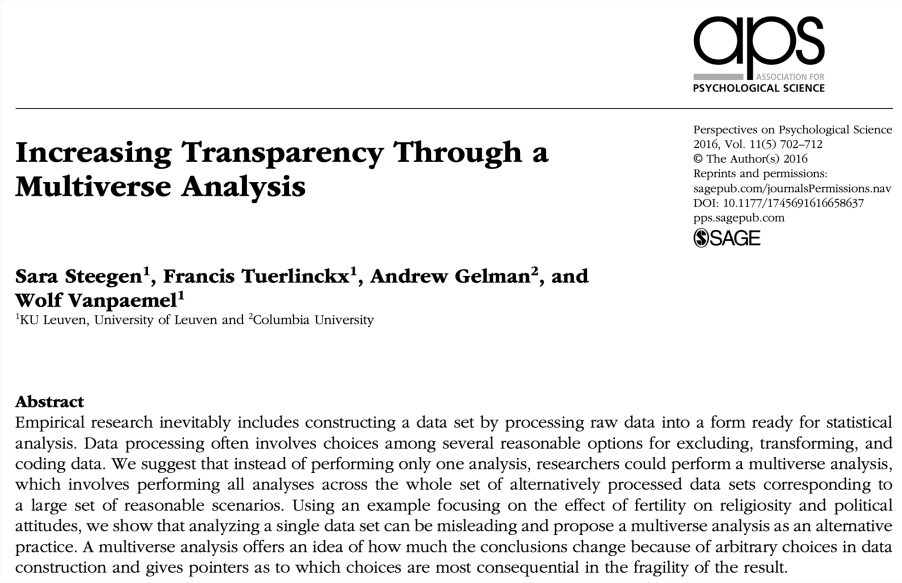
Predictor variables	*Regression 1*	*Regression 2*	*Regression 3*
Gender (1, female; 0, male)	0.06***	0.05***	0.05***
Hometown (1, Texas; 0, elsewhere)	−0.04**	−0.03*	−0.03*
Ethnicity			
Black (1, yes; 0, no)	−0.04	−0.04	−0.05
Latino (1, yes; 0, no)	0.02	0.02	0.01
White (1, yes; 0, no)	0.01	−0.00	0.00
Year in school (1, freshman; 6, doctoral)	−0.21**	−0.21***	−0.21*
Parents' education (1, less than high school; 5, graduate)	0.02***	0.02***	0.02**
Life satisfaction		0.27***	0.25***
Social trust		0.23***	0.20***
Needs for using Groups			
Socializing			−0.01
Entertainment			−0.10
Self-status seeking			0.01
Information			0.14***
R^2	0.04	0.08	0.16
R^2 change	0.04	0.05	0.08

*$p < 0.05$; **$p < 0.01$; ***$p < 0.001$.

`Credit` Park et al. 2009, Being immersed in social networking environment: Facebook groups, uses and gratifications, and social outcomes

Exploring the model space

- **Another idea:** run not an arbitrary (small) set of models but as many (plausible ones) as possible to get an idea how much conclusions change depending on arbitrary data wrangling and modeling choices (the "model distribution").
- There are various related procedures and labels used in different subfields to promote this idea, including:
  - Multiverse analysis (Steegen et al. 2016)
  - Specification curves (Simonsohn et al. 2020)
  - Computational multimodel analysis (Young and Holsteen 2015)
- Also check out critical perspective on "Mülltiverse Analysis" (Julia Rohrer). The bottom line: Mindless multiversing doesn't give more robustness or insight.



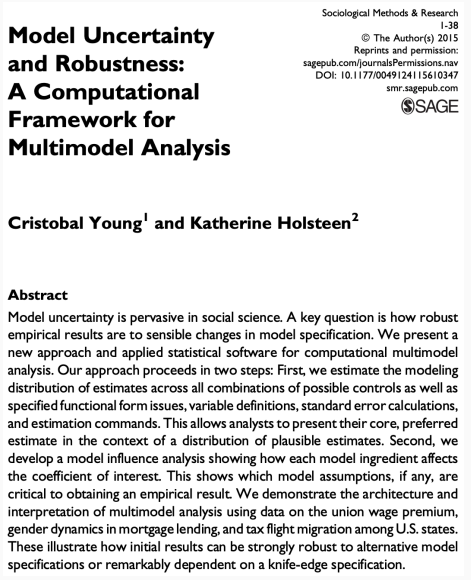
aps ASSOCIATION FOR PSYCHOLOGICAL SCIENCE

Perspectives on Psychological Science
2016, Vol. 11(5) 702–712
© The Author(s) 2016
Reprints and permissions:
sagepub.com/journalsPermissions.nav
DOI: 10.1177/1745691616658637
pps.sagepub.com
SAGE

Increasing Transparency Through a Multiverse Analysis

Sara Steegen[1], Francis Tuerlinckx[1], Andrew Gelman[2], and Wolf Vanpaemel[1]
[1]KU Leuven, University of Leuven and [2]Columbia University

Abstract
Empirical research inevitably includes constructing a data set by processing raw data into a form ready for statistical analysis. Data processing often involves choices among several reasonable options for excluding, transforming, and coding data. We suggest that instead of performing only one analysis, researchers could perform a multiverse analysis, which involves performing all analyses across the whole set of alternatively processed data sets corresponding to a large set of reasonable scenarios. Using an example focusing on the effect of fertility on religiosity and political attitudes, we show that analyzing a single data set can be misleading and propose a multiverse analysis as an alternative practice. A multiverse analysis offers an idea of how much the conclusions change because of arbitrary choices in data construction and gives pointers as to which choices are most consequential in the fragility of the result.



Sociological Methods & Research
1-38
© The Author(s) 2015
Reprints and permission:
sagepub.com/journalsPermissions.nav
DOI: 10.1177/0049124115610347
smr.sagepub.com
SAGE

Model Uncertainty and Robustness: A Computational Framework for Multimodel Analysis

Cristobal Young[1] and Katherine Holsteen[2]

Abstract
Model uncertainty is pervasive in social science. A key question is how robust empirical results are to sensible changes in model specification. We present a new approach and applied statistical software for computational multimodel analysis. Our approach proceeds in two steps: First, we estimate the modeling distribution of estimates across all combinations of possible controls as well as specified functional form issues, variable definitions, standard error calculations, and estimation commands. This allows analysts to present their core, preferred estimate in the context of a distribution of plausible estimates. Second, we develop a model influence analysis showing how each model ingredient affects the coefficient of interest. This shows which model assumptions, if any, are critical to obtaining an empirical result. We demonstrate the architecture and interpretation of multimodel analysis using data on the union wage premium, gender dynamics in mortgage lending, and tax flight migration among U.S. states. These illustrate how initial results can be strongly robust to alternative model specifications or remarkably dependent on a knife-edge specification.

Specification curves

- Specification curve analysis (SCA) facilitates the visual identification of the source of variation in results across multiple specifications.
- The key feature, the specification curve, provides all gathered estimates sorted by effect size and highlighted by significance.
- SCA is carried out in three main steps:
  1. Define the set of reasonable specifications to estimate;
  2. Estimate all specifications and report the results in a descriptive specification curve; and
  3. Conduct joint statistical tests using an inferential specification curve.
- As of now there are two R packages that offer high-level functions for specification: `specr` (see here) and `Multiverse` (see here).

nature human behaviour

RESOURCE

https://doi.org/10.1038/s41562-020-0912-z

Check for updates

Specification curve analysis

Uri Simonsohn[1]✉, Joseph P. Simmons[2] and Leif D. Nelson[3]

Empirical results hinge on analytical decisions that are defensible, arbitrary and motivated. These decisions probably introduce bias (towards the narrative put forward by the authors), and they certainly involve variability not reflected by standard errors. To address this source of noise and bias, we introduce specification curve analysis, which consists of three steps: (1) identifying the set of theoretically justified, statistically valid and non-redundant specifications; (2) displaying the results graphically, allowing readers to identify consequential specifications decisions; and (3) conducting joint inference across all specifications. We illustrate the use of this technique by applying it to three findings from two different papers, one investigating discrimination based on distinctively Black names, the other investigating the effect of assigning female versus male names to hurricanes. Specification curve analysis reveals that one finding is robust, one is weak and one is not robust at all.

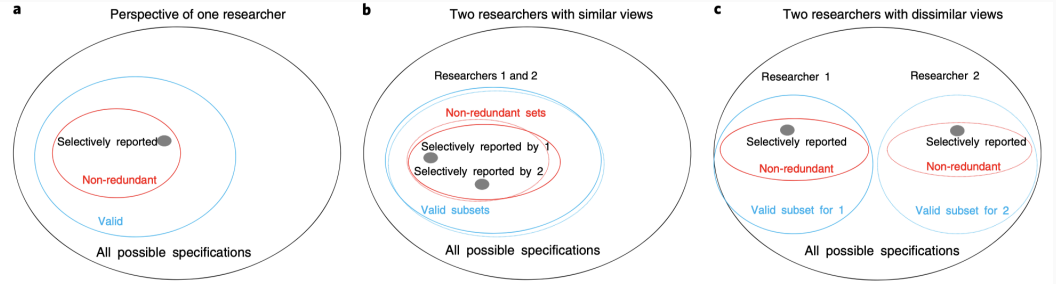


Fig. 1 | Sets of possible specifications as perceived by researchers. a, The set of specifications reported in an article are a small subset of those the researcher would consider valid to report. **b**, Different researchers may have similar views on the set of valid specifications but report quite different subsets of them. **c**, Different researchers may also disagree on the set of specifications they consider valid.

Specification curves (cont.)

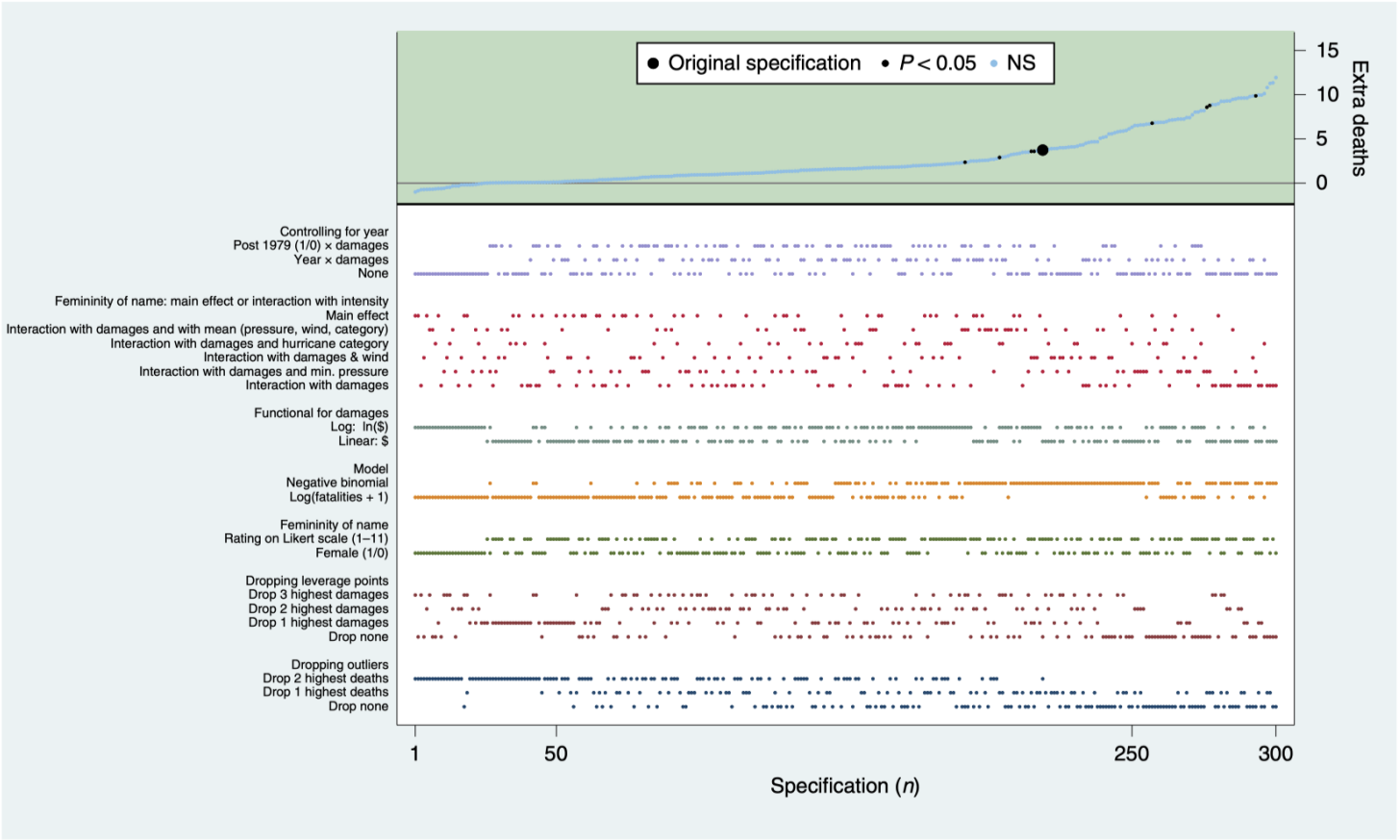
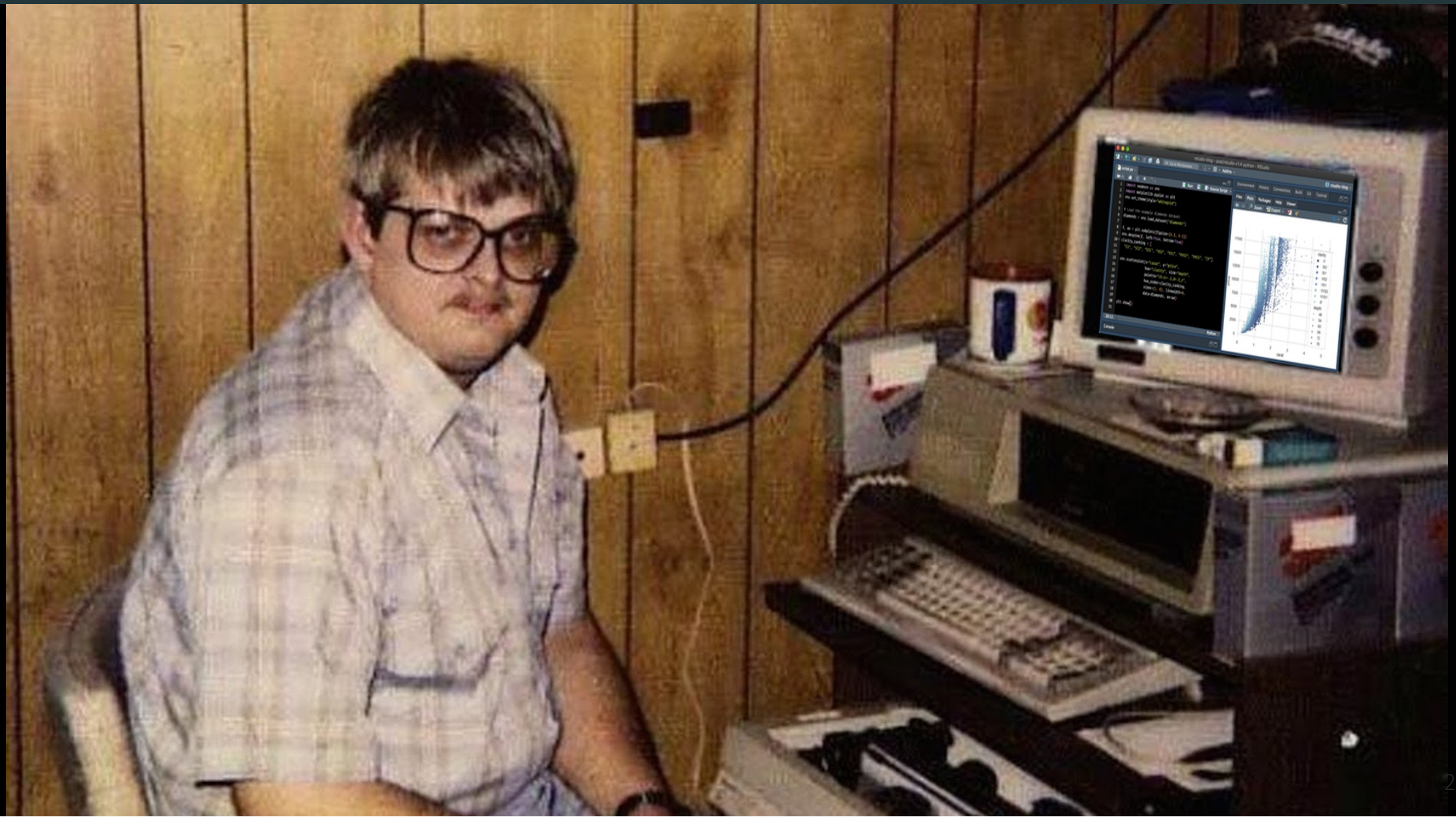


Fig. 2 | Descriptive specification curve. Each dot in the top panel (green area) depicts the marginal effect, estimated at sample means, of a hurricane having a female rather than male name; the dots vertically aligned below (white area) indicate the analytical decisions behind those estimates. A total of 1,728 specifications were estimated; to facilitate visual inspection, the figure depicts the 50 highest and lowest point estimates and a random subset of 200 additional ones, but the inferential statistics for specification curve analysis include all 1,728 specifications. NS, not significant.

Processing estimation output

Processing estimation output



Why model processing?

When estimating a model, we usually estimate parameters (or simulate distributions thereof). There is, however, more that we can take away from the estimation, including:

```
summary(model_out)

Call:
lm(formula = arr_delay ~ distance + origin, data = flights)

Residuals:
    Min      1Q  Median      3Q     Max
 -89.04  -24.00  -11.83    7.26 1281.45

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 13.4140488  0.1748144   76.73   <2e-16 ***
distance    -0.0040451  0.0001097  -36.87   <2e-16 ***
originJFK   -2.7042552  0.1887083  -14.33   <2e-16 ***
originLGA   -4.4561694  0.1935123  -23.03   <2e-16 ***
---
Sig. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 44.51 on 327342 degrees of freedom
(9430 observations deleted due to missingness)
Multiple R-squared: 0.005503, Adjusted R-squared: 0.005493
F-statistic: 603.7 on 3 and 327342 DF,  p-value: < 2.2e-16
```

Why model processing?

When estimating a model, we usually estimate parameters (or simulate distributions thereof). There is, however, more that we can take away from the estimation, including:

- **Estimated coefficients** and associated standard errors, T-statistics, p-values, confidence intervals

```
summary(model_out)

Call:
lm(formula = arr_delay ~ distance + origin, data = flights)

Residuals:
    Min      1Q  Median      3Q     Max
 -89.04  -24.00  -11.83    7.26 1281.45

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) 13.4140488  0.1748144   76.73   <2e-16 ***
distance    -0.0040451  0.0001097  -36.87   <2e-16 ***
originJFK   -2.7042552  0.1887083  -14.33   <2e-16 ***
originLGA   -4.4561694  0.1935123  -23.03   <2e-16 ***
---
Sig. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 44.51 on 327342 degrees of freedom
(9430 observations deleted due to missingness)
Multiple R-squared: 0.005503, Adjusted R-squared: 0.005493
F-statistic: 603.7 on 3 and 327342 DF,  p-value: < 2.2e-16
```

Why model processing?

When estimating a model, we usually estimate parameters (or simulate distributions thereof). There is, however, more that we can take away from the estimation, including:

- **Estimated coefficients** and associated standard errors, T-statistics, p-values, confidence intervals
- **Model summaries**, including goods of fit measures, information on model convergence, number of observations used

```
summary(model_out)

Call:
lm(formula = arr_delay ~ distance + origin, data = flights)

Residuals:
    Min      1Q  Median      3Q     Max
 -89.04  -24.00  -11.83    7.26 1281.45

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 13.4140488  0.1748144   76.73   <2e-16 ***
distance    -0.0040451  0.0001097  -36.87   <2e-16 ***
originJFK   -2.7042552  0.1887083  -14.33   <2e-16 ***
originLGA   -4.4561694  0.1935123  -23.03   <2e-16 ***
---
Sig. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 44.51 on 327342 degrees of freedom
(9430 observations deleted due to missingness)
Multiple R-squared: 0.005503, Adjusted R-squared: 0.005493
F-statistic: 603.7 on 3 and 327342 DF,  p-value: < 2.2e-16
```

Why model processing?

When estimating a model, we usually estimate parameters (or simulate distributions thereof). There is, however, more that we can take away from the estimation, including:

- **Estimated coefficients** and associated standard errors, T-statistics, p-values, confidence intervals
- **Model summaries**, including goods of fit measures, information on model convergence, number of observations used
- **Observation-level information** that arises from the estimated model, such as fitted/predicted values, residuals, estimates of influence

```
summary(model_out)

Call:
lm(formula = arr_delay ~ distance + origin, data = flights)

Residuals:
    Min      1Q  Median      3Q     Max
-89.04  -24.00  -11.83    7.26 1281.45

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 13.4140488  0.1748144   76.73   <2e-16 ***
distance    -0.0040451  0.0001097  -36.87   <2e-16 ***
originJFK   -2.7042552  0.1887083  -14.33   <2e-16 ***
originLGA   -4.4561694  0.1935123  -23.03   <2e-16 ***
---
Sig. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 44.51 on 327342 degrees of freedom
(9430 observations deleted due to missingness)
Multiple R-squared: 0.005503, Adjusted R-squared: 0.005493
F-statistic: 603.7 on 3 and 327342 DF,  p-value: < 2.2e-16
```

Processing estimation output in R

- Fitting a model returns an object of a certain model class (here: `lm`).
- Printing that object returns a quite minimalist set of information - just the input formula and coefficients.

```
R> model_out ← lm(arr_delay ~ distance + origin, data = flights)
R> class(model_out)
```

```
[1] "lm"
```

```
R> model_out
```

```
Call:
lm(formula = arr_delay ~ distance + origin, data = flights)

Coefficients:
(Intercept)     distance    originJFK    originLGA
  13.414049    -0.004045    -2.704255    -4.456169
```

Processing estimation output in R

- Fitting a model returns an object of a certain model class (here: `lm`).
- Printing that object returns a quite minimalist set of information - just the input formula and coefficients.
- The anatomy of the object is considerably more complex. It comes as a list of various components, including the coefficients, residuals, fitted values, and original model input.

```
R> names(model_out)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "na.action"     "contrasts"     "xlevels"       "call"
[13] "terms"         "model"
```

Processing estimation output in R

- Fitting a model returns an object of a certain model class (here: `lm`).
- Printing that object returns a quite minimalist set of information - just the input formula and coefficients.
- The anatomy of the object is considerably more complex. It comes as a list of various components, including the coefficients, residuals, fitted values, and original model input.
- There's no way to print this list the slide in full - it's just too long.

```
R> str(model_out)

List of 14
 $ coefficients : Named num [1:4] 13.41405 -0.00405 -2.70426 -4.45617
  ..- attr(*, "names")= chr [1:4] "(Intercept)" "distance" "originJFK" "
 $ residuals    : Named num [1:327346] 3.25 16.77 26.7 -22.33 -30.88 ...
  ..- attr(*, "names")= chr [1:327346] "1" "2" "3" "4" ...
 $ effects      : Named num [1:327346] -3945.1 1579.9 204.2 1025 -30.8 .
  ..- attr(*, "names")= chr [1:327346] "(Intercept)" "distance" "originJ
 $ rank         : int 4
 $ fitted.values: Named num [1:327346] 7.75 3.23 6.3 4.33 5.88 ...
  ..- attr(*, "names")= chr [1:327346] "1" "2" "3" "4" ...
 $ assign       : int [1:4] 0 1 2 2
 $ qr           :List of 5
  ..$ qr   : num [1:327346, 1:4] -5.72e+02 1.75e-03 1.75e-03 1.75e-03 1.7
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:327346] "1" "2" "3" "4" ...
  .. .. ..$ : chr [1:4] "(Intercept)" "distance" "originJFK" "originLGA"
  .. ..- attr(*, "assign")= int [1:4] 0 1 2 2
  .. ..- attr(*, "contrasts")=List of 1
  .. .. ..$ origin: chr "contr.treatment"
  ..$ qraux: num [1:4] 1 1 1 1
  ..$ pivot: int [1:4] 1 2 3 4
```

Processing estimation output in R

- However, there are some high-level functions we can apply to do something useful with the model object, including:
  - `coef()` to extract the coefficients

```
R> coef(model_out)
```

```
  (Intercept)      distance    originJFK    originLGA
13.414048769 -0.004045067 -2.704255237 -4.456169356
```

Processing estimation output in R

- However, there are some high-level functions we can apply to do something useful with the model object, including:
  - `coef()` to extract the coefficients
  - `fitted.values()` to extract the outcome values predicted by the model

```
R> coef(model_out)
```

```
(Intercept)    distance   originJFK   originLGA
13.414048769 -0.004045067 -2.704255237 -4.456169356
```

```
R> fitted.values(model_out)[1:5]
```

```
        1        2        3        4        5
7.750955 3.230064 6.304715 4.334768 5.875538
```

Processing estimation output in R

- However, there are some high-level functions we can apply to do something useful with the model object, including:
  - `coef()` to extract the coefficients
  - `fitted.values()` to extract the outcome values predicted by the model
  - `residuals()` to extract the residuals

```
R> coef(model_out)
```

```
  (Intercept)      distance    originJFK    originLGA
13.414048769 -0.004045067 -2.704255237 -4.456169356
```

```
R> fitted.values(model_out)[1:5]
```

```
        1        2        3        4        5
7.750955 3.230064 6.304715 4.334768 5.875538
```

```
R> residuals(model_out)[1:5]
```

```
          1           2           3           4           5
 3.249045  16.769936  26.695285 -22.334768 -30.875538
```

Processing estimation output in R

- However, there are some high-level functions we can apply to do something useful with the model object, including:
  - `coef()` to extract the coefficients
  - `fitted.values()` to extract the outcome values predicted by the model
  - `residuals()` to extract the residuals
  - `model.matrix()` to extract the matrix of original input variables (predictors)

```
R> coef(model_out)
```

```
  (Intercept)     distance    originJFK    originLGA
13.414048769 -0.004045067 -2.704255237 -4.456169356
```

```
R> fitted.values(model_out)[1:5]
```

```
       1        2        3        4        5
7.750955 3.230064 6.304715 4.334768 5.875538
```

```
R> residuals(model_out)[1:5]
```

```
         1          2          3          4          5
 3.249045  16.769936  26.695285 -22.334768 -30.875538
```

```
R> model.matrix(model_out) %>% head(4)
```

```
  (Intercept) distance originJFK originLGA
1           1     1400         0         0
2           1     1416         0         1
3           1     1089         1         0
4           1     1576         1         0
```

Processing estimation output in R

- To learn more about the estimated model, we can apply the `summary()` function.
- The `summary` method is specific to the model class it is applied to (here: "`lm`"). To learn more, you'd have to call, e.g., `?summary.lm` or `?summary.glm`.

```
R> summary(model_out)
```

```
Call:
lm(formula = arr_delay ~ distance + origin, data = flights)

Residuals:
    Min      1Q  Median      3Q     Max
 -89.04  -24.00  -11.83    7.26 1281.45

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 13.4140488  0.1748144   76.73   <2e-16 ***
distance    -0.0040451  0.0001097  -36.87   <2e-16 ***
originJFK   -2.7042552  0.1887083  -14.33   <2e-16 ***
originLGA   -4.4561694  0.1935123  -23.03   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 44.51 on 327342 degrees of freedom
  (9430 observations deleted due to missingness)
Multiple R-squared:  0.005503,    Adjusted R-squared:  0.005493
F-statistic: 603.7 on 3 and 327342 DF,  p-value: < 2.2e-16
```

Processing estimation output in R

- To learn more about the estimated model, we can apply the `summary()` function.
- The `summary` method is specific to the model class it is applied to (here: "`lm`"). To learn more, you'd have to call, e.g., `?summary.lm` or `?summary.glm`.
- The function creates more than a printed summary in the console. It returns an object of class `summary.lm`, which can be further dissected.

```
R> class(summary(model_out))
```

```
[1] "summary.lm"
```

Processing estimation output in R

- To learn more about the estimated model, we can apply the `summary()` function.
- The `summary` method is specific to the model class it is applied to (here: "`lm`"). To learn more, you'd have to call, e.g., `?summary.lm` or `?summary.glm`.
- The function creates more than a printed summary in the console. It returns an object of class `summary.lm`, which can be further dissected.
- Again, there's no way to print this list on the slide in full - it's just too long.

```
R> str(summary(model_out))
```

```
List of 12
 $ call         : language lm(formula = arr_delay ~ distance + origin, da
 $ terms        :Classes 'terms', 'formula'  language arr_delay ~ distanc
  .. ..- attr(*, "variables")= language list(arr_delay, distance, origin
  .. ..- attr(*, "factors")= int [1:3, 1:2] 0 1 0 0 0 1
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:3] "arr_delay" "distance" "origin"
  .. .. .. ..$ : chr [1:2] "distance" "origin"
  .. ..- attr(*, "term.labels")= chr [1:2] "distance" "origin"
  .. ..- attr(*, "order")= int [1:2] 1 1
  .. ..- attr(*, "intercept")= int 1
  .. ..- attr(*, "response")= int 1
  .. ..- attr(*, ".Environment")⩽environment: R_GlobalEnv>
  .. ..- attr(*, "predvars")= language list(arr_delay, distance, origin)
  .. ..- attr(*, "dataClasses")= Named chr [1:3] "numeric" "numeric" "cha
  .. .. ..- attr(*, "names")= chr [1:3] "arr_delay" "distance" "origin"
 $ residuals    : Named num [1:327346] 3.25 16.77 26.7 -22.33 -30.88 ...
  ..- attr(*, "names")= chr [1:327346] "1" "2" "3" "4" ...
 $ coefficients : num [1:4, 1:4] 13.41405 -0.00405 -2.70426 -4.45617 0.1
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:4] "(Intercept)" "distance" "originJFK" "originLGA"
```

Dissecting model objects

The problem

"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. This makes it more difficult to combine results from multiple models. 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. (...) 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."

Hadley Wickham, "Tidy Data"

The solution?

See next slide!

Processing estimation output with broom

`broom` is a suite of tools that summarizes key information about models. It takes the messy output of built-in functions in R, such as `lm` or `t.test`, and turns them into tidy `tibbles()` (think: dataframes). The output is not ready for publication but an important intermediary step that makes post-processing of estimation results more convenient. It is part of the tidyverse and tidymodels.

There are three key `broom` verbs that you need to learn.[1]

1. `tidy()`: Summarizes information about model components.
2. `glance()`: Reports information about the entire model.
3. `augment()`: Adds information about observations to a dataset.

[1] For a more detailed and comprehensive introduction, see the official documentation at https://broom.tidymodels.org/.

Tidy model objects with tidy()

`broom`'s `tidy()` function extracts the coefficient block (the model component) together with inferential statistics:

```
R> broom::tidy(model_out, conf.int = TRUE, conf.level = 0.95)
```

```
  # A tibble: 4 × 7
    term        estimate std.error statistic   p.value conf.low conf.high
    <chr>          <dbl>     <dbl>     <dbl>     <dbl>    <dbl>     <dbl>
  1 (Intercept) 13.4      0.175         76.7 0          13.1      13.8
  2 distance    -0.00405  0.000110     -36.9 5.53e-297 -0.00426  -0.00383
  3 originJFK   -2.70     0.189        -14.3 1.46e- 46 -3.07     -2.33
  4 originLGA   -4.46     0.194        -23.0 3.04e-117 -4.84     -4.08
```

Tidy model objects with tidy()

`broom`'s `tidy()` function extracts the coefficient block (the model component) together with inferential statistics:

```
R> broom::tidy(model_out, conf.int = TRUE, conf.level = 0.95)
```

```
  # A tibble: 4 × 7
    term        estimate std.error statistic   p.value conf.low conf.high
    <chr>          <dbl>     <dbl>     <dbl>     <dbl>    <dbl>     <dbl>
  1 (Intercept) 13.4      0.175         76.7 0          13.1      13.8
  2 distance    -0.00405  0.000110     -36.9 5.53e-297 -0.00426  -0.00383
  3 originJFK   -2.70     0.189        -14.3 1.46e- 46 -3.07     -2.33
  4 originLGA   -4.46     0.194        -23.0 3.04e-117 -4.84     -4.08
```

Here, we also extract the upper and lower bounds on the 95% confidence intervals for the estimates. What makes the function so convenient is the fact that the output comes as a tidy `tibble` with useful variable names.

What exactly is extracted depends on the model type. You can learn more about the tidying function by typing `?tidy.[model class]`, e.g.: `?tidy.lm`.

Summarize model statistics with glance()

`broom`'s `glance()` function extracts summary statistics of the model and provides them in a single-row `tibble`:

```
R> broom::glance(model_out)
```

```
  # A tibble: 1 × 12
    r.squared adj.r.squared sigma statistic p.value    df   logLik     AIC     BIC
        <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>    <dbl>   <dbl>   <dbl>
  1   0.00550       0.00549  44.5      604.       0     3  -1.71e6  3.41e6  3.41e6
  # … with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Augment data with model information with augment()

`broom`'s `augment()` function adds model information about each observation in a dataset, including, e.g.:

- predicted values (in the `.fitted` column)
- residuals (`.resid`)
- standard errors of fitted values (`.se.fit`; optional)

```
R> broom::augment(model_out, se_fit = TRUE) %>% head(3)
```

```
# A tibble: 3 × 11
  .rownames arr_delay distance origin .fitted .se.fit .resid       .hat .sigma
  <chr>         <dbl>    <dbl> <chr>    <dbl>   <dbl>  <dbl>      <dbl>  <dbl>
1 1                11     1400 EWR       7.75   0.135   3.25 0.00000922   44.5
2 2                20     1416 LGA       3.23   0.156  16.8  0.0000123    44.5
3 3                33     1089 JFK       6.30   0.136  26.7  0.00000938   44.5
# … with 2 more variables: .cooksd <dbl>, .std.resid <dbl>
```

Augment data with model information with augment()

`broom`'s `augment()` function adds model information about each observation in a dataset, including, e.g.:

- predicted values (in the `.fitted` column)
- residuals (`.resid`)
- standard errors of fitted values (`.se.fit`; optional)

```
R> broom::augment(model_out, se_fit = TRUE) %>% head(3)
```

```
# A tibble: 3 × 11
  .rownames arr_delay distance origin .fitted .se.fit .resid       .hat .sigma
  <chr>         <dbl>    <dbl> <chr>    <dbl>   <dbl>  <dbl>      <dbl>  <dbl>
1 1                11     1400 EWR       7.75   0.135   3.25 0.00000922   44.5
2 2                20     1416 LGA       3.23   0.156  16.8  0.0000123    44.5
3 3                33     1089 JFK       6.30   0.136  26.7  0.00000938   44.5
# … with 2 more variables: .cooksd <dbl>, .std.resid <dbl>
```

It is also possible to pass on data that was not used during model fitting using the `newdata` argument. This requires that at least all predictor variable columns used to fit the model are present. Providing new data can be useful if one is interested to generate predictions for a test set.

Unlocking the power of broom with multiple models

The true power of broom unfolds in settings where you want to combine results from multiple analyses (using subgroups of data, different models, bootstrap replicates of the original data frame, permutations, imputations, ...).

Unlocking the power of broom with multiple models

The true power of broom unfolds in settings where you want to combine results from multiple analyses (using subgroups of data, different models, bootstrap replicates of the original data frame, permutations, imputations, ...).

Does this ring a bell? Well, let's go back to our covariate selection sensitivity analysis. Recall that in **Steps 1 to 3**, we had specified and run 63 models. Let's evaluate the results now. First, extract the results in a tidy fashion:

Unlocking the power of broom with multiple models

The true power of broom unfolds in settings where you want to combine results from multiple analyses (using subgroups of data, different models, bootstrap replicates of the original data frame, permutations, imputations, ...).

Does this ring a bell? Well, let's go back to our covariate selection sensitivity analysis. Recall that in **Steps 1 to 3**, we had specified and run 63 models. Let's evaluate the results now. First, extract the results in a tidy fashion:

Step 4 (continuing the analysis from above): Extract results from all models

```
R> models_broom ← map(models_list, broom::tidy)
R> models_broom[[1]] # inspect one list entry
```

```
  # A tibble: 2 × 5
    term        estimate std.error statistic   p.value
    <chr>          <dbl>     <dbl>     <dbl>     <dbl>
  1 (Intercept)    -5.86   0.187       -31.3 1.50e-205
  2 dep_delay       1.02   0.00462     221.  0
```

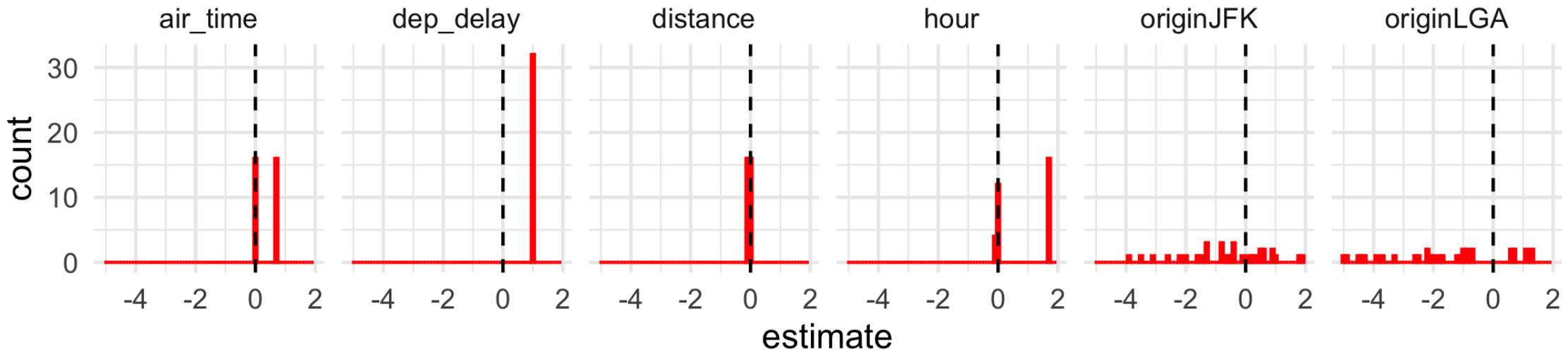
Next, let's merge them all into one data frame:

```
R> models_broom_df ← map_dfr(models_broom, rbind)
```

Unlocking the power of broom with multiple models

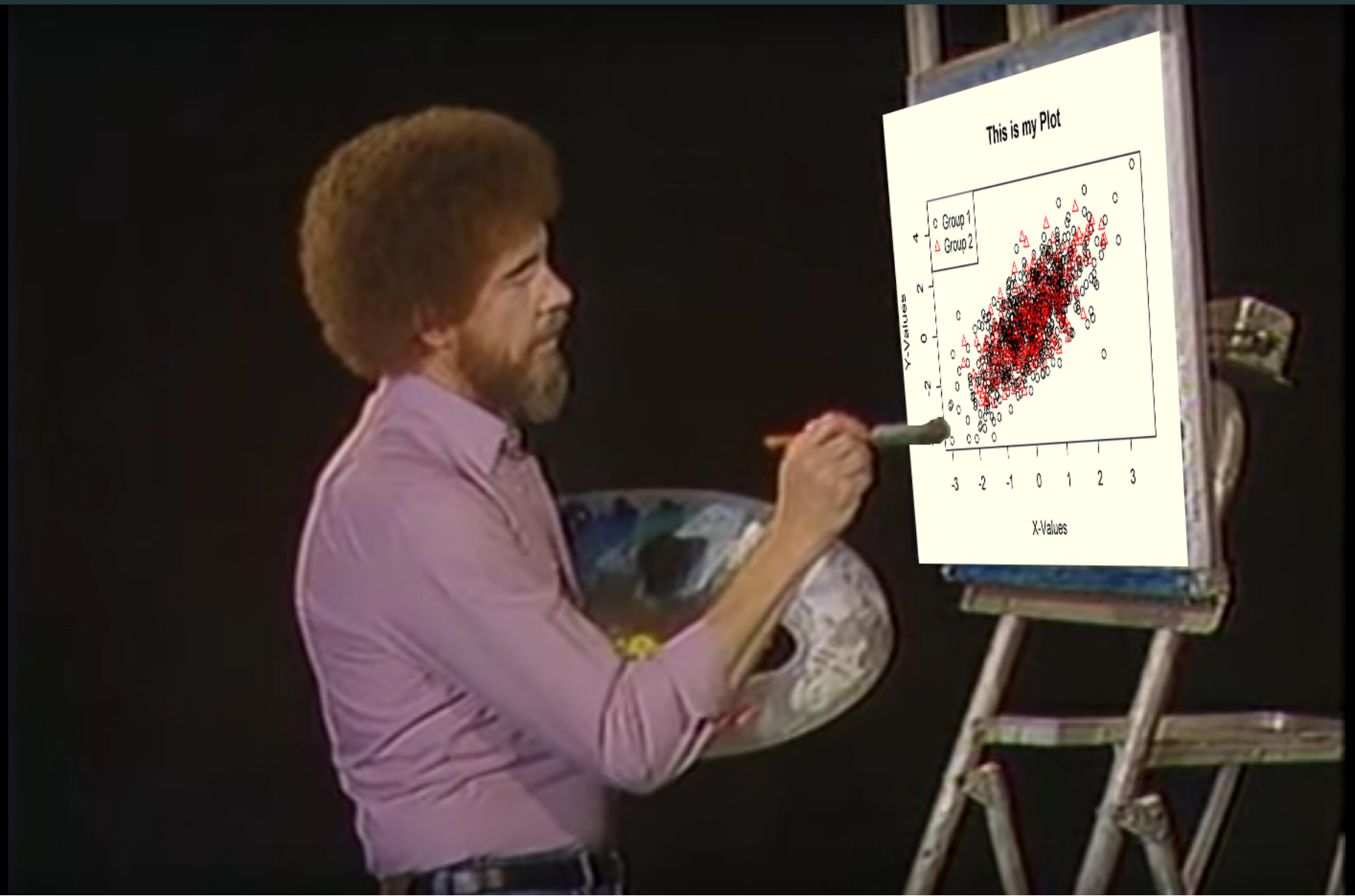
Step 5: Summarize the estimates for subset of key predictors

```
R> models_broom_df %>%
+   filter(!str_detect(term, "Intercept|carrier")) %>%
+   ggplot(aes(estimate)) +
+   geom_histogram(binwidth = .1, color = "red") +
+   geom_vline(xintercept = 0, linetype="dashed") +
+   facet_grid(cols = vars(term), scales = "free_y") +
+   theme_minimal()
```



Reporting modeling results

Reporting modeling results



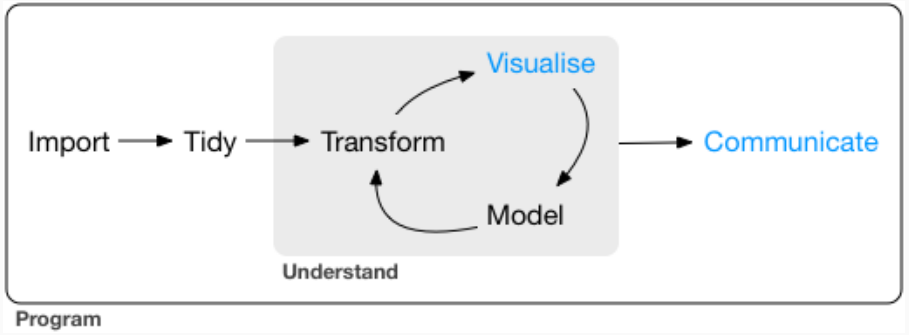
Reporting modeling results

Why good reporting is as important as good modeling

- Hardly anybody will read your code. Most stakeholders will not even be able to understand what you've done.
- Not all components of your model are equally relevant.
- Good communication of your model can save a lot of time a space.

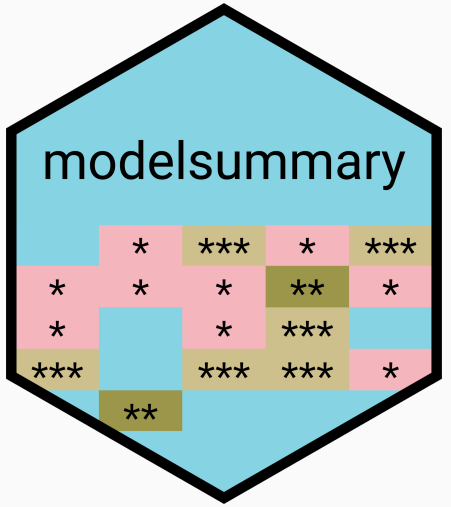
Making reporting part of the workflow

- Our vision of the data science workflow is to **automate as much as possible** in order to have time for the really important decisions.
- Since reporting results is usually at the end of the workflow, this step is affected by any change in the previous steps. Any manual work here hurts twice.
- Reporting and publishing results should be seen as part of the workflow. Even if we don't work with RMarkdown to write our reports, we want avoid copy-and-paste work into other software.



Summarizing models (and data) with modelsummary

`modelsummary` is a suite of tools to create table and plot summaries of models and data. It supports hundreds of model types out-of-the-box and the output can be saved to a wide variety of formats, including HTML, PDF, Text/Markdown, LaTeX, MS Word, JPG, and PNG.



There are three key `modelsummary` verbs that you need to learn.[1]

1. `modelsummary()`: Create regression tables with side-by-side models.
2. `modelplot()`: Create coefficient plots of model results.
3. `datasummary_*()`: Create data summaries such as cross-tabs or balance tables.

[1] There is more in `modelsummary` than what we can cover today. Have a glimpse at Vincent Arel-Bundock's page.

Creating regression tables with modelsummary()

- `modelsummary()` takes one or several models as input. Multiple models are provided as (optionally named) list.
- The extraction of information (estimates, standard errors, model summaries etc.) is taken care of by the function.
- On this slide you see how `modelsummary()` generates content ready to be rendered as HTML table. With the `output` argument, we can also ask the function to creat `.tex`, `.rmd`, `.txt`, `.png`, and `.jpg`.
- Although the defaults are good, be sure to refine the table before publishing it.

```
R> modelsummary(list(model1_out, model2_out, model3_out))
```

	Model 1	Model 2	Model 3
(Intercept)	−5.899	10.829	−3.213
	(0.033)	(0.136)	(0.056)
dep_delay	1.019		1.018
	(0.001)		(0.001)
distance		−0.004	−0.003
		(0.000)	(0.000)
Num.Obs.	327346	327346	327346
R2	0.837	0.004	0.839
R2 Adj.	0.837	0.004	0.839
AIC	2822272.7	3414551.8	2818708.3
BIC	2822304.8	3414583.9	2818751.1

Modifying regression tables

- Here's one example for that.
- With `estimate` we define a `glue` string to display estimates alongside confidence intervals.
- We suppress uncertainty statistics with `statistic`.
- We omit any goodness-of-fit stats with `gof_omit` and a regular expression.
- We provide a `title`.

```
R> # estimate three toy models
R> model1_out ← lm(arr_delay ~ dep_delay, data = flights)
R> model2_out ← lm(arr_delay ~ distance, data = flights)
R> model3_out ← lm(arr_delay ~ dep_delay + distance, data = flights)
R> models ← list(model1_out, model2_out, model3_out)
R>
R> # create table
R> modelsummary(models,
+               estimate = "{estimate} [{conf.low}, {conf.high}]",
+               statistic = NULL,
+               gof_omit = ".+",
+               title = "Linear regression of flight delay at arrival (in mins)")
```

Linear regression of flight delay at arrival (in mins)

	Model 1	Model 2	Model 3
(Intercept)	-5.899 [-5.964, -5.835]	10.829 [10.564, 11.095]	-3.213 [-3.322, -3.104]
dep_delay	1.019 [1.018, 1.021]		1.018 [1.017, 1.020]
distance		-0.004 [-0.004, -0.004]	-0.003 [-0.003, -0.002]

Modifying regression tables (cont.)

The `modelsummary()` function is extremely versatile. The defaults are good, but it will pay off to invest some time to learn the details, which are documented here.

In addition, it supports other table-making packages to further customize the appearance of tables. The details are documented here.

But I don't want to force anything on you. As always in R, there are several other excellent packages that help to create tables, including:

- `gtsummary` by Daniel Sjoberg
- `textreg` by Philip Leifeld
- `stargazer` by Marek Hlavac
- `sjPlot` by Daniel Lüdecke

In any case, do invest some time in learning the function's options and in actually producing readable and informative tables before publishing them. (1-3 hours per table are fine!)

```
R> modelsummary(
+   models,
+   output = "default",
+   fmt = 3,
+   estimate = "estimate",
+   statistic = "std.error",
+   vcov = NULL,
+   conf_level = 0.95,
+   stars = FALSE,
+   coef_map = NULL,
+   coef_omit = NULL,
+   coef_rename = NULL,
+   gof_map = NULL,
+   gof_omit = NULL,
+   group = term ~ model,
+   group_map = NULL,
+   add_rows = NULL,
+   align = NULL,
+   notes = NULL,
+   title = NULL,
+   escape = TRUE,
+   ...
+ )
```

modelsummary() tables: more examples

modelsummary package for R					
	Donations		Crimes (persons)		Crimes (property)
	OLS 1	Poisson 1	OLS 2	Poisson 2	OLS 3
Literacy (%)	-39.121	0.003***	3.680	-0.000***	-68.507***
	(37.052)	(0.000)	(46.552)	(0.000)	(18.029)
Priests/capita[1]	15.257		77.148**		-16.376
	(25.735)		(32.334)		(12.522)
Patents/capita		0.011***		0.001***	
		(0.000)		(0.000)	
Constant	7948.667***	8.241***	16259.384***	9.876***	11243.544***
	(2078.276)	(0.006)	(2611.140)	(0.003)	(1011.240)
Num.Obs.	86	86	86	86	86
R2	0.020		0.065		0.152
Adj.R2	-0.003		0.043		0.132
AIC	1740.8	274160.8	1780.0	257564.4	1616.9
BIC	1750.6	274168.2	1789.9	257571.7	1626.7
Log.Lik.	-866.392	-137077.401	-886.021	-128779.186	-804.441

[1] Very important variable.

* p < 0.1, ** p < 0.05, *** p < 0.01

The most important parameter is printed in red.

Table 1: modelsummary package for R

	Donations		Crimes (person)		Crimes (property)
	OLS 1	Poisson 1	OLS 2	Poisson 2	OLS 3
Literacy (%)	-39.121	0.003***	3.680	-0.000***	-68.507***
	(37.052)	(0.000)	(46.552)	(0.000)	(18.029)
Priests/capita	**15.257**		**77.148****		**-16.376**
	(25.735)		(32.334)		(12.522)
Patents/capita		0.011***		0.001***	
		(0.000)		(0.000)	
Constant	7948.667***	8.241***	16259.384***	9.876***	11243.544***
	(2078.276)	(0.006)	(2611.140)	(0.003)	(1011.240)
Num.Obs.	86	86	86	86	86
R2	0.020		0.065		0.152
Adj.R2	-0.003		0.043		0.132
AIC	1740.8	274160.8	1780.0	257564.4	1616.9
BIC	1750.6	274168.2	1789.9	257571.7	1626.7
Log.Lik.	-866.392	-137077.401	-886.021	-128779.186	-804.441

* p < 0.1, ** p < 0.05, *** p < 0.01

First custom note to contain text.

Second custom note with different content.

Tables vs. plots to communicate model results

The limits of tables

- Tables of coefficients work ok when models are linear and additive.
- They are good to communicate "precise" information.[1]
- The are, however, less informative for
  - non-linear relationships between x and y (x^2, $log(x)$, etc.),
  - interaction effects,
  - models for categorical data.

The promise of plots

- *Coefficient plots* can make it more straightforward to focus on two key features of estimated parameters: effect size and uncertainty.
- What's more, they make comparisons across effects and models much easier. We humans are visual animals.
- Other plots can go a long way to display other implications of models that are not visible from a tabular output, e.g., *predicted probability plots*, *marginal effects plots*, ...

[1] But sometimes give more precision than warranted. As a rule of thumb, never report more than three decimal points. In most cases, 0-2 is enough. Your estimates are less precise than that anyway.

Coefficient plots in the wild

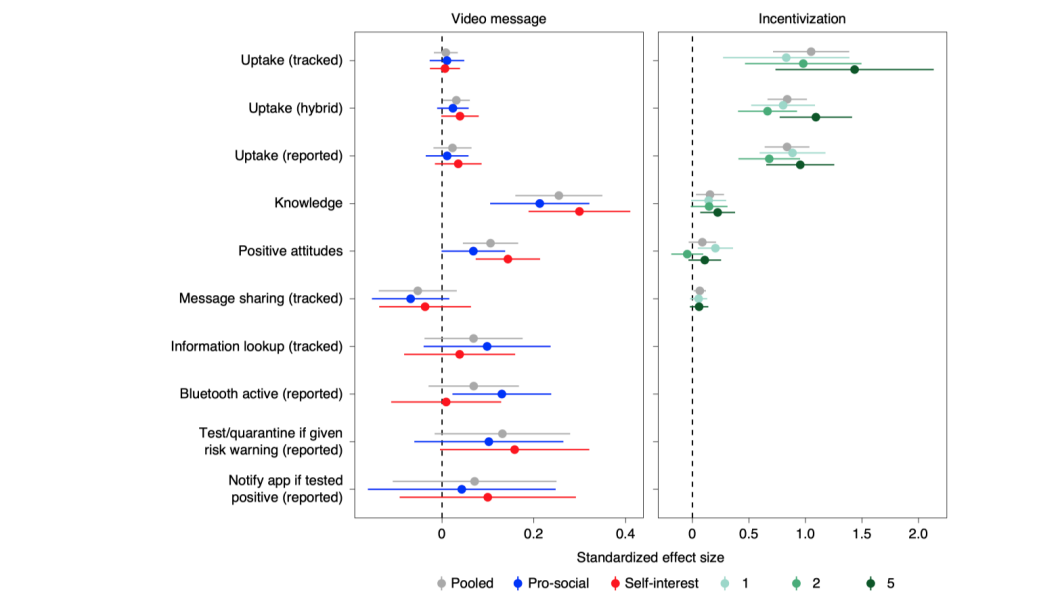
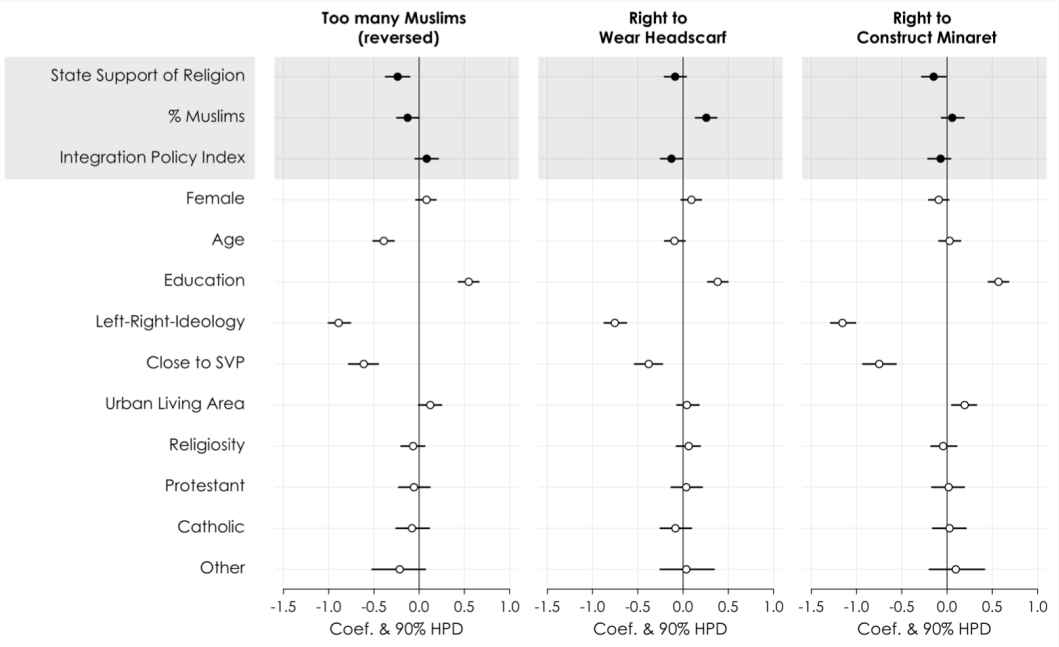


Fig. 3 | Effect of message and incentive treatments on uptake, knowledge, attitudes and behaviour. Each plot shows standardized ITT estimates with 95% CIs from fully saturated ordinary least squares regression models fit using the pre-registered LASSO covariate selection procedure. The video message sample comprises $n = 2{,}044$, 1,356 and 1,337 respondents for estimation of the pooled, pro-social and self-interest treatment effects, respectively. The incentive sample comprises $n = 1{,}015$, 513, 516 and 494 respondents for estimation of the pooled, €1, €2 and €5 treatment effects, respectively.

`Credit` Munzert et al. 2020

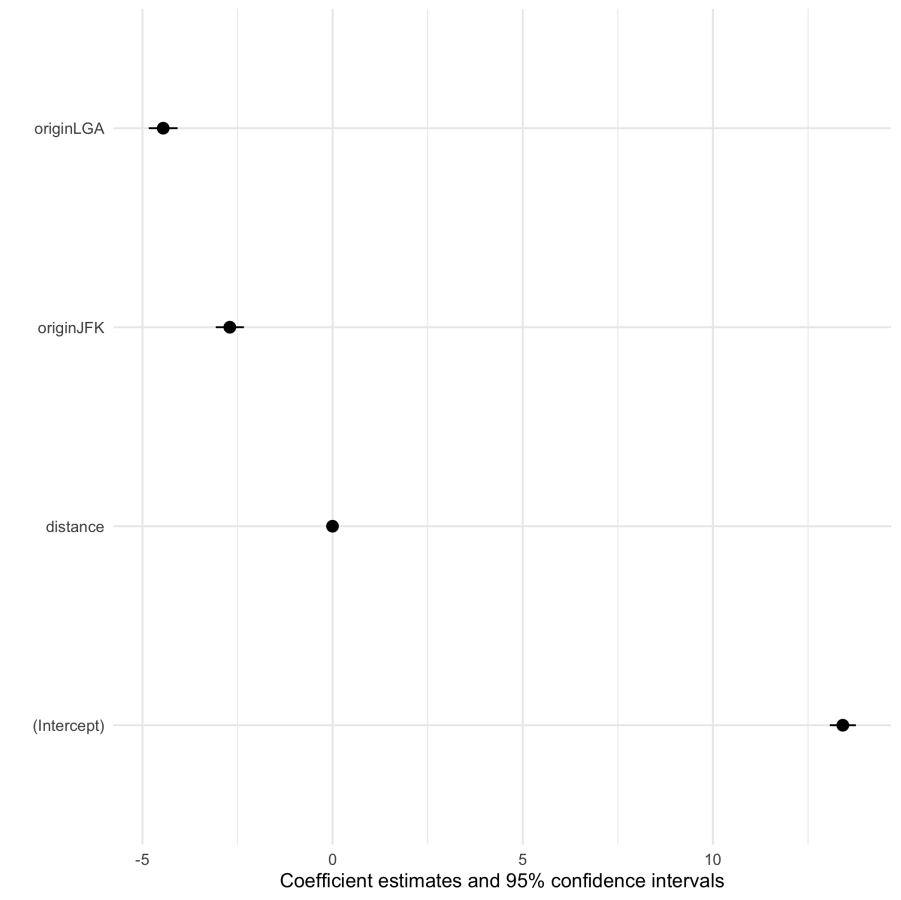


`Credit` Helbling and Traunmüller 2016

Creating coefficient plots with modelplot()

- `modelplot()` takes one or several models as input. Multiple models are provided as (optionally named) list.
- Again, the extraction of information (estimates, standard errors, model summaries etc.) is taken care of by the function.
- The graphs produced by `modelplot()` are simple `ggplot2` objects. They can be post-processed (and exported) accordingly.
- Although the defaults are good, be sure to refine the plot before publishing it.

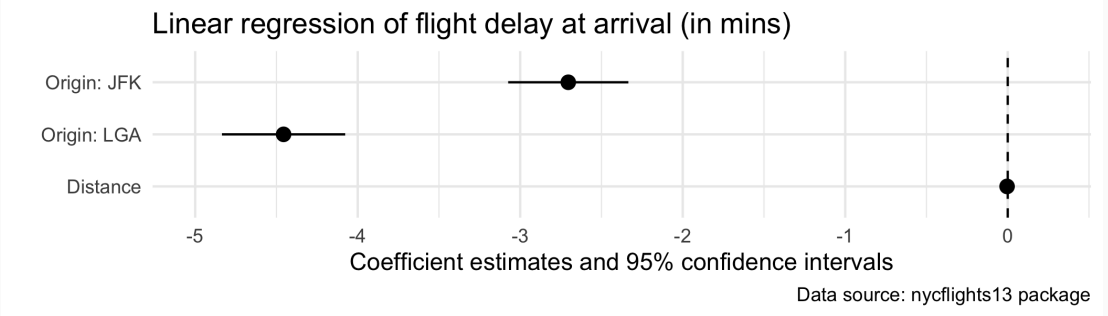
```
R> modelplot(model_out)
```



Modifying coefficient plots

- Here's one example for that.
- We provide intuitive variable names with `coef_map`.
- We drop the intercept with `coef_omit`.
- We make more layout adaptations with `ggplot2` functions that make the plot better readable.

```
R> cm ← c("distance" = "Distance",
+         "originLGA" = "Origin: LGA",
+         "originJFK" = "Origin: JFK")
R> modelplot(model_out,
+           coef_omit = "Interc",
+           coef_map = cm) +
+   xlim(-5, .25) +
+   geom_vline(xintercept = 0, linetype="dashed") +
+   labs(title = "Linear regression of flight delay at arrival (in mins)",
+        caption = "Data source: nycflights13 package") +
+   theme_minimal()
```



Working with interpretable effect sizes

Take care when plotting effect sizes

- One of the major perks of coefficient plots is the comparability of effects across coefficients.
- This can be, however, also one of the major problems of these kinds of plots
- In order for them to make visual sense, the underlying covariates have to be inherently comparable. By showing slopes, the plot shows the effect of a unit change in each covariate on the outcome, but unit changes may not be comparable across variables.
- Also check out the documentation of the `effectsize` package for a more thorough discussion of the problem (and how to tackle it).

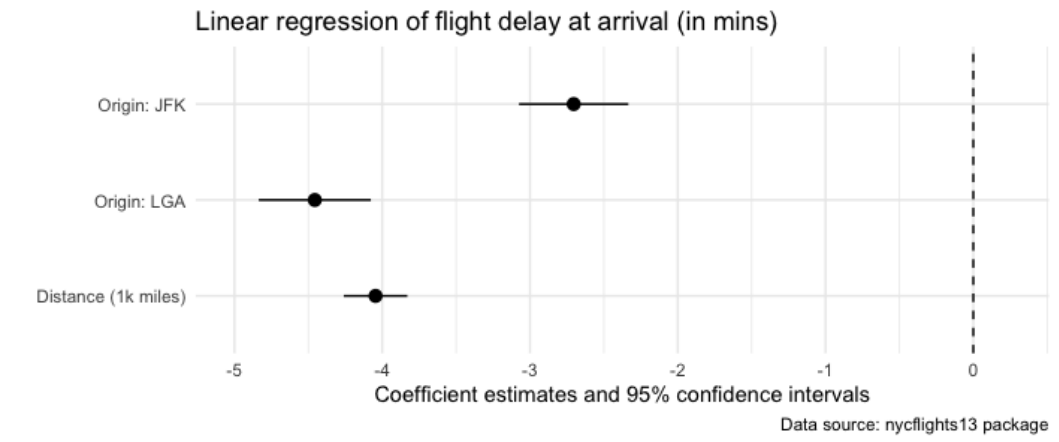
Addressing the issue

- There are several ways to address the problem, including:
  - Rescale variables to show intuitive unit changes in X (e.g., 1km instead of 1m)
  - Rescale to full scale (minimum to maximum) changes in X
  - Standardize variables to show standard deviation changes in X
- Note that any rescaling operation also affects how you interpret the coefficients (and we're only talking about the linear case!). Sometimes it also makes sense to standardize the response variable. In that case, the coefficients can be interpreted as the change in the response in standard deviations for a 1 unit change in the predictor (whatever that is).

Working with interpretable effect sizes (cont.)

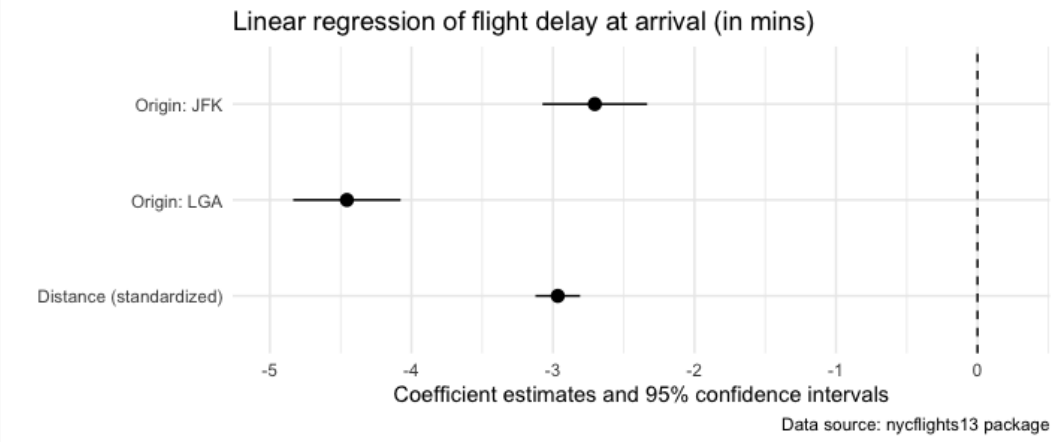
Rescaling (1 unit = 1000 miles)

```
R> # rescale continuous variable
R> flights$distance1kmiles ← flights$distance/1000
R> model_out_kmiles ← lm(arr_delay ~
+        distance1kmiles + origin, data = flights)
R>
R> # plot model (detailed fine-tuning not shown)
R> modelplot(model_out_kmiles)
```



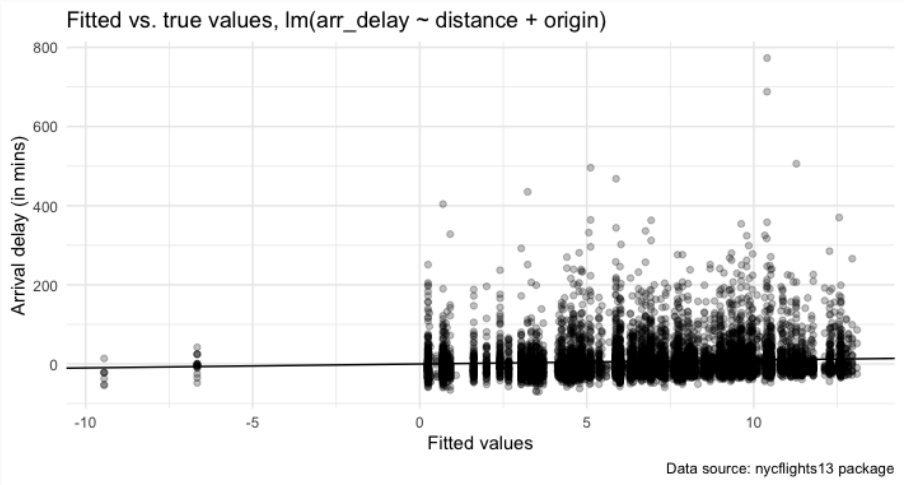
Standardization (1 unit = 1 s.d. on the covariate)

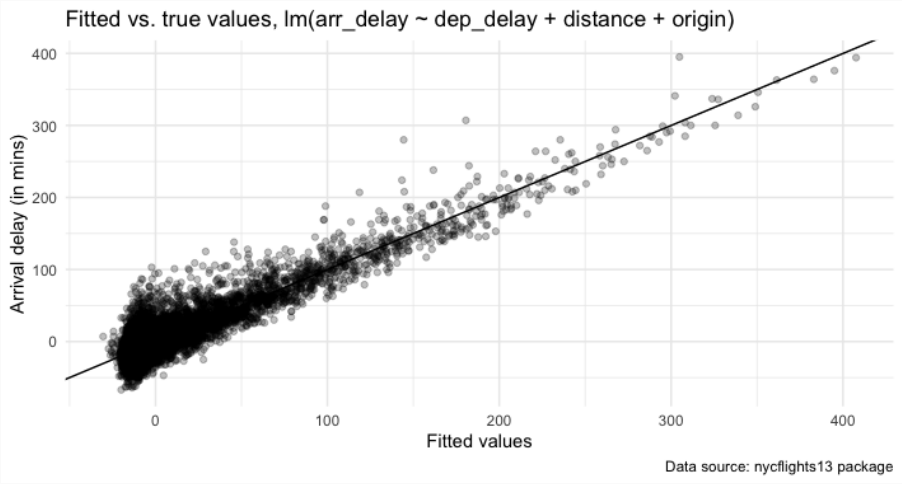
```
R> # rescale continuous variable
R> flights$distance_std ← standardize(flights$dista
R> model_out_std ← lm(arr_delay ~
+        distance_std + origin, data = flights)
R>
R> # plot model (detailed fine-tuning not shown)
R> modelplot(model_out_std)
```



More reporting with plots

- There is much, much more that can be done with plots and model reporting (stay tuned for the session on visualization!).
- Always be aware about what a model gives you and which relationships you want to explore or highlight.
- As another example, see the fitted vs. true plots on the right. The upper scatter plot compares fitted (x-axis) vs. true (y-axis) values from our standard model. It illustrates a really poor fit.
- The lower scatter plot compares fitted-vs-true for a slight modification of our standard model in which we also take `dep_delay` into account as predictor. Apparently, its a very powerful one.





Summary

Summary

Some final bits of advice that didn't fit on the main slides:

1. Before you actually run models, **describe the data**. That's not only for you. Every report should begin with a visualization of the phenomena of interest, plotting the rawest data available that is also legible in a graph.

2. **Plot early, and plot often**! Visualization is not only a great tool for communication, but also for exploration and statistical analysis (more on that later).

3. **Fit many models**. At least if you don't commit to one particular subset (→ preregistration). Think in terms of series of models, starting with the utterly simple and continuing through to the hopelessly complex.

4. **Table results with care**. Please don’t just report which effects were significant and which were non-significant. Please don’t report just p-values. Please don’t just report the estimated effects of the significant effects. Report all estimates that you also discuss.

5. **Invest more time in refining plots** than you planned to. Good figures aren't only plots. Think of informative headers and notes. Ideally, your figures are self-contained.

Coming up

Assignment

Assignment 4 is about to go online on GitHub Classroom. Check it out and start modeling!

Next lecture

Visualization. But first: Mid-term exam week - no class, enjoy the break!