Introduction to Data Science

Dr T Prathima, Dept of IT, CBIT(A), Hyderabad

Unit-1

Introduction to data science: The Data Science Process: Roles in a data science project, Stages of a data science project, Setting expectations.

Starting with R and data: Starting with R,working with data from files, Working with relational databases.

Exploring data: Using Summary Statistics to spot problems, Spotting problems with graphics and visualization.

Zumel, N., Mount, J., &Porzak, J., "Practical data science with R", 2nd edition. Shelter Island, NY: Manning, 2019.

Data Science Process

- The statistician William S. Cleveland defined data science as an interdisciplinary field larger than statistics itself.
- We define data science as managing the process that can transform hypotheses and data into actionable predictions.
- Typical predictive analytic goals include predicting who will win an election, what products will sell well together, which loans will default, and which advertisements will be clicked on.
- The data scientist is responsible for acquiring and managing the data, choosing the modeling technique, writing the code, and verifying the results.
- Because data science draws on so many disciplines, many of the best data scientists we meet started as programmers, statisticians, business intelligence analysts, or scientists.
- By adding a few more techniques to their repertoire, they became excellent data scientists.

Examples

- Much of the theoretical basis of data science comes from statistics.
- But data science as we know it is strongly influenced by technology and software engineering methodologies, and has largely evolved in heavily computer science– and information technology– driven groups.
- Engineering flavor of data science - some famous examples:
  - Amazon’s product recommendation systems
  - Google’s advertisement valuation systems
  - LinkedIn’s contact recommendation system
  - Twitter’s trending topics
  - Walmart’s consumer demand projection systems

Features

- **All of these systems are *built off large datasets*.** That’s not to say they’re all in the realm of big data. But none of them could’ve been successful if they’d only used small datasets. To manage the data, these systems require concepts from computer science: database theory, parallel programming theory, streaming data techniques, and data warehousing.
- **Most of these systems are *online or live*.** Rather than producing a single report or analysis, the data science team deploys a decision procedure or scoring procedure to either directly make decisions or directly show results to a large number of end users. The production deployment is the last chance to get things right, as the data scientist can’t always be around to explain defects.
- **All of these systems are *allowed to make mistakes* at some non-negotiable rate.**
- **None of these systems are *concerned with cause*.** They’re successful when they find useful correlations and are not held to correctly sorting cause from effect.

Supplementary material

- https://github.com/WinVector/PDSwR2
- *Beyond Spreadsheets with R* by Jonathan Carroll (Manning, 20108) or
- *R in Action* by Robert Kabacoff (now available in a second edition http://www.manning.com/kabacoff2/), along with the text's associated website
- *Quick-R* (http://www.statmethods.net).
- For statistics, we recommend *Statistics,* Fourth Edition, by David Freedman, Robert Pisani, and Roger Purves (W. W. Norton & Company, 2007).
- zip file (https://github.com/WinVector/PDSwR2/raw/master/CodeExamples.zip)
- https://forums.manning.com/forums/practical-data-science-with-rsecond-edition

Defining Data Science

- Data science is a cross-disciplinary practice that draws on methods from data engineering, descriptive statistics, data mining, machine learning, and predictive analytics.
- Much like operations research, data science focuses on implementing data-driven decisions and managing their consequences.
- The data scientist is responsible for guiding a data science project from start to finish.
- Success in a data science project comes not from access to any one exotic tool, but from having quantifiable goals, good methodology, cross-discipline interactions, and a repeatable workflow.

Roles in a Data Science Project

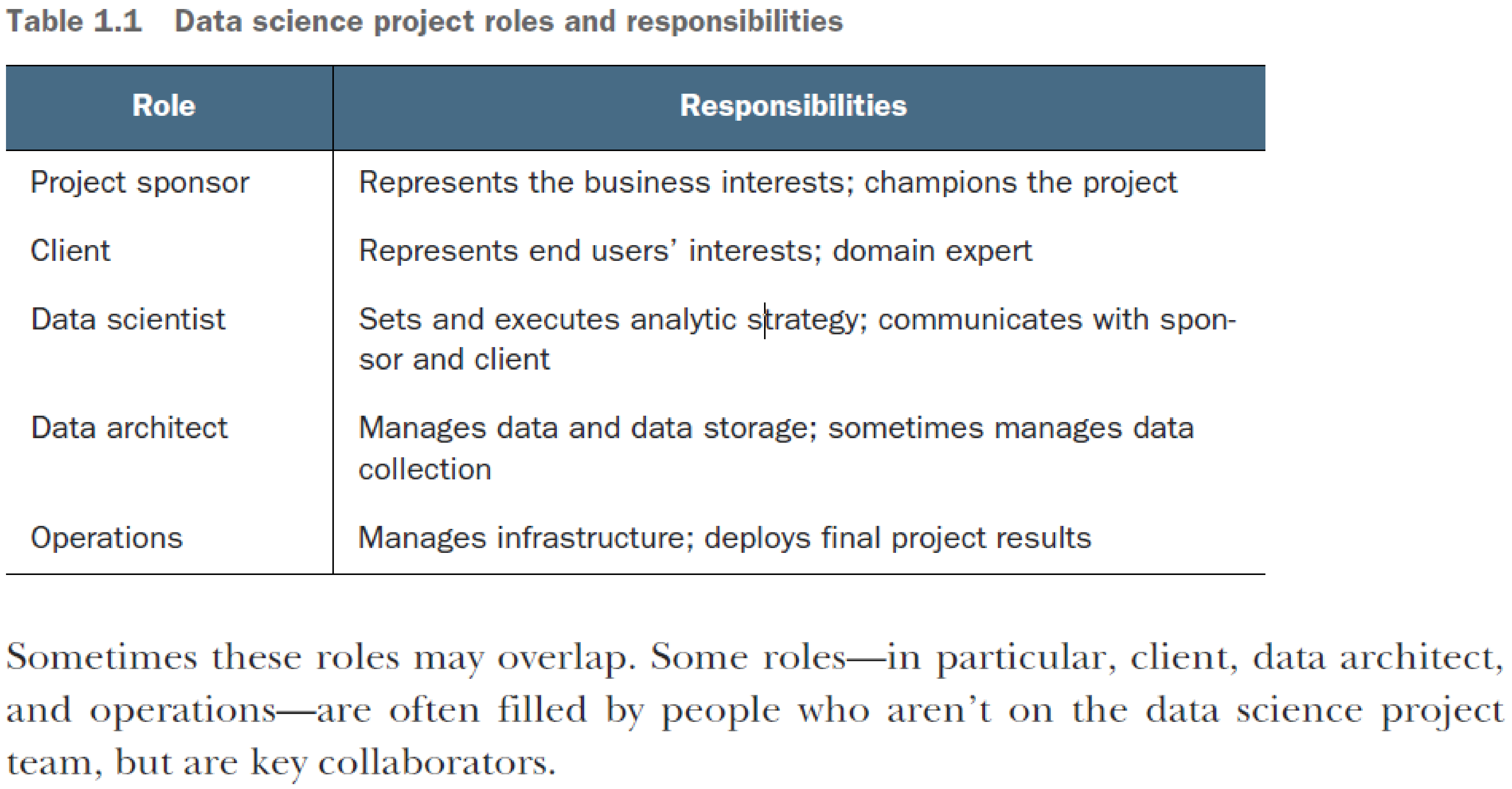


Table 1.1 Data science project roles and responsibilities

Role	Responsibilities
Project sponsor	Represents the business interests; champions the project
Client	Represents end users' interests; domain expert
Data scientist	Sets and executes analytic strategy; communicates with sponsor and client
Data architect	Manages data and data storage; sometimes manages data collection
Operations	Manages infrastructure; deploys final project results

Sometimes these roles may overlap. Some roles—in particular, client, data architect, and operations—are often filled by people who aren't on the data science project team, but are key collaborators.

Roles – Project Sponsor

Suppose you're working for a German bank. The bank feels that it's losing too much money to bad loans and wants to reduce its losses. To do so, they want a tool to help loan officers more accurately detect risky loans.

- *The most important role in a data science project is the project sponsor.* The sponsor is the person who wants the data science result; generally, they represent the business interests.
- The sponsor is responsible for deciding whether the project is a success or failure. The data scientist may fill the sponsor role for their own project if they feel they know and can represent the business needs, but that's not the optimal arrangement.
- The ideal sponsor meets the following condition: if they're satisfied with the project outcome, then the project is by definition a success.
- *Getting sponsor sign-off becomes the central organizing goal of a data science project.*
- **In the loan application example, the sponsor might be the bank's head of Consumer Lending.**

- **To ensure sponsor sign-off, you must get clear goals from them through directed interviews. You attempt to capture the sponsor's expressed goals as quantitative statements. An example goal might be "Identify 90% of accounts that will go into default at least two months before the first missed payment with a false positive rate of no more than 25%."**

Roles – Client

- While the sponsor is the role that represents the business interests, the client is the role that represents the model’s end users’ interests.
- Sometimes, the sponsor and client roles may be filled by the same person. Again, the data scientist may fill the client role if they can weight business trade-offs, but this isn’t ideal.
- The client is more hands-on than the sponsor; they’re the interface between the technical details of building a good model and the day-to-day work process into which the model will be deployed.
- They aren’t necessarily mathematically or statistically sophisticated, but are familiar with the relevant business processes and serve as the domain expert on the team.
- In the loan application example, the client may be a loan officer or someone who represents the interests of loan officers.
- As with the sponsor, you should keep the client informed and involved. Ideally, you’d like to have regular meetings with them to keep your efforts aligned with the needs of the end users.
- Generally, the client belongs to a different group in the organization and has other responsibilities beyond your project. Keep meetings focused, present results and progress in terms they can understand, and take their critiques to heart.
- If the end users can’t or won’t use your model, then the project isn’t a success, in the long run.

Data Scientist

- The next role in a data science project is the data scientist, who’s responsible for taking all necessary steps to make the project succeed, including setting the project strategy and keeping the client informed.
- They design the project steps, pick the data sources, and pick the tools to be used. Since they pick the techniques that will be tried, they have to be well informed about statistics and machine learning.
- They’re also responsible for project planning and tracking, though they may do this with a project management partner.
- At a more technical level, the data scientist also looks at the data, performs statistical tests and procedures, applies machine learning models, and evaluates results—the science portion of data science.

Data Architect

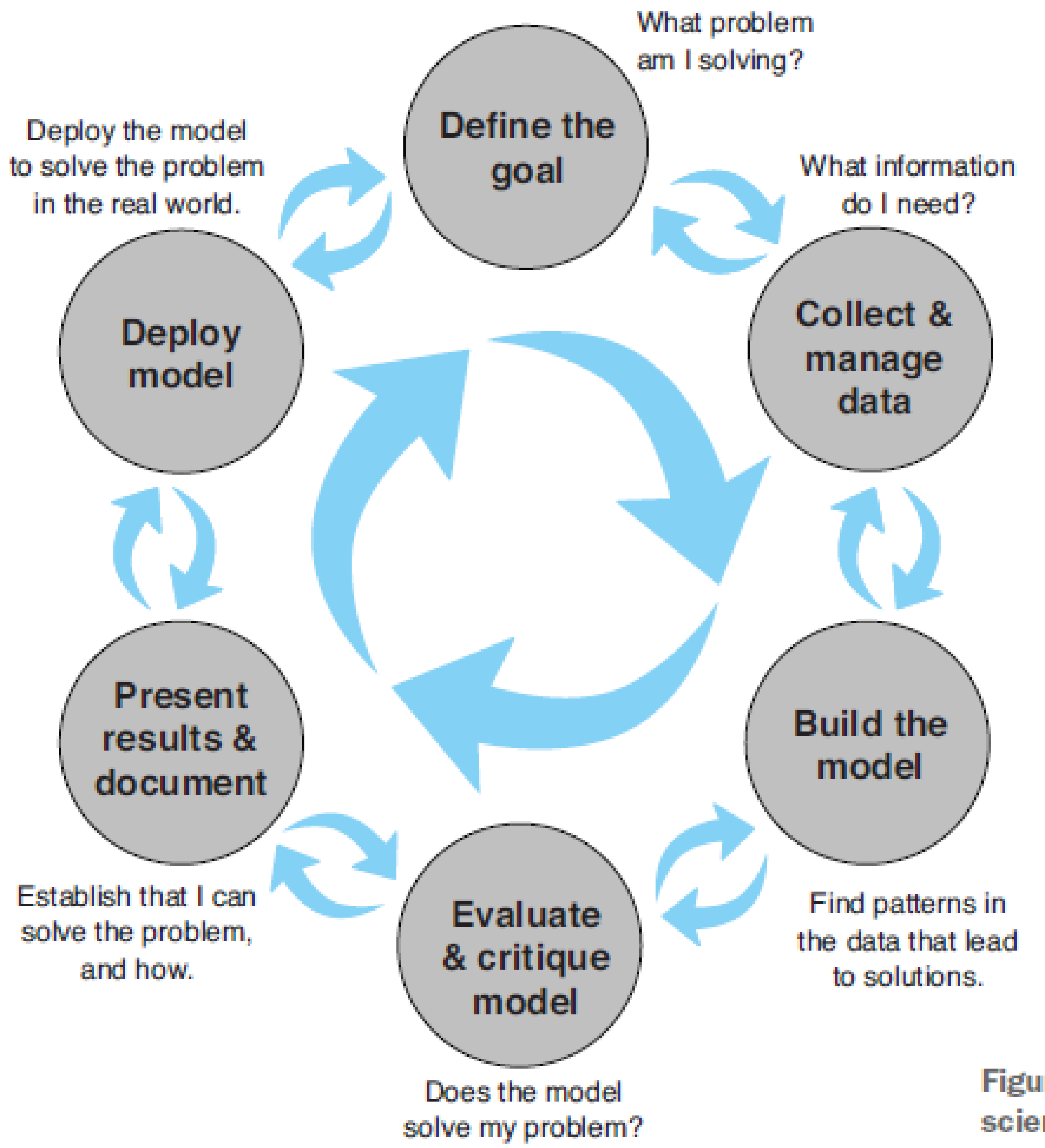
- The data architect is responsible for all the data and its storage.
- Often this role is filled by someone outside of the data science group, such as a database administrator or architect.
- Data architects often manage data warehouses for many different projects, and they may only be available for quick consultation.

Operations

- The operations role is critical both in acquiring data and delivering the final results.
- The person filling this role usually has operational responsibilities outside of the data science group.
- For example, if you’re deploying a data science result that affects how products are sorted on an online shopping site, then the person responsible for running the site will have a lot to say about how such a thing can be deployed.
- This person will likely have constraints on response time, programming language, or data size that you need to respect in deployment.
- The person in the operations role may already be supporting your sponsor or your client, so they’re often easy to find (though their time may be already very much in demand).

Stages of a Data Science Project

- The ideal data science environment is one that encourages feedback and iteration between the data scientist and all other stakeholders.
- This is reflected in the lifecycle of a data science project.


What problem am I solving?
Define the goal
What information do I need?
Collect & manage data
Build the model
Find patterns in the data that lead to solutions.
Evaluate & critique model
Does the model solve my problem?
Present results & document
Establish that I can solve the problem, and how.
Deploy model
Deploy the model to solve the problem in the real world.
Figu
scie

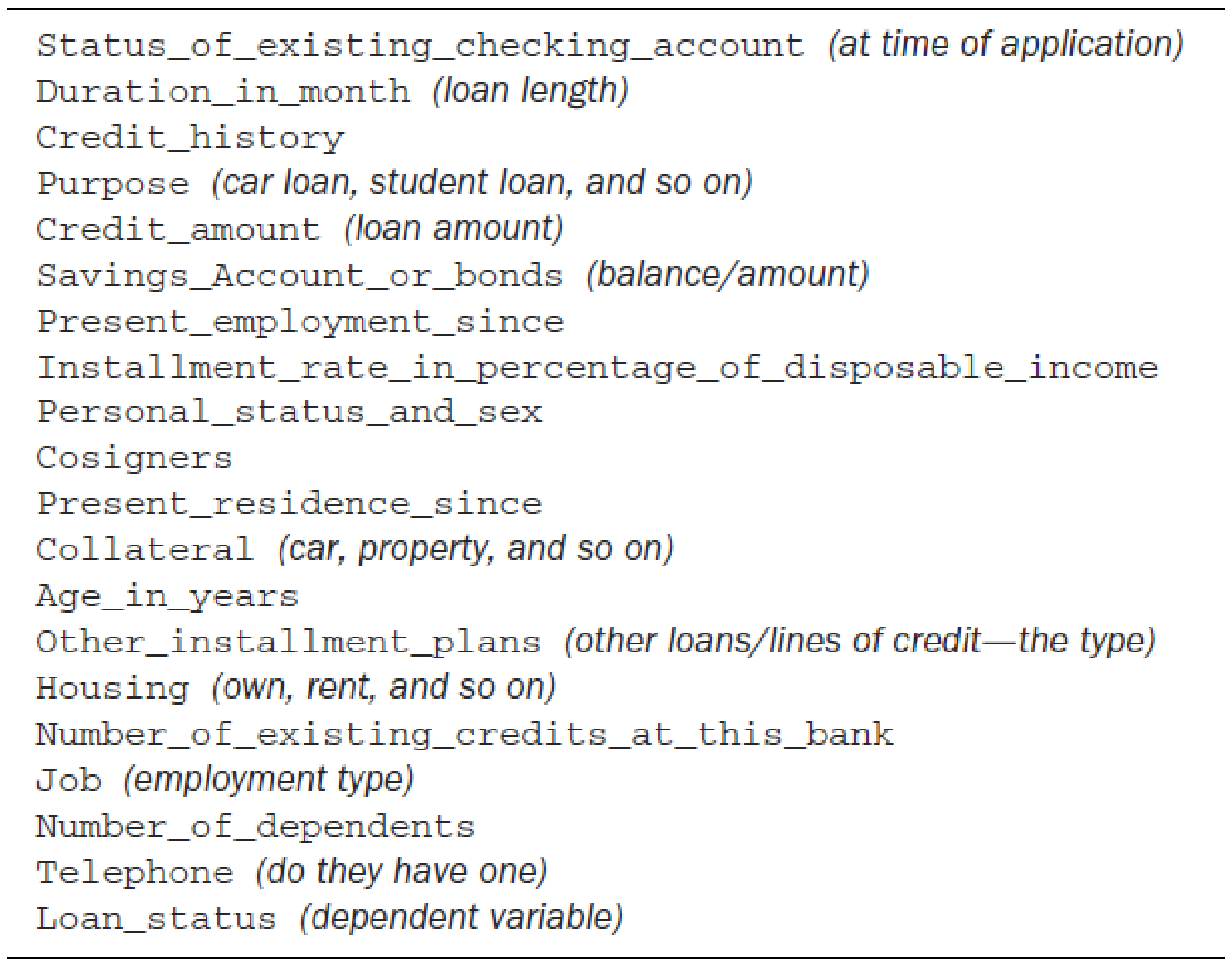
Defining the Goal

- The first task in a data science project is to define a measurable and quantifiable goal. At this stage, learn all that you can about the context of your project:
  - Why do the sponsors want the project in the first place? What do they lack, and what do they need?
  - What are they doing to solve the problem now, and why isn’t that good enough?
  - What resources will you need: what kind of data and how much staff? Will you have domain experts to collaborate with, and what are the computational resources?
  - How do the project sponsors plan to deploy your results? What are the constraints that have to be met for successful deployment?
- Once you and the project sponsor and other stakeholders have established preliminary answers to these questions, you and they can start defining the precise goal of the project.
- The goal should be specific and measurable; not “We want to get better at finding bad loans,” but instead “We want to reduce our rate of loan charge-offs by at least 10%, using a model that predicts which loan applicants are likely to default.”

Data Collection and Management

- This step encompasses identifying the data you need, exploring it, and conditioning it to be suitable for analysis. This stage is often the most time-consuming step in the process. It’s also one of the most important:
    - What data is available to me?
    - Will it help me solve the problem?
    - Is it enough?
    - Is the data quality good enough?

Table 1.2 Loan data attributes



`Status_of_existing_checking_account` *(at time of application)*
`Duration_in_month` *(loan length)*
`Credit_history`
`Purpose` *(car loan, student loan, and so on)*
`Credit_amount` *(loan amount)*
`Savings_Account_or_bonds` *(balance/amount)*
`Present_employment_since`
`Installment_rate_in_percentage_of_disposable_income`
`Personal_status_and_sex`
`Cosigners`
`Present_residence_since`
`Collateral` *(car, property, and so on)*
`Age_in_years`
`Other_installment_plans` *(other loans/lines of credit—the type)*
`Housing` *(own, rent, and so on)*
`Number_of_existing_credits_at_this_bank`
`Job` *(employment type)*
`Number_of_dependents`
`Telephone` *(do they have one)*
`Loan_status` *(dependent variable)*

Modelling

- The most common data science modeling tasks are these:
- *Classifying*—Deciding if something belongs to one category or another
- *Scoring*—Predicting or estimating a numeric value, such as a price or probability
- *Ranking*—Learning to order items by preferences
- *Clustering*—Grouping items into most-similar groups
- *Finding relations*—Finding correlations or potential causes of effects seen in the data
- *Characterizing*—Very general plotting and report generation from data

The loan application problem is a classification problem: you want to identify loan applicants who are likely to default.

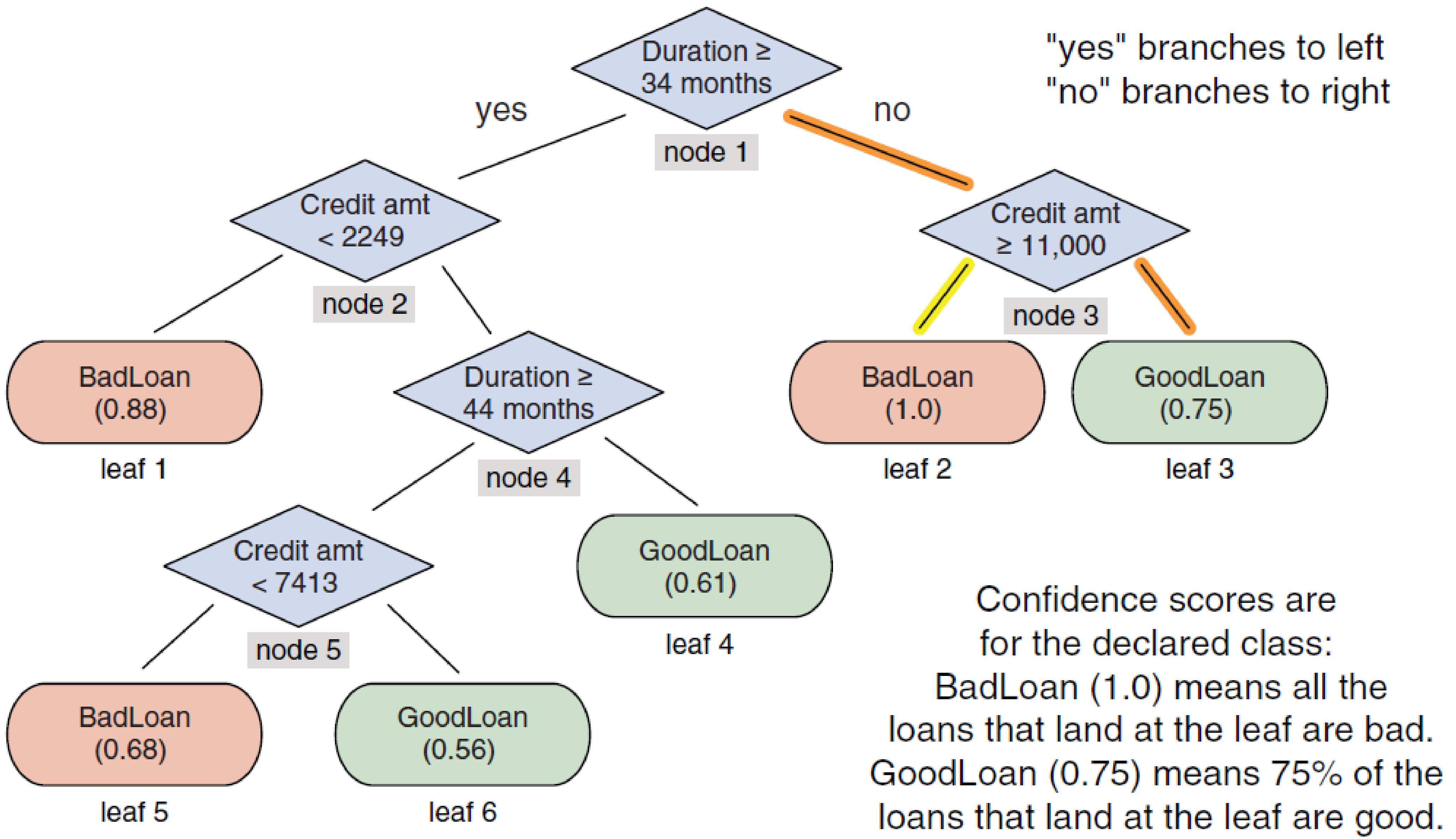


Figure 1.3 A decision tree model for finding bad loan applications. The outcome nodes show confidence scores.

Model Evaluation and Critique

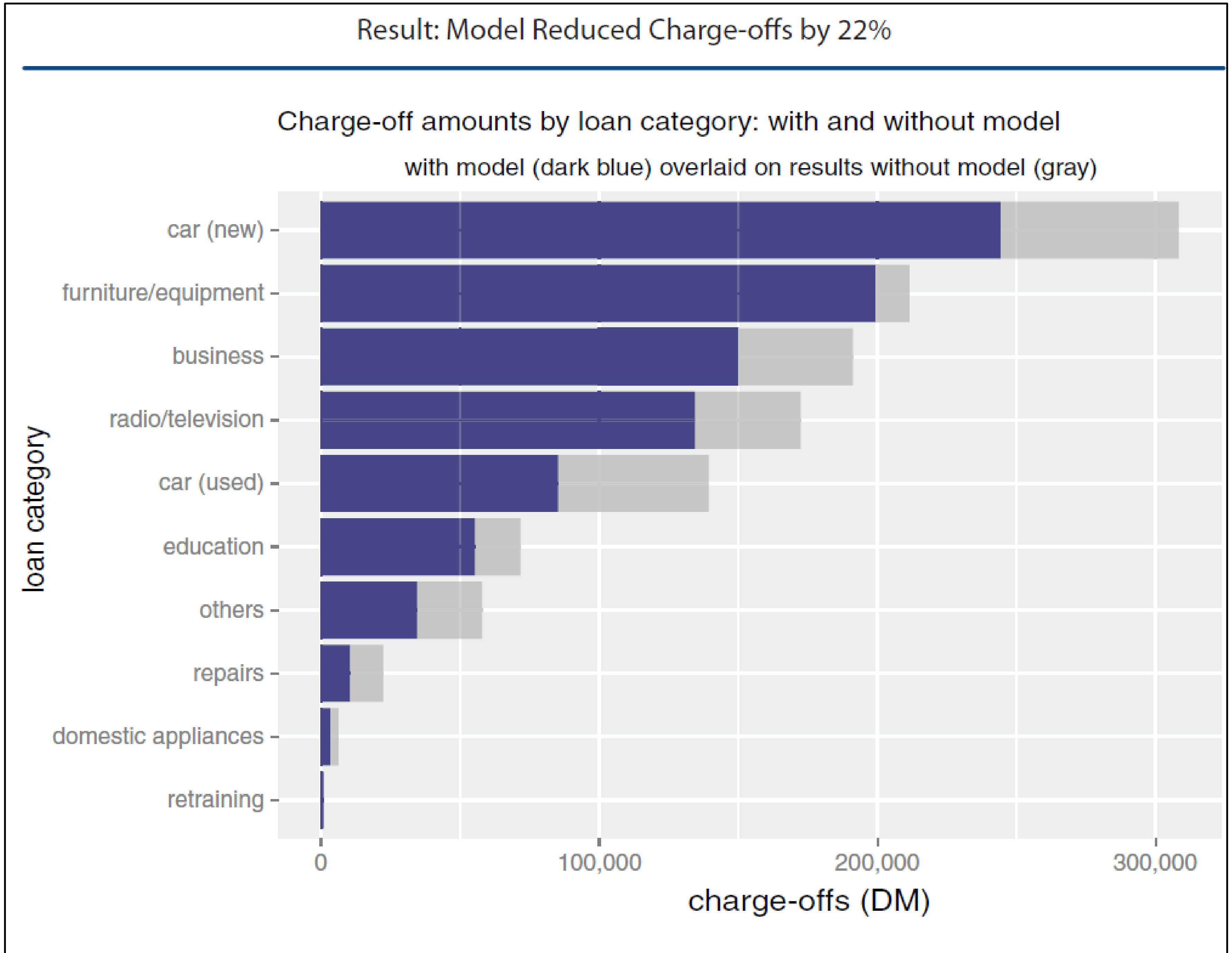
- Once you have a model, you need to determine if it meets your goals:
- Is it accurate enough for your needs? Does it generalize well?
- Does it perform better than “the obvious guess”? Better than whatever estimate you currently use?
- Do the results of the model make sense in the context of the problem domain?

- Confusion Matrix

Presentation and Documentation

A presentation for the model’s end users (the loan officers) would instead emphasize how the model will help them do their job better:

- How should they interpret the model?
- What does the model output look like?
- If the model provides a trace of which rules in the decision tree executed, how do they read that?
- If the model provides a confidence score in addition to a classification, how should they use the confidence score?
- When might they potentially overrule the model?



Result: Model Reduced Charge-offs by 22%

Figure 1.4 Example slide from an executive presentation

Model Deployment and Maintenance

- Finally, the model is put into operation.
- In many organizations, this means the data scientist no longer has primary responsibility for the day-to-day operation of the model.
- But you still should ensure that the model will run smoothly and won’t make disastrous unsupervised decisions.
- You also want to make sure that the model can be updated as its environment changes.
- And in many situations, the model will initially be deployed in a small pilot program.
- The test might bring out issues that you didn’t anticipate, and you may have to adjust the model accordingly.

Setting Expectations

- Setting expectations is a crucial part of defining the project goals and success criteria.
- The business-facing members of your team (in particular, the project sponsor) probably already have an idea of the performance required to meet business goals:
- For example, the bank wants to reduce their losses from bad loans by at least 10%. Before you get too deep into a project, you should make sure that the resources you have are enough for you to meet the business goals.
- You get to know the data better during the exploration and cleaning phase;
- after you have a sense of the data, you can get a sense of whether the data is good enough to meet desired performance thresholds.
- If it’s not, then you’ll have to revisit the project design and goalsetting stage.

Determining lower bounds on model performance

- Understanding how well a model *should* do for acceptable performance is important when defining acceptance criteria.
- The *null model* represents the lower bound on model performance that you should strive for.
- You can think of the null model as being “the obvious guess” that your model must do better than.
- In situations where there’s no existing model or solution, the null model is the simplest possible model:
- A model that labels all loans as `GoodLoan` (in effect, using only the existing process to classify loans) would be correct 70% of the time.
- So you know that any actual model that you fit to the data should be better than 70% accurate to be useful—if accuracy were your only metric.
- Since this is the simplest possible model, its error rate is called the *base error rate*.

Determining lower bounds on model performance

- How much better than 70% should you be? In statistics there's a procedure called *hypothesis testing*, or *significance testing*, that tests whether your model is equivalent to a null model (in this case, whether a new model is basically only as accurate as guessing `GoodLoan` all the time).
- You want your model accuracy to be "significantly better"—in statistical terms—than 70%.
- if there is an existing model or process in place, you'd like to have an idea of its precision, recall, and false positive rates;
- improving one of these metrics is almost always more important than considering accuracy alone. If the purpose of your project is to improve the existing process, then the current model must be unsatisfactory for at least one of these metrics.
- Knowing the limitations of the existing process helps you determine useful lower bounds on desired performance.

Starting with R and data

Few resources

- R in Action, Second Edition, Robert Kabacoff, Manning, 2015
- Beyond Spreadsheets with R, Jonathan Carroll, Manning, 2018
- The Art of R Programming, Norman Matloff, No Starch Press, 2011
- R for Everyone, Second Edition, Jared P. Lander, Addison-Wesley, 2017

R coding style guides

- The Google R Style Guide (https://google.github.io/styleguide/Rguide.html)
- Hadley Wickham's style guide from Advanced R (http://adv-r.had.co.nz/Style.html)

EXAMPLES AND THE COMMENT CHARACTER (#)

- R commands as free text, with the results prefixed by the hash mark, `#`, which is R's comment character.
- `print(seq_len(25))`
- `# [1] 1 2 3 4 5 6 7 8 9 10 11 12`
- `# [13] 13 14 15 16 17 18 19 20 21 22 23 24`
- `# [25] 25`
- Notice the numbers were wrapped to three lines, and each line starts with the index of the first cell reported on the line inside square brackets

VECTORS AND LISTS

- Vectors (sequential arrays of values) are fundamental R data structures.
- Lists can hold different types in each slot;
- vectors can only hold the same primitive or atomic type in each slot.
- In addition to numeric indexing, both vectors and lists support name-keys.
- Retrieving items from a list or vector can be done by the operators shown next.
- VECTOR INDEXING R vectors and lists are indexed from 1, and not from 0 as with many other programming languages.

Builds an example vector. c() is R's concatenate operator—it builds longer vectors and lists from shorter ones without nesting. For example, c(1) is just the number 1, and c(1, c(2, 3)) is equivalent to c(1, 2, 3), which in turn is the integers 1 through 3 (though stored in a floating-point format).

```
example_vector <- c(10, 20, 30)
example_list <- list(a = 10, b = 20, c = 30)
```

Builds an example list

```
example_vector[1]
 ## [1] 10
example_list[1]
## $a
## [1] 10
```

Demonstrates vector and list use of [ ]. Notice that for the list, [ ] returns a new short list, not the item.

```
example_vector[[2]]
 ## [1] 20
example_list[[2]]
## [1] 20
```

Demonstrates vector and list use of [[ ]]. In common cases, [[ ]] forces a single item to be returned, though for nested lists of complex type, this item itself could be a list.

```
example_vector[c(FALSE, TRUE, TRUE)]
 ## [1] 20 30
example_list[c(FALSE, TRUE, TRUE)]
## $b
## [1] 20
##
## $c
## [1] 30
```

Vectors and lists can be indexed by vectors of logicals, integers, and (if the vector or list has names) characters.

```
example_list$b
 ## [1] 20

example_list[["b"]]
## [1] 20
```

For named examples, the syntax example_list$b is essentially a short-hand for example_list[["b"]] (the same is true for named vectors).

```
x <- 1:5
print(x)
# [1] 1 2 3 4 5
```

Defines a value we are interested in and stores it in the variable x

```
x <- cumsumMISSPELLED(x)
# Error in cumsumMISSPELLED(x) : could not find function "cumsumMISSPELLED"
```

Attempts, and fails, to assign a new result to x

```
print(x)
# [1] 1 2 3 4 5
```

Notice that in addition to supplying a useful error message, R preserves the original value of x.

```
x <- cumsum(x)
print(x)
# [1]  1  3  6 10 15
```

Tries the operation again, using the correct spelling of cumsum(). cumsum(), short for cumulative sum, is a useful function that computes running totals quickly.

Vectorised

- Another aspect of vectors in R is that most R operations are *vectorized*.
- A function or operator is called vectorized when applying it to a vector is shorthand for applying a function to each entry of the vector independently.
- For example, the function `nchar()` counts how many characters are in a string. In R this function can be used on a single string, or on a vector of strings

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

LOGICAL OPERATIONS

- R’s logical operators come in two flavors.
- R has standard infix scalar-valued operators that expect only one value and have the same behavior and same names as you would see in C or Java: `&&` and `||`.
- R also has vectorized infix operators that work on vectors of logical values: `&` and `|`.
- Be sure to always use the scalar versions (`&&` and `||`) in situations such as `if` statements, and the vectorized versions (`&` and `|`) when processing logical vectors.

NULL AND NANA (NOT AVAILABLE) VALUES

- In R `NULL` is just a synonym for the empty or length-zero vector formed by using the concatenate operator `c()` with no arguments.
- `NULL` is simply a length-zero vector.
- For example, `c(c(), 1, NULL)` is perfectly valid and returns the value `1`.
- For example, the vector `c("a", NA, "c")` is a vector of three character strings where we do not know the value of the second entry.
- Having `NA` is a great convenience as it allows us to annotate missing or unavailable values in place, which can be critical in data processing.
- Also, `NA` means “not available,” not invalid (as `NaN` denotes), so `NA` has some convenient rules such as the logical expression `FALSE & NA` simplifying to `FALSE`.

IDENTIFIERS

- Identifiers or symbol names are how R refers to variables and functions.
- The Google R Style Guide insists on writing symbol names in what is called “CamelCase” (word boundaries in names are denoted by uppercase letters, as in “CamelCase” itself).
- The Advanced R guide recommends an underscore style where names inside identifiers are broken up with underscores (such as “day_one” instead of “DayOne”).
- Also, many R users use a dot to break up names with identifiers (such as “day.one”).
- In particular, important built-in R types such as `data.frame` and packages such as `data.table` use the dot notation convention.

LINE BREAKS

- It is generally recommended to keep R source code lines at 80 columns or fewer.
- R accepts multiple-line statements as long as where the statement ends is unambiguous.
- For example, to break the single statement “`1 + 2`" into multiple lines, write the code as follows:
- `1 +`
- `  2`
- Do not write code like the following, as the first line is itself a valid statement, creating ambiguity:
- `1`
- `+ 2`

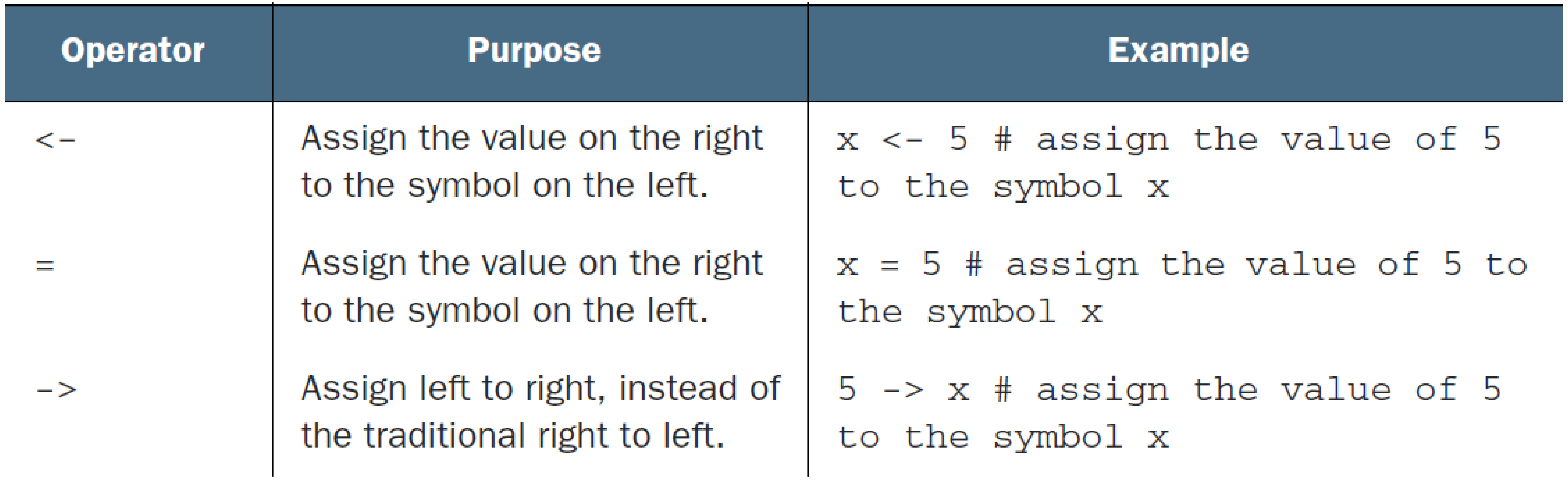
SEMICOLONS

- R allows semicolons as end-of-statement markers, but does not require them.
- Most style guides recommend not using semicolons in R code and certainly not using them at ends of lines.

ASSIGNMENT

- R has many assignment operators; the preferred one is <−.
- = can be used for assignment in R, but is also used to bind argument values to argument names by name during function calls (so there is some potential ambiguity in using =).

Table 2.1 Primary R assignment operators



Operator	Purpose	Example
`<-`	Assign the value on the right to the symbol on the left.	`x <- 5 # assign the value of 5 to the symbol x`
`=`	Assign the value on the right to the symbol on the left.	`x = 5 # assign the value of 5 to the symbol x`
`->`	Assign left to right, instead of the traditional right to left.	`5 -> x # assign the value of 5 to the symbol x`

LEFT-HAND SIDES OF ASSIGNMENTS

- R allows slice expressions on the left-hand sides of assignments, and both numeric and logical array indexing.
- This allows for very powerful array-slicing commands and coding styles.
- For example, we can replace all the missing values (denoted by “`NA`") in a vector with zero as shown in the following example:

```
d <- data.frame(x = c(1, NA, 3))
print(d)
#    x
# 1  1
# 2 NA
# 3  3
```

“data.frame ” is R’s tabular data type, and the most important data type in R. A data.frame holds data organized in rows and columns.

When printing data.frames, row numbers are shown in the first (unnamed) column, and column values are shown under their matching column names.

```
d$x[is.na(d$x)] <- 0
print(d)
#   x
# 1 1
# 2 0
# 3 3
```

We can place a slice or selection of the x column of d on the left-hand side of an assignment to easily replace all NA values with zero.

Factors

- R can handle many kinds of data: numeric, logical, integer, strings (called *character* types), and *factors*.
- Factors are an R type that encodes a fixed set of strings as integers.
- Factors can save a lot on storage while appearing to behave as strings
- Factors also encode the entire set of allowed values, which is useful—but can make combining data from different sources (that saw different sets of values) a bit of a chore.
- To avoid issues, we suggest delaying conversion of strings to factors until late in an analysis.
- This is usually accomplished by adding the argument `stringsAsFactors = FALSE` to functions such as `data.frame()` or `read.table()`.
- We, however, do encourage using factors when you have a reason, such as wanting to use `summary()` or preparing to produce dummy indicators

NAMED ARGUMENTS

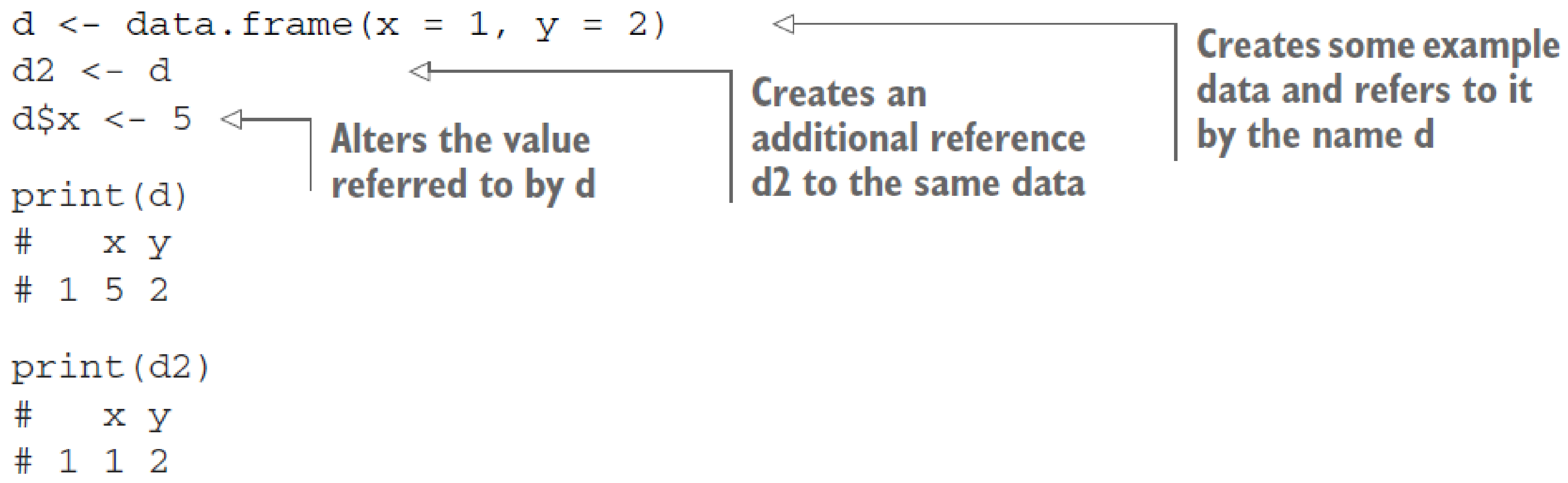
- R is centered around applying functions to data.
- Functions that take a large number of arguments rapidly become confusing and illegible.
- This is why R includes a named argument feature.
- As an example, if we wanted to set our working directory to “/tmp” we would usually use the `setwd()` command like so: `setwd("/tmp")`.
- However, `help(setwd)` shows us the first argument to `setwd()` has the name `dir`, so we could also write this as `setwd(dir = "/tmp")`.
- This becomes useful for functions that have a large number of arguments, and for setting optional function arguments.
- Note: named arguments must be set by `=`, and not by an assignment operator such as `<-`.

PACKAGE NOTATION

- In R there are two primary ways to use a function from a package.
- The first is to attach the package with the `library()` command and then use the function name.
- The second is to use the package name and then `::` to name the function.
- An example of this second method is `stats::sd(1:5)`.
- The `::` notation is good to avoid ambiguity or to leave a reminder of which package the function came from for when you read your own code later.

VALUE SEMANTICS

- R is unusual in that it efficiently simulates "copy by value" semantics.
- Any time a user has two references to data, each evolves independently: changes to one do not affect the other.
- This is very desirable for part-time programmers and eliminates a large class of possible aliasing bugs when writing code.
- Notice `d2` keeps the old value of `1` for `x`. This feature allows for very convenient and safe coding.
- Many programming languages protect references or pointers in function calls in this manner; however, R protects complex values and does so in all situations (not just function calls).
- Some care has to be taken when you want to share back changes, such as invoking a final assignment such as `d2 <- d` after all desired changes have been made.



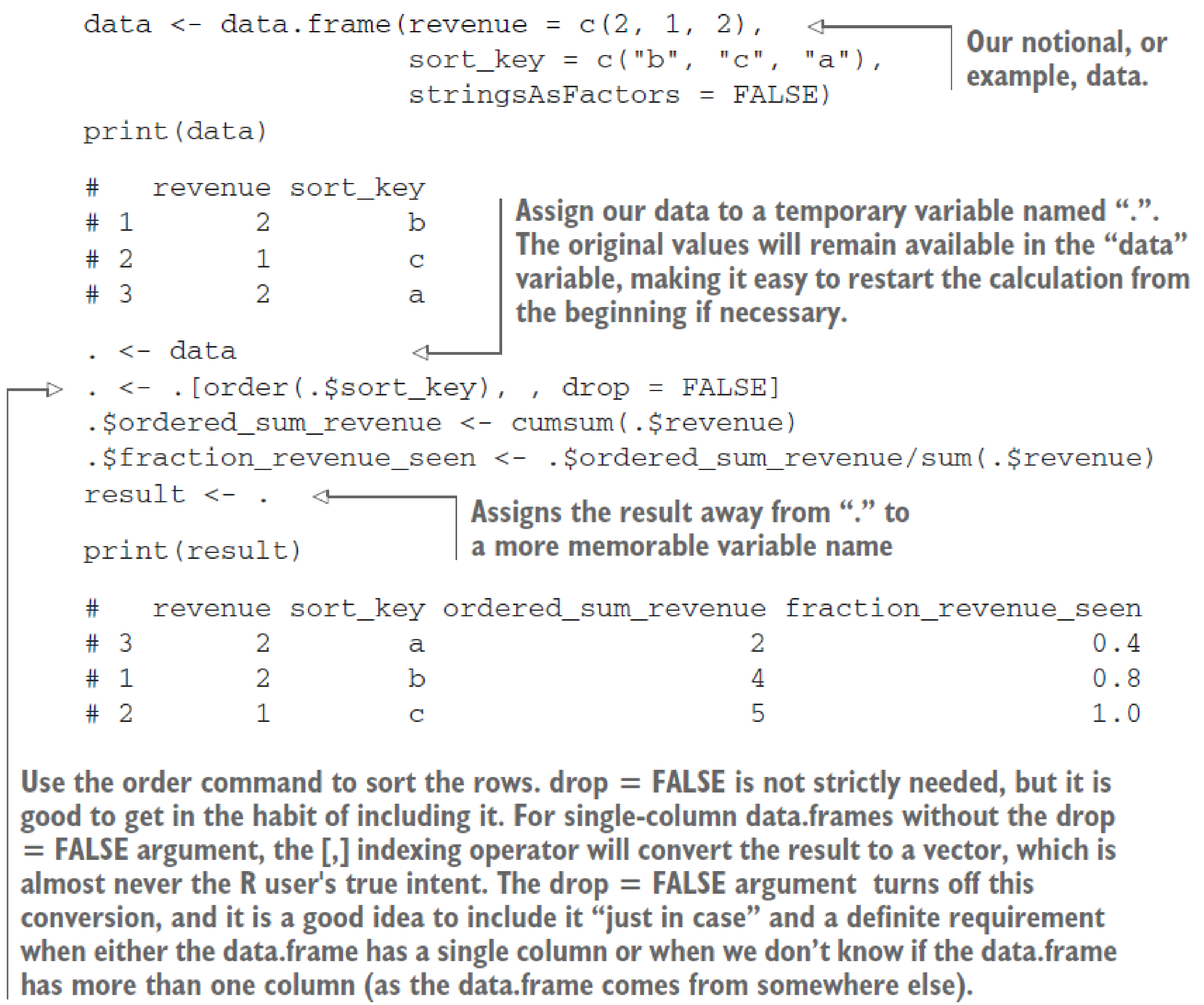
```
d <- data.frame(x = 1, y = 2)
d2 <- d
d$x <- 5

print(d)
#   x y
# 1 5 2

print(d2)
#   x y
# 1 1 2
```

Creates some example data and refers to it by the name d

Creates an additional reference d2 to the same data

Alters the value referred to by d

ORGANIZING INTERMEDIATE VALUES



```
data <- data.frame(revenue = c(2, 1, 2),
                   sort_key = c("b", "c", "a"),
                   stringsAsFactors = FALSE)
print(data)

#   revenue sort_key
# 1       2        b
# 2       1        c
# 3       2        a

. <- data
. <- .[order(.$sort_key), , drop = FALSE]
.$ordered_sum_revenue <- cumsum(.$revenue)
.$fraction_revenue_seen <- .$ordered_sum_revenue/sum(.$revenue)
result <- .

print(result)

#   revenue sort_key ordered_sum_revenue fraction_revenue_seen
# 3       2        a                   2                   0.4
# 1       2        b                   4                   0.8
# 2       1        c                   5                   1.0
```

Our notional, or example, data.

Assign our data to a temporary variable named “.”. The original values will remain available in the “data” variable, making it easy to restart the calculation from the beginning if necessary.

Assigns the result away from “.” to a more memorable variable name

Use the order command to sort the rows. drop = FALSE is not strictly needed, but it is good to get in the habit of including it. For single-column data.frames without the drop = FALSE argument, the [,] indexing operator will convert the result to a vector, which is almost never the R user's true intent. The drop = FALSE argument turns off this conversion, and it is a good idea to include it “just in case” and a definite requirement when either the data.frame has a single column or when we don’t know if the data.frame has more than one column (as the data.frame comes from somewhere else).

- Long sequences of calculations can become difficult to read, debug, and maintain.
- To avoid this, we suggest reserving the variable named “ . ” to store intermediate values.

Piped Notation

- The R package `dplyr` replaces the dot notation with what is called *piped notation*
- `library("dplyr")`

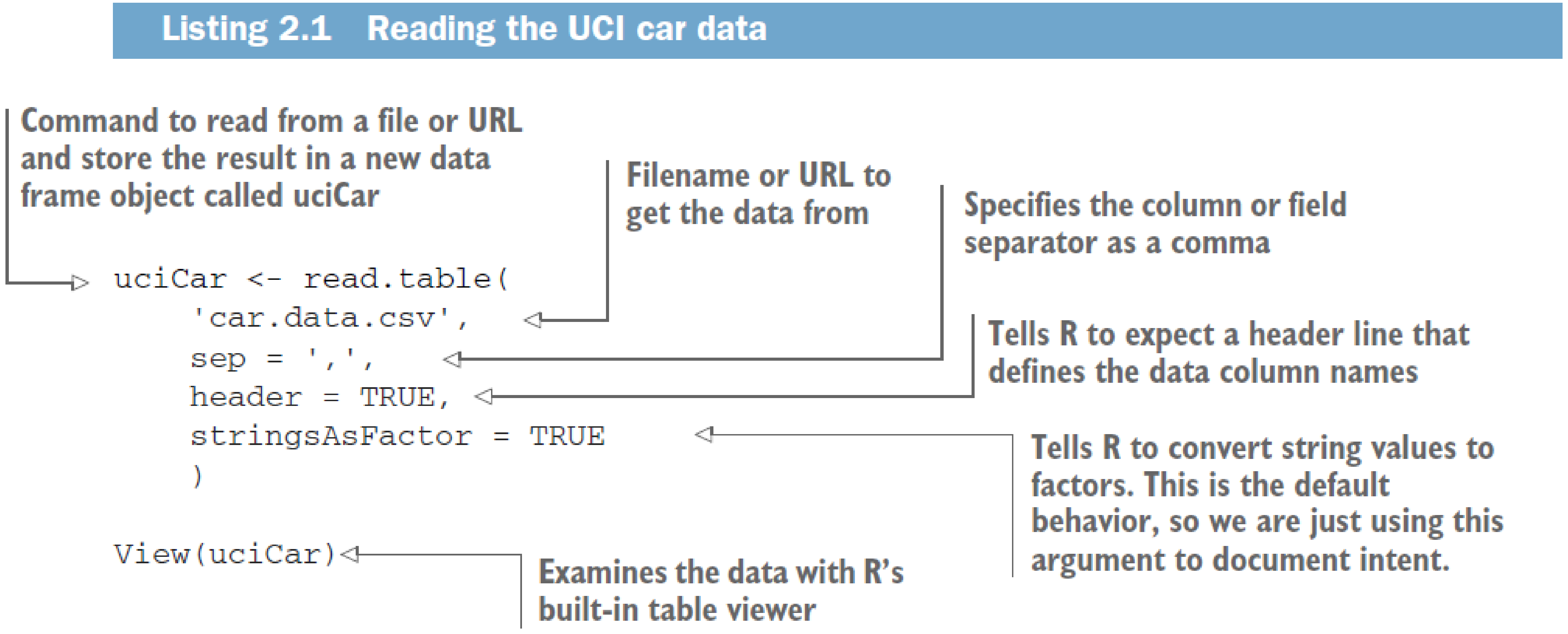
```
      result <- data %>%
      arrange(., sort_key) %>%
      mutate(., ordered_sum_revenue = cumsum(revenue)) %>%
      mutate(., fraction_revenue_seen = ordered_sum_revenue/sum(revenue))
```

- Each step of this example has been replaced by the corresponding `dplyr` equivalent.
- `arrange()` is `dplyr`'s replacement for `order()`, and `mutate()` is `dplyr`'s replacement for assignment.
- The code translation is line by line, with the minor exception that assignment is written first (even though it happens after all other steps).
- The calculation steps are sequenced by the `magrittr` pipe symbol `%>%`.
- The `magrittr` pipe allows you to write any of `x %>% f`, `x %>%`

data.frame

- The R `data.frame` class is designed to store data in a very good "ready for analysis"
- format. `data.frame`s are two-dimensional arrays where each column represents a variable, measurement, or fact, and each row represents an individual or instance.
- In this format, an individual cell represents what is known about a single fact or variable for a single instance.
- `data.frame`s are implemented as a named list of column vectors (list columns are possible, but they are more of the exception than the rule for `data.frame`s).
- In a `data.frame`, all columns have the same length, and this means we can think of the kth entry of all columns as forming a row.
- Operations on `data.frame` columns tend to be efficient and vectorized.
- Adding, looking up, and removing columns is fast. Operations per row on a `data.frame` can be expensive, so you should prefer vectorized column notations for large `data.frame` processing.
- R's `data.frame` is much like a database table in that it has schema-like information: an explicit list of column names and column types.
- Most analyses are best expressed in terms of transformations over `data.frame` columns.
- `d <- data.frame(col1 = c(1, 2, 3), col2 = c(-1, 0, 1))`
- `d$col3 <- d$col1 + d$col2`
- `print(d)`

Working with files



Listing 2.1 Reading the UCI car data

```
uciCar <- read.table(
     'car.data.csv',
     sep = ',',
     header = TRUE,
     stringsAsFactor = TRUE
     )

View(uciCar)
```

Command to read from a file or URL and store the result in a new data frame object called uciCar

Filename or URL to get the data from

Specifies the column or field separator as a comma

Tells R to expect a header line that defines the data column names

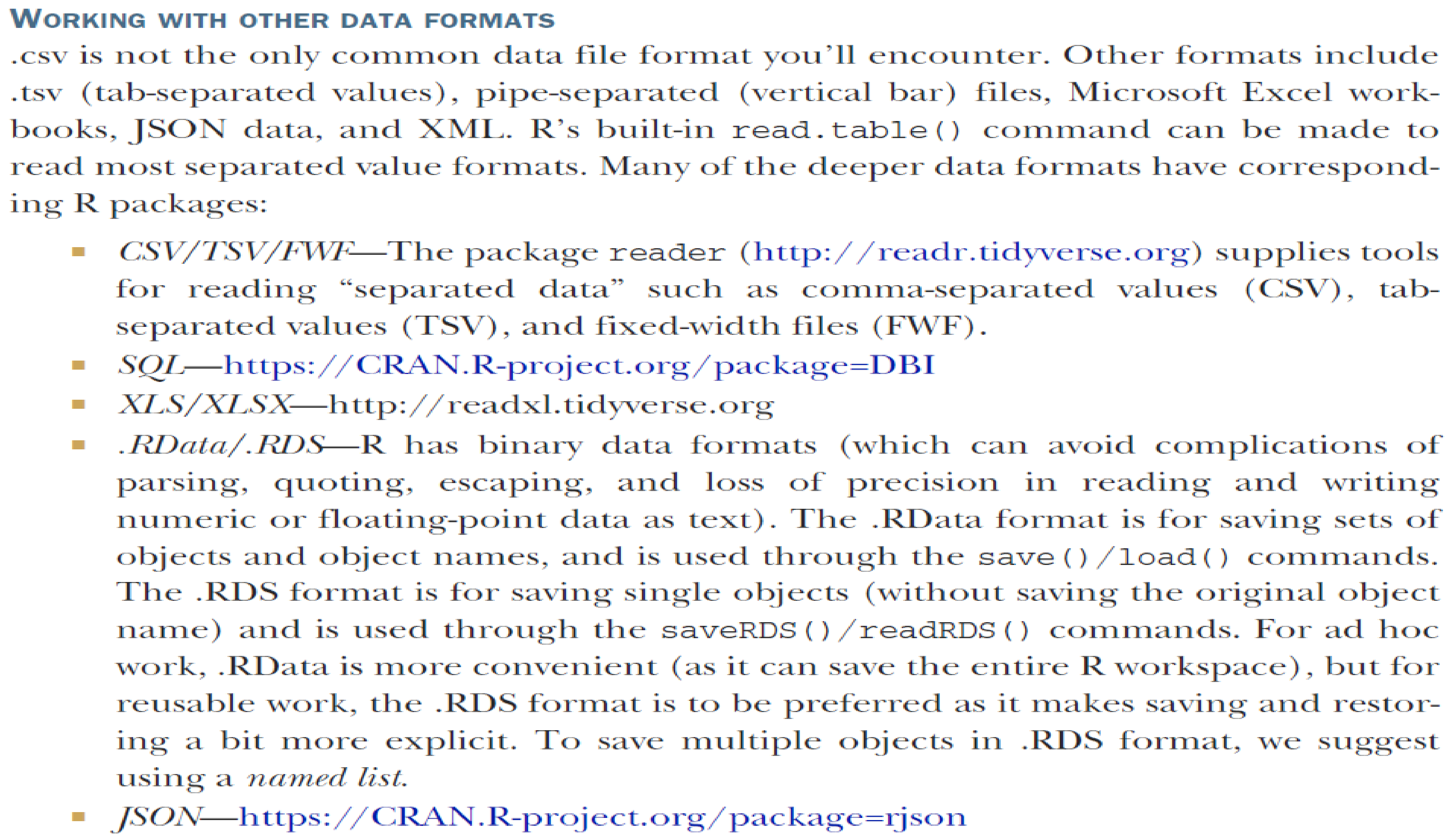
Tells R to convert string values to factors. This is the default behavior, so we are just using this argument to document intent.

Examines the data with R's built-in table viewer

Examining our data

Once we've loaded the data into R, we'll want to examine it. These are the commands to always try first:

- `class()`—Tells you what kind of R object you have. In our case, `class(uciCar)` tells us the object `uciCar` is of class `data.frame`. Class is an object-oriented concept, which describes how an object is going to behave. R also has a (less useful) `typeof()` command, which reveals how the object's storage is implemented.
- `dim()`—For data frames, this command shows how many rows and columns are in the data.
- `head()`—Shows the top few rows (or "head") of the data. Example: `head(uciCar)`.
- `help()`—Provides the documentation for a class. In particular, try `help(class(uciCar))`.
- `str()`—Gives you the structure for an object. Try `str(uciCar)`.
- `summary()`—Provides a summary of almost any R object. `summary(uciCar)` shows us a lot about the distribution of the UCI car data.
- `print()`—Prints all the data. Note: for large datasets, this can take a very long time and is something you want to avoid.
- `View()`—Displays the data in a simple spreadsheet-like grid viewer.



Working with other data formats

.csv is not the only common data file format you'll encounter. Other formats include .tsv (tab-separated values), pipe-separated (vertical bar) files, Microsoft Excel workbooks, JSON data, and XML. R's built-in `read.table()` command can be made to read most separated value formats. Many of the deeper data formats have corresponding R packages:

- *CSV/TSV/FWF*—The package `reader` (http://readr.tidyverse.org) supplies tools for reading "separated data" such as comma-separated values (CSV), tab-separated values (TSV), and fixed-width files (FWF).
- *SQL*—https://CRAN.R-project.org/package=DBI
- *XLS/XLSX*—http://readxl.tidyverse.org
- *.RData/.RDS*—R has binary data formats (which can avoid complications of parsing, quoting, escaping, and loss of precision in reading and writing numeric or floating-point data as text). The .RData format is for saving sets of objects and object names, and is used through the `save()`/`load()` commands. The .RDS format is for saving single objects (without saving the original object name) and is used through the `saveRDS()`/`readRDS()` commands. For ad hoc work, .RData is more convenient (as it can save the entire R workspace), but for reusable work, the .RDS format is to be preferred as it makes saving and restoring a bit more explicit. To save multiple objects in .RDS format, we suggest using a *named list.*
- *JSON*—https://CRAN.R-project.org/package=rjson

XML—https://CRAN.R-project.org/package=XML
MongoDB—https://CRAN.R-project.org/package=mongolite

Working with relational databases

- Relational databases scale easily to hundreds of millions of records and supply important production features such as parallelism, consistency, transactions, logging, and audits.
- Relational databases are designed to support online transaction processing (OLTP)
- Loading the data
- Querying
- Summarizing
- Etc
- #PUMS American Community Survey data

Listing 2.7 PUMS data provenance documentation (PDSwR2/PUMS/download/LoadPUMS.Rmd)

When we downloaded the data

Where we found the data documentation. This is important to record, as many data files don't contain links back to the documentation.

```
Data downloaded 4/21/2018 from:Reduce Zoom

  https://www.census.gov/data/developers/data-sets/acs-1year.2016.html
   https://www.census.gov/programs-surveys/acs/
➥technical-documentation/pums.html
  http://www2.census.gov/programs-
     surveys/acs/tech_docs/pums/data_dict/PUMSDataDict16.txt
  https://www2.census.gov/programs-surveys/acs/data/pums/2016/1-Year/

First in a `bash` shell perform the following steps:

wget https://www2.census.gov/programs-surveys/acs/data/
➥pums/2016/1-Year/csv_hus.zip
 md5 csv_hus.zip
# MD5 (csv_hus.zip) = c81d4b96a95d573c1b10fc7f230d5f7a
 wget https://www2.census.gov/programs-surveys/acs/data/pums/2016/1-
     Year/csv_pus.zip
md5 csv_pus.zip
# MD5 (csv_pus.zip) = 06142320c3865620b0630d74d74181db
wget http://www2.census.gov/programs-
     surveys/acs/tech_docs/pums/data_dict/PUMSDataDict16.txt
md5 PUMSDataDict16.txt
# MD5 (PUMSDataDict16.txt) = 56b4e8fcc7596cc8b69c9e878f2e699aunzip csv_hus.zip
```

The exact steps we took

Cryptographic hashes of the file contents we downloaded. These are very short summaries (called hashes) that are highly unlikely to have the same value for different files. These summaries can later help us determine if another researcher in our organization is using the same data.

Listing 2.8 Loading data into R from a relational database

Attaches some packages we wish to use commands and functions from.

Loads the data from the compressed RDS disk format into R memory. Note: You will need to change the path PUMSsample to where you have saved the contents of PDSwR2/PUMS.

Connects to a new RSQLite in-memory database. We will use RSQLite for our examples. In practice you would connect to a preexisting database, such as PostgreSQL or Spark, with preexisting tables.

Copies the data from the in-memory structure dlist into the database

Removes our local copy of the data, as we are simulating having found the data in the database

Uses the SQL query language for a quick look at up to five rows of our data

Builds dplyr handles that refer to the remote database data

Uses dplyr to examine and work with the remote data

Uses the rquery package to get a summary of the remote data

```
library("DBI")
library("dplyr")
library("rquery")

dlist <- readRDS("PUMSsample.RDS")
db <- dbConnect(RSQLite::SQLite(), ":memory:")
dbWriteTable(db, "dpus", as.data.frame(dlist$ss16pus))
dbWriteTable(db, "dhus", as.data.frame(dlist$ss16hus))
rm(list = "dlist")

dbGetQuery(db, "SELECT * FROM dpus LIMIT 5")

dpus <- tbl(db, "dpus")
dhus <- tbl(db, "dhus")

print(dpus)
glimpse(dpus)

View(rsummary(db, "dpus"))
```

Listing 2.9 Loading data from a database

Copies data from the database into R memory. This assumes we are continuing from the previous example, so the packages we have attached are still available and the database handle db is still valid.

All the columns in this copy of PUMS data are stored as the character type to preserve features such as leading zeros from the original data. Here we are converting columns we wish to treat as numeric to the numeric type. Non-numeric values, often missing entries, get coded with the symbol NA, which stands for not available.

```
dpus <- dbReadTable(db, "dpus")

dpus <- dpus[, c("AGEP", "COW", "ESR",  "PERNP",
                 "PINCP","SCHL", "SEX", "WKHP")]

for(ci in c("AGEP", "PERNP", "PINCP", "WKHP")) {
   dpus[[ci]] <- as.numeric(dpus[[ci]])
}

dpus$COW <- strtrim(dpus$COW, 50)

str(dpus)
```

Selects a subset of columns we want to work with. Restricting columns is not required, but improves legibility of later printing.

The PUMS level names are very long (which is one of the reasons these columns are distributed as integers), so for this dataset that has level names instead of level codes, we are shortening the employment codes to no more than 50 characters.

Looks at the first few rows of data in a column orientation.

Listing 2.10 Remapping values and selecting rows from data

```
target_emp_levs <- c(
  "Employee of a private for-profit company or busine",
  "Employee of a private not-for-profit, tax-exempt, ",
  "Federal government employee",
  "Local government employee (city, county, etc.)",
  "Self-employed in own incorporated business, profes",
  "Self-employed in own not incorporated business, pr",
  "State government employee")

complete <- complete.cases(dpus)
```

Defines a vector of employment definitions we consider "standard"

Builds a new logical vector indicating which rows have valid values in all of our columns of interest. In real applications, dealing with missing values is important and cannot always be handled by skipping incomplete rows. We will return to the issue of properly dealing with missing values when we discuss managing data.

Builds a new logical vector indicating which workers we consider typical full-time employees. All of these column names are the ones we discussed earlier. The results of any analysis will be heavily influenced by this definition, so, in a real task, we would spend a lot of time researching the choices in this step. It literally controls who and what we are studying. Notice that to keep things simple and homogeneous, we restricted this study to civilians, which would be an unacceptable limitation in a complete work.

```
stdworker <- with(dpus,
                   (PINCP>1000) &
                    (ESR=="Civilian employed, at work") &
                    (PINCP<=250000) &
                    (PERNP>1000) & (PERNP<=250000) &
                    (WKHP>=30) &
                    (AGEP>=18) & (AGEP<=65) &
                    (COW %in% target_emp_levs))

dpus <- dpus[complete & stdworker, , drop = FALSE]
```

Restricts to only rows or examples that meet our definition of a typical worker

```
no_advanced_degree <- is.na(dpus$SCHL) |
   (!(dpus$SCHL %in% c("Associate's degree",
                      "Bachelor's degree",
                      "Doctorate degree",
                      "Master's degree",
                      "Professional degree beyond a bachelor's degree")))
dpus$SCHL[no_advanced_degree] <- "No Advanced Degree"
```

Recodes education, merging the less-than-bachelor's-degree levels to the single level No Advanced Degree

```
dpus$SCHL <- relevel(factor(dpus$SCHL),
                      "No Advanced Degree")
dpus$COW <- relevel(factor(dpus$COW),
                    target_emp_levs[[1]])
dpus$ESR <- relevel(factor(dpus$ESR),
                    "Civilian employed, at work")
dpus$SEX <- relevel(factor(dpus$SEX),
                    "Male")
```

Converts our string-valued columns to factors, picking the reference level with the relevel() function

```
saveRDS(dpus, "dpus_std_employee.RDS")
```

Save this data to a file so we can use it in later examples. This file is also already available at the path PDSwR2/PUMS/dpus_std_employee.RDS.

```
summary(dpus)
```

Takes a look at our data. One of the advantages of factors is that summary() builds up useful counts for them. However, it was best to delay having string codes as factors until after we finished with remapping level codes.

Shows the possible levels for SCHL

```
levels(dpus$SCHL)
## [1] "No Advanced Degree"                                   "Associate's degree"
## [3] "Bachelor's degree"                                     "Doctorate degree"
## [5] "Master's degree"                                       "Professional degree
     beyond a bachelor's degree"
```

Shows how the first few levels are represented as codes

```
head(dpus$SCHL)
## [1] Associate's degree Associate's degree Associate's degree No Advanced D
     egree Doctorate degree Associate's degree
##   6 Levels: No Advanced Degree Associate's degree Bachelor's degree Doctor
     ate degree ... Professional degree beyond a bachelor's degree
```

Shows the first few string values for SCHL

```
str(dpus$SCHL)
##  Factor w/ 6 levels "No Advanced Degree",..: 2 2 2 1 4 2 1 5 1 1 ...
```

Non-statisticians are often surprised that you can use non-numeric columns (such as

The cbind operator combines two data frames by columns, or each row is built by matching columns from rows in each data frame.

Builds a data.frame with the SCHL column recoded as character strings instead of as a factor

Builds a matrix with dummy variables generated from the SCHL factor column

```
d <- cbind(
   data.frame(SCHL = as.character(dpus$SCHL),
              stringsAsFactors = FALSE),
   model.matrix(~SCHL, dpus)
 )
d$'(Intercept)' <- NULL
 str(d)

## 'data.frame':    41305 obs. of  6 variables:
##  $ SCHL                                              : chr  "Associate's d
     egree" "Associate's degree" "Associate's degree" "No Advanced Degree" ..
     .
##  $ SCHLAssociate's degree                            : num  1 1 1 0 0 1 0
     0 0 0 ...
##  $ SCHLBachelor's degree                             : num  0 0 0 0 0 0 0
     0 0 0 ...
##  $ SCHLDoctorate degree                              : num  0 0 0 0 1 0 0
     0 0 0 ...
##  $ SCHLMaster's degree                               : num  0 0 0 0 0 0 0
     1 0 0 ...
##  $ SCHLProfessional degree beyond a bachelor's degree: num  0 0 0 0 0 0 0
     0 0 0 ...
```

Removes a column named "(Intercept)" from the data.frame, as it is a side effect of model.matrix that we are not interested in at this time.

Shows the structure that presents the original SCHL string form along with the indicators. str() presents the first few rows in transpose format (flipped so rows are now up and down and columns are across).

Listing 2.6 Summary of Good_Loan and Purpose

```
setwd("PDSwR2/Statlog")
 d <- readRDS("creditdata.RDS")

table(d$Purpose, d$Good_Loan)

##                      BadLoan GoodLoan
##   business                34       63
##   car (new)               89      145
##   car (used)              17       86
##   domestic appliances      4        8
##   education               22       28
##   furniture/equipment     58      123
##   others                   5        7
##   radio/television        62      218
##   repairs                  8       14
##   retraining               1        8
```

Sets the working directory. You will have to replace PDSwR2/Statlog with the actual full path to Statlog on your machine.

Reads the prepared statlog data

Exploring Data

- Summary statistics
- ***Typical problems revealed by data summaries***
  - common issues:
  - Missing values
  - Invalid values and outliers
  - Data ranges that are too wide or too narrow
  - The units of the data

Listing 3.1 The `summary()` command

Change this to your actual path to the directory where you unpacked PDSwR2

The variable is_employed is missing for about a third of the data. The variable income has negative values, which are potentially invalid.

```
setwd("PDSwR2/Custdata")
customer_data = readRDS("custdata.RDS")
summary(customer_data)
##     custid                sex          is_employed        income
##  Length:73262        Female:37837   FALSE: 2351    Min.    :  -6900
##  Class :character    Male  :35425   TRUE :45137    1st Qu.:  10700
##  Mode  :character                   NA's :25774    Median :  26200
##                                                    Mean   :  41764
##                                                    3rd Qu.:  51700
##                                                    Max.   :1257000
##
```

```
##             marital_status  health_ins
##  Divorced/Separated:10693   Mode :logical
##  Married           :38400   FALSE:7307
##  Never married     :19407   TRUE :65955
##  Widowed           : 4762
##
##
##
##                        housing_type   recent_move      num_vehicles
##  Homeowner free and clear     :16763   Mode :logical   Min.   :0.000
##  Homeowner with mortgage/loan:31387   FALSE:62418     1st Qu.:1.000
##  Occupied with no rent        : 1138   TRUE :9123      Median :2.000
##  Rented                       :22254   NA's :1721      Mean   :2.066
##  NA's                         : 1720                   3rd Qu.:3.000
##                                                        Max.   :6.000
##                                                        NA's   :1720
##       age                  state_of_res     gas_usage
##  Min.   :  0.00   California  : 8962   Min.   :  1.00
##  1st Qu.: 34.00   Texas       : 6026   1st Qu.:  3.00
##  Median : 48.00   Florida     : 4979   Median : 10.00
##  Mean   : 49.16   New York    : 4431   Mean   : 41.17
##  3rd Qu.: 62.00   Pennsylvania: 2997   3rd Qu.: 60.00
##  Max.   :120.00   Illinois    : 2925   Max.   :570.00
##                   (Other)     :42942   NA's   :1720
```

About 90% of the customers have health insurance.

The variables housing_type, recent_move, num_vehicles, and gas_usage are each missing 1720 or 1721 values.

The average value of the variable age seems plausible, but the minimum and maximum values seem unlikely. The variable state_of_res is a categorical variable; summary() reports how many customers are in each state (for the first few states).

Listing 3.2 Will the variable `is_employed` be useful for modeling?

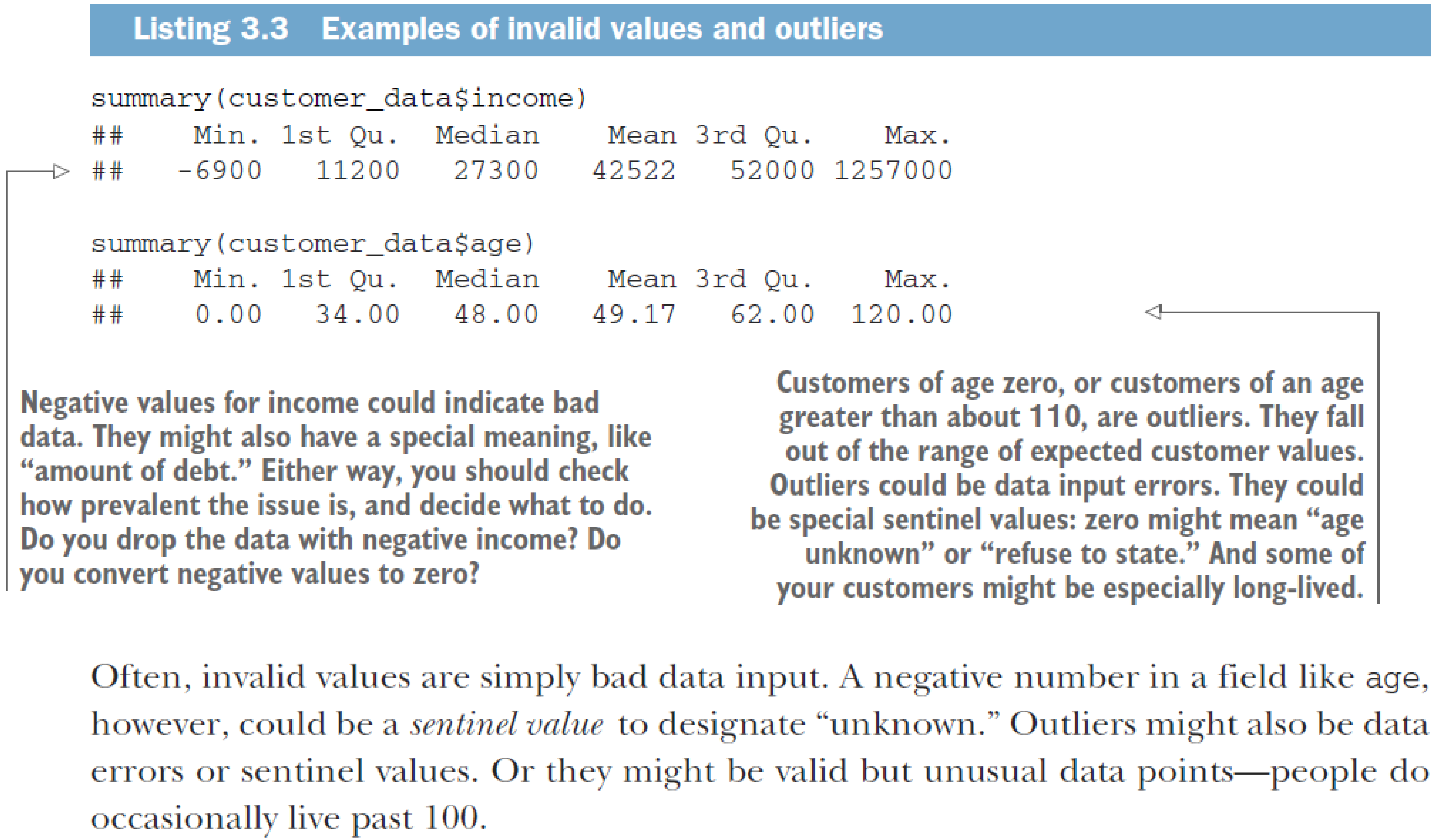
```
## is_employed
## FALSE: 2321
## TRUE :44887
## NA's :24333
```

The variable is_employed is missing for more than a third of the data. Why? Is employment status unknown? Did the company start collecting employment data only recently? Does NA mean "not in the active workforce" (for example, students or stay-at-home parents)?

```
##                       housing_type    recent_move
## Homeowner free and clear    :16763    Mode :logical
## Homeowner with mortgage/loan:31387    FALSE:62418
## Occupied with no rent       : 1138    TRUE :9123
## Rented                      :22254    NA's :1721
## NA's                        : 1720
##
##
##   num_vehicles     gas_usage
##  Min.   :0.000   Min.   :  1.00
##  1st Qu.:1.000   1st Qu.:  3.00
##  Median :2.000   Median : 10.00
##  Mean   :2.066   Mean   : 41.17
##  3rd Qu.:3.000   3rd Qu.: 60.00
##  Max.   :6.000   Max.   :570.00
##  NA's   :1720    NA's   :1720
```

The variables housing_type, recent_move, num_vehicles, and gas_usage are missing relatively few values—about 2% of the data. It's probably safe to just drop the rows that are missing values, especially if the missing values are all in the same 1720 rows.

Invalid values and outliers



Listing 3.3 Examples of invalid values and outliers

```
summary(customer_data$income)
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    -6900   11200   27300   42522   52000 1257000

summary(customer_data$age)
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.00   34.00   48.00   49.17   62.00  120.00
```

Negative values for income could indicate bad data. They might also have a special meaning, like "amount of debt." Either way, you should check how prevalent the issue is, and decide what to do. Do you drop the data with negative income? Do you convert negative values to zero?

Customers of age zero, or customers of an age greater than about 110, are outliers. They fall out of the range of expected customer values. Outliers could be data input errors. They could be special sentinel values: zero might mean "age unknown" or "refuse to state." And some of your customers might be especially long-lived.

Often, invalid values are simply bad data input. A negative number in a field like age, however, could be a *sentinel value* to designate "unknown." Outliers might also be data errors or sentinel values. Or they might be valid but unusual data points—people do occasionally live past 100.

Exploring Data

- ***Spotting problems using graphics and visualization***
- – Avoid too many superimposed elements, such as too many curves in the same graphing space.
- – Find the right aspect ratio and scaling to properly bring out the details of the data.
- – Avoid having the data all skewed to one side or the other of your graph.
- Visualization is an iterative process. Its purpose is to answer questions about the data.
- ***Visually checking distributions for a single variable***
- Histograms
- Density plots
- Bar charts
- Dot plots
- The visualizations in this section help you answer questions like these:
- What is the peak value of the distribution?
- How many peaks are there in the distribution (unimodality versus bimodality)?
- How much does the data vary? Is it concentrated in a certain interval or in a certain category?

```
summary(custdata$age)
# Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
# 0.00   34.00   48.00   49.16   62.00  120.00
```

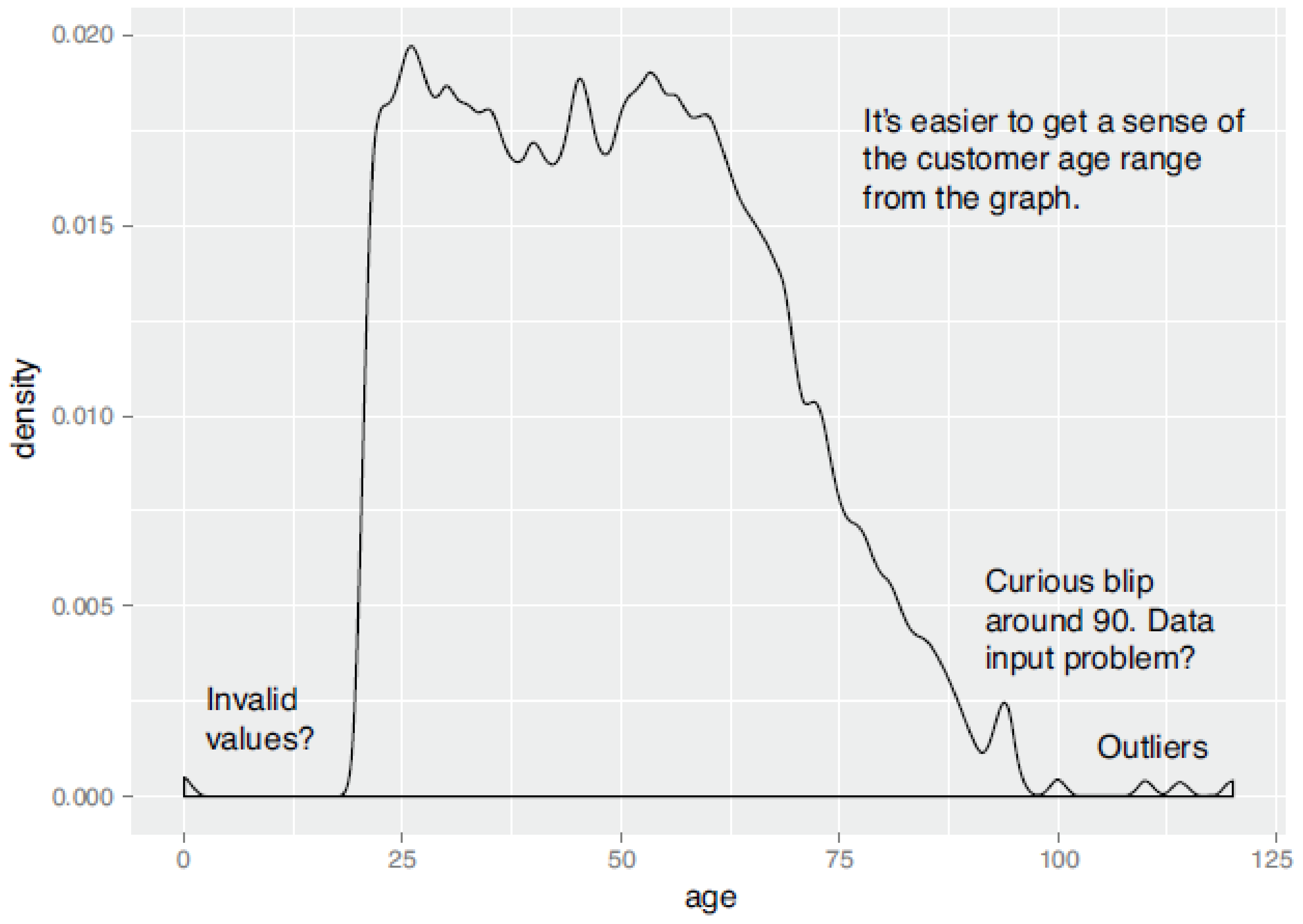


Figure 3.2 Some information is easier to read from a graph, and some from a summary.

A note on ggplot2

The theme of this section is how to use visualization to explore your data, not how to use `ggplot2`. The `ggplot2` package is based on Leland Wilkinson's book, *Grammar of Graphics*. We chose `ggplot2` because it excels at combining multiple graphical elements together, but its syntax can take some getting used to. Here are the key points to understand when looking at our code snippets:

- Graphs in `ggplot2` can only be defined on data frames. The variables in a graph—the *x* variable, the *y* variable, the variables that define the color or the size of the points—are called *aesthetics*, and are declared by using the `aes` function.
- The `ggplot()` function declares the graph object. The arguments to `ggplot()` can include the data frame of interest and the aesthetics. The `ggplot()` function doesn't itself produce a visualization; visualizations are produced by *layers*.
- Layers produce the plots and plot transformations and are added to a given graph object using the + operator. Each layer can also take a data frame and aesthetics as arguments, in addition to plot-specific parameters. Examples of layers are `geom_point` (for a scatter plot) or `geom_line` (for a line plot).

Visually checking relationships between two variables

- Is there a relationship between the two inputs *age* and *income* in my data?
- If so, what kind of relationship, and how strong?
- Is there a relationship between the input *marital status* and the output *health insurance*? How strong?
- Line plots and scatter plots for comparing two continuous variables
- Smoothing curves and hexbin plots for comparing two continuous variables at high volume
- Different types of bar charts for comparing two discrete variables
- Variations on histograms and density plots for comparing a continuous and discrete variable

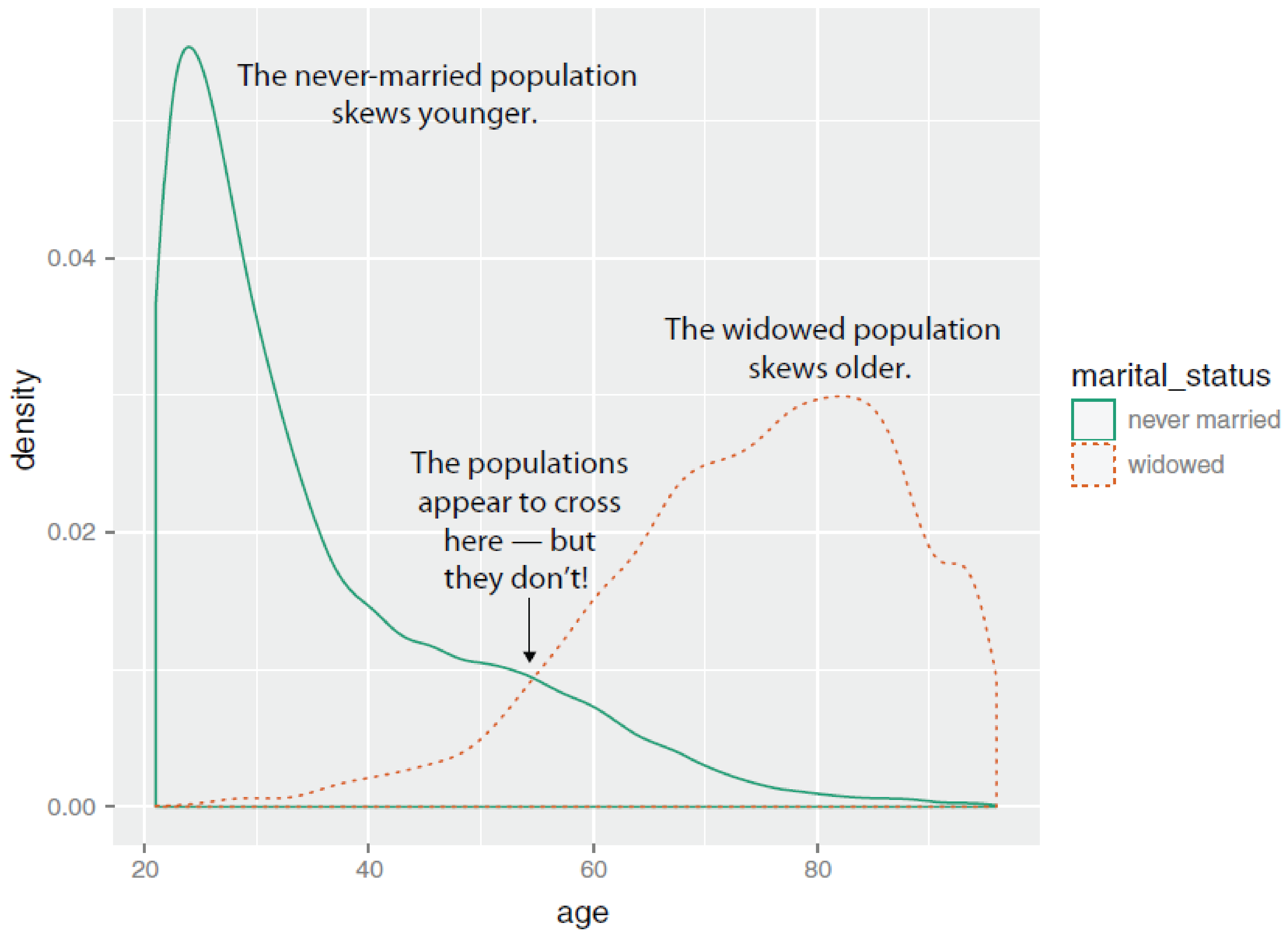


Figure 3.22 Comparing the distribution of marital status for widowed and never married populations

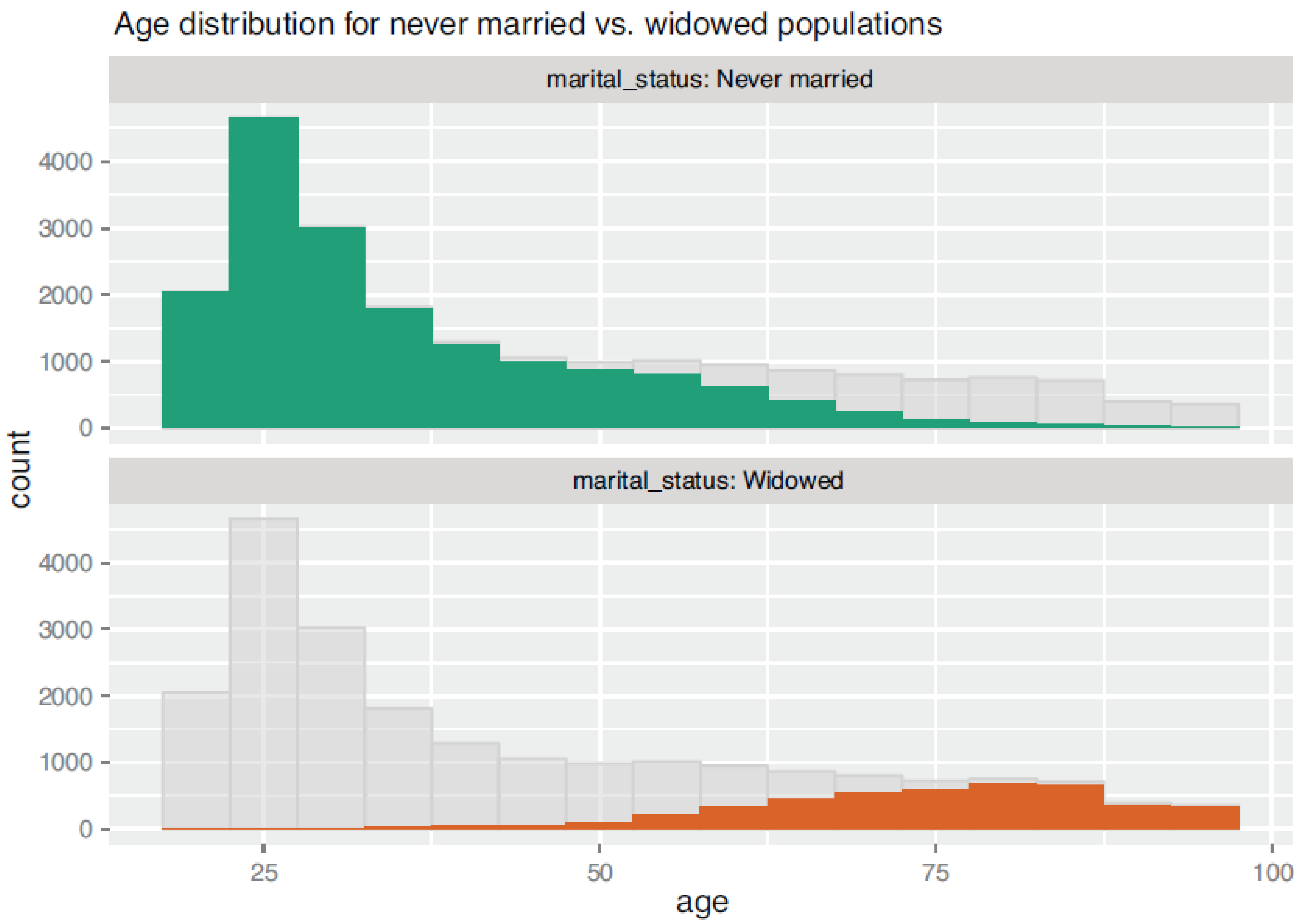


Figure 3.23 `ShadowHist` **comparison of the age distributions of widowed and never married populations**

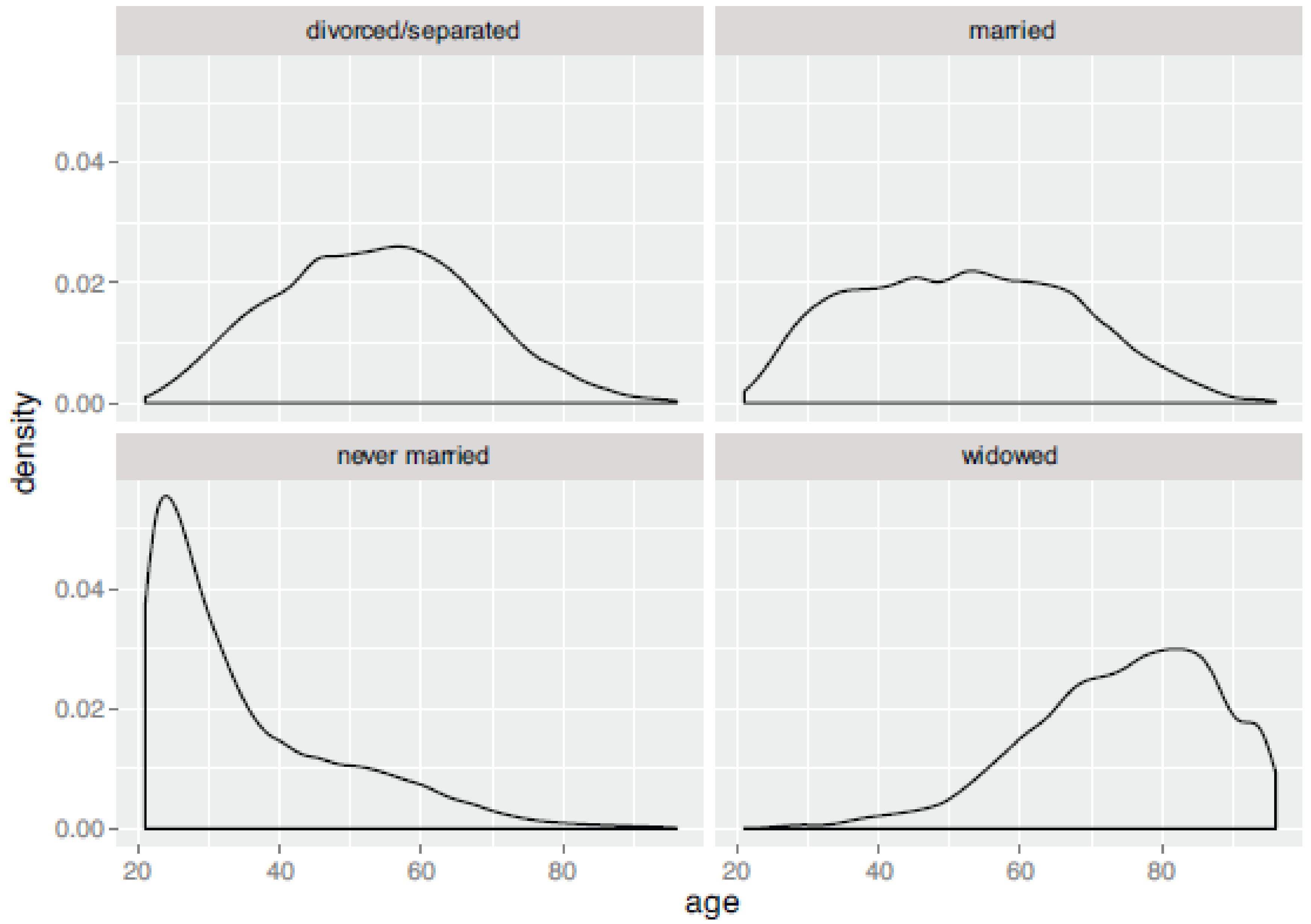


Figure 3.24 Faceted plot of the age distributions of different marital statuses

Facet plots, also known as trellis plots or small multiples, are **figures made up of multiple subplots which have the same set of axes, where each subplot shows a subset of the data**

Graph type	Uses	Examples
Smoothing curve	Shows underlying "average" relationship, or trend, between two continuous variables. Can also be used to show the relationship between a continuous and a binary or Boolean variable: the fraction of `true` values of the discrete variable as a function of the continuous variable.	Estimate the "average" relationship of income to years in the workforce.
Hexbin plot	Shows the relationship between two continuous variables when the data is very dense.	Plot income vs. years in the workforce for a large population.
Stacked bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Highlights the frequencies of each value of `var1`. Works best when `var2` is binary.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to retain information about the number of people in each marital category.
Side-by-side bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Good for comparing the frequencies of each value of `var2` across the values of `var1`. Works best when `var2` is binary.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to directly compare the number of insured and uninsured people in each marital category.

Shadow plot	Shows the relationship between two categorical variables (`var1` and `var2`). Displays the frequency of each value of `var1`, while allowing comparison of `var2` values both within and across the categories of `var1`.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to directly compare the number of insured and uninsured people in each marital category *and* still retain information about the total number of people in each marital category.
Filled bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Good for comparing the relative frequencies of each value of `var2` within each value of `var1`. Works best when `var2` is binary.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to compare the *ratio* of uninsured to insured people in each marital category.
Bar chart with faceting	Shows the relationship between two categorical variables (`var1` and `var2`). Best for comparing the relative frequencies of each value of `var2` within each value of `var1` when `var2` takes on more than two values.	Plot the distribution of marital status (`var2`) as a function of housing type (`var1`).

Graph type	Uses	Examples
Overlaid density plot	Compares the distribution of a continuous variable over different values of a categorical variable. Best when the categorical variable has only two or three categories. Shows whether the continuous variable is distributed differently or similarly across the categories.	Compare the age distribution of married vs. divorced populations.
Faceted density plot	Compares the distribution of a continuous variable over different values of a categorical variable. Suitable for categorical variables with more than three or so categories. Shows whether the continuous variable is distributed differently or similarly across the categories.	Compare the age distribution of several marital statuses (never married, married, divorced, widowed).
Faceted histogram or shadow histogram	Compares the distribution of a continuous variable over different values of a categorical variable while retain-	Compare the age distribution of several marital statuses (never married, married, divorced, wid-