Emily Riederer

Data Disasters

To all the mistakes I've made (data, and otherwise) and those who tolerated my making them.

Contents

List of Tables

List of Figures

Preface

Training in data analysis often begins with Statistics 101 course. Students learn the "happy path" of answer data that adheres to specific assumptions (such as "independent and identically distributed with a Normal density") and answers pre-specified questions (most notably, the infamous null hypothesis significance test). Then, they venture out into the world of real-world data analysis where non-experimental data is rarely so well behaved and the questions asked of it are far more nuanced.

No one course should aim to teach students everything they should know

about statistics. In fact, one of the best parts about a career in statistics is the responsibility and privilege of life-long learning. However, **the flaw of introductory statistics is not that it's incomplete, but that it's not obvious how it is *not* complete**. Statistics is a bad salesman. There's no season finale, no cliff hanger, no teasing and hinting and promising more and better to come. Student may leave their studies believing that answering more complex data analysis questions is trivially easy (by relying on the one-size-fits-all "panacea" that they learned) or intractably difficult (when the assumptions of that method are not met.)

This book attempts to add more color to all the dimensions of data analysis while showcasing the nuances throughout the true *life cycle* of data analysis using two strategies.

First, it attempts to highlight common pitfalls in all the parts of data analysis: from data management and computation to visualization, interpretation, and modeling and even to communication and collaboration. Data analysis is fundamentally a *creative* task, so there are rarely canonical one-size-fits-all solutions. Curiously, however, there are plenty of canonical *issues* even if they require different solutions in different settings. Thus, the goal of this book is to highlight common *data disasters* and, in doing so, help students cultivate an intuition for how to detect common problems before they occur in an important analysis.

Second, while exploring these *data disasters*, we humbly put forth a (woefully incomplete!) literature review of more advanced methods from statistics and other quantitative disciplines (e.g. economics, epidemiology), to help learners build a "mental index" of terms to search and techniques to study should they encounter a relevant problem.

The content in this book is currently being developed and is all subject to change.

Chapters and sections tagged as **WIP** (work-in-progress) have substantial content and are suitable for reading.

Chapters and sections tagged as **TODO** have minimal outlines or code examples (if that).

0.1 Main Topics

In particular, we will aim to help you avoid eleven types of data disasters:

- **Data Dalliances**: Misinterpreting or misuing data based on how it was collected or what it represents

- **Computational Quandaries**: Letting computers do what you said and not what you meant
- **Egregious Aggregations**: Losing critical information when information is condensed
- **Vexing Visualization**: Confusing ourselves or others with plotting choices
- **Incredible Inferences**: Drawing incorrect conclusions for analytical results
- **Cavalier Causality**: Falling prey to spurious correlations masquerading as causality
- **Mindless Modeling**: Failing to get the most value out of models by not tailoring the features, targets, and performance metrics
- **Alternative Algorithms**: Lacking an understanding of alternative methods which may be better suited for the problem at hand
- **Futile Findings**: Asking and answering questions that aren't useful
- **Complexifying Code**: Making projects unwieldy or more difficult to understand than necessary
- **Rejecting Reproducibility**: Working inefficiently instead of an efficient, reproducible, and sharable workflow

0.2 Common Themes

In each chapter, we will see numerous examples of each disaster and consider strategies to help us mitigate. Along the way, we'll emphasize:

- The importance of **domain knowledge** and the **data-generating process** to decide what it is you want to do
- The utility of **simulation** as a tool to explore if, in fact, you are doing it
- The exploration of **counterexamples** to build **intuition for common patterns** of problems even where common solutions don't exist

As we go, we will notice how three common themes that challenge the focus of introductory statistics:

- Summary statistics mask interesting stories that we see when focusing on the **variation**
- Similarly, observations and variables are rarely independent; the story is in the **covariance**
- Assumptions of Normality, or more broadly symmetry, are often in appropriate in wonky, **highly skewed** world

0.3 Acknowledgements

This book is built by the excelent **bookdown** R package with styling tips heavily inspired by the Tidy Maching Leaning with R[1] repo.

Sidebar icons used throughout the book are credited to Vector Markets[2] (light-bulb) and Freepik[3] (notepad and caution sign) on www.flaticon.com[4]. All other images are my own and made with either Excalidraw[5] or Inkscape[6].

[1] https://github.com/tidymodels/TMwR
[2] https://www.flaticon.com/authors/vectors-market
[3] https://www.freepik.com%22
[4] https://www.flaticon.com/
[5] https://excalidraw.com/
[6] https://inkscape.org/

About the Author

Emily Riederer is...

Find me on my website[7] and on Twitter[8]

[7] https://emilyriederer.netlify.com?utm_source=datadisasters
[8] https://twitter.com/EmilyRiederer

1

Introduction (TODO)

Statistics is not synonymous with data analysis; rigor vs practicality

"Evaluating the Success of a Data Analysis" (Hicks and Peng, 2019)

"Data Alone is not Ground Truth" (Bassa, 2017)

1.1 Case Study

1.2 What is data?

Data is...

1.3 What is analysis?

Analysis is the process of turning information into insight...S

1.4 What is workflow?

Workflows are an intentional process for accomplishing a complex goal...

1.5 What is data analysis?

Data analysis altogether is…

1.6 What are data disasters?

Data disasters occur when…

Data

2

Data Dalliances (WIP)

The first step to data analysis is, in fact, data. While this may seem obvious, statistics textbooks often dodge this detail. Discussions of regression analysis often begin with a statement like:

"Let X be the nxp design matrix of independent variables…"

---

but in practice this statement is as absurd as writing a book about how to win a basketball game, assuming your team already has a 20 point lead with 1 minute left to play.

It's very convenient but typically incorrect to assume that the data we happen to have is the ideal (or, more humbly, sufficient) data for the questions we wish to analyze. The specific vagaries of data vary greatly by domain, but a commonality across many fields (such as political science, economics, epidemiology, and market research) is that we are often called to work with *found data* (or, more formally, "observational data") from administrative sources or production systems. In contrast to artisanally crafted data experimental data (like the carefully controlled agricultural experiments which motivated many early methods developments in statistics), this data was generated neither by us nor for us. To quote Angela Bassa, the head of data science at an e-commerce company: "Data isn't ground truth. Data are artifacts of systems" (Bassa, 2017).

The analytical implications of observational versus experimental data are well explored in the field of causal inference (which we will discuss some in Chapters 6 and 7). However, this distinction has implications far earlier in the data analysis process, as well. To name a few:

- Records and fields many not represent the entities or measures most conducive to analysis
- Data collection methods may capture a different subset of events or do so at a different frequency than we expected, leading to systemic biases

- Data movement between systems can insert errors (or, at minimum, challenges to our intuition)
- Data transformations may be fragile or transient, reflecting the primary purpose of the system not our unrelated analytical use

In this chapter, we will explore data structures and the full data generating process to better understand how different types of data challenges emerge. In doing so, we will hone sharper intuition for how our data can deceive us and what to watch out for when beginning an analysis.

2.1 Preliminaries

Before we begin our exploration of data dalliances, we must first establish a baseline understanding of data structure, data production, and data quality.

2.1.1 Data Structure Basics

2.1.1.1 Relational Data Structure

Understanding the content and structure of the data you are using is a critical prerequisite to analysis. In this book, we focus on tabular, structured data like one might find in an Excel spreadsheet or relational database.[1]

In particular, many tools work best with what R developer Hadley Wickham describes as "tidy data" (Wickham, 2014). Namely:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

This is analogous to how one generally finds data arranged in a database and how statisticians are used to conceptualizing it. For example, the design matrix of a linear model consists of one column of data for each independent variable to be included in the model and one row for each observation.[2] As

[1] Other types of data that one might encounter in the wild include free text, JSON, spatial data, and more. However, many of these require either more advanced analysis techniques or structuring that converts them into tabular data, so they are out of the scope of this discussion.

[2] When data is arrange this way in a matrix X, linear regression coefficients can be computed as $\beta = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$

Wickham points out, this is also similar to what is called "3rd normal form" in the world of relational database management systems.

Using this data structure is valuable not only because it is similar to what many modern data tools expect, but also because it provides us a framework to think critically about what defined each observation and each variable in our dataset.

2.1.1.2 Schemas (TODO)

2.1.2 Data Production Processes

In statistical modeling we discuss the **data generating process**: we can build models that describe the mechanisms that create our observations. We can broaden this notion to think about the generating process of each of these steps of data production.

Regardless of the type of data (experimental, observational, survey, etc.), there are generally four main steps to production: collection, extraction, loading, and transformation.[3]

- **Collect**: The way in which signals from the real world are captured as data. This could include logging (e.g. for web traffic or system monitoring), sensors (e.g. temperature collection), surveys, and more
- **Extract**: The process of removing data from the place in which it was originally captured in preparation of moving it somewhere in which analysis can be done
- **Load**: The process of loading the extracted data to its final destination
- **Transform**: The process of modeling and transforming data so that its structure is useful for analysis and its variables are interpretable

To better theorize about data quality issues, it's useful to think of four DGPs (as shown in Figure 2.1): the real-world DGP, the data collection/extraction DGP[4], the data loading DGP, and the data transformation DGP.

2.1.2.1 E-Commerce Data Example

For example, consider the role of each of these four DGPs for e-commerce data:

[3] You may hear the last three referred to as ELT or ETL

[4] I don't mean to imply statisticians do not regularly think about the data collection DGP! The rich literature on missing data imputation, censored data in survival analysis, and non-response bias is survey data collection are just a few examples of how carefully statisticians think about how data collection impacts analysis. I chose to break it out here to discuss the more technical aspects of collection

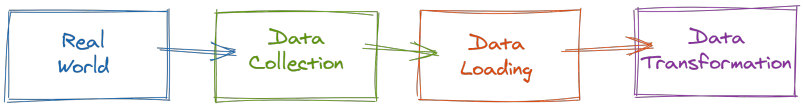


FIGURE 2.1: A schematic of the data production process

- **Real-world DGP**: Supply, demand, marketing, and a range of factors motivate a consumer to visit a website and make a purchase
- **Data collection DGP**: Parts of the website are instrumented to log certain customer actions. This log is then extracted from the different operational system (login platforms, payment platforms, account records) to be used for analysis
- **Data loading DGP**: Data recorded by different systems is moved to a data warehouse for further processing through some sort of manual, scheduled, or orchestrated job. These different systems may make data available at different frequencies.
- **Data transformation DGP**: To arrive at that final data presentation requires creating a data model[5] to describe domain-specific attributes with key variables crafted with data transformations

2.1.2.2 Subway Data Example

Or, consider the role of each of these four DGPs for subway ridership data[6]:

- **Real-world DGP**: Riders are motivated to use public transportation to commute, run errands, or visit friends. Different motivating factors may cause different weekly and annual seasonality
- **Data collection DGP**: To ride the subway, riders go to a station and enter and exit through turnstiles. The mechanical rotation of the turnstile caused by a rider passing through is recorded
- **Data loading DGP**: Data recorded at each turnstile is collected through a centralized computer system at the station. Once a week, each station uploads a flat file of this data to a data lake owned by the city's Department of Transportation
- **Data transformation DGP**: Turnstiles from different companies may have different data formats. Transformation may include harmonizing disparate sources, coding system-generated codes (e.g. Station XYZ) to semantically meaningful names (e.g. Main Street Station), and publishing a final unified representation across stations and across time

[5] https://en.wikipedia.org/wiki/Data_model

[6] Like NYC's infamously messy turnstile data[7]. I don't claim to know precisely how this dataset is created, but many of the specific challenges it contains are highly relevant.

Throughout this chapter, we'll explore how understanding key concepts about each of these DGPs can help guide our intuition on where to look for problems.

2.1.3 Data Quality Dimension

To guide our discussion of how data production can affect aspects of data quality, we need a guiding definition of data quality. This is challenging because data quality is *subjective* and *task-specific*. It matters much more if data is "fit for purpose" and operates in a way that is *transparent* to its users moreso than meeting some preordained quality standard.

Regardless, it's useful for our discussion to think about general dimensions of data quality. Here, we will rely on six dimensions of data quality outlined by Data Management Association (dam, 2020). Their official definitions are:

1. **Completeness**: The proportion of stored data against the potential of "100% complete"
2. **Uniqueness**: Nothing will be recorded more than once based upon how that thing is identified. It is the inverse of an assessment of the level of duplication
3. **Timeliness**: The degree to which data represent reality from the required point in time
4. **Validity**: Data are valid if it conforms to the syntax (format, type, range) of its definition
5. **Accuracy**: The degree to which data correctly describes the "real world" object or event being described.
6. **Consistency**: The absence of difference, when comparing two or more representations of a thing against a definition

2.1.4 Questions to Ask (TODO)

Our goal of understanding data is to ensure can assess its data quality and fit for our purpose. Understanding both its structure and its production process helps to accomplish this.

2.2 Data Collection

One of the tricky nuances of data collection is understanding what precisely is getting captured and logged in the first place. No matter how robust the

sensors, loggers, or other mechanisms are that record our dataset, that data is still unfit for its purpose so long as the analyst does not fully understand what it represents. In the next section, we will see how what data gets collected (and our understanding of it) can alter our notions of data completion and how we must handle it in our computations.

2.2.1 What Makes a Record (Row)

The first priority when starting to work with a dataset is understanding what a single record (row) represents and what causes it to be generated.

Consider something as simple as a login system where users must enter their credentials, endure a Captcha-like verification process to prove that they are not a robot, and enter a multi-factor authentication code. Figure 2.2 depicts such a process.

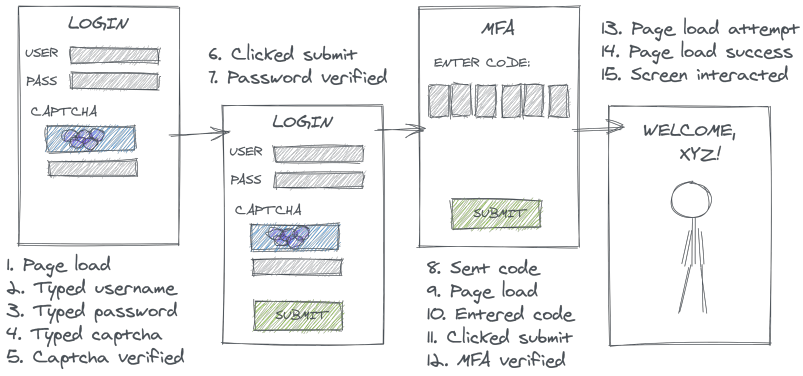


FIGURE 2.2: A diagram illustrating a multi-step process for a user to login to a website or app

Which of these events gets collected and recorded has a significant impact on subsequent data processing. In a technical sense, no inclusion/exclusion decision here is *incorrect*, per say, but if the producers' choices don't match the consumers' understandings, it can lead to misleading results.

For example, an analyst might seek out a `logins` table in order to calculate the rate of successful website logins. Reasonably enough, they might compute this rate as the sum of successful events over the total. Now, suppose two users attempt to login to their account, and ultimately, one succeeds in accessing their private information and the other doesn't. The analyst would probably hope to compute and report a 50% login success rate. However, depending on how the data is represented, they could quite easily compute nearly any value from 0% to 100%.

TABLE 2.1: Success rate naively computed under different data collection schemes

	Session	Attempt	Event	Outcome	Intermediate
Success	1	1	6	1	2
Total	2	4	7	1	2
Rate (%)	50	25	86	100	100

As a thought experiment, we can consider what types of events might be logged:

- **Per Attempt**: If data is logged once per overall login attempt, successful attempts only trigger one event, but a user who forgot their password may try (and fail) to login multiple times. In the case illustrated above, that deflates the successful login rate to **25%**.
- **Per Event**: If the logins table contains a row for every login-related event, each 'success' will trigger a large number of positive events and each 'failure' will trigger a negative event preceded by zero or more positive events. In the case illustrated below, this inflates our successful login rate to **86%**.
- **Per Conditional**: If the collector decided to only look at downstream events, perhaps to circumvent record duplication, they might decide to create a record only to denote the success or failure of the final step in the login process (MFA). However, login attempts that failed an upstream step would not generate any record for this stage because they've already fallen out of the funnel. In this case, the computed rate could reach **100%**
- **Per Intermediate**: Similarly, if the login was defined specifically as successful password verification, the computed rate could his **100%** even if some users subsequently fail MFA

These different situations are further illustrated in Figure 2.3, and their calculations are shown in Table 2.1.

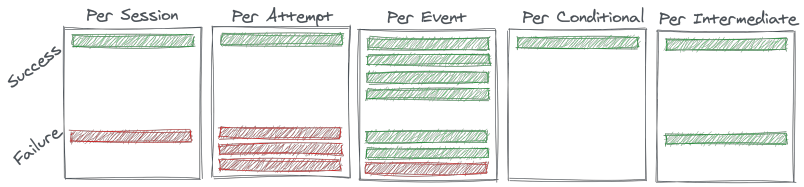


FIGURE 2.3: Login events recorded under different data collection paradigms

(Note that we did not explicitly "simulate" data here, but the workflow is largely the same. We imagine a real-world process, how that might translate

into a digital representation, and created a numerical example to understand the implications. Not all simulations require fancy code; sometimes paper and a pencil or a thought experiment works just fine.)

While humans have a shared intuition of what concepts like a user, session, or login are, the act of collecting data forces us to map that intuition onto an atomic event. Any misunderstanding in precisely what that definition is can have massive impact on the perceived data quality; "per event" data will appear heavily duplicated if it is assumed to be "per session" data.

In some cases, this could be obvious to detect. If the system outputs fields that are incredibly specific (e.g. with some hyperbole, imagine a `step_in_the_login_process` field with values taking any of the human-readable descriptions of the fifteen processes listed in the image above), but depending how this source is organized (e.g. in contrast to above, if we only have fields like `sourceid` and `processid` with unintuitive alphanumeric encoded values) and defined, it could be nearly impossible to understand the nuances without uncovering quality metadata or talking to a data producer.

2.2.2 What Doesn't Make a Record (Row)

Along with thinking about what *does* count (or gets logged), it's equally important to understand what systemically does not generate a record. Consider users who have the intent or desire to login (motivated by a real-world DGP) but cannot find the login page, or users who load the login page but never click a button because they know that they've forgotten their password and see no way to request it. Often, some of these corner cases may be some of the most critical and informative (e.g. here, demonstrating some major flaws in our UI). It's hard to *computationally* validate what data doesn't exist, so *conceptual* data validation is critical.

2.2.3 Records versus Keys

The preceding discussion on what types of real-world observations will or will not generate records in our resulting dataset is related to but distinct from another important concept from the world of relational databases: **primary keys**.

Primary keys are a minimal subset of variables in a dataset than define a unique record. For example, in the previous discussion of customer logins this might consist of **natural keys**[8] such as the combination of a `session_id` and

[8] Keys with semantic meanings that are naturally part of the dataset

a `timestamp` or **surrogate keys**[9] such as a global `event_id` that is generated every time the system logs any event.

Understanding a table's primary keys can be useful for many reasons. To name a few reasons, these fields are often useful for linking data from one table to another and for identifying data errors (if the uniqueness of these fields are not upheld). They also can be suggestive of the true granularity of the table.

However, simply knowing a table's primary keys does *not* resolve the issues we discussed in the prior two sections. Any of the many different data collection strategies we considered are *unique* by session and timestamp; however, as we've seen, that is no guarantee that they *must* contain every session and timestamp in the universe of events.

2.2.4 What Defines a Variable (Column)

Just as critical as understanding what constitutes a record (row) in a dataset is understanding the precise definition of each variable (column). Superficially, this task seems easier: after all, each variable has a name which hopefully includes some semantic information. However, quite often this information can provide a false sense of security. Just because you identify a variable with a promising sounding name, that does not mean that it is the most relevant data for your analysis.

For example, consider wanting to analyze patterns in customer spend amounts across orders on an e-commerce website. You might find a table of orders with a field called `amt_spend`. But what might this mean?

- If the dataset is sourced from a payment processor, it likely includes the total amount billed to a customers' credit card: including item prices less any discounts, shipping costs, taxes, etc. Alternatively, if this order was split across a gift card and a credit card, this field might only reflect the amount charged to the credit card
- If the dataset is created for Finance, it might perhaps include only the total of item prices less discounts if this best corresponded to the data the Finance team needs for revenue reporting
- Someone, somewhere, at some point might have assigned `amt_spend` to the name of the variable containing gross spend (before accounting for any discounts) and there might be a different variable `amt_spend_net` which accounts for discounts applied

It's critical to understand what each variable *actually* means. The upside of

[9] Keys without semantic meaning that exist primarily for the purpose of being keys

this is that it forces analysts to think more crisply about their research questions and what the *ideal* variables for their analysis would be. As we've seen, concepts like "spend" may seem deceptively simple, but are not unambiguous.

2.3 Data Extraction & Loading

Checking that data contains expected and *only* expected records (that is, completeness, uniqueness, and timeliness) is one of the most common first steps in data validation. However, the superficially simple act of loading data into a data warehouse or updating data between tables can introduce a variety of risks to data completeness which require different strategies to detect. Data loading errors can result in data that is stale, missing, duplicate, inconsistently up-to-date across sources, or complete but for only a subset of the range you think.

While the data quality principles of **completeness**, **uniqueness**, and **timeliness** would suggest that records should exist once and only once, the reality of many haphazard data loading process means data may appear sometime between zero and a handful of times. Data loads can occur in many different ways. For example, they might be:

- manually executed
- scheduled (like a cron[10] job)
- orchestrated (with a tool like Airflow[11] or Prefect[12])

No approach is free from challenges. For example, scheduled jobs risk executing before an upstream process has completed (resulting in stale or missing data); poorly orchestrated jobs may be prevented from working due to one missing dependency or might allow multiple stream to get out of sync (resulting in multisource missing data). Regardless of the method, all approaches must be carefully configured to handle failures gracefully to avoid creating duplicates, and the frequency at which they are executed may cause partial loading issues if it is incompatible with the granularity of the source data.

2.3.0.1 Data Load Failure Modes

To develop our understanding of the true data generating process and to formulate theories on how our data could be broken (and what we should

[10] https://en.wikipedia.org/wiki/Cron
[11] https://airflow.apache.org/
[12] https://www.prefect.io/

validate), it is useful to understand the different ways data extraction and loading can fail.

Figure 2.4 illustrates a number of examples. Suppose that each row of boxes in the diagram represents one day of records in a table.

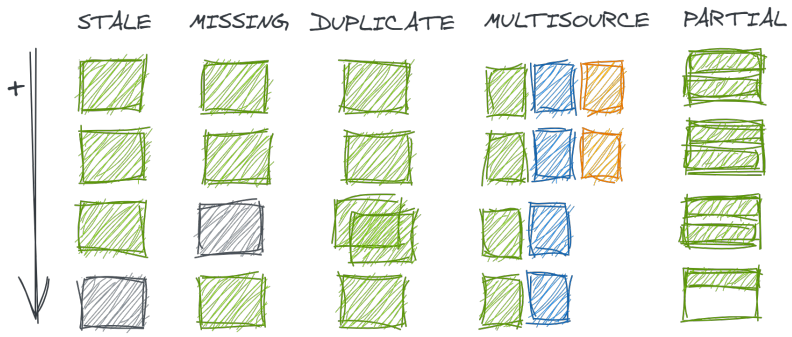


FIGURE 2.4: Different modes of data loading failure

Our dataset might be susceptible to:

- **Stale data** occurs when the data is not as up-to-date as would be expected from is regular refresh cadence. This could happen if a manual step was skipped, a scheduled job was executed before the upstream source was available, or orchestrated data checks found errors and quarantined new records
- **Missing data** occurs when one data load fails but subsequent loads have succeeded
- **Duplicate data** occurs when one data load is executed multiple times
- **Multisource missing data** occurs when a table is loaded from multiple sources, and some have continued to update as expected while others have not
- **Partial data** occurs when a table is loaded correctly as intended by the producer but contains less data than expected by the consumer (e.g. a table loads ever 12 hours but because there is some data for a given date, the user assumes that all relevant records for that date have been loaded)

The differences in these failure modes become important when an analyst attempts to assess data completeness. One of the first approaches an analyst might consider is simply to check the `min()` and `max()` event dates in their table. However, this can only help detect stale data. To catch missing data, an analyst might instead attempt to count the number of `distinct` days represented in the data; to detect duplicate data, that analyst might need to count records by day and examine the pattern.

Metric	Stale	Missing	Duplicate	Multi	Partial
`min(date)`	**13**	14	14	14	14
`max(date)`					
`count(distinct date)`	**3**	**3**	4	4	4
`count(1) by date`	**100100100010010001001001002001001**			1001006666	10010010050
`count(1)`	300300	300300	**400300**	332332	350350
`count(distinct PKs)`					

In a case like the toy example above where the correct number of rows per date is highly predictable and the number of dates is small, such eyeballing is feasible; however when the expected number of records varies day-to-day or time series are long, this approach becomes subjective, error-prone, and intractable. Additionally, it still might be hard to catch errors in mutli-source data or partial loads if the lower number of records was still within the bounds of reasonable deviation for a series. These last two types deserve further exploration.

2.3.0.2 Multi-Source

A more effective strategy for assessing data completeness requires a better understanding of how data is being collected and loaded. In the case of multi-source data, one single source stopping loading may not be a big enough change to disrupt aggregate counts but could still jeopardize meaningful analysis. It would be more useful to conduct completeness checks by *subgroup* to identify these discrepancies.

But not any subgroup will do; the subgroup must correspond to the various data sources. For example, suppose we run an e-commerce store and wish to look at sales from the past month by category. Naturally, we might think to check the completeness of the data by category. But what if sales data is sourced from three separate locations: our Shopify site (80%), our Amazon Storefront (15%), and phone sales (5%). Unless we explicitly check completeness by channel (a dimension we don't particularly care about for our analysis), it would be easy to miss if our data source for phone sales has stopped working or loads at a different frequency.

Another interesting aspect of multi-source data, is multiple sources can contribute either to different *rows/records* or different *columns/variables*. Table-level frequency counts won't help us in the latter case since other sources might create the right total number of records but result in some specific fields in those records being missing or inaccurate.

2.3.0.3 Partial Loads

Partial loads really are not data errors at all, but are still important to detect since they can jeopardize an analysis. A common scenario might occur if a job loads new data every 12 hours (say, data from the morning and afternoon of day n-1 loads on day n at 12AM and 12PM, respectively). An analyst retrieving data at 11AM may be concerned to see an approximate ~50% drop in sales in the past day, despite confirming that their data looks to be "complete" since the maximum record date is, in fact, day n-1. Of course, this concern could be somewhat easily allayed if they then checked a timestamp field, but such a field might not exists or might not have been used for validation since its harder to anticipate the appropriate maximum timestamp than it is the maximum date.

2.3.0.4 Delayed or Transient Records

The interaction between choices made in the data collection and data loading phases can introduce their own sets of problems.

Consider an `orders` table for an e-commerce company that analysts may use to track customer orders. It might contain one record per `order_id` x `event` (placement, processing, shipment), one record per order placed, one record per order shipping, or one record per order with a `status` field that changes over time to denote the order's current stage of life. Some of these options are illustrated in Figure 2.5.

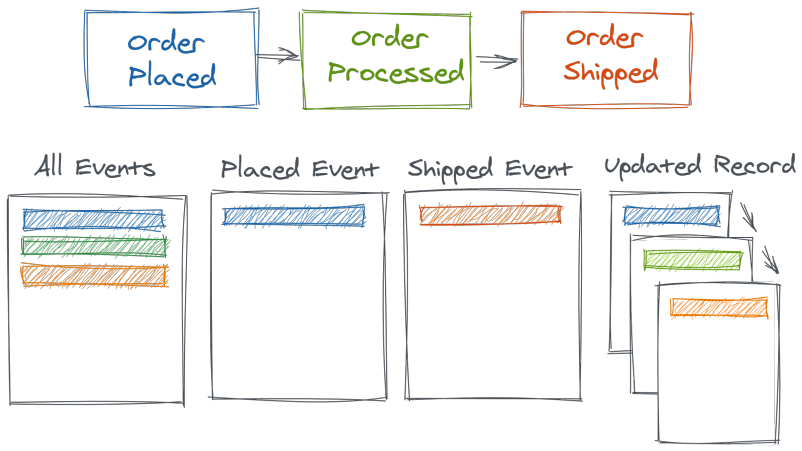


FIGURE 2.5: Illustration of alternative data collection and extraction strategies for order data

Any of these modeling choices seem reasonable and the difference between

them might appear immaterial. But consider the *collection* choice to record and report *shipped* events. Perhaps this might be operationally easier if shipment come from one source system whereas orders could come from many. However, an interesting thing about shipments is that they are often lagged in a variable way from the order date.

Suppose the e-commerce company in question offers three shipping speeds at checkout. Figure 2.6 shows the range of possible shipment dates based on the order dates for the three different speeds (shown in different bars/colors).

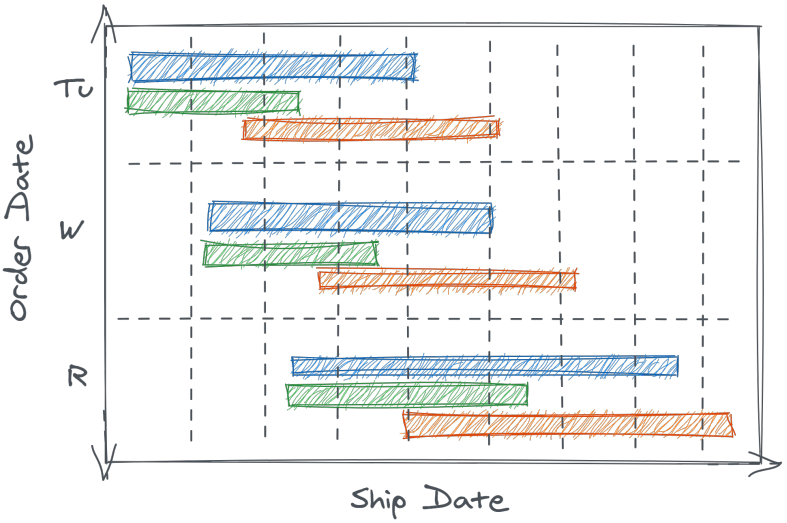


FIGURE 2.6: A conceptual chart of when different classes of real-world events might materialize as records in our dataset

How might this effect our perceived data quality?

- Order data could appear **stale** or not timely since orders with a given `order_date` would only load days later once shipped
- Similar to **missing** or **multisource** data, the data *range* in the table could lead to deceptive and incomplete data validation because some orders from a later order date might ship (and thus be logged) before all orders from a previous order date
- Put another way, we could have multiple order dates demonstrating **partial** data loads
- These features of the data might behave inconsistently across time due to seasonality (e.g. no shipping on weekends or federal holidays), so heuristics developed to clean the data based on a small number of observations could fail

- From an analytical perspective, orders with faster shipping would be disproportionately overrepresented in the "tail" (most recent) data. If shipping category correlated with other characteristics like total order spend, this could create an artificial trend in the data

Once again, understanding that data is *collected* at point of shipment and reasoning how shipment timing varies and impacts *loading* is necessary for successful validation.

If this thought experiment seems to vague, we can make it more concrete by mocking up a dataset with which to experiment.

In the simplest version, we will simply suppose one order is submited on each of 10 days with dates (represented for convenience as integers and not calendar dates) given by the `dt_subm` vector. Suppose shipping always takes three days, so we can easily calculate the shipment date (`dt_ship`) based on the submission date. The shipment date is the same as the date the data will be logged and loaded (`dt_load`).

```
# data simulation: single orders + deterministic ship dates ----
dt_subm <- 1:10
days_to_ship <- 3
dt_ship <- dt_subm + days_to_ship
dt_load <- dt_ship
df <- data.frame(dt_subm, dt_ship, dt_load)
head(df)
```

```
##   dt_subm dt_ship dt_load
## 1       1       4       4
## 2       2       5       5
## 3       3       6       6
## 4       4       7       7
## 5       5       8       8
## 6       6       9       9
```

Suppose we are an analyst living in day 5 and wonder how many orders were submitted on day 3. We can observe all shipments loaded before day 5 so we filter our data accordingly. However, when we count how many records exist for day 3 we find none. Instead, when we move ahead to an analysis date of day 7, we are able to observe the orders submitted on day 3.

```
library(dplyr)

# how many day-3 orders do we observe as of day-5? ----
df %>%
  filter(dt_load <= 5) %>%
  filter(dt_subm == 3) %>%
  nrow()
```

```
## [1] 0
```

```
# how many day-3 orders do we observe as of day-7? ----
df %>%
  filter(dt_load <= 7) %>%
  filter(dt_subm == 3) %>%
  nrow()
```

```
## [1] 1
```

(Note that these conditions could be checked much more succinctly with a base R expression such as `sum(df$dt_load < 7 & df$dt_subm == 3)`. However, there is sometimes virtue in option for more readable code even if it is less compact. Here, we prefer the more verbose option for the claritfy of our exposition. Such trade-offs, and general thoughts on coding style, are explored further in Chapter 11.)

Now, this may seem to trivial. Clearly, if there were *zero* records for a day, we would catch this in data validation, right? We can make our synthetic data slightly more realistic to better illustrate the problem. Let's not imagine that there are 10 orders each day, and each order is shipped sometime between 2 and 4 days after the order with equal probability.

```
# data simulation: multiple orders + random ship dates ----
dt_subm <- rep(x = 1:10, each = 10)
days_to_ship <- sample(x = 2:4, size = length(dt_subm), replace = TRUE)
dt_ship <- dt_subm + days_to_ship
dt_load <- dt_ship
df <- data.frame(dt_subm, dt_ship, dt_load)
head(df)
```

```
##   dt_subm dt_ship dt_load
## 1       1       5       5
## 2       1       4       4
## 3       1       3       3
## 4       1       5       5
## 5       1       5       5
## 6       1       5       5
```

When we repeat the prior analysis, we now see that we have *some* records for orders submitted on day 3 by the time we begin analysis on day 5. In this case, we might be more easily tricked to believe this is *all* orders. However, when we repeat the analysis on day 7, we see the the number of orders on day 3 has increased.

Of course, you can imagine the real world is yet much more complicated than this example. In reality, we would have a random number of orders each day. Additionally, we might have a *mixture* of different *types* of orders. There might be high-priced orders where customers tended to be willing to pay for faster shipping, and low-priced orders where customers tend to chose slower shipping. In a case like this, not only might naive validation miss the lack of data completeness, but the *sample* of shipments we begin to see on day 5 could be unrepresentative of the population of orders placed on day 3. This is a type of **selection bias** that we will examine further in Chapter 6 (Incredible Inferences).

2.4 Data Encoding & Transformation (WIP)

Once data is loaded into a more suitable location for processing and analysis (such as a data warehouse), it often undergoes numerous transformations to change its shape, structure, and content to be more suited for analytical use.

For example, recall the `logins` table that we discussed above. It might be filtered to a cleaner versions which represents only a subset of events, or event identifiers like `'1436'` might be recoded to more human-readable names like `'mfa-success'`.

Unfortunately, although these steps attempt to increase the data's usability, they are also not immune to inserting bugs.

2.4.1 Data Encoding

2.4.1.1 Data Types

One critical set of decisions in data encoding is what sort of *data types* each field of interest should be. Data types such as integers, reals, character strings, logicals, dates, and times determine how data is stored and the types of manipulations that can be done to it.

We'll see more examples of the computational implications for our data types in Chapter 3 (Computational Quandaries). This chapter specifically explores the unique complexities of string and date types.

2.4.1.2 Indicator Variables (TODO)

what is positive case?

> "We had a bunch of zeros that should have been coded ones and the ones should have been coded zeroes."

(https://retractionwatch.com/2013/01/04/paper-on-evidence-for-environmental-racism-in-epa-polluter-fines-retracted-for-coding-error/)

2.4.1.3 General Representation (TODO)

> "These data sets often have multiple files that...have unclear and sometimes duplicative variables. Such complexities are commonplace among many data systems... I would not be surprised if coding errors were fairly common, and that the ones discovered constitute only the "tip of the iceberg." "

(https://retractionwatch.com/2015/09/10/divorce-study-felled-by-a-coding-error-gets-a-second-chance/)

2.4.1.4 The Many Meanings of Null

Another major encoding decision is how to handle null values. Previously, in the discussion of Data Collection, we considered the presence and absence of full *records*. However, when preparing data for analysis, both data produces and consumers need to decide how to cope with the presence or absence of individual *fields*.

If records contain some but not all relevant information, they may be published with explicitly missing fields or the full record may not be published at all. The difference between implicit and explicit missingness on the resulting data is illustrated in Figure 2.7.

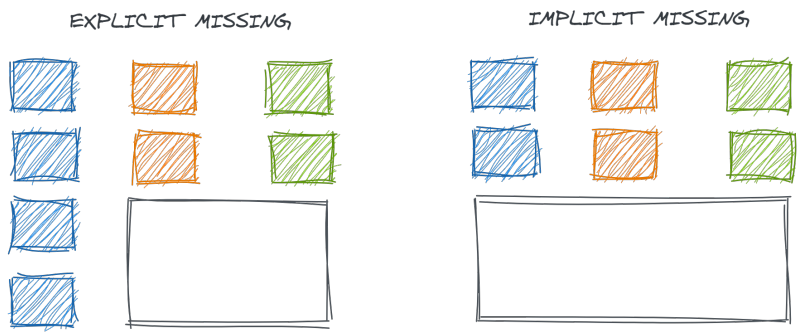


FIGURE 2.7: A comparison of explicit versus implicit missingness

Understanding what the system implies by each *explicitly* missing data field is also critical for validation and analysis. Checks for data completeness usually include counting null values, but null data isn't always incorrect. In fact, null data can be highly informative if we know what it means. Some meanings of null data might include:

- **Field is not relevant**: Perhaps our `logins` table reports the mobile phone operating system (iOS or Android) that was used to access the login page to track platform-specific issues. However, there is no valid value for this
- **Relevant value is not known**: Our `logins` table might also have an `account_id` field which attempts to match login attempts to known accounts/customers using different metadata like cookies or IP addresses. In theory, almost everyone trying to log in should have an account identifier, but our methods may not be good enough to identify them in all cases
- **Relevant value is null**: Of course, sometimes someone without an account at all might try to log in for some reason. In this case, the correct value for an `account_id` field truly *is* null
- **Relevant value was recorded incorrectly**: Sometimes systems have

glitches. Without a doubt, every single login attempt *should* have a timestamp, but such a field could be null if this data was somehow lost or corrupted at the source

Similarly, different systems might or might not report out these nulls in different ways such as:

- **True nulls**: Literally the entry in the resulting dataset is null
- **Null-like non-nulls**: Blank values like an empty string (`''`) that contain a null amount of information but won't be detected when counting null values
- **Placeholder values**: Meaningless values like an `account_id` of `00000000` for all unidentified accounts which preserve data *validity* (the expected structure) but have no intrinsic meaning
- **Sentinel/shadow values**: Abnormal values which attempt to indicate the reasons for null-ness such as an `account_id` of `-1` when no browser cookies were found or `-2` when cookies were found but did not help link to any specific customer record

Each of these encoding choices changes the definitions of appropriate completeness and validity for each field and, even more critically, impacts the expectations and assertions we should form for data accuracy. We can't expect 100% completeness if nulls are a relevant value; we can't check validity of ranges as easily if sentinel values are used with values that are outside the normal range (hopefully, or we have much bigger problems!) So, understanding how upstream systems *should* work is essential for assessing if they *do* work.

Similarly, understanding how our null data is collected has significant implications for how we subsequently process it. We will discuss this more in Chapter 3 (Computational Quandaries).

2.4.2 Data Transformation

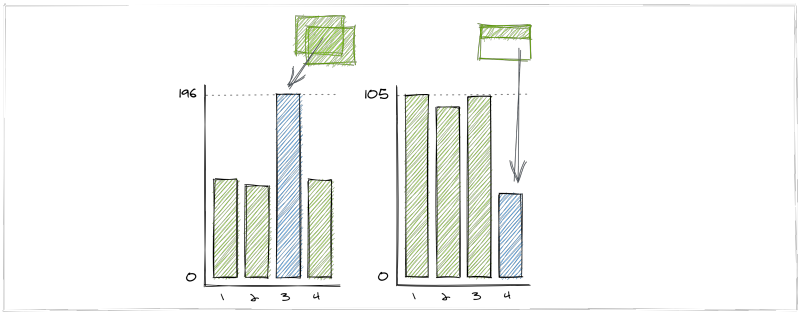
Finally, once the data is roughly where we want it, it likely undergoes many transformations to translate all of the system-generated fields we discussed in data collection into semantically-relevant dimensions for analytical consumers. Of course, the types of transformations that could be done are innumerable with far more variation than data loading. So, we'll just look at a few examples of common failure patterns.

2.4.2.1 Pre-Aggregation

Data transformations may include aggregating data up to higher levels of granularity for easier analysis. For example, a transformation might add up

item-level purchase data to make it easier for an analyst to look at spend per *order* of a specific user.

Data transformations not only transform our data, but they also transform how the dimensions of data quality manifest. If data with some of the **completeness** or **uniqueness** issues we discussed with data loading is pre-aggregated, these problems can turn into problems of **accuracy**. For example, the duplicate or partial data loads that we discussed when aggregated could suggest inaccurately high or low quantities respectively.



2.4.2.2 Field Encoding

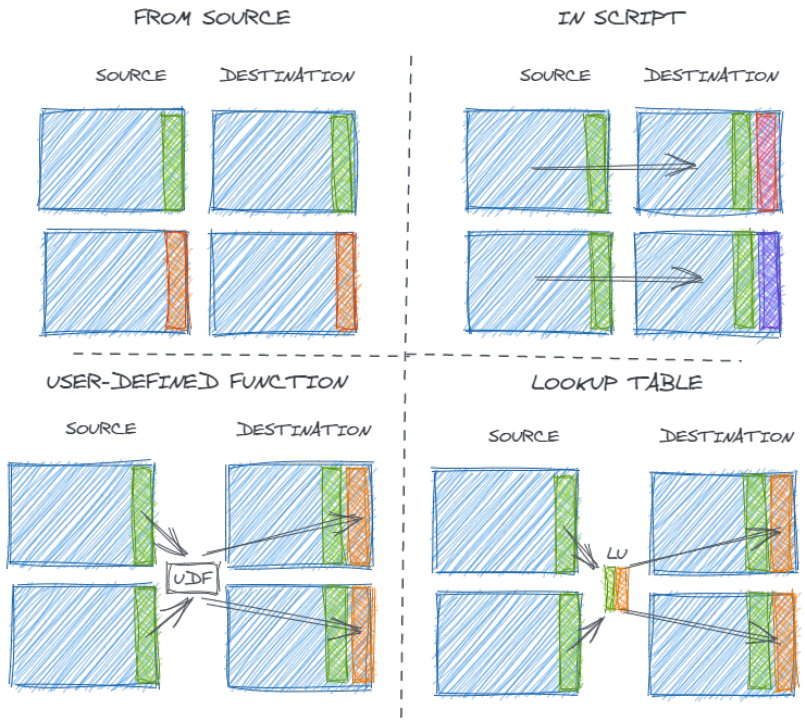
When we assess data consistency across tables,

Categorical fields in a data set might be created in any number of ways including:

- Directly taken from the source
- Coded in a transformation script
- Transformed with logic in a shared user-defined function (UDFs[13]) or macro[14]
- Joined from a shared look-up table

Each approach has different implications on data consistency and usability.

[13] https://docs.snowflake.com/en/sql-reference/user-defined-functions.html
[14] https://docs.getdbt.com/docs/building-a-dbt-project/jinja-macros/#macros



Using fields from the source simply is what it is – there's no subjectivity or room for manual human error. If multiple tables come from the same source, it's likely but not guaranteed that they will be encoded in the same way.

Coding transformations in the ELT process is easy for data producers. There's no need to coordinate across multiple processes or use cases, and the transformation can be immediately modified when needed. However, that same lack of coordination can lead to different results for fields that should be the same.

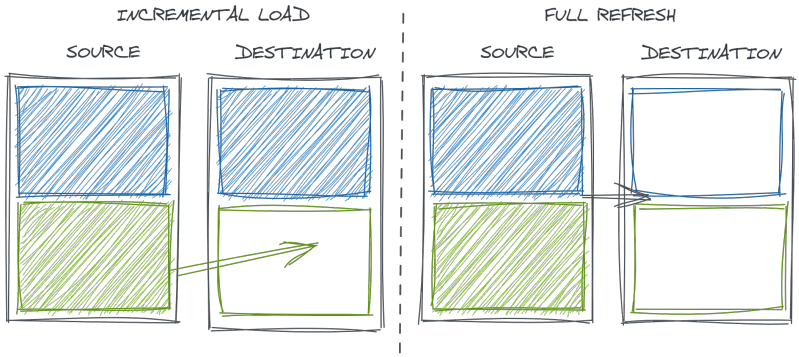
Alternatively, macros, UDFs, and look-up tables provided centralized ways to map source data inputs to desired analytical data outputs in a systemic and consistent way. Of course, centralization has its own challenges. If something in the source data changes, the process of updating a centralized UDF or look-up table may be slowed down by the need to seek consensus and collaborate. So, data is more *consistent* but potentially less *accurate*.

Regardless, such engineered values require scrutiny – paticularly if they are being used as a key to join multiple tables – and the distinct values in them should be carefully examined.

2.4.2.3 Updating Transformations

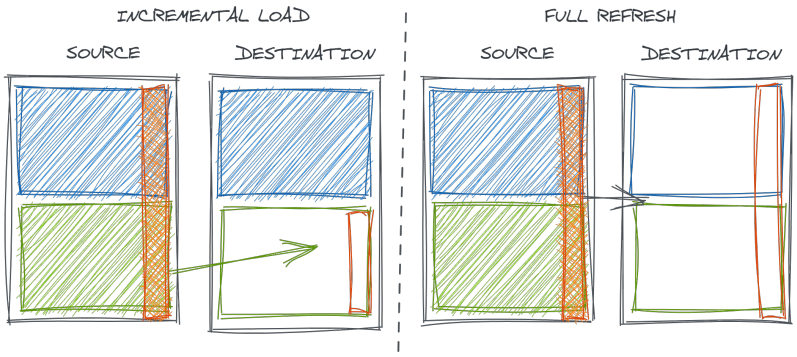
Of course, data consistency is not only a problem across different data sources but within one data source. Regardless of the method of field encoding used in the previous step, the intersection of data loading and data transformation strategies can introduce data consistency errors over time.

Often, for computation efficiency, analytical tables are loaded using an *incremental* loading strategy. This means that only new records (determined by time period, a set of unique keys, or other criteria) from the upstream source are loaded to the downstream table. This is in contrast to a *full refresh* where the entire downstream table is recreated on each update.

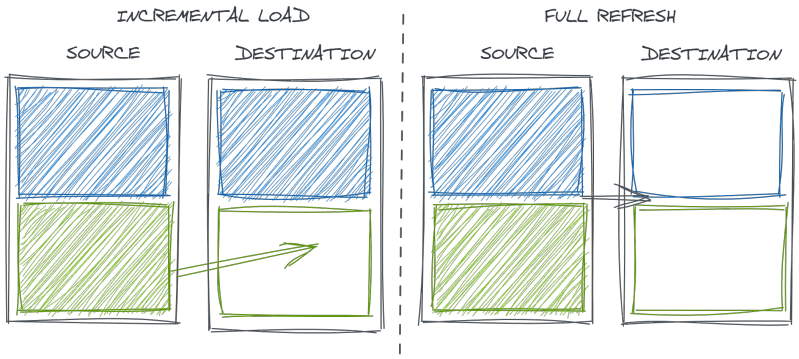


Incremental loads have many advantages. Rebuilding tables in entirety can be very time consuming and computationally expensive. In particular, in non-cloud data warehouses that are not able to scale computing power on demand, this sort of heavy duty processing job can noticeably drain resources from other queries that are trying to run in the database. Additionally, if the upstream staging data is ephemeral, fully rebuilding the table could mean failing to retain history.

However, in the case that our data transformations change, incremental loads may introduce inconsistency in our data overtime as only new records are created and inserted with the new logic.



This is also a problem more broadly if some short-term error is discovered either with data loading or transformation in historical data. Incremental strategies may not always update to include the corrected version of the data.



Regardless, this underscores the need to validate entire datasets and to re-validate when repulling data.

2.5 More on Missing Data (TODO)

MCAR / MAR / MNAR

2.6 Other Types of Data (TODO)

2.6.1 Survey Data

2.6.2 Human-Generated

2.7 Strategies (TODO)

2.7.1 Understand the intent

- Why was data originally collected? By whom and for what purpose?
- Are there clear, unique definitions of key concepts (e.g. entites, measures)? What are they?
- Documentation (metadata, dictionaries)

2.7.2 Understand the execution

- learn about the real-world systems
- understand key steps in data production process
- documentation (lineage, provenance)

2.7.3 Seek expertise

- talking to experts (upstream and downstream)

2.7.4 Trust but verify

- always on data validation
  - summaries
  - context-informed assertions
- exploratory analysis

2.8 Real World Disasters

2.8.1 Data loading artificially spikes COVID cases

Data loading, particularly when a new process is introduced can go astray and lead to spurious results. The city of El Paso discovered this after reporting an exceedingly high number of daily COVID cases (Borunda, 2020):

> El Paso city and public health officials on Thursday admitted to a major blunder, saying that the more than 3,100 new cases reported a day earlier was incorrect.

> The number was the result of a two-day upload of cases in one day after the public health department went from imputing data manually to an automatic data upload that was intended to increase efficiency, Public Health Director Angela Mora said at news conference.

This instance demonstrates not only how *data* goes wrong but also how easy it is to trust discrepancies and find explanations in real-world phenomena. The article goes on to suggest that the extremely high case numbers were not immediately caught given the overall "alarming" levels:

> Mora said she accepted responsibility and that she should have taken a deeper look after noticing the abnormally high number of new cases.El Paso County is still seeing more than 1,000 new cases on a daily basis, officials said. The correct number of new cases Wednesday was 1,537.

> "The numbers are still high and they are still alarming but they were not as high as we reported," Mora said.

Similar data artifacts also arose from a variety of other factors besides system changes. For example, delays in 9-to-5 office paperwork (the real-world events triggering the creation of some key data elements) over the holidays led to more artificial trends in data reporting around the end-of-year holidays:

> While 238 deaths is the most reported in Illinois since the start of the pandemic, the IDPH noted in its release that some data was, "delayed from the weekends, including this past holiday weekend."

- Illinois sees biggest spike in reported COVID-19 deaths to date after holidays delay some data, officials say[15], WGN9 (TODO CITATION)

2.8.2 Data encoding leads to incorrect BMI calculation

Data that is encoded inconsistently may be mishandled in automated decision processes. For example, data manually entered into electronic health records (EHR) by countless care provides might be compiled into a single source although the meanings could be inconsistent.

In the early days of the COVID19 vaccination push, some regions chose to prioritize their vaccine supply by certain individual characteristics including a high body mass index (BMI), which is a function of one's weight relative to their height.

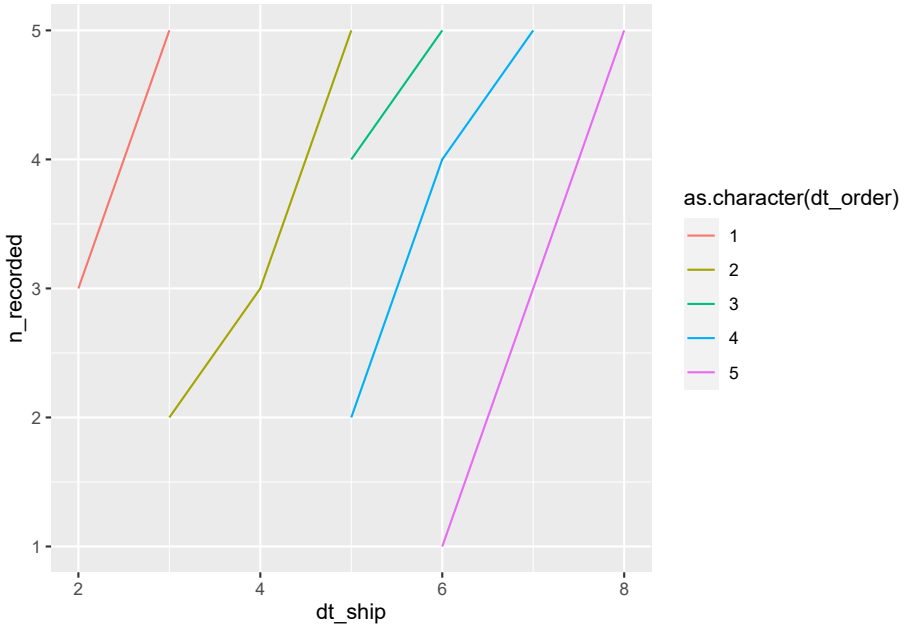
One 6ft 2in tall man's data was misinterpreted to suggest that he was 6.2 *centimeters* tall; this caused his to get high priority for a vaccine.

As described in the BBC (bbc, 2021):

[15] https://wgntv.com/news/coronavirus/illinois-sees-biggest-spike-in-reported-covid-19-deaths-to-date-after-officials-say-holidays-delayed-some-data/

> A man in his 30s with no underlying health conditions was offered a Covid vaccine after an NHS error mistakenly listed him as just 6.2cm in height. Liam Thorp was told he qualified for the jab because his measurements gave him a body mass index of 28,000.

```
# but in reality this is just a data loading/recording issue ----
df %>%
  group_by(dt_order, dt_ship) %>%
  count() %>%
  group_by(dt_order) %>%
  arrange(dt_ship) %>%
  mutate(n_recorded = cumsum(n)) %>%
  ggplot(aes(x = dt_ship, y = n_recorded, col = as.character(dt_order))) + geom_line()
```



2.8.3 Data Transformation

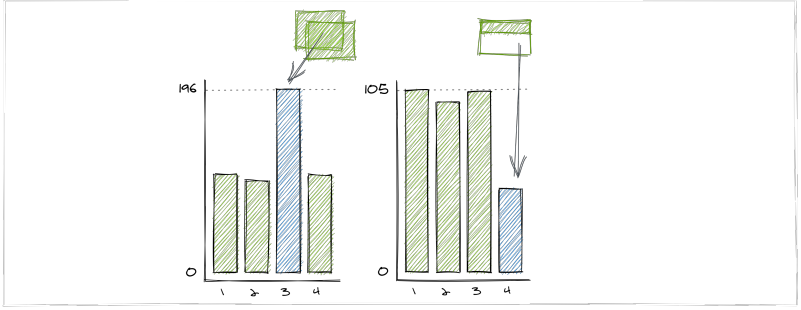
Finally, once the data is roughly where we want it, it likely undergoes many transformations to translate all of the system-generated fields we discussed in

data collection into semantically-relevant dimensions for analytical consumers. Of course, the types of transformations that could be done are innumerable with far more variation than data loading. So, we'll just look at a few examples of common failure patterns.

2.8.3.1 Pre-Aggregation

Data transformations may include aggregating data up to higher levels of granularity for easier analysis. For example, a transformation might add up item-level purchase data to make it easier for an analyst to look at spend per *order* of a specific user.

Data transformations not only transform our data, but they also transform how the dimensions of data quality manifest. If data with some of the **completeness** or **uniqueness** issues we discussed with data loading is pre-aggregated, these problems can turn into problems of **accuracy**. For example, the duplicate or partial data loads that we discussed when aggregated could suggest inaccurately high or low quantities respectively.



2.8.3.2 Field Encoding

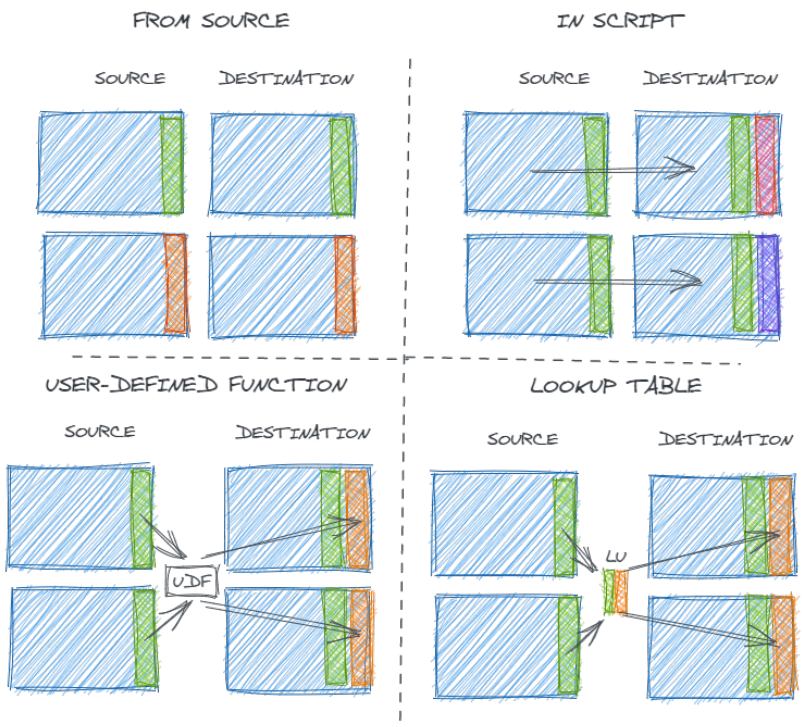
When we assess data consistency across tables,

Categorical fields in a data set might be created in any number of ways including:

- Directly taken from the source
- Coded in a transformation script
- Transformed with logic in a shared user-defined function (UDFs[16]) or macro[17]
- Joined from a shared look-up table

[16] https://docs.snowflake.com/en/sql-reference/user-defined-functions.html
[17] https://docs.getdbt.com/docs/building-a-dbt-project/jinja-macros/#macros

Each approach has different implications on data consistency and usability.



Using fields from the source simply is what it is – there's no subjectivity or room for manual human error. If multiple tables come from the same source, it's likely but not guaranteed that they will be encoded in the same way.

Coding transformations in the ELT process is easy for data producers. There's no need to coordinate across multiple processes or use cases, and the transformation can be immediately modified when needed. However, that same lack of coordination can lead to different results for fields that should be the same.

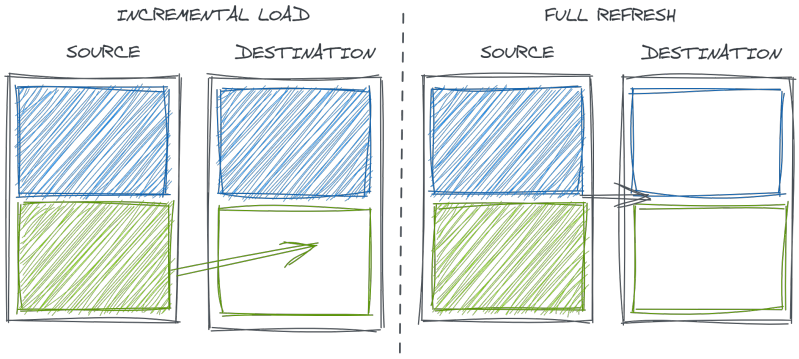
Alternatively, macros, UDFs, and look-up tables provided centralized ways to map source data inputs to desired analytical data outputs in a systemic and consistent way. Of course, centralization has its own challenges. If something in the source data changes, the process of updating a centralized UDF or look-up table may be slowed down by the need to seek consensus and collaborate. So, data is more *consistent* but potentially less *accurate*.

Regardless, such engineered values require scrutiny – paticularly if they are being used as a key to join multiple tables – and the distinct values in them should be carefully examined.

2.8.3.3 Updating Transformations

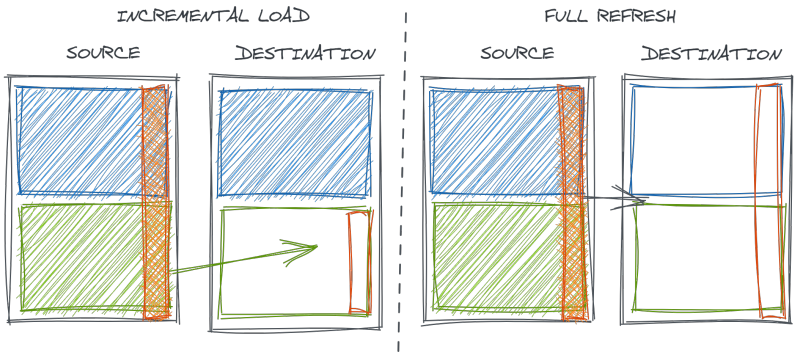
Of course, data consistency is not only a problem across different data sources but within one data source. Regardless of the method of field encoding used in the previous step, the intersection of data loading and data transformation strategies can introduce data consistency errors over time.

Often, for computation efficiency, analytical tables are loaded using an *incremental* loading strategy. This means that only new records (determined by time period, a set of unique keys, or other criteria) from the upstream source are loaded to the downstream table. This is in contrast to a *full refresh* where the entire downstream table is recreated on each update.

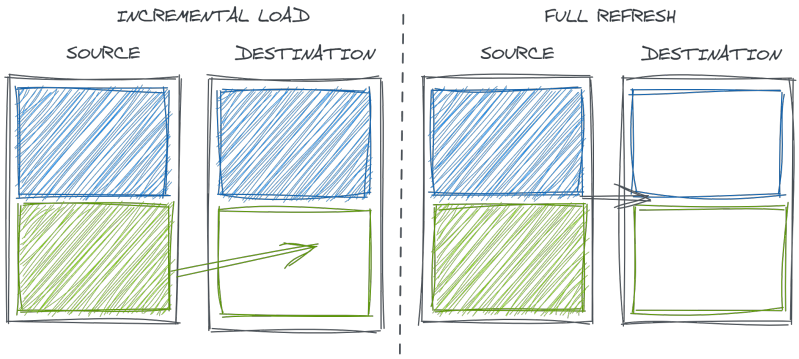


Incremental loads have many advantages. Rebuilding tables in entirety can be very time consuming and computationally expensive. In particular, in non-cloud data warehouses that are not able to scale computing power on demand, this sort of heavy duty processing job can noticeably drain resources from other queries that are trying to run in the database. Additionally, if the upstream staging data is ephemeral, fully rebuilding the table could mean failing to retain history.

However, in the case that our data transformations change, incremental loads may introduce inconsistency in our data overtime as only new records are created and inserted with the new logic.



This is also a problem more broadly if some short-term error is discovered either with data loading or transformation in historical data. Incremental strategies may not always update to include the corrected version of the data.



Regardless, this underscores the need to validate entire datasets and to re-validate when repulling data.

2.9 Human-Generated Data

Ahhhh

2.10 Other Encoding Issues

- for indicators which is 1?
- field values changing over time

2.11 Strategies

–>

3

Computational Quandaries (WIP)

After gaining confidence in one's data (or, at least, making peace with it), the next step in a data analysis is often to start cleaning and exploring that data with summary statistics, plots, and models. Generally, this requires a computational tool like SQL, R, or python.

The process of computation itself can be fraught with challenges. Computational tools are extremely literal; they are excellent at doing *precisely what they were told to do* but not often what analysts might have *meant* or *wished* that they would do. Additionally, the moment an analyst begins to use a tool, the conversation is no longer between them and the data; suddenly, the mental model of how every single tool developer thought you might want to do analysis affects the tools' behaviors and the analysts' results.

In this chapter, we will explore common ways that tools may do something technically correct, reasonable, and as-intended but very much not what analysts may expect. Along the way, we will see how computational methods interact with the data encoding choices we discussed in Chapter 2 (Data Dalliances).

3.1 Preliminaries - Data Computation

Before we think about specific tools or failure modes, we can first consider the common types of operations that the analytical tools allow us to do with our data.

3.1.1 Single Table Operations

Given a single data table, we may wish to do operations (illustrated in Figure 3.1) such as:

- **Filtering**: Extracting a subset of a dataset for analysis based on certain inclusion criteria for each record

- **Aggregation**: Grouping our data table by one or more variables and condensing information across records with *aggregate functions* like counts, sums, and averages
- **Transformation**: Create new columns or modifying existing columns to represent more complex or domain-specific context

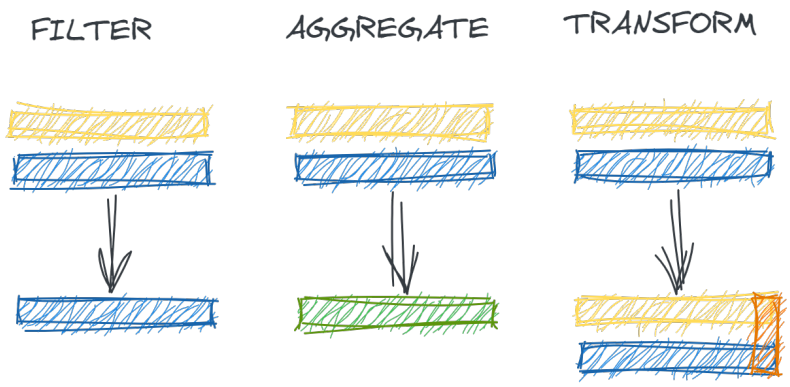


FIGURE 3.1: Illustration of basic single-table data wrangling operations

3.1.2 Multiple Table Operations

Often, we can get additional value in an analysis by combining multiple types of information from difference tables. When working with multiple tables, we may be interested in:

- **Combining Row-wise**: Taking multiple tables with the same schemas (column names and data types) and creating a single table which contains the union (all records), intersection (only matching), or difference (only in one) of the records in the two tables
- **Combining Column-wise**: Appending additional fields to existing records through joining (also known as merging) multiple tables

3.1.3 Mechanics

All of these operations rely on a few core computational tasks:

- **Arithmetic**: Basic addition, subtraction, multiplication, and division to aggregate and transform data

- **Equality**: Comparing whether or not two values are equal is critical for data filtering, column-wise combination, and certain types of data transformation
- **Casting**: Converting data types of different elements into a comparable format is necessary for row-wise combination and often a prerequisite to certain equality and arithmetic tasks

While these operations may seem simple, their behavior within certain tools and when employed for certain data types may sometimes lead to unintuitive or misleading results.

3.2 Null Values

In Chapter 2 (Data Dalliances), we discuss how null values may represent many different concepts and be encoded in multiple different ways. In addition to those semantic challenges, various representations of null values may cause different computational problems.[1] In this section, we will explore these potential failure modes.

3.2.1 Types of Null Values

Not only can null values represent many different things (as explored in Chapter 2), they also may be represented in many different ways. Understanding how nulls are encoded in one's dataset is a critical prerequisite to attempting any of the computations described in the subsequent sections.

3.2.1.1 Language representations

Different programming languages each offer their own versions of null values – and sometimes more than one. For example, the R language includes `NA`, typed `NA`s (e.g. `NA_integer_`, `NA_character_`), `NaN`, and `NULL`; meanwhile, core python has `None` and the `numpy` module provides a `nan`.

These different values carry different semantic and functional meanings. For example R's `NA` generally means "the presence of an absence" whereas `NULL` is "the absence of a presence". This is articulated more clearly if we examine

[1] This problem is not isolated to data analysis tools. For an entertaining example, see the 2019 WIRED article "How a 'NULL' License Plate Landed One Hacker in Ticket Hell" (Barrett, 2019) which a real-world software system producing unintended and undesirable behavior when asked to deal with a word `'NULL'`.

the lengths of these objects and observe that `NA` has a length 1 whereas `NULL` has a length 0.

```
length(NA)
length(NULL)
```

```
## [1] 1
## [1] 0
```

As further proof that these are not interchangeable, we may use the helper functions `is.na()` and `is.null()`. It's false that `NA` is `NULL` and essentially unable to be evaluated if `NULL` is `NA` because `NULL`s are truly nothing.[2]

```
is.na(NA)
is.null(NULL)
is.na(NULL)
is.null(NA)
```

```
## [1] TRUE
## [1] TRUE
## logical(0)
## [1] FALSE
```

To further complicate matters, we have `NaN` ("not a number"), along with `-Inf` and `Inf`, which generally arise when we attempt to abuse R's calculator. Somewhat charmingly, `Inf` and `-Inf` may be used in some rudimentary calculations where the limit is returned.[3]

```
1/0
0/0
1/Inf
```

```
## [1] Inf
## [1] NaN
## [1] 0
```

[2] You'll notice that the code chunk below contains four commands but only three boolean outputs. Why is that? `is.na(NULL)` returns `logical(0)`, a zero-length value. It's yet another form of missingness!

[3] From calculus, we know 1/Inf approaches 0, but Inf/Inf is undefined.

3.2.1.2 Sentinel value encoding

Beyond these null types offered natively by different programming languages, there are also many different data management *conventions* for null values. Because null values can have many meanings, sometimes missing fields are encoded with "out of range" values which intend to suggest a type of missingness.

For example, the US Census Bureau's Medical Expenditure Panel Survey uses the following reserved codes to denote different types of missingness: (TODO: cite p10 https://www.meps.ahrq.gov/data_stats/download_data/pufs/h206a/h206adoc.pdf)

```
- -1 INAPPLICABLE Question was not asked due to skip pattern
- -7 REFUSED Question was asked and respondent refused to answer question
- -8 DK Question was asked and respondent did not know answer
- -14 NOT YET TAKEN/USED Respondent answered that the medicine has not yet been used
- -15 CANNOT BE COMPUTED Value cannot be derived from data
```

This approach preserves a lot of relevant information while, at the same time, being readily apparent that these values are not valid when the data is manually inspect. Unfortunately, manually inspecting every data field is rarely possible, and such sentinel values may go undetected when looking at higher-level summaries.

Consider a survey of a population of retired adults where age is coded as `999` if not provided. Below, we simulate 100,000 such observations that are uniformly distributed between the age of 65 and 95 (hence, have an expected value of 80). Next, we replace merely *half of a percent* with our "null" values of `999`. Taking the mean with these false values results in a mean of about 85. This number alone might not raise the alarm; after all, we know the dataset's population is older adults. However, accidentally treating these as valid values biases our results by a somewhat remarkable five years.

```
set.seed(123)

n <- 100000
p <- 0.01 / 2
ages <- runif(n, 65, 95)

ages_nulls <- ages
ages_nulls[1:(n*p)] <- 999

mean(ages)
mean(ages_nulls)
```

```
## [1] 79.98
## [1] 84.57
```

So, the first order of business with null values is understanding how they are encoded and translating them to the most computationally appropriate form. However, that is only the beginning of the story.

3.2.1.3 Other types of nulls (TODO)

might be blank string which won't get detected in standard null checks

```
x <- ""
is.na(x)
```

```
## [1] FALSE
```

same thing is true in dataframes

```
library(dplyr)
df <- data.frame(x = c("a", "b", "", "d"))
summarize_all(df, ~sum(is.na(.)))
```

```
##   x
## 1 0
```

in such cases, need to explicitly check for such alternative encodings like blank strings

```
x <- ""
is.na(x) | x == ""
```

```
## [1] TRUE
```

of course, a blank string isn't the only choice
could also have an empty string of any length

```
x <- "  "
is.na(x) | x == ""
is.na(x) | trimws(x) == ""
is.na(x) | nchar(trimws(x)) == 0
```

```
## [1] FALSE
## [1] TRUE
## [1] TRUE
```

we'll see more about how fluid strings can be in the string section below

3.2.2 Aggregation

Once null values are coded as "true" nulls, how these nulls are handled in the simple aggregation of data varies both across different languages and across different functions within a language. To better understand the problems this might cause, we will look at examples in R and SQL.

To explore aggregation, let's build a simple dataset. We will suppose that we are working with a subscription-based e-commerce service and that we are looking at a `spend` dataset with one record per customer and information about the amount they spent and returned in a given month:

```
spend <-
  data.frame(
    AMT_SPEND = c(10, 20, NA),
    AMT_RETURN = rep(NA, 3)
  )

head(spend)
```

```
##   AMT_SPEND AMT_RETURN
## 1        10         NA
## 2        20         NA
## 3        NA         NA
```

To compute the average amount spent (`AMT_SPEND`) with the `dplyr` package, an analyst might first reasonably write the following `summarize()` statement. However, as we can see, due to the presence of null values within the `AMT_SPEND` column, the result of this aggregation is for the whole quantity of `AVG_SPEND` to be set to the value `NA`.

A glance at the documentation for the `mean()` function[4] reveals that the function has a parameter called `na.rm`. This parameter's default value is `FALSE`, but, when it is set to `TRUE`, it removes null values from our dataset. Adding this argument to the previous statement allows us to reach a numerical answer.

```
summarize(spend,
          AVG_SPEND = mean(AMT_SPEND),
          AVG_SPEND_NARM = mean(AMT_SPEND, na.rm = TRUE))
```

```
##   AVG_SPEND AVG_SPEND_NARM
## 1        NA             15
```

However, is this the *right* numerical answer? Recall that what `na.rm = TRUE` does is *drop* the null values from the set of numbers being averaged. However, suppose the null values represent that no purchases were made for a given customer in a given month. That is, zero dollars were spent. In effect, we have removed all non-purchasers from the data being averaged.

More precisely, we have switched from taking the average

$$\frac{\sum_1^n Spend}{\sum_1^n 1}$$

over all n customers

to taking the average

$$\frac{\sum_{Spend>0} Spend}{\sum_{Spend>0} 1}$$

over only those customers with spend.

At face value, we could say that the code above is giving the incorrect answer; by dropping some low (zero) purchase amounts, the average amount spend per customer is inflated. A second perspective, which is someone more philosophically troubling, is that this tiny change to the code which fixed the *obvious* problem (returning a null value) has introduced a *non-obvious* problem by fundamentally changing the question that we are asking. By dropping all accounts from our table who made no purchases, we are no longer answering "What is the average amount spent by all of our customer?" but rather "What is the average amount spent by an actively engaged customer?" In the language of probability theory, we might say that we have then changed our

[4] This documentation can be obtained in R by typing `?mean` in the console.

estimand from the expected value of spend to expected value of spend conditional on spend being positive. This technical quirk has significant analytical impact.

To answer the real question at hand, we have a couple of options. We could manually `sum()` the amount spent with the option to drop nulls but then divide by the correct denominator (all observations – not just those with spend) or we could explicitly recode null values in `AMT_SPEND` to zero before taking the average.[5] Either of these options lead to the correct conclusion of a lower average spend amount.

```
summarize(
    spend,
    AVG_SPEND_MANUAL = sum(AMT_SPEND, na.rm = TRUE) / n(),
    AVG_SPEND_RECODE = mean(coalesce(AMT_SPEND, 0))
  )
```

```
##   AVG_SPEND_MANUAL AVG_SPEND_RECODE
## 1               10               10
```

This is all well and good if we could just accept that the behaviors above are simply how nulls work, but further complexity comes as we see that there is no industry standard across tools. For example, as the SQL code below shows, SQL's `avg()` function behaves more like R's `mean()` *with* the `na.rm = TRUE` option set (whereas, you may recall that R's `mean()` behaves with `na.rm = FALSE` by default). That is, the default behavior of SQL is to only operate on the valid and available values. The result of this default may mean that it is less obvious when our dataset has null values. SQL, unlike R, does not ask for "permission" to drop out nulls; instead, it unilaterally makes a decision how to handle these variables.

```
SELECT
  avg(amt_spend) as AVG_SPEND
FROM spend
```

```
##   AVG_SPEND
## 1        15
```

[5] Recoding can be done with a number of different general purpose functions like `ifelse` or `dplyr::case_when` in R. Different SQL varaints often offer different options for this purpose with functions such as `nvl()` or `zeroifnull()`. A common version across many language is `coalesce()` which takes the first non-null argument listed.

However, this is not to suggest that null values cannot also be "destructive" in SQL (that is, returning null). While aggregation functions (which compute over the *rows/records*) like `sum()` and `avg()` drop nulls, operators like `+` and `-` (which compute *across columns/variables* in the *same row/record*) do not exhibit the same behavior. Consider, for example, if we wish to calculate the average net purchase amount (purchases minus returns) instead of the gross (total) purchase amount.

```
SELECT
  avg(amt_spend - amt_return) as AVG_SPEND_NET
FROM spend
```

```
##   AVG_SPEND_NET
## 1            NA
```

Despite what we learned above about SQL's `avg()` function, the query above returns only a null value. What has happened? In our `spend` dataset, the `amt_return` column is completely null (representing no return purchases). Because the subtraction occurs before the average is taken, subtracting null values in the `amt_return` variable from valid numbers in `amt_spend` variable creates a new variable of all null values. This new variable, which is already all null, is passed to the `avg()` function. This process is shown step-by-step below.

```
SELECT
  amt_spend,
  amt_return,
  amt_spend-amt_return
FROM spend
```

```
##   AMT_SPEND AMT_RETURN amt_spend-amt_return
## 1        10         NA                   NA
## 2        20         NA                   NA
## 3        NA         NA                   NA
```

3.2.3 Comparison

Null values don't just introduce complexity when doing arithmetic. Difficulties also arise any time multiple variables are assessed for equality or inequality. Since a null value is unknown, most programming languages generally will *not* consider nulls to be comparable with other nulls.

We can observe simple examples of this in both R and SQL. In neither language can a null value be assessed for equality or inequality versus either another number or another null.

```
NA == 3
NA > 10
NA == NA
```

```
## [1] NA
## [1] NA
## [1] NA
```

```
SELECT
  (NULL = 3) as NULL_EQ_NUM,
  (NULL > 10) as NULL_GT_NUM,
  (NULL = NULL) as NULL_EQ_NULL
```

```
##   NULL_EQ_NUM NULL_GT_NUM NULL_EQ_NULL
## 1          NA          NA           NA
```

In these toy examples, such outcomes may seem perfectly logical. However, this same reasoning can arise in sneakier ways and lead to uninteded results when equality evaluations are *implicit* in the task at hand instead of the singular focus. We'll now see examples from data filtering, joining, and transformation.

If you're familiar with SQL, you may have been surprised to notice that there is no `FROM` clause in the query above. In fact, SQL queries can treat values just like variables containing only a single record.

We will use this trick throughout the chapter for exploring how SQL works when we don't have an ideal sample dataset to test certain scenarios. Beyond exposition in this book, this trick is also useful in practice.

3.2.3.1 Filtering

Suppose we want to split our dataset into two datasets based on high or low values of spend. We might assume the following two lines of code will create a clear partition (implying that each record would fall into exactly one group.)

```
spend_lt20 <- filter(spend, AMT_SPEND < 20)
spend_gte20 <- filter(spend, AMT_SPEND >= 20)
```

However, if we examine the resulting datasets, we see that *neither* contains the third record which had a null value for the `AMT_SPEND` variable.

```
spend_lt20
```

```
##   AMT_SPEND AMT_RETURN
## 1        10         NA
```

```
spend_gte20
```

```
##   AMT_SPEND AMT_RETURN
## 1        20         NA
```

The same situation results in SQL.

```
SELECT *
FROM spend
WHERE AMT_SPEND < 20
```

```
##   AMT_SPEND AMT_RETURN
## 1        10         NA
```

```
SELECT *
FROM spend
WHERE AMT_SPEND >= 20
```

```
##   AMT_SPEND AMT_RETURN
## 1        20         NA
```

This is a direct result of the fact that nulls cannot be compared for equality any inequality. We can think of data filtering as a two-step process: first evaluate whether the condition is TRUE or FALSE, then return only the records for which the condition holds true. When we conduct the more manual process of filtering step-by-step, we see that the null value of `AMT_SPEND` does not get a "truth value" when compared with a number. Thus, it is not contained in either "truth value" subset.

```
mutate(spend, is_lt20 = (AMT_SPEND < 20))
```

```
##   AMT_SPEND AMT_RETURN is_lt20
## 1        10         NA    TRUE
## 2        20         NA   FALSE
## 3        NA         NA      NA
```

Thus, whenever our data has null values, the very common act of data filtering risks excluding important information.

3.2.3.2 Joining

The same phenomenon as described above also happens when joining multiple datasets.

Suppose we have multiple datasets we wish to merge based on columns denoting a record's name and date of birthday. For ease of exploration, we will make the simplest possible such dataset and simply try to merge it to itself. (This may seem silly, but often when trying to understand *computationally* complex things, it is a good idea to make the scenario as simple as possible. In fact, this idea is core to the concept of computational unit tests which we will discuss at the end of this chapter.)

```
bday <- data.frame(NAME = c('Anne', 'Bob'), BIRTHDAY = c('2000-01-01', NA))
bday
```

```
##   NAME   BIRTHDAY
## 1 Anne 2000-01-01
## 2  Bob       <NA>
```

In SQL, if we try to join this table, the records in row 1 will match because `'Anne' == 'Anne'` and `'2000-01-01' == '2000-01-01'`. However, poor Bob's record is eliminated because his birth date is logged as null, and `NA == NA` is false.

```
SELECT a.*
FROM
  bday as a
  INNER JOIN
  bday as b
  ON
  a.NAME = b.NAME and
  a.BIRTHDAY = b.BIRTHDAY
```

```
##   NAME   BIRTHDAY
## 1 Anne 2000-01-01
```

In contrast, R's `dplyr::inner_join()` function will not do this by default. This function lets us specifically control how nulls are matches with the `na_matches` argument, with a default option to match on `NA` values. (You may read more about the argument by typing `?dplyr::inner_join` in the R console to pull up the documentation.)

```
inner_join(bday, bday, by = c('NAME', 'BIRTHDAY'))
```

```
##   NAME   BIRTHDAY
## 1 Anne 2000-01-01
## 2  Bob       <NA>
```

This example then is not only a cautionary tale for how null values may unintentionally corrupt our data transformations but also how "brittle" our knowledge and intuition may be when moving between tools. Neither of these default behaviors is strictly better or worse, but they are definitely different and have real implications on our analysis.

3.2.3.3 Transformation

A common task in data analysis is to aggregate results by subgroup. For example, we might want to summarize how many customers (rows/records) spent more or less than $10. To discern this, we might create a categorical variable for high versus low purchase amounts, group by this variable and count.

The psuedocode would read something like this:

```
data %>%
  mutate(HIGH_LOW = << transform AMT_SPEND >>) %>%
  group_by(HIGH_LOW) %>%
  count()
```

To define the `HIGH_LOW` variable, we might use a function like `ifelse()`, `dplyr::if_else()`, or `dplyr::case_when()`. However, once again, we have the issue of how values are *partitioned* when nulls are included. If we recode any records with `AMT_SPEND` of less than or equal to 10 to "Low" and default the rest to "High", we will accidentally count all null values in the "High" group.

```
spend %>%
  mutate(HIGH_LOW = case_when(
    AMT_SPEND <= 10 ~ "Low",
    TRUE ~ "High")
    ) %>%
  group_by(HIGH_LOW) %>%
  count()
```

```
## # A tibble: 2 x 2
## # Groups:   HIGH_LOW [2]
##   HIGH_LOW     n
##   <chr>    <int>
## 1 High         2
## 2 Low          1
```

Instead, it is more accurate and transparent (unless we know specifically what null values mean and what group they should be part of) to not let one of our "core" categories by the "default" case in our logic. We can explicitly encode any residual values as something like "OTHER" or "ERROR" to help us see that there is a problem requiring extra attention.

```
spend %>%
  mutate(HIGH_LOW = case_when(
    AMT_SPEND <= 10 ~ "Low",
    AMT_SPEND > 10 ~ "High",
    TRUE ~ "OTHER")
    ) %>%
  group_by(HIGH_LOW) %>%
  count()
```

```
## # A tibble: 3 x 2
## # Groups:   HIGH_LOW [3]
##   HIGH_LOW     n
##   <chr>    <int>
## 1 High         1
## 2 Low          1
## 3 OTHER        1
```

While nulls contribute to this issue, it's important to realize that nulls are not the only factor causing this error nor or they the solution. The more substantial issue is the careless use of defaults and *implicit* encoding versus *explicit* encoding. In the second form of the SQL query above, we are more specific about exactly what is allowed in each category which ensures any unexpected inputs will not be allowed to "sneak" into ordinary outputs.

3.3 Logicals (TODO)

Much like the different versions of nulls that we met in the last section, logical data types use reserved keywords to represent `TRUE` and `FALSE`. (The exact formats of logical reserved keywords vary by language. R and SQL use `TRUE` and `FALSE` and python uses `TRUE` and `FALSE`.) This means that these names function like a number or a letter which intrinsically hold one specific value and cannot take on a different value besides their own.

```
TRUE = 5
```

```
## Error in TRUE = 5: invalid (do_set) left-hand side to assignment
```

```
2 = 5
```

```
## Error in 2 = 5: invalid (do_set) left-hand side to assignment
```

Across languages, `TRUE` and `FALSE` are considered equivalent to the numeric representations of 1 and 0 respectively.

TABLE 3.1: 1 records

a	b	c	d	e
1	1	1	2	5

```
as.numeric(TRUE)
as.numeric(FALSE)
```

```
## [1] 1
## [1] 0
```

A consequence of this numerical equivalency is that `TRUE` and `FALSE` may be compared to each other or other numbers and be included in mathematical expressions.

```
TRUE > FALSE
TRUE < 5
FALSE > -1
TRUE + 1
TRUE * 5
```

```
## [1] TRUE
## [1] TRUE
## [1] TRUE
## [1] 2
## [1] 5
```

similar in SQL

```
select
  TRUE > FALSE as a,
  TRUE < 5 as b,
  FALSE > -1 as c,
  TRUE + 1 as d,
  TRUE * 5 as e
```

3.3.1 Language-specific nuances (CUT?)

3.3.1.1 Keyword abbrevations

R also recognizes the abbreviations of `T` and `F` as `TRUE` and `FALSE` respectively; however this is not recommended. `T` and `F` are *not* reserved keywords, so they can be overwritten with a different value. This makes code using such abbreviations "brittle" and less reliable.

```
if (T) {"Hello"} else {"Goodbye"}
if (F) {"Hello"} else {"Goodbye"}
T = 0
if (T) {"Hello"} else {"Goodbye"}
```

```
## [1] "Hello"
## [1] "Goodbye"
## [1] "Goodbye"
```

3.3.1.2 Alternative representations

We've previously seen that logicals have associated numerical values. Different languages may also treat their string representations differently.

For example, R believes that the string `"TRUE"` is equal to the logical value `TRUE` when directly compared. However, this relationships breaks the transitive property of mathematics[6] because `TRUE` equals both `1` and `"TRUE"`, yet `"TRUE"` does not equal `1` so the mathematical operations that can be done with logical `TRUE` cannot be done with string `"TRUE"`.

```
TRUE == 1
TRUE == "TRUE"
TRUE == "True"
TRUE * 5
"TRUE" * 5
```

```
## Error in "TRUE" * 5: non-numeric argument to binary operator
```

```
## [1] TRUE
## [1] TRUE
## [1] FALSE
## [1] 5
```

[6] If a = b and b = c then a = c

In contrast neither SQL not python honor the string forms of their respective logical reserved keywords at all.

```
select
  (TRUE == 1) as is_int_true,
  (TRUE == 'TRUE') as is_char_true,
  TRUE*5 as true_times_five,
  'TRUE'*5 as true_str_time_five
```

```
##   is_int_true is_char_true true_times_five
## 1           1            0               5
##   true_str_time_five
## 1                  0
```

```
True == 1
True == "True"
True == "TRUE"
True * 5
"True" * 5
```

```
## True
## False
## False
## 5
## 'TrueTrueTrueTrueTrue'
```

3.3.2 Comparison (TODO)

The nuances of logical representation and handling seem straightforward in isolation. However, when encountered in real-world data problems, they are not isolated and are unlikely to be our main focus.

Imagine two datasets which all encode the same information but use boolean, string, and integer representations of a logical respectively.

```
df1 <- data.frame(ID = 1:3, IND = rep(TRUE, 3), X = 1:3)
df2 <- data.frame(ID = 1:5, IND = rep('TRUE', 5), Y = 11:15)
df3 <- data.frame(ID = 1:5, IND = rep(1, 5), Z = 21:25)
```

By simple inspection, the logical and string representations in particular look superficially similar and yet they will behave differently.

```
head(df1)
```

```
##   ID  IND X
## 1  1 TRUE 1
## 2  2 TRUE 2
## 3  3 TRUE 3
```

```
head(df2)
```

```
##   ID  IND  Y
## 1  1 TRUE 11
## 2  2 TRUE 12
## 3  3 TRUE 13
## 4  4 TRUE 14
## 5  5 TRUE 15
```

If we use R's `dplyr::filter()` or `base::subset()` function to subset the data, the value of `df1` will correctly subset based on the boolean values of `IND`. However, R will not know how to interpret the string version in `df2`.

```
filter(df1, IND)
filter(df2, IND)
```

```
## Error: Problem with `filter()` input `..1`.
## x Input `..1` must be a logical vector, not a character.
## i Input `..1` is `IND`.
```

```
subset(df2, df2$IND)
```

```
## Error in subset.data.frame(df2, df2$IND): 'subset' must be logical
```

```
##   ID  IND X
## 1  1 TRUE 1
## 2  2 TRUE 2
## 3  3 TRUE 3
```

```
filter(df1, isTRUE(IND))
filter(df2, isTRUE(IND))
```

```
## [1] ID  IND X
## <0 rows> (or 0-length row.names)
## [1] ID  IND Y
## <0 rows> (or 0-length row.names)
```

```
filter(df2, isTRUE(IND))
```

```
## [1] ID  IND Y
## <0 rows> (or 0-length row.names)
```

```
left_join(df1, df2, by = c("ID", "IND"))
```

```
## Error: Can't join on `x$IND` x `y$IND` because of incompatible types.
## i `x$IND` is of type <logical>>.
## i `y$IND` is of type <character>>.
```

```
merge(df1, df2, by.x = c("ID", "IND"), by.y = c("ID", "IND"), all.x = TRUE)
```

```
##   ID  IND X  Y
## 1  1 TRUE 1 11
## 2  2 TRUE 2 12
## 3  3 TRUE 3 13
```

```
select df1.*, df2.*
from
 df1
 left join
 df2
 on
 df1.id = df2.id and
 df1.ind = df2.ind
```

TABLE 3.2: 3 records

ID	IND	X	ID	IND	Y
1	1	1	NA	NA	NA
2	1	2	NA	NA	NA
3	1	3	NA	NA	NA

```
import pandas as pd

data1 = {'ID': list(range(1, 4)),
        'IND': [True] * 3,
        'X': list(range(1,4))
        }
data2 = {'ID': list(range(1, 6)),
         'IND': ['True'] * 5,
        'Y': list(range(11, 16))}

df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

df1
df2
```

```
##    ID   IND  X
## 0   1  True  1
## 1   2  True  2
## 2   3  True  3
##    ID   IND   Y
## 0   1  True  11
## 1   2  True  12
## 2   3  True  13
## 3   4  True  14
## 4   5  True  15
```

```
pd.merge(left = df1, right = df2, how = "left", on = ['ID', 'IND'])
```

```
##    ID   IND  X   Y
## 0   1  True  1 NaN
## 1   2  True  2 NaN
## 2   3  True  3 NaN
```

3.4 Numbers (TODO)

3.4.1 Integer division

In R (mostly what we'd expect)

```
1/2
```

```
## [1] 0.5
```

```
ifelse(3/4 > 0.5, 'High', 'Low')
```

```
## [1] "High"
```

In some SQL dialect SQL (SQLite shown. "Modern" interfaces like Snowflake and BigQuery don't do this)

```
SELECT (1/2) as one_div_two
```

```
##   one_div_two
## 1           0
```

This rounding can get masked when we are recoding or doing subsequent calculations

```
SELECT
  case
    when 3/4 > 0.5 then 'High'
    else 'Low'
  end as high_low_int,
  case
    when 3.0 / 4 > 0.5 then 'High'
    else 'Low'
  end as high_low_float
```

```
##   high_low_int high_low_float
## 1          Low           High
```

The above works because of implicit casting. We can also do explicit casting, but where we do this matters

```
SELECT
  case
    when cast(3 as float) / 4 > 0.5 then 'High'
    else 'Low'
  end as cast_first,
  case
    when cast(3/4 as float) > 0.5 then 'High'
    else 'Low'
  end as cast_last
```

```
##   cast_first cast_last
## 1       High       Low
```

3.4.2 Inexact storage and comparison

In R

```
(3 - 2.9) <= 0.1
(2 - 1.9) <= 0.1
(1 - 0.9) <= 0.1
```

```
## [1] FALSE
## [1] FALSE
## [1] TRUE
```

```
(3 - 2.9) == 0.1
(2 - 1.9) == 0.1
(1 - 0.9) == 0.1
```

```
## [1] FALSE
## [1] FALSE
## [1] FALSE
```

despite the fact that we see no difference

```
3 - 2.9
2 - 1.9
1 - 0.9
```

```
## [1] 0.1
## [1] 0.1
## [1] 0.1
```

In python - same problem but in slightly different cases

```
(3 - 2.9) <= 0.1
(2 - 1.9) <= 0.1
(1 - 0.9) <= 0.1
```

```
## False
## False
## True
```

here we can see the differences

```
3 - 2.9
2 - 1.9
1 - 0.9
```

```
## 0.10000000000000009
## 0.10000000000000009
## 0.09999999999999998
```

Same thing with SQL where differences are masked

```
select
  3 - 2.9 as sub_three,
  2 - 1.9 as sub_two,
  1 - 0.9 as sub_one,
  (3 - 2.9) <= 0.1 as sub_compare_three,
  (2 - 1.9) <= 0.1 as sub_compare_two,
  (1 - 0.9) <= 0.1 as sub_compare_one
```

```
##   sub_three sub_two sub_one sub_compare_three
## 1       0.1     0.1     0.1                 0
##   sub_compare_two sub_compare_one
## 1               0               1
```

Instead, can use either built-in equality checker (python equivalent is `math.isclose`) or check that difference between two numbers is in very small window

```
all.equal(1 - 0.9, 0.1)

abs( (1-0.9) - 0.1 ) <= 1e-10
```

```
## [1] TRUE
## [1] TRUE
```

3.4.3 Division by zero

more of a design issue about the right way to handle

we've seen before how we have to understand other peoples use of null values

this is a case where we he to decide what makes most sense

3.5 Strings (WIP)

String data can be inherently appealing. At their best, strings are used to bring more readable and human interpretable values into a dataset. However, string data and the processing thereof comes with its own challenges.

First, unlike numbers, human language strings can be ambiguously defined. `2` is the only number to represent the value of two. However, the incorporation of human language means that many different words, phrases, and formatting choices can represent the same concept. This is confounded by instances where string data was manually entered, as is the case with user-input data.[7]

[7] I can accurately say my name is "Emily", "Emily Riederer", "Emily E. Riederer", "Ms. Emily Riederer". Additionally, if I spell my name over the phone, I can likely end up "Emily Rieder" or if I type it and mindlessly accept autocorrect, I'm "Emily Reindeer". This inconsistently may be problematic if you are trying to combine my data across different sources where I provided my name in different ways.

Secondly, string data is one of the most flexible datatypes and can contain any other types of information – from should-be-logical values (`"yes"/"no"`, `"true"/"false"`), should-be-numeric values (`"27"`), should-be-date values (`"2020-01-01"`), and even complex data encodings like JSON blobs (`"{"name":{"first":"emily","last":"riederer"},"social":{"twitter":"emilyriederer","github"` with hideous formatting for emphasis.) For a data publisher, this may be a convenience, but as we will see it can turn into a frustration or a liability when functions and comparison operations are attempted with strings that semantically represent a different type of value.

3.5.1 Dirty Strings (TODO)

whitespace

```
"a" == "a"
"a b" == "a b"
"a b" == "a  b"
"a b" == "a b "
```

```
## [1] TRUE
## [1] TRUE
## [1] FALSE
## [1] FALSE
```

"fancy" characters (alternate encodings like ms word)

```
' " ' == ' " '
' " ' == ' " '
```

```
## [1] TRUE
## [1] FALSE
```

special characters and display versus values

```
x <- "a\tb"
cat(x) # what you see...
x == "a    b" # ...is not what you get
```

```
## a    b[1] FALSE
```

3.5.2 Regular Expressions (TODO)

we promised not to be solution oriented, but

not knowing regex is a disaster when trying to work with string data…

3.5.3 Comparison

TODO

3.5.3.1 String ordering

Strings are ranked based on *alphabetical order* just like a dictionary. Some properties of this ordering include that:

- numbers are smaller than letters (`1 < "a"`)
- lower-case is smaller than upper case (`"a" < "A"`)
- fewer characters are smaller than more characters (`"a" < "aa"`)

Such rules make perfect sense for true characters. However, when strings are used as a "catch all" to represent other structures, typical comparison operators can produce odd results. For example, it is generally uncontroverisal that ninety-one is less than one hundred twenty. However, the string `"91"` is *greater than* `"120"` because only the character `"9"` is compared to the character `"1"`.[8]

```
91 < 120
"91" < "120"
```

```
## [1] TRUE
## [1] FALSE
```

When strings are used to represent dates and times, comparison operators may or may not work depending on the precise formatting conversions. Below, we see that "YYYYQQ"-formats sort correctly because the information is hierarchically nested; millenia are compared before centuries, centuries before decades, decades before years, and years before quarters. However, many other string representations of dates, like "QQ-YYYY" will not order correctly. Related topics will be discussed in the "Dates and Times" section.

[8] Similarly, `"91" > "905"` because since they both start with 9, we move on to compare 1 which is greater than 0.

```
"20190Q4" < "2020Q3" # string (alphabetic) ordering same as semantic ordering
"Q4-2019" < "Q3-2020" # string and semantic orderings are different
```

```
## [1] TRUE
## [1] FALSE
```

These examples demonstrate that we shouldn't rely on sorting schemes that follow different rules. Before doing comparisons on such types, its a safer bet to cast them to the format most truly representative of their types. If, for some reason, you do wish to keep them as strings, the second example shows that its is wise to format them in the most conducive format possible so things just work.

3.5.3.2 Type coercion

We discussed string comparison before when looking at "dirty" strings. More unexpected behavior arises when strings are compared across different data types. Many computing programs will attempt to *coerce* the objects to a similar and comparable type. Sometimes, this can be convenient as operations "just work", but as always there is a cost for convenience. As we'll see, delegating important decisions to our computing engine may not always capture the semantic relationships that we are most interested in.

For example, consider compare a string and a number. To make them more comparable, R will convert them both to strings before checking for equality. Thus, the number `2020` is equivalent to the string `"2020"` but not the string `"02020"`.

```
"2020" == 2020
"02020" == 2020
```

```
## [1] TRUE
## [1] FALSE
```

In contrast, SQLite[9] thinks that the string `'2020'` is greater than the number 2020 and that these two quantities are not equal.

[9] Different versions of SQL may differ

```
SELECT
  case when     '2020' =  2020 then 1 else 0 end as is_eq,
  case when not '2020' == 2020 then 1 else 0 end as not_eq,
  case when     '2020' <  2020 then 1 else 0 end as is_lt,
  case when     '2020' >  2020 then 1 else 0 end as is_gt
```

```
##   is_eq not_eq is_lt is_gt
## 1     0      1     0     1
```

^TODO: where this could cause problems (FIPS example?)

3.5.4 Transformation (TODO)

basic things like addition differ by language

in R, returns error:

```
'a' + 'b'
```

```
## Error in "a" + "b": non-numeric argument to binary operator
```

```
'a' * 5
```

```
## Error in "a" * 5: non-numeric argument to binary operator
```

in SQLite, goes to zero:

```
SELECT
  'a' + 'b' as string_add,
  'a'*5 as string_mult
```

```
##   string_add string_mult
## 1          0           0
```

in python, does concatenation for `+` and analogous (concatenation of repeat) for `*`:

```
'a' + 'b'
```

```
## 'ab'
```

```
'a' * 5
```

```
## 'aaaaa'
```

3.6 Dates and Times (WIP)

Unlike character strings, dates and times seem like they should be well defined with distinct, quantifiable components like years, months, and days. However, many different conventions for date formatting and underlying storage formats exist. This leads to similar challenges with dates and times as we saw with strings before.

Some common formats in the wild are:

- YYYYMMDD
- YYYYMM
- MMDDYYYY
- DDMMYYYY
- MM/DD/YYYY
- MM/DD/YY
- DD/MM/YYYY
- YYYY-MM-DD (ISO8601)

In addition to how dates are *formatted*, they may be stored in a variety of different ways "under the hood" such as Unix time (seconds since 1970-01-01 00:00:00 UTC) and Julian time (days since noon in Greenwich on November 24, 4714 B.C) (TODO).

To complicate matters further, many of these formats may be represented either by native date types in various programs or by more basic data types (such as integers for the first four and strings for the last four). In addition, analogous formats exist for *timestamps* which encode both calendar date and time of day (hour, minute, and second information).

TODO: why ISO8601?

3.6.1 Comparison

Automatic conversion of data types Dates versus timestamps

```
df_dt <-
data.frame(
  DT_ENROLL = as.Date("2020-01-01"),
  DT_PURCH  = 20200101,
  DT_LOGIN  = as.POSIXlt("2020-01-01T12:00:00")
  )
```

none of these are equal so nothing returns on filtering

```
filter(df_dt, DT_ENROLL == DT_PURCH) %>% nrow()
```

```
## [1] 0
```

```
filter(df_dt, DT_ENROLL == DT_LOGIN) %>% nrow()
```

```
## Warning in mask$eval_all_filter(dots, env_filter):
## Incompatible methods ("Ops.Date", "Ops.POSIXt") for
## "=="
```

```
## [1] 0
```

the same thing happens in sql

```
select * from df_dt where DT_ENROLL = DT_PURCH
```

```
## [1] DT_ENROLL DT_PURCH  DT_LOGIN
## <0 rows> (or 0-length row.names)
```

```
select * from df_dt where DT_ENROLL = DT_LOGIN
```

```
## [1] DT_ENROLL DT_PURCH  DT_LOGIN
## <0 rows> (or 0-length row.names)
```

in what way aren't they equal? to understand this its helpful to know how the computer encodes these dates

with `as.numeric()` in R we can see the numeric representation of the date

```
as.numeric(df_dt$DT_ENROLL)
```

```
## [1] 18262
```

this works the same way in SQL

```
select cast(DT_ENROLL as integer), cast(DT_PURCH as integer) from df_dt
```

```
##   cast(DT_ENROLL as integer) cast(DT_PURCH as integer)
## 1                      18262                  20200101
```

this has the implication that things that are on the same date have an inequality relationship in both languages

```
filter(df_dt, DT_ENROLL < DT_PURCH) %>% nrow()
```

```
## [1] 1
```

```
select
  cast(DT_ENROLL as integer),
  case when DT_ENROLL < 18000 then 1 else 0 end as lt_18000,
  case when DT_ENROLL < 19000 then 1 else 0 end as lt_19000,
  case when DT_ENROLL < DT_PURCH then 1 else 0 end as lt_purch
from df_dt
```

```
##   cast(DT_ENROLL as integer) lt_18000 lt_19000
## 1                      18262        0        1
##   lt_purch
## 1        1
```

Note this this can affect both filters and joins

and this similarly causes a more general problem when comparing a date to a date-as-an-integer

```
as.Date("2020-01-01") > 20160501
```

```
## [1] FALSE
```

```
select cast('2020-01-01' as date) > 20160501
```

```
##   cast('2020-01-01' as date) > 20160501
## 1                                     0
```

3.7 Programming Errors (TODO)

3.7.1 Default Cases (WIP)

see case-when example in nulls section

3.7.2 Order of Operations (WIP)

PEMDAS but sometimes still ambiguous

```
1 + 1  * 2 / 3 - 1
(1 + 1) * 2 / 3 - 1
1 + 1 * 2 / (3 - 1)
```

```
## [1] 0.6667
## [1] 0.3333
## [1] 2
```

SQL clause order of evaluations

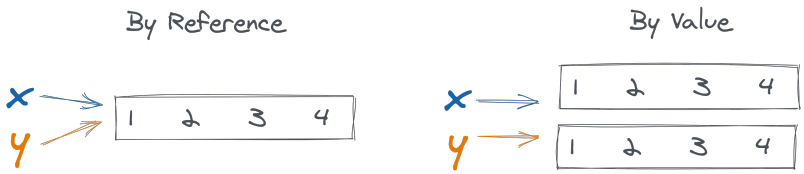


FIGURE 3.2: Different relationships between named variables and their values

3.7.3 Object References (WIP)

Copying and modifying object overview

When might each be preferred?

What risks are there if we don't understand which we are doing?

In Python

```
x = [1,2,3]
y = x
y.append(4)
print(y)
print(x)
```

```
## [1, 2, 3, 4]
## [1, 2, 3, 4]
```

```
z = x.copy()
z.append(5)
print(z)
print(x)
```

```
## [1, 2, 3, 4, 5]
## [1, 2, 3, 4]
```

pandas DataFrame methods with `inplace` arg (`False` is default)

In R

```
library(data.table)

DT <- data.table(a=c(1,2), b=c(11,12))
print(DT)

newDT <- DT         # reference, not copy
newDT[1, a := 100] # modify new DT

print(DT)           # DT is modified too.

DT = data.table(a=c(1,2), b=c(11,12))
newDT <- DT
newDT$b[2] <- 200  # new operation
newDT[1, a := 100]

print(DT)
```

```
##    a  b
## 1: 1 11
## 2: 2 12
##      a  b
## 1: 100 11
## 2:   2 12
##    a  b
## 1: 1 11
## 2: 2 12
```

From https://stackoverflow.com/questions/10225098/understanding-exactly-when-a-data-table-is-a-reference-to-vs-a-copy-of-another

```
import pandas as pd

# set-up sample data ----
data = {'a': [1, 2],
        'b': [11, 12]}
df = pd.DataFrame(data = data)

# rename columns without replacing ----
df.rename(columns = {'a':'x'})
```

```
##    x   b
```

```
## 0  1  11
## 1  2  12
```

```
df

# rename columns with replacing ----
```

```
##    a   b
## 0  1  11
## 1  2  12
```

```
df.rename(columns = {'a':'x'}, inplace = True)
df
```

```
##    x   b
## 0  1  11
## 1  2  12
```

```
add_ones <- function(data) {

  data$x0 <- rep(0, nrow(data))

}

df <- data.frame(x1 = 1:5)
df
```

```
##   x1
## 1  1
## 2  2
## 3  3
## 4  4
## 5  5
```

```
add_ones(df)
df
```

```
##   x1
## 1  1
## 2  2
## 3  3
## 4  4
## 5  5
```

```
add_ones <- function(data) {

  data$x0 <- rep(0, nrow(data))
  return(data)

}

df <- data.frame(x1 = 1:5)
df
```

```
##   x1
## 1  1
## 2  2
## 3  3
## 4  4
## 5  5
```

```
add_ones(df)
```

```
##   x1 x0
## 1  1  0
## 2  2  0
## 3  3  0
## 4  4  0
## 5  5  0
```

```
df
```

```
##   x1
## 1  1
## 2  2
## 3  3
## 4  4
## 5  5
```

```
df2 <- add_ones(df)
df2
```

```
##   x1 x0
## 1  1  0
## 2  2  0
## 3  3  0
## 4  4  0
## 5  5  0
```

3.8 Trusting Tools

3.8.1 Delegating decisions

A theme throughout this book is the fundamentally *social* nature of data analysis. Data analysis is fraught without understanding the countless decisions made along the way by those who generated it (whose data is reflected), those who collected it, those who migrated it, and those who have posed questions of it. On one hand, this is a beautiful aspect of analysis; on the other hand, it means that analysts and their analyses are subject to all of the cognitive and social psychological biases of everyday humans.

One such bias is "social proof": assuming that if a tool behaves a certain way, it must be because it is correct.

Assuming that our tools know best is admittedly an attractive proposition. It appeals to a desire to think that someone, somewhere is "in charge" and, perhaps more critically, helps us avoid a domino effect of distrust (If we *don't* trust our tools how can we trust our results? And if we can't trust our results,

how can we trust anything at all?) Unfortunately, there are many reasons are tools might not know best. For example, the tool's developer might have:

- Made a mistake
- Had a different analysis problem in mind with a different optimal approach
- Been optimizing for a different constraint (e.g. explainability vs. accuracy, speed vs. theoretical properties)
- Come from a community with different norms
- Been affording users the flexibility to do things many ways even if they don't agree
- Built a certain feature for a different purpose than how you are using it
- Not thought about it at all

As a few concrete examples from popular open source tools. We'll look briefly at the prominent python library `scikitlearn` for machine learning and Apache Spark, an engine for large-scale distributed data processing.

3.8.1.1 Defaults in `scikitlearn`

`scikitlearn`'s default behavior for logistic regression modeling[10] automatically applies L2 regularization. You might or might not know what this means, and you might or might not want to apply it to your problem. That's fine. The important thing is that it *will* change your estimates and predictions, and it is *not* a part of the classical definition of that algorithm (for modelers coming from a statistical background.)

Of course, there's nothing inherently wrong about this choice; the library authors just had different goals than a typical statistical. `scikitlearn` developer Olivier Grisel explains on Twitter[11] that this choice (and others in the library) is explained because "Scikit-learn was always designed to make it easy to get good predictive accuracy (eg as measured by CV) rather than as statistical inference library." Additionally, this choice is documented in bold in the function documentation[12].

However, an analyst could easily miss this nuance if they do not *read* the documentation. Or, if they *misinterpret* this choice as social proof that regularization is always the right approach, they might not make the best choice for their own analysis.

[10] A classic modeling technique for predicting binary (yes/no) outcomes

[11] https://twitter.com/ogrisel/status/1167438229655773186?s=20

[12] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

3.8.1.2 Algorithms in `Spark`

As a second example, according to a 2015 Jira ticket[13], developers of Spark considered multiple methodologies they could use when adding the functionality to compute feature importance for a random forest. Ultimately, a core contributor advised against permutation importance due to its computational cost.

> It's high time we add this to MLlib, so I'm adding this to the 1.5 roadmap. Peter Prettenhofer If you are still interested in this, please feel free to take it. Or if others are interested, please comment on this JIRA.
>
> The initial API should be quite simple; I'm imagining a single method returning importance for each feature, modeled after what R or other libraries return.
>
> I think we should calculate importance based on the learned model. The permutation test would be nice in the future but would be much more expensive (shuffling data).

FIGURE 3.3: JIRA ticket for Spark with a discussion of which random forest variable importance algorithm to implement

Clearly, no one wants a workflow that is too costly or timely to run. So, once again, there is no right or wrong. However, since every approach to feature importance has its own biases, pitfalls, and challenges in interpretation, it's a mistake for an end-user to not carefully understand which algorithm is used and why.

3.8.2 "Off-Label" Use (TODO)

coined in https://www.rstudio.com/resources/rstudioglobal-2021/maintaining-the-house-the-tidyverse-built/

3.8.3 Security (TODO)

namespace squatting

executable code

[13] https://issues.apache.org/jira/browse/SPARK-5133

3.9 Inefficient Processing (TODO)

3.10 Strategies (WIP)

Paragraph 1 TODO

Some computational quandaries are inherent to our tools themselves, but often they are a function both of the tools and the ways we chose to use them. More strategies related to writing robust and resilient code will be discussed in Chapter 11 (Complexify Code).

3.10.1 Understand the intent

- read the docs
- look at examples
- don't carry default knowledge between languages

3.10.2 Understand the execution

- test out simple examples (like we've been doing)
- specificlly try out corner cases

3.10.3 Be explicit not implicit

- default arguments
- examples above with casting, coalescingW

3.11 Real World Disasters (WIP)

https://www.theguardian.com/politics/2020/oct/05/how-excel-may-have-caused-loss-of-16000-covid-tests-in-england

> The data error, which led to 15,841 positive tests being left off the official daily figures, means than 50,000 potentially infectious people may have been missed by contact tracers and not told to self-isolate.

Analysis

4

Egregious Aggregations (WIP)

Once armed with an understanding of the data and tools available for analysis, a common start to analysis is exploring data with *aggregation*. At its heart, any sort of data analysis is the process of condensing raw data into something more manageable and useful while giving up as little of the information as possible. From linear regressions and hypothesis testing to random forests and beyond, much of data analysis could truly be called "applied sums an averages".

Many elementary tools for this task are much better at the comprehension task than the preservation one. We learn rigorous assumptions to consider and validate when studying linear regression, but basic arithmetic aggregation presents itself as agnostic and welcome to any type of data. However, the underlying distributions of our variables and the relationships between them have a significant impact on the how informative and interpretable various summarizations are.

In this chapter, we will explore different ways that univariate and multivariate aggregations can be naive or uninformative.

4.1 Motivating Example: Similar in Summary

To begin, we will look at a whimsical toy example. This may feel trite or manufactured, but the subsequent sections will aim to convince you that these issues are not just esoteric. Consider the "datasaurus dozen" dataset (Matejka and Fitzmaurice, 2017) which is available within the `datasauRus` R package (Locke and D'Agostino McGowan, 2018).

```
library(datasauRus)
```

This dataset contains 12 sets of data stacked on top of one another and identi-

TABLE 4.1: Summary statistics for Datasaurus Dozen datasets

dataset	mean(x)	mean(y)	var(x)	var(y)	cor(x, y)
away	54.27	47.84	281.2	725.8	-0.064
bullseye	54.27	47.83	281.2	725.5	-0.069
circle	54.27	47.84	280.9	725.2	-0.068
dino	54.26	47.83	281.1	725.5	-0.064
dots	54.26	47.84	281.2	725.2	-0.060
h_lines	54.26	47.83	281.1	725.8	-0.062
high_lines	54.27	47.84	281.1	725.8	-0.069
slant_down	54.27	47.84	281.1	725.6	-0.069
slant_up	54.27	47.83	281.2	725.7	-0.069
star	54.27	47.84	281.2	725.2	-0.063
v_lines	54.27	47.84	281.2	725.6	-0.069
wide_lines	54.27	47.83	281.2	725.7	-0.067
x_shape	54.26	47.84	281.2	725.2	-0.066

fied by the `dataset` column.[1]. Besides the identifier column, the data is fairly small and contains only two variables `x` and `y`.

```
df <- datasauRus::datasaurus_dozen
head(df)
```

```
## # A tibble: 6 x 3
##   dataset     x     y
##   <chr>   <dbl> <dbl>
## 1 dino     55.4  97.2
## 2 dino     51.5  96.0
## 3 dino     46.2  94.5
## 4 dino     42.8  91.4
## 5 dino     40.8  88.3
## 6 dino     38.7  84.9
```

A quick analysis of summary statistics reveals that each of the 12 datasets is very consistent in its summary statistics. The means and variances of `x` and `y` and even their correlations are nearly identifcal.

However, as shown in Figure 4.1, when we visualize this data, we find that the 12 datasets reveal remarkably different patterns.

This dataset is a more elaborate version of Anscombe's Quartet, a well-known

[1] If you are following along in R, you might run `unique(df$dataset)` to see all the values of this column. We won't do that now as to not ruin the surprise

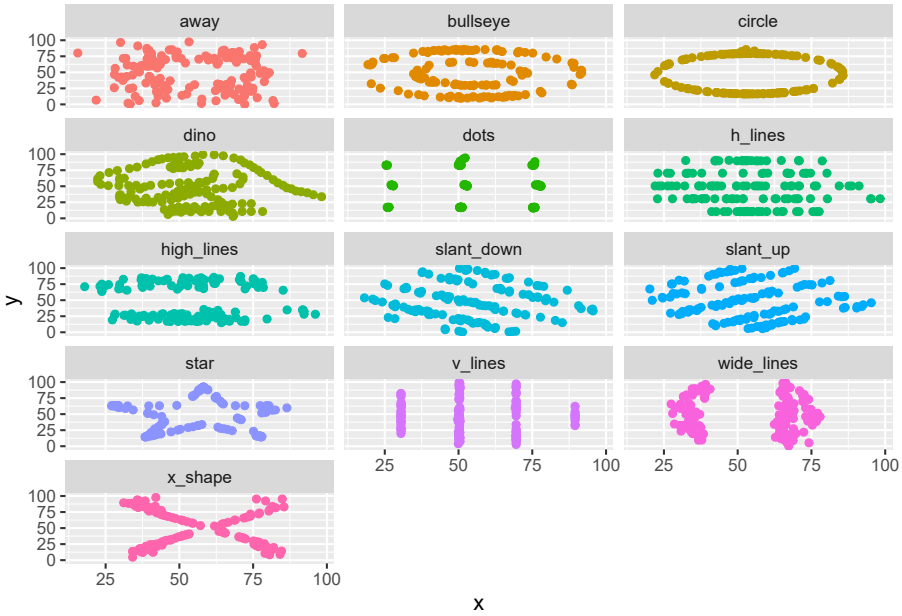


FIGURE 4.1: Scatterplots for Datasaurus Dozen datasets

set of four datasets which exhibit similar properties. Examining a simiilar plot for Anscombe's Quartet (with data from the `anscombe` dataset which ships in R's `datasets` package), we can get better intuition for how the phenomenon is manufactured. Figure 4.2 shows as similar plot to 4.1. Comparing datasets 1 and 3, for example, we can see a trade-off between a semi-strong trend with a consistent-seeming amount of noise and an nearly perfect linear trend with a single outlier.

```
## Warning: Values are not uniquely identified; output will contain list-cols.
## * Use `values_fn = list` to suppress this warning.
## * Use `values_fn = length` to identify where the duplicates arise
## * Use `values_fn = {summary_fun}` to summarise duplicates
```

```
## Warning: `cols` is now required when using unnest().
## Please use `cols = c(x, y)`
```

```
## `geom_smooth()` using formula 'y ~ x'
```

Figure 4.2 also plots the simple linear regression line for each dataset. Similar to the summary statistics, these are also identical. We know this because the regression coefficient for a simple linear regression is given by

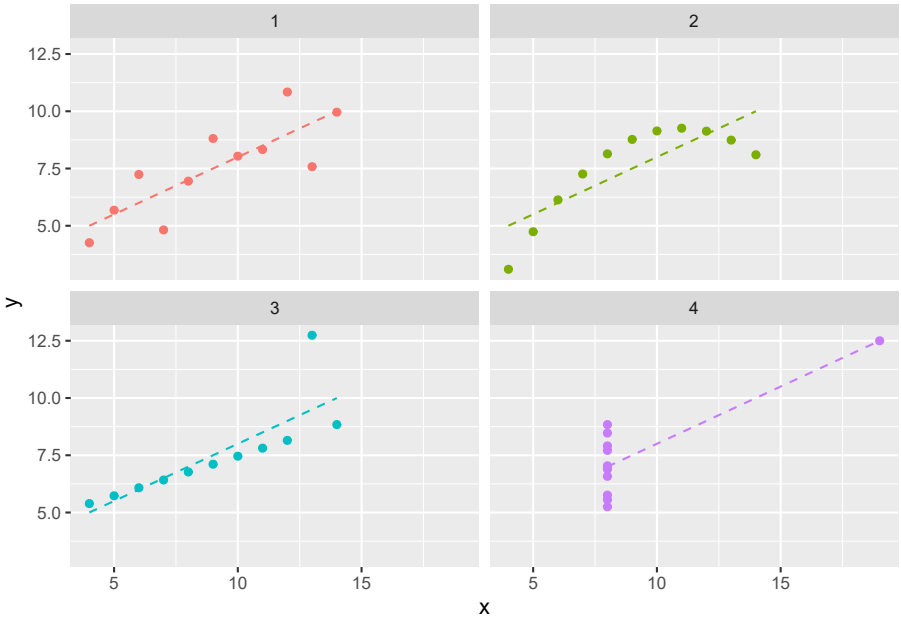


FIGURE 4.2: Scatterplots for Anscombe's Quartet

`cov(x,y)/sd(x)sd(y)`. You'll notice I do not write "we can see that…" because, in fact, we can only *see* similarity not equality. The message of this section may seem to be "don't summarize your data without plotting it", but conducting "visual analytics" without looking at the numbers is also problematic. We'll explore the latter topic more in Chapter **??** (Vexing Visualiztions).

While there are clearly a contrived example (and, if you so chose to check out the "Datasaurus Dozen" paper, a very cleverly contrived example!), its also a cautionary tale. Summary statistics are not just insufficient when they focus on central tendency (e.g. mean) instead of spread. In this example, even an examination of variation and covariation led to an overly simplistic view of the underlying data.

4.2 Averages (WIP)

4.2.1 Implicit assumptions (TODO)

When statistics students study linear regression, they are introduced to a number of canonical assumptions including:

- The true functional form between the dependent and independent variables is linear / additive
- Errors are independent
- Errors have constant variance (that is, they are homoskedastic)
- Errors have a normal distribution

Of course, whether or not these assumptions hold, there's nothing stopping anyone from *mechanically* fit at linear regression[2]. Instead, these assumptions are required to make the output of a linear regression *meaningful* and, more specifically, for conducting correct inference.

Similarly, there are no limitations on mechanically computing an average

4.2.2 Averaging skewed data

Arithmetic average versus colloquial meaning of average as "typical"

Skewed data

Multimodal data / mixture models

4.2.3 No "average" observation

In the previous section, the average represented a point in the relevant data *range* even if it was not perhaps the one most representative of a "typical" observation. We discussed how in some situations this quantity may be a reasonable answer to certain types of questions and an aid for certain types of decisions.

However, when we seek an average *profile* over multiple variables, the problems of averages are further compounded. We may end up with a set of "average" summary statistics that are not representative of any part of our population.

To see this, let's assume we are working with data for a company with a subscription business model. We might be interested in profiling the age of each account (how long they have been a subscriber) and their activity (measured by amount spent on an e-commerce platform, files downloaded on a streaming service, etc.)

The following code simulates a set of observations: 80% of accounts are between 0 to 3 years in age and have an average activity level of 100 while 20% of accounts are older than 3 years in age and have an average activity level

[2]In fact, the only mechanical constraint to computing linear regression output is that no column of the design matrix (no independent variable) is a precise linear combination of the other columns. Yet this constraint is not typically included among the standard statement of assumptions.

of 500. (Don't over-think the specific probability distributions lived here. We are concerned with interrogating the properties of the average and not with simulating a realistic data generating process. Giving ourselves permission to be wrong or "lazy" about unimportant things gives us more energy to focus on what matters.)

```
set.seed(123)

# define simulation parameters ----
## n: total observations
## p: proportion of observations in group 1
n <- 5000
p <- 0.8
n1 <- n*p
n2 <- n*(1-p)

# generate fake dataset with two groups ----
df <-
  data.frame(
    age = c(runif(n1,   0,  3), runif(n2,   3, 10)),
    act = c(rnorm(n1, 100, 10), rnorm(n2, 500, 10))
  )
```

Figure 4.3 shows a scatterplot of the relationship between account age (x-axis) and activity level (y-axis). Meanwhile, the marginal rug plots shows the univariate distribution of each variable. The sole red dot denotes the coordinates of the average age and average activity. Notably, this dot exists in a region of "zero density"; that is, it is not representative of *any* customer. Strategic decisions made with this sort of observation in mind as the "typical" might not be destined for success.

4.2.4 The product of averages

As the above example shows, averages of multivariate data can produce poor summaries – particularly when these variables are interrelated[3].

A second implication of this observation is that deriving additional computations based on pre-averaged numbers is likely to obtain inaccurate results.

For example, consider that we wish to estimate the average dollar amount of returns per any e-commerce order. Orders may generally be a mixture of

[3]We intentionally avoid the word *correlated* here to emphasize the fact that *correlation* refers more strictly to linear relationships

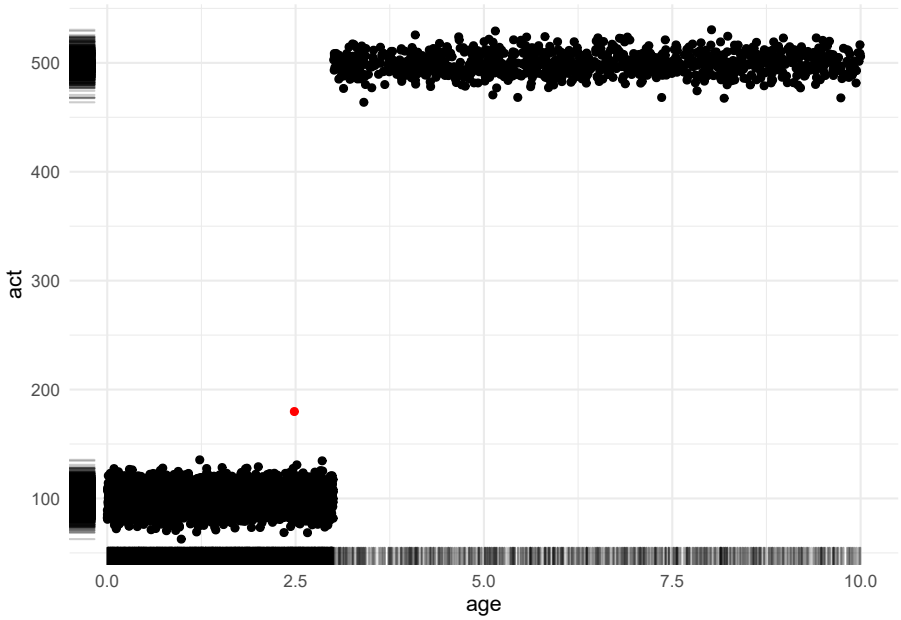


FIGURE 4.3: A scatterplot of two variables and their averages

low-price orders (around $50 on average) and high-price orders (around $250 on average). Low-price orders may have a 10% probability of being returned while high price orders have a 20% probability. (Again, are these numbers, distributions, or relationships hyper-realistic? Not at all. However, once again we are telling ourselves a story just to reason about numerical properties, so we have to give ourselves permission to not focus on irrelevant details.)

```
set.seed(123)

# define simulation parameters ----
## n: observations per group
## pr[1|2]: mean price per group
n <- 100
pr1 <- 50
pr2 <- 250
pr_sd <- 5
re1 <- 0.1
re2 <- 0.2

# simulate spend amounts and return indicators ----
amt_spend  <- c(rnorm(n, pr1, pr_sd), rnorm(n, pr2, pr_sd))
```

```
ind_return <- c(rbinom(n, 1, re1),    rbinom(n, 1, re2))

# compute summary statistics ----
average_of_product <- mean(amt_spend * ind_return)
product_of_average <- mean(amt_spend) * mean(ind_return)
```

The *true* average amount returned across all of our orders is 36.0438 (from the `average_of_product` variable). However, if instead we already knew an average spend amount and an average return proportion, we might be inclined to compute the `product_of_average` method which returns a value of 26.9923. (This is a difference of 9.05 relative to an average purchase amount of 150.)

At first, this may seem unintuitive until we write out the formulas and realize that these metrics are, in fact, two very different quantities:

$$\frac{\sum_1^n Spend}{\sum_1^n 1} * \frac{\sum_1^n I(Return)}{\sum_1^n 1}$$

over all n orders

versus

$$\frac{\sum_1^n Spend * I(Return)}{\sum_1^n 1}$$

If this still feels counterintuitive, we can see how much of the difference is accounted for by the interrelation between our two variables. In the following code, we break the relationship between the variables by randomly reordering the `ind_return` variable so it is no longer has any true relationship to the corresponding `amt_spend` variable.

```
# randomly reorder one of two variables to break relationships ----
ind_return <- sample(ind_return, size = 200)

# recompute variables ----
average_of_product <- mean(amt_spend * ind_return)
product_of_average <- mean(amt_spend) * mean(ind_return)
```

After redoing the calculations, we find that th two values are much closer. `average_of_product` is now 24.1041 and `product_of_average` is now 26.9923. These are notably still not the same number so that does not mean that these two equations are equivalent if variables are unrelated; however, this second result once again illustrates the extent to which interrelations can defy our naive intuitions.

4.2.5 Average over what? (TODO)

no such thing as an unweighted average (just sometimes weights are equal)

formal definition of expected value forces you to pick a probability distribution

eg avg mpg by time vs by mileage?

not strictly an error but our language allows an ill-defined problem

4.2.6 Dichotomization and distributions

```
n <- 1000

# simulate x and y: uniformly distributed x ----
x1 <- runif(n)
y1 <- 5 + 3*x1 + rnorm(n)

# simulate x and y: same relationship, more concentrated distribution of x ----
x2 <- c( runif(n*0.1, 0.00, 0.44),
         runif(n*0.8, 0.45, 0.55),
         runif(n*0.1, 0.55, 1.00)
        )
y2 <- 5 + 3*x2 + rnorm(n)

# com
g1 <- ifelse(x1 < 0.5, 0, 1)
means1 <- c(mean(y1[g1 == 0]), mean(y1[g1 == 1]))
means1
```

```
## [1] 5.814 7.254
```

```
g2 <- ifelse(x2 < 0.5, 0, 1)
means2 <- c(mean(y2[g2 == 0]), mean(y2[g2 == 1]))
means2
```

```
## [1] 6.218 6.763
```

```
means1
```

```
## [1] 5.814 7.254
```

```
means2
```

```
## [1] 6.218 6.763
```

```
cor(x1, y1)
```

```
## [1] 0.6424
```

```
cor(x2, y2)
```

```
## [1] 0.3688
```

4.2.7 Small sample sizes

4.3 Proportions (WIP)

note that these are of course just a type of average (average of indicators) but helpful to examine challenges separately

4.3.1 Picking the right denominator

4.3.2 Sample size effects

4.4 Variation (TODO)

```
x <- c(-20, -10, -5, 0, 5, 10, 20)
mean(x)
var(x)
```

```
## [1] 0
## [1] 175
```

```
x <- c(-15, -15, -5, 0, 5, 15, 15)
mean(x)
var(x)
```

```
## [1] 0
## [1] 158.3
```

4.5 Correlation (WIP)

As shown in 4.4,

4.5.1 Linear relationships only

```
x <- 1:10
y <- list(linear = x, quadratic = x**2, cubic = x**3, quartic = x**4)

vapply(y, FUN = function(z) cor(x, z, method = "pearson"), FUN.VALUE = numeric(1))
```

```
##    linear quadratic     cubic   quartic
##    1.0000    0.9746    0.9284    0.8817
```

4.5.2 Multiple forms

Traditional (Pearson) correlation depends on specific values whereas Spearman and Kendall focus on order statistics

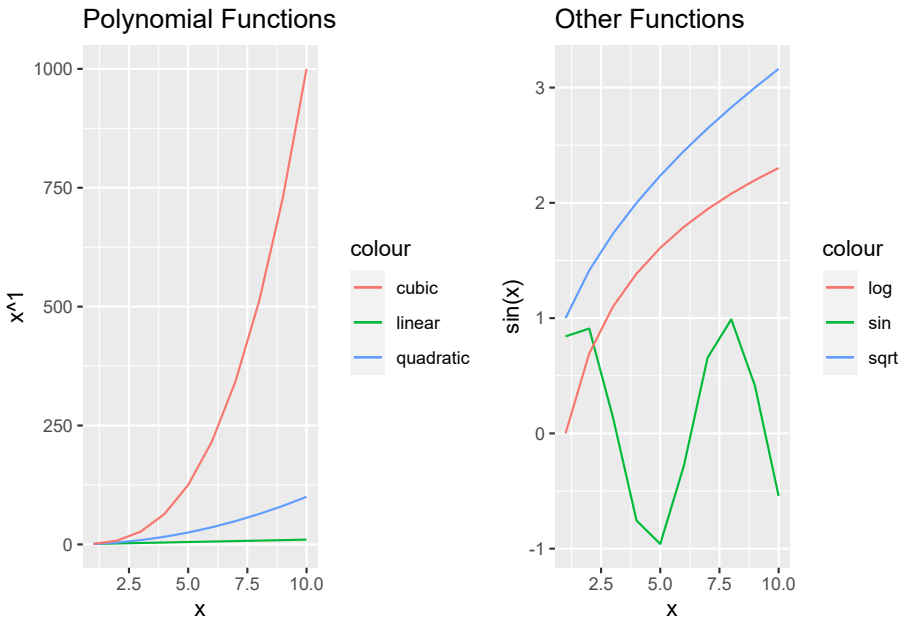


FIGURE 4.4: Plots of x from 1 to 10 over a range of common functions

```
# polynomials ----
x <- 1:10
y <- list(linear = x, quadratic = x**2, cubic = x**3, quartic = x**4)

vapply(y, FUN = function(z) cor(x, z, method = "pearson"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "spearman"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "kendall"), FUN.VALUE = numeric(1))
```

```
##    linear quadratic     cubic   quartic
##    1.0000    0.9746    0.9284    0.8817
##    linear quadratic     cubic   quartic
##         1         1         1         1
##    linear quadratic     cubic   quartic
##         1         1         1         1
```

Similar results with a different set of functions

```
# other functional forms ----
x <- 1:10
```

```
y <- list(sin(x), sqrt(x), exp(x), log(x))

vapply(y, FUN = function(z) cor(x, z, method = "pearson"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "spearman"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "kendall"), FUN.VALUE = numeric(1))
```

```
## [1] -0.1705  0.9892  0.7169  0.9517
## [1] -0.1394  1.0000  1.0000  1.0000
## [1] -0.1111  1.0000  1.0000  1.0000
```

4.5.3 Sensitivity to domain

The "strength of relationship" (completely deterministic) is the same in both cases
However, the summarization of the relationships changes

Here's same case as before:

```
# polynomials ----
x <- 1:10
y <- list(linear = x, quadratic = x**2, cubic = x**3, quartic = x**4)

vapply(y, FUN = function(z) cor(x, z, method = "pearson"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "spearman"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "kendall"), FUN.VALUE = numeric(1))
```

```
##    linear quadratic     cubic   quartic
##    1.0000    0.9746    0.9284    0.8817
##    linear quadratic     cubic   quartic
##         1         1         1         1
##    linear quadratic     cubic   quartic
##         1         1         1         1
```

And here's a different range:

```
# polynomials, diff range ----
x <- -10:10
y <- list(linear = x, quadratic = x**2, cubic = x**3, quartic = x**4)
```

```
vapply(y, FUN = function(z) cor(x, z, method = "pearson"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "spearman"), FUN.VALUE = numeric(1))
vapply(y, FUN = function(z) cor(x, z, method = "kendall"), FUN.VALUE = numeric(1))
```

```
##    linear quadratic     cubic   quartic
##    1.0000    0.0000    0.9179    0.0000
##    linear quadratic     cubic   quartic
##         1         0         1         0
##    linear quadratic     cubic   quartic
##         1         0         1         0
```

4.5.4 Partial correlation

A lot of EDA starts with some sort of correlation matrix
This won't always account for the fact that some variables can mask correlation between others

Consider two groups with trends in different directions

```
n_obsvs <- 10
n_group <-  2

group <- rep(1:n_group, each = n_obsvs)
var1  <- rep(1:n_obsvs, times = n_group)
var2  <- var1 * rep(c(5, -5), each = n_obsvs)
var3  <- var1 * rep(c(1,  5), each = n_obsvs)
```

As Figure 4.5 shows

Because of the opposing trends, their correlation becomes zero

```
cor(var1, var2, method = "pearson")
cor(var1, var2, method = "spearman")
cor(var1, var2, method = "kendall")
```

```
## [1] 0
## [1] 0
## [1] 0
```

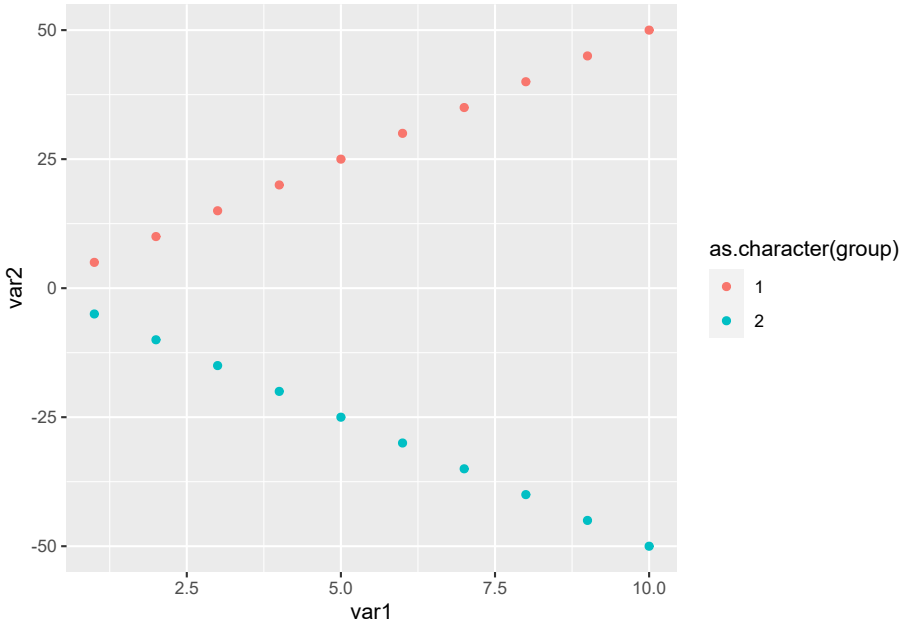


FIGURE 4.5: Subgroups demonstrating opposing linear relationships

However, by group the correlation is 1

```
cor(var1[group == 1], var2[group == 1])
cor(var1[group == 2], var2[group == 2])
```

```
## [1] 1
## [1] -1
```

A similar thing happens when the relationship has the same sign but different slopes

```
cor(var1, var3, method = "pearson")
```

```
## [1] 0.5704
```

while the correlation is still one within group

```
cor(var1[group == 1], var3[group == 1])
cor(var1[group == 2], var3[group == 2])
```

```
## [1] 1
## [1] 1
```

Even partial correlation doesn't help in case of opposing signs

```
library(ppcor)
```

```
## Warning: package 'ppcor' was built under R version
## 4.0.5
```

```
## Loading required package: MASS
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked from 'package:patchwork':
##
##     area
```

```
## The following object is masked from 'package:dplyr':
##
##     select
```

```
pcor(data.frame(var1, var2, group))$estimate
```

```
##       var1    var2   group
## var1     1  0.0000  0.0000
## var2     0  1.0000 -0.8864
## group    0 -0.8864  1.0000
```

It *improves* the strength of the estimated correlation in the case of `var3` but still failes to estimate it correctly. Partial correlation would be assuming a form like `var3 ~ var1 + group` and not `var3 ~ var1 * group`

```
pcor(data.frame(var1, var3, group))$estimate
```

```
##           var1   var3   group
## var1   1.0000 0.8321 -0.7375
## var3   0.8321 1.0000  0.8864
## group -0.7375 0.8864  1.0000
```

4.6 Trends

4.6.1 "If trends continue..."

Figure 4.6 shows that...

```
library(ggplot2)

n <- 300
x <- runif(n)
y <- x + rnorm(n)

ggplot(
  data.frame(x, y),
  aes(x, y)
  ) +
  geom_point() +
  geom_smooth(method = "lm", formula = y ~ I(x**1), se = FALSE, fullrange = TRUE) +
  geom_smooth(method = "lm", formula = y ~ I(x**2), se = FALSE, fullrange = TRUE) +
  geom_smooth(method = "lm", formula = y ~ I(x**3), se = FALSE, fullrange = TRUE) +
  scale_x_continuous(limit = c(0,2))
```

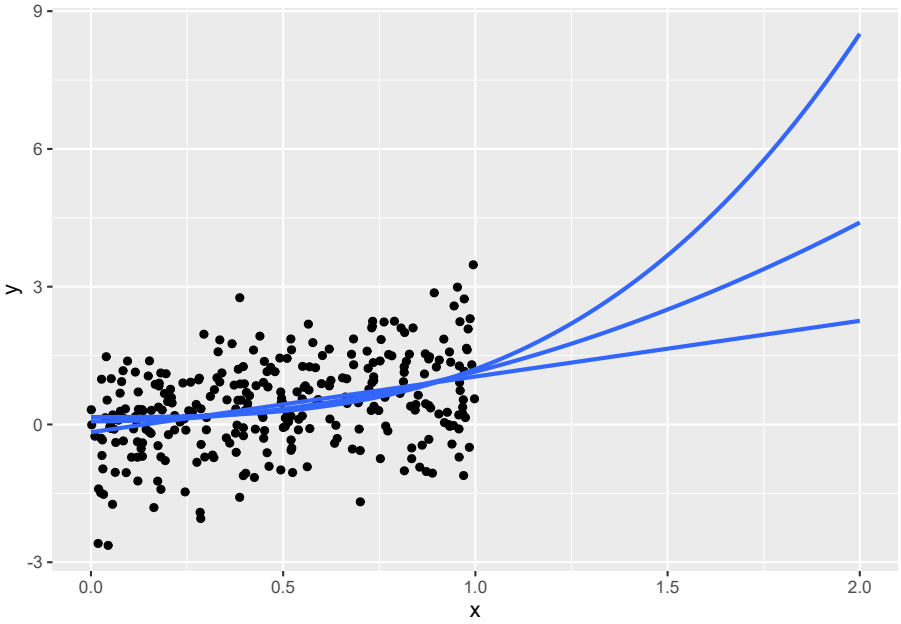


FIGURE 4.6: Extrapolated linear, quadratic, and cubic fits of data

4.6.2 Seasonality

4.7 Comparisons (TODO)

4.7.1 Percents versus percentage points

4.7.2 Changes with small bases

```
(0.015 - 0.010) / 0.010
(0.805 - 0.800) / 0.800
```

```
## [1] 0.5
## [1] 0.00625
```

```
0.015 / 0.010
0.805 / 0.800
```

```
## [1] 1.5
## [1] 1.006
```

4.8 Strategies (TODO)

4.9 Real World Disasters (TODO)

Straight vs weighted averages in COVID positivity rates (May, 2020)

> The changes could result in real-world differences for Hoosiers, because the state uses a county's positivity rate as one of the numbers to determine which restrictions that county will face. Those restrictions determine how many people may gather, among other items.

> Some Hoosiers may see loosened restrictions because of the changes. While Box said the county-level impact will be mixed, she predicted some smaller counties will see a decline in positivity rate after the changes.

> "The change to the methodology is how we calculate the seven-day positivity rate for counties. In the past, similar to many states, we've added each day's positivity rate for seven days and divided by seven to obtain the week's positivity rate. Now we will add all of the positive tests for the week and divide by the total tests done that week to determine the week's positivity rate. This will help to minimize the effect that a high variability

> in the number of tests done each day can have on the week's overall positivity, especially for our smaller counties."

---

three issues here

first straight versus weighted averages

```
avg_of_ratios <- (10/100 + 90/100) / 2

ratio_of_sums <- (10 + 90) / (100 + 100)

avg_of_ratios == ratio_of_sums

avg_of_ratios_uneq <- (10/100 + 180 / 200) / 2

ratio_of_sums_uneq <- (10 + 180) / (100 + 200)

avg_of_ratios_uneq == ratio_of_sums_uneq

weightavg_of_ratios_uneq <- (100/300)*(10/100) + (200/300)*(180/200)

ratio_of_sums_uneq == weightavg_of_ratios_uneq
```

```
## [1] TRUE
## [1] FALSE
## [1] TRUE
```

then back to the data for why it matters.

if data is from same distribution, this could increase variance but shouldn't effect mean

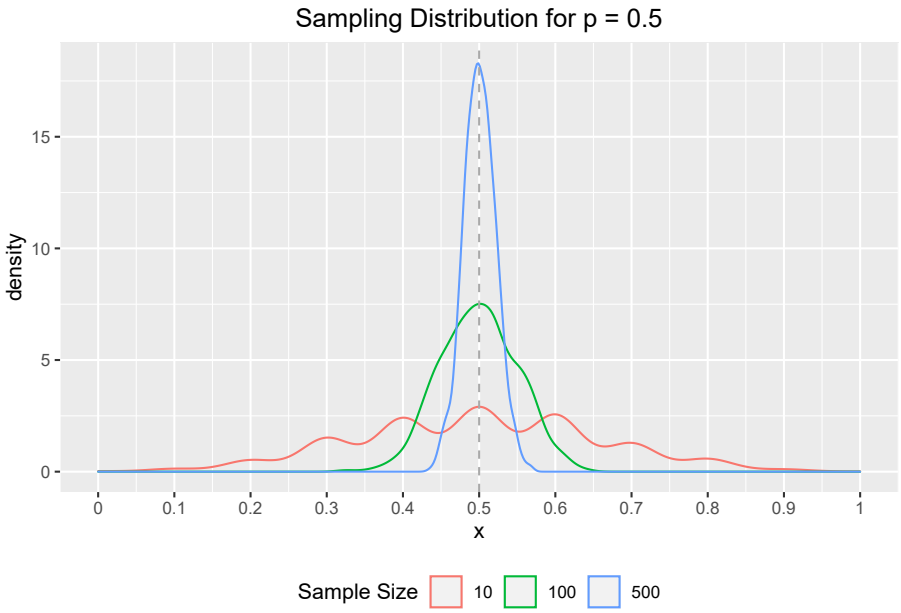
Recall that the standard deviation of sample proportion is $\sqrt{(p * (1-p)/n)}$

link to discussions of sample size and different types of averages

```
set.seed(123)

# define simulation parameters ----
## n: total draws from binomial distribution
## p: proportion of successes
```

```
p <- 0.5
n <- 1000

# sample from binomials of different size ----
s010 <- rbinom(n,  10, p) /  10
s100 <- rbinom(n, 100, p) / 100
s500 <- rbinom(n, 500, p) / 500

# set results as dataframe for inspection ----
df <- data.frame(
  s = rep(c(10, 100, 500), each = n),
  x = c(s010, s100, s500)
)
```

```
library(ggplot2)

ggplot(data = df) +
  aes(x = x, col = as.character(s)) +
  geom_density() +
  geom_vline(xintercept = p, col = 'darkgrey', linetype = 2) +
  labs(
    title = "Sampling Distribution for p = 0.5",
    col = "Sample Size"
  ) +
  scale_x_continuous(breaks = seq(0, 1, 0.1), labels = seq(0, 1, 0.1)) +
  theme(
    plot.title = element_text(hjust = 0.5),
    legend.position = "bottom"
  )
```



but low sample days based on real world are probably also a sign of a different distribution (only very urgent cases get tested?)

5

Vexing Visualization (TODO)

6

Incredible Inferences (TODO)

Previously, we have seen how different inputs like data, tools, and methods can add risks to our data analysis. However, the battle is not won simply when we get our first set of *outputs*. In this chapter, we will explore common errors in interpreting the results of our analysis by exploring aspects of bias, missingness, and confounding.

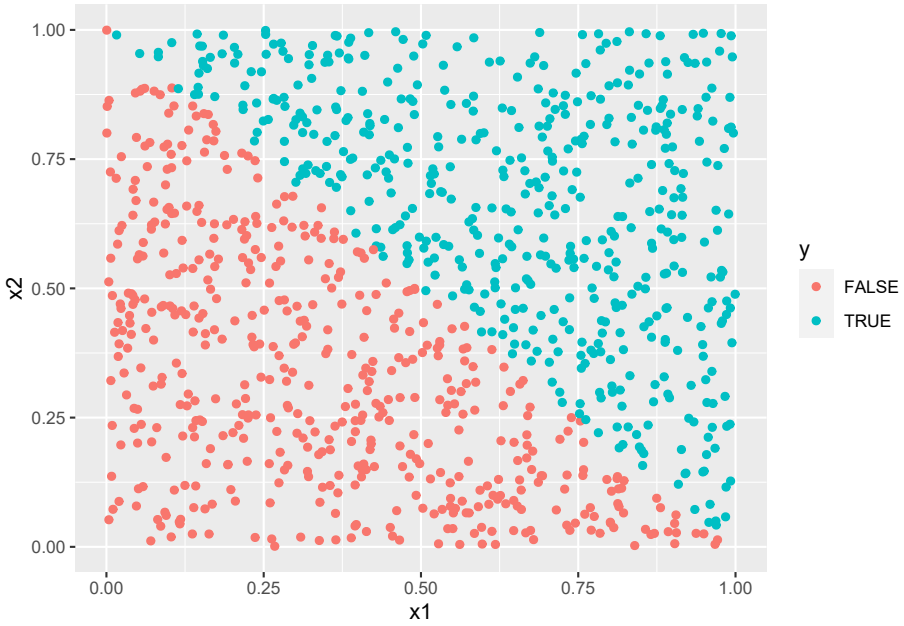
6.1 Common Biases

6.2 Policy-induced relationships

```
set.seed(123)

n <- 1000
x1 <- runif(n)
x2 <- runif(n)
y <- x1 + x2 > 1
df <- data.frame(x1, x2, y)

with(df, cor(x1, x2))
with(df[df$y,], cor(x1, x2))
```

```
## [1] -0.05928
## [1] -0.5003
```

```
library(ggplot2)

ggplot(df) +
  aes(x = x1, y = x2, col = y) +
  geom_point()
```



6.3 Feature leakage

```
n <- 1000
minutes_month1 <- runif(n, 60, 1200)
minutes_month2 <- runif(n, 60, 1200)
minutes_tot <- minutes_month1 + minutes_month2
df <- data.frame(minutes_month1, minutes_month2, minutes_tot)
```

Figure 6.1 shows...

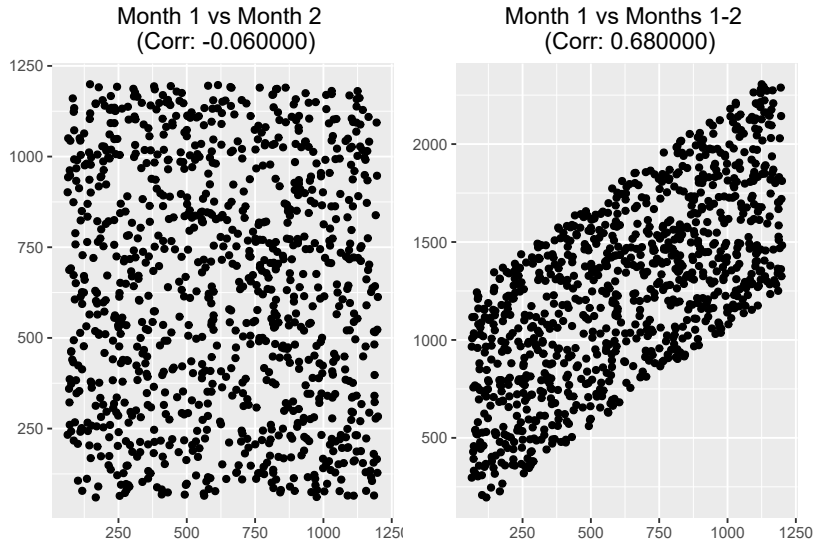


FIGURE 6.1: Correlation of independent versus cumulative quantities

6.4 "Diligent" data dredging

```
set.seed(123)

n <- 1000
x <- rnorm(n)

random_test <- function(x) {

  indices <- sample(1:length(x), length(x)/2, replace = FALSE)
  group1 <- x[indices]
  group2 <- x[-indices]
  tt <- t.test(group1, group2)
  return(tt$p.value)

}

p <- vapply(1:10000, FUN = function(...) {random_test(x)}, FUN.VALUE = numeric(1))
sum(p < 0.05)
```

```
## [1] 500
```

```
n_obsv <- 1000
n_vars <- 100
mat_cat <- matrix(
  data = rbinom(n_obsv * n_vars, 1, 0.5),
  nrow = n_obsv,
  ncol = n_vars
  )
mat_all <- cbind(x, mat_cat)
df <- as.data.frame(mat_all)
names(df) <- c("x", paste0("v", 1:n_vars))
head(df)
```

```
##          x v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13
## 1 -0.56048  1  1  0  1  1  0  1  0  0   1   1   0   1
## 2 -0.23018  1  1  0  0  0  0  0  1  1   1   1   0   0
## 3  1.55871  1  0  0  0  1  0  1  1  0   1   0   0   0
## 4  0.07051  1  0  1  1  0  0  1  0  1   1   1   0   0
## 5  0.12929  1  1  1  0  0  0  0  0  0   1   1   1   1
## 6  1.71506  1  0  0  1  1  0  1  1  1   1   1   0   1
##   v14 v15 v16 v17 v18 v19 v20 v21 v22 v23 v24 v25 v26
## 1   1   0   1   0   0   0   1   0   0   1   1   1   0
## 2   1   1   1   1   1   0   0   1   0   1   0   1   0
## 3   1   1   1   1   1   0   0   1   0   1   1   0   1
## 4   0   0   0   1   0   0   0   1   1   0   1   1   0
## 5   0   1   1   0   0   0   1   1   0   0   0   1   0
## 6   1   0   0   0   0   1   1   0   0   0   1   1   0
##   v27 v28 v29 v30 v31 v32 v33 v34 v35 v36 v37 v38 v39
## 1   0   0   0   1   0   0   1   0   0   0   1   1   1
## 2   1   0   0   1   1   0   1   0   0   0   1   0   0
## 3   0   0   0   1   0   1   0   0   1   1   1   1   0
## 4   0   0   0   1   1   1   1   0   0   1   0   0   0
## 5   1   0   0   1   0   0   0   0   1   0   1   0   1
## 6   0   1   1   0   1   1   0   0   0   1   1   0   1
##   v40 v41 v42 v43 v44 v45 v46 v47 v48 v49 v50 v51 v52
## 1   0   0   0   1   1   1   0   1   0   0   1   1   0
## 2   0   0   0   0   0   0   0   1   0   1   0   1   1
## 3   0   0   1   0   1   1   0   0   1   0   1   1   1
## 4   0   1   0   1   0   0   0   1   0   0   0   0   1
## 5   0   0   1   0   1   0   1   1   1   1   0   1   0
## 6   1   1   1   1   0   0   0   1   0   0   1   0   0
##   v53 v54 v55 v56 v57 v58 v59 v60 v61 v62 v63 v64 v65
```

```
## 1   0   0   0   1   0   1   1   0   0   1   0   0   0
## 2   0   0   1   0   1   0   1   1   0   1   0   1   1
## 3   1   1   1   0   0   1   0   1   0   1   1   1   1
## 4   1   1   0   0   0   0   1   0   1   1   1   1   1
## 5   0   0   1   1   1   0   0   0   0   0   1   0   1
## 6   1   1   0   0   1   0   1   1   0   1   1   0   1
##   v66 v67 v68 v69 v70 v71 v72 v73 v74 v75 v76 v77 v78
## 1   1   1   1   0   0   0   0   1   0   0   1   0   0
## 2   1   1   0   1   1   0   1   1   1   1   1   1   1
## 3   1   0   1   0   1   1   0   1   1   0   0   0   0
## 4   0   1   0   0   0   1   0   1   0   0   0   0   0
## 5   0   0   0   1   0   1   1   1   0   1   0   0   0
## 6   0   1   0   1   0   0   0   1   1   0   1   0   0
##   v79 v80 v81 v82 v83 v84 v85 v86 v87 v88 v89 v90 v91
## 1   1   0   1   0   0   0   0   1   1   1   0   0   1
## 2   1   1   0   1   1   1   0   0   1   1   0   0   0
## 3   1   0   0   1   0   0   0   0   1   0   1   0   0
## 4   0   0   0   0   1   0   1   1   0   0   0   0   0
## 5   1   1   1   0   0   1   1   0   0   1   1   0   1
## 6   1   0   1   1   0   0   0   0   1   1   0   1   0
##   v92 v93 v94 v95 v96 v97 v98 v99 v100
## 1   1   1   0   0   0   1   1   0    0
## 2   0   1   0   1   1   0   1   0    1
## 3   1   0   0   0   1   0   1   0    1
## 4   0   0   1   1   0   0   1   1    1
## 5   0   1   1   1   1   1   0   1    0
## 6   0   0   0   0   1   1   0   0    1
```

```
t.test(x ~ v1, data = df)$p.value
t.test(x ~ v2, data = df)$p.value
t.test(x ~ v3, data = df)$p.value
t.test(x ~ v4, data = df)$p.value
# etc.
```

```
## [1] 0.09771
## [1] 0.8734
## [1] 0.02182
## [1] 0.1525
```

Success! ..Or success?

sample splitting with "train"

(obviously a very ugly way to do this, but that's the point)

```
t.test(x ~ v1, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v2, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v3, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v4, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v5, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v6, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v7, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v8, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v9, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v10, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v11, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v12, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v13, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v14, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v15, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v16, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v17, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v18, data = df[1:(n_obsv/2),])$p.value
t.test(x ~ v19, data = df[1:(n_obsv/2),])$p.value
```

```
## [1] 0.6022
## [1] 0.4947
## [1] 0.196
## [1] 0.3682
## [1] 0.2115
## [1] 0.7113
## [1] 0.3127
## [1] 0.8142
## [1] 0.9033
## [1] 0.8212
## [1] 0.4416
## [1] 0.2564
## [1] 0.5292
## [1] 0.1715
## [1] 0.0855
## [1] 0.2285
## [1] 0.6277
## [1] 0.01318
## [1] 0.2556
```

and “test”

```
t.test(x ~ v18, data = df[(n_obsv/2 + 1):n_obsv,])$p.value
```

```
## [1] 0.1691
```

6.5 Superficial stories

6.5.1 Regression to the mean

simulate truly independent spend amounts across two periods

```
set.seed(123)

n  <- 1000
mu <- 100
sd <- 10
spend1 <- rnorm(n, mu, sd)
spend2 <- rnorm(n, mu, sd)

df <- data.frame(spend1, spend2)
```

```
library(dplyr)

df %>%
  group_by(spend1 > mu) %>%
  summarize_at(vars(starts_with("spend")), mean) %>%
  mutate(pct_change = round((spend2 - spend1) / spend1, 3))
```

```
## # A tibble: 2 x 4
##   `spend1 > mu` spend1 spend2 pct_change
##   <lgl>          <dbl>  <dbl>      <dbl>
## 1 FALSE           92.2   99.7      0.081
## 2 TRUE           108.   101.      -0.063
```

```
df %>%
  mutate(spend1_bin = cut(spend1, 5)) %>%
  group_by(spend1_bin) %>%
  summarize_at(vars(starts_with("spend")), mean) %>%
  mutate(pct_change = round((spend2 - spend1) / spend1, 3))
```

```
## # A tibble: 5 x 4
##   spend1_bin spend1 spend2 pct_change
##   <fct>       <dbl>  <dbl>      <dbl>
## 1 (71.8,84]    80.5   97.8      0.215
## 2 (84,96.1]    91.1  100.       0.098
## 3 (96.1,108]  102.   101.      -0.012
## 4 (108,120]   113.   101.      -0.101
## 5 (120,132]   124.   103.      -0.167
```

```
df %>%
  mutate(spend1_bin = cut(spend1, 5)) %>%
  group_by(spend1_bin) %>%
  summarize(corr = cor(spend1, spend2))
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## # A tibble: 5 x 2
##   spend1_bin    corr
##   <fct>        <dbl>
## 1 (71.8,84]   0.281
## 2 (84,96.1]  -0.0149
## 3 (96.1,108]  0.0438
## 4 (108,120]   0.101
## 5 (120,132]  -0.165
```

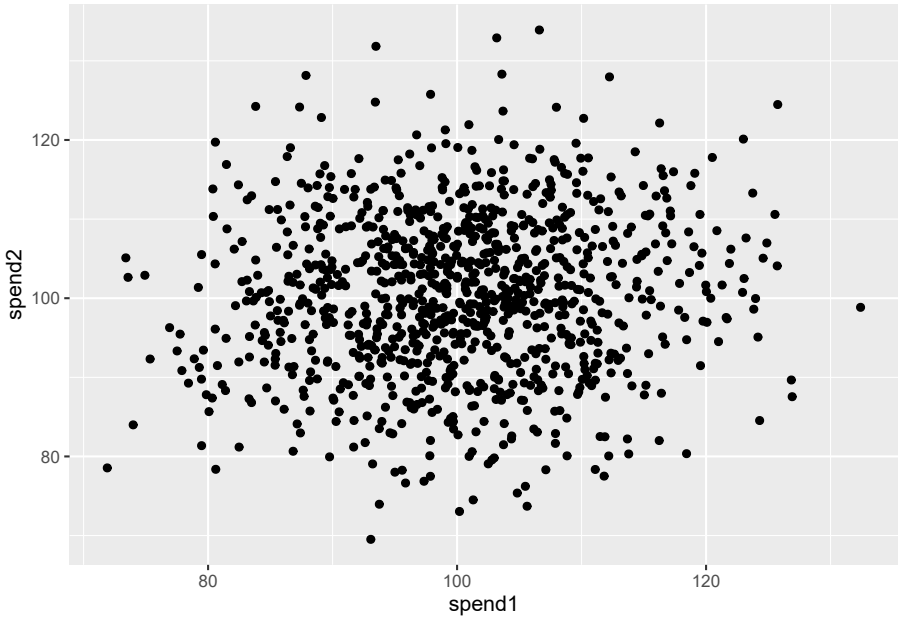
```
mean(spend1 > spend2)
mean(spend1 < spend2)
```

```
## [1] 0.49
## [1] 0.51
```

```
sum((spend1 > mu) * (spend1 > spend2)) / sum(spend1 > mu)
sum((spend1 < mu) * (spend1 < spend2)) / sum(spend1 > mu)
```

```
## [1] 0.7168
## [1] 0.7267
```

```
library(ggplot2)

ggplot(df) +
  aes(x = spend1, y = spend2) +
  geom_point()
```



6.5.2 Distribution shifts

Figure 6.2 shows that...

Figure 6.3 shows that...

```
## Warning: Removed 1 row(s) containing missing values
## (geom_path).
```

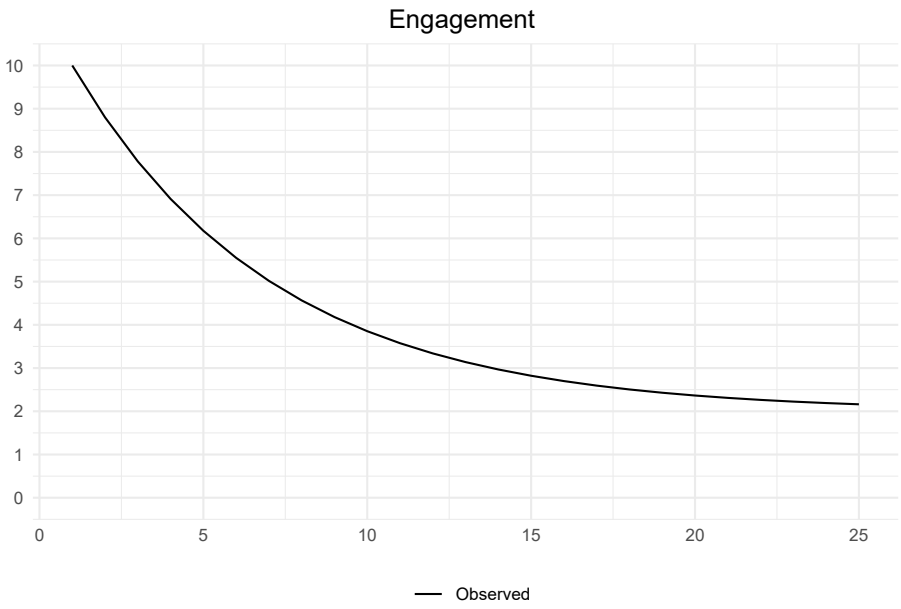


FIGURE 6.2: Trends within and between customer behavioral groups

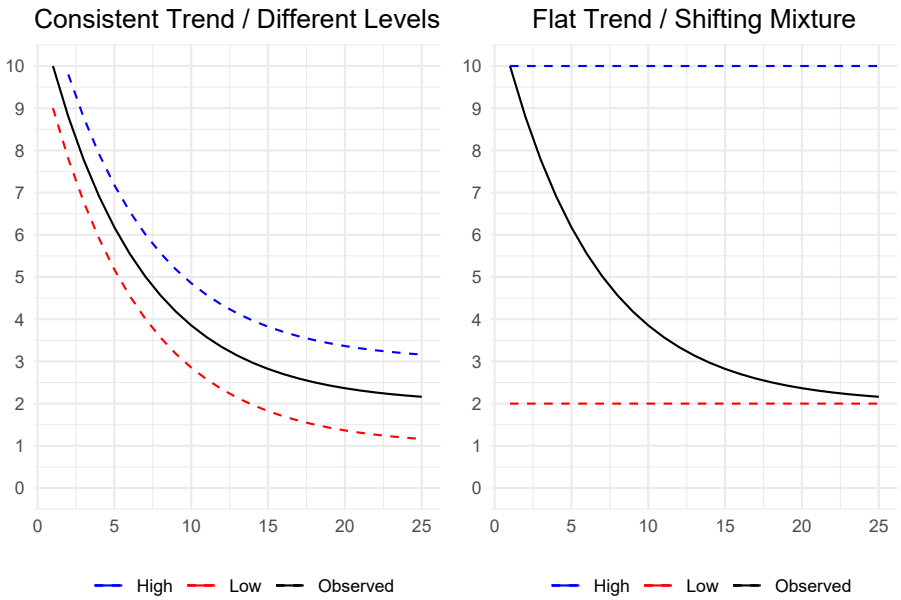


FIGURE 6.3: Possible subgroup trends contributing to aggregate trend

The code used to generate this mock dataset is shown below.

```
hi_engagement <- 10
lo_engagement <- 2
pr_engagement <- 0.85^(0:24)
avg_engagement <- 10*pr_engagement + 2*(1-pr_engagement)

df <-
  data.frame(
    t = 1:length(avg_engagement),
    avg_engagement,
    hi_engagement,
    lo_engagement
    )
```

6.6 Tricky timing issues (WIP)

6.6.1 Censored data

Suppose we are wondering how long our subscription customers will stay put
mean lifetime of customers in 24 and uses exponential distrib (see appendix on distribs)
we are analyzing a cohort of customers 18 months after they first subscribed

```
# time-to-event censored ----
set.seed(123)
n <- 1000
curr_time <- 18
mean_lifetime <- 24

lifetime <- rexp(n, rate = 1 / mean_lifetime)
mean(lifetime)
```

```
## [1] 24.72
```

Because we are only 18 months in, we cannot observe the lifetimes of all customers
for those that left before 18 months we have complete data

but for those who left after 18 months we only know their lifetime exceeds 18 months.
Thus, if we look at the mean only where we can observe it, it's biased towards lower lifetimes. (Recall that we know what the correct value is)

```
#> observed ----
lifetime_observed <- lifetime
lifetime_observed[lifetime > curr_time] <- NA
mean(lifetime_observed, na.rm = TRUE)
```

```
## [1] 7.919
```

Of course, we do know more than nothing (null) about the "surviving customers". We know that their lifetime is *at least* as large as the current time. So alternatively, we could use the current time in our calculations. This makes for a slightly less biased estimate, but it is still wrong and guaranteed to underestimate the actual average.

```
#> max ----
lifetime_max <- pmin(lifetime, curr_time)
mean(lifetime_max)
```

```
## [1] 12.88
```

This scenario illustrates the concept of **censored data**. Figure 6.4 illustrates the fundamental problem more clearly.

So what can we do instead? A common approach is to examine *quantiles* (such as the median) which can make more full use of the data we have. Since we know that rank of our observations (that is, that the censored observations are all larger than the observed datapoints), we can reliable calculate the p-th quantile so long as p percent of the data is not censored.

```
#> quantile ----
sum(!is.na(lifetime_observed)) / n
```

```
## [1] 0.508
```

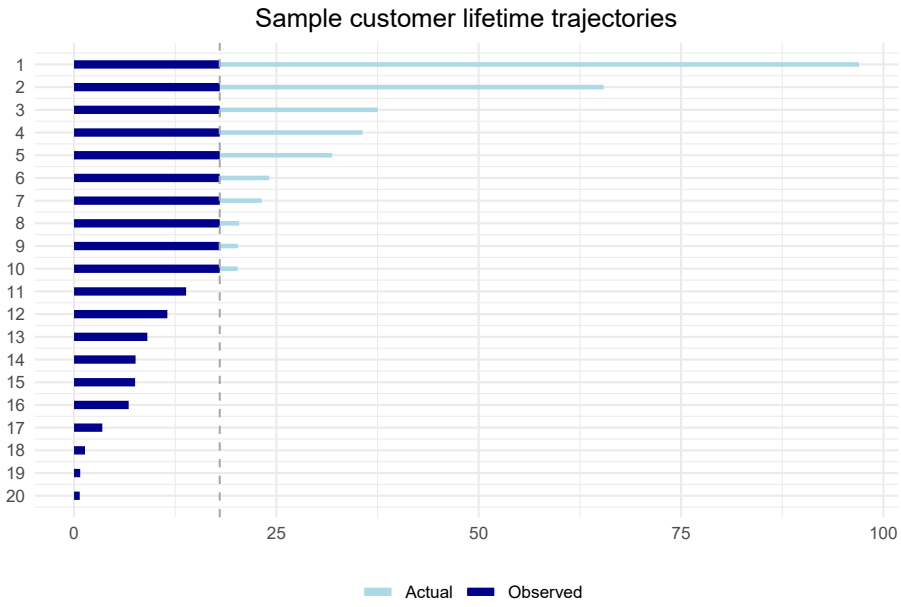


FIGURE 6.4: A sample of observations of customer lifetimes showing observed and censored data

```
lifetime_quantile <- lifetime_observed
lifetime_observed[is.na(lifetime_observed)] <- 100*curr_time
quantile(lifetime_observed, p = c(0.5))
```

```
##   50%
## 17.55
```

6.6.2 Immortal time bias

```
rollout_time <- 12
used_feature <- (lifetime > rollout_time) * rbinom(n, size = 1, prob = 0.5)
aggregate(lifetime, by = list(used_feature), FUN = mean)
```

```
##   Group.1     x
## 1       0 19.51
## 2       1 36.81
```

6.7

7

Cavalier Causality (TODO)

In Chapter 6 (Incredible Inferences), we began to see that we can be tricked by biases when we lack *causal* thinking and an underlying theory for the data generating process. In this chapter, we will revisit some of these same disasters and introduce some specific frameworks to help us more rigorously explore our analysis for errors and biases and, even better, strategize the best ways to fix them.

8

Mindless Modeling (TODO)

8.1 Features

8.2 Targets

8.3 Evaluation Metrics

8.4 Unsupervised Learning

8.5 Lifecycle Management

8.6 Fair and Ethical Modeling

9

Alternative Algorithms (TODO)

As the consummate showman, P.T. Barnum is often quoted as saying "Leave them wanting more". Unfortunately, statistics professors have less of a flare for drama. Introductory statistics courses will typically introduce a few types of models (for example, linear and *perhaps* logistic regression), and that's a wrap. It's often until students start *taking* the subsequent courses that they are exposed to the true limitations of previous techniques and taught to demand more.

This chapter attempts to flip that paradigm by briefly surveying a broad number of modeling techniques. The goal is not to go into all of the rigorous deals that one should understand to use these models. Instead, we hope to build a "mental toolbox" of techniques so that you know where to focus your study when you encounter a problem in the real world.

9.1 Not Modeling

9.1.1 First Principles

9.1.2 Simple Analyses

9.2 Extending Linear Regression

9.2.1 Modeling Binary Outcomes

9.2.2 Modeling Counts

9.2.3 Modeling Time Until an Event

9.2.4 Modeling Repeated Measures on a Population

9.2.5 Modeling Observations in a Nested Hierarchy

9.3 Causal Analysis Patterns

Similar to https://emilyriederer.netlify.app/post/causal-design-patterns/

9.4 Special Data Types

9.4.1 Duration Analysis

9.4.2 Time & Space Data

9.5 Bayesian Methods

9.6 Simulation Methods

9.6.1 Agent-Based

9.6.2 Discrete Event

9.7 Clustering (beyond K-Means)

9.7.1 Density-Based

9.7.2 Mixture Models

Workflow

10

Futile Findings (TODO)

11

Complexifying Code (TODO)

11.1 Making code unreadable

11.1.1 Naming

structure for sorting, ordering, clear semantics

11.1.2 Whitespace

linters / stylers

11.2 Making a monolith

not using functions / files / templates / variables

11.2.1 Making code inflexible (variables)

11.2.2 Making useful code chunks hard to reuse (functions)

11.3 Project organization

11.4 Decoding

rubber duck decoding

12

Rejecting Reproducibility (TODO)

"Good Enough Practices in Computational Reproducibility" (Wilson et al., 2017)

Package managment (Ushey, 2020)

environment management

A

Useful Data Generation Functions (TODO)

B

Common Probability Distributions (TODO)

Bibliography

(2020). Six dimensions of data quality assessment. *DAMA UK Working Group*.

(2021). Covid: Man offered vaccine after error lists him as 6.2cm tall. *BBC North West*.

Barrett, B. (2019). How a 'null' license plate landed one hacker in ticket hell. *WIRED*.

Bassa, A. (2017). Data alone isn't ground truth.

Borunda, D. (2020). El paso officials admit massive covid-19 spike of 3,100 new cases was error. *El Paso Times*.

Hicks, S. C. and Peng, R. D. (2019). Evaluating the success of a data analysis.

Locke, S. and D'Agostino McGowan, L. (2018). *datasauRus: Datasets from the Datasaurus Dozen*. R package version 0.1.4.

Matejka, J. and Fitzmaurice, G. (2017). *Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing*. Association for Computing Machinery, New York, NY, USA.

May, E. (2020). How an error in the calculation of indiana's positivity rate may affect you. *Indianapolis Star*.

Ushey, K. (2020). *renv: Project Environments*. R package version 0.12.2.

Wickham, H. (2014). Tidy data. *Journal of Statistical Software, Articles*, 59(10):1–23.

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., and Teal, T. K. (2017). Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6).