Profiling Your Data

Section 2

Understanding Data

Once your data has been acquired (e.g., importing a .CSV file), you need to understand the data.

Understanding data comes in two forms:

- Domain Knowledge – Understanding the data based on the processes that generated the data
- Profiling – Understanding the technical aspects of the data "as-is"

Domain knowledge can be considered *semantics* – the meaning of the data in business terms. Examples:

- The *flights* dataset's *arr_delay* feature is the count of minutes a flight arrived early/late/on-time
- The *titanic_train* dataset's *SibSp* feature is the count of the siblings/spouses traveling with the passenger

Profiling can be considered *characteristics* – what is contained in the data:

- The maximum value of the *flights* dataset's *distance* feature is 4,983
- The *titanic_train* dataset's *age* feature is missing 177 values

Both types of understanding are critical for success!

Introducing skimr

The *skimr* package provides quick, easy, and robust data profiling.

From the *skimr* package documentation:

- "*skimr* is designed to provide summary statistics about variables in data frames, tibbles, data tables and vectors. It is opinionated in its defaults, but easy to modify."

The primary means of using *skimr* to profile your data is the *skim* function…

Load packages

```
library(skimr)
library(nycflights13)

data("flights")

skim(flights)
```

Load the *flights* dataset

Profile the *flights* dataset using the *skim* function

Data Profiling with skimr – Data Summary

The *skim* function produces a wealth of output via the RStudio *Console*.

First, *skim* produces the *Data Summary*:

Console | Terminal | Background Jobs

R 4.2.2 · ~/

```
> skim(flights)
── Data Summary ────────────────────────
                           Values
Name                       flights
Number of rows             336776
Number of columns          19
_______________________
Column type frequency:
  character                4
  numeric                  14
  POSIXct                  1
________________________
Group variables            None
```

Data Profiling with skimr – Character Data

While the *Data Summary* is useful, it isn't critical when wrangling data for machine learning.

The *skim* function first shows its power by profiling *character* data:

```
── Variable type: character ──────────────────────────────────────────
  skim_variable n_missing complete_rate min max empty n_unique whitespace
1 carrier               0         1       2   2     0       16          0
2 tailnum            2512         0.993   5   6     0     4043          0
3 origin                0         1       3   3     0        3          0
4 dest                  0         1       3   3     0      105          0
```

In terms of machine learning, the following profiling is most valuable:

n_missing: The count of values that are *NA*. Many machine learning algorithms will throw errors when encountering missing data.

empty: The count of values that are empty strings (i.e., “”). Some machine learning algorithms will interpret this as a category.

n_unique: The count of unique values. String features with many unique values (“high cardinality”) are a problem.

whitespace: The count of string values only containing whitespace (e.g., tabs). These can be problematic for many algorithms.

Data Profiling with skimr – Numeric Data

The mighty *skim* function continues by profiling *numeric* data:

```
── Variable type: numeric ──────────────────────────────────────────────────────────
  skim_variable  n_missing complete_rate    mean     sd   p0  p25  p50  p75 p100 hist
1 year                   0         1     2013       0   2013 2013 2013 2013 2013 ▁▁▇▁▁
2 month                  0         1        6.55   3.41    1    4    7   10   12 ▇▇▇▇▇
3 day                    0         1       15.7    8.77    1    8   16   23   31 ▇▇▇▇▆
4 dep_time            8255         0.975 1349.   488.      1  907 1401 1744 2400 ▁▇▆▇▃
5 sched_dep_time         0         1     1344.   467.    106  906 1359 1729 2359 ▁▇▇▇▃
```

In terms of machine learning, the following profiling is most valuable:

n_missing: The count of values that are *NA*. Many missing values are problematic for machine learning algorithms.

p0, p25, p50, p75, p100: The values corresponding to various feature distribution percentiles. Useful for determining problematic numeric data: single values (e.g., *year*), uniform distributions (e.g., *month*), and unique identifiers.

hist: A visualization of the distribution of the numeric features' values. Useful for determining problematic numeric data: single values (e.g., *year*), uniform distributions (e.g., *month*), and unique identifiers.

Data Profiling with skimr – POSIXct Data

The mighty *skim* function finishes by profiling date/time (i.e., *POSIXct*) data:

```
── Variable type: POSIXct ──────────────────────────────────────────────────────────────────────────
  skim_variable n_missing complete_rate min                 max                 median              n_unique
1 time_hour             0             1 2013-01-01 05:00:00 2013-12-31 23:00:00 2013-07-03 10:00:00     6936
```

In terms of machine learning, the following profiling is most valuable:

n_missing: The count of values that are *NA*. Many machine learning algorithms will throw errors when encountering missing data, and many missing values are problematic for algorithms.

min, max: Useful for determining problematic date/time values. For example, default dates (e.g., 1900-01-01).

Missing Data

The most crucial understanding provided by data profiling is knowing where data is missing.

The reason - most machine learning algorithms cannot handle missing data.

There are five data wrangling for machine learning strategies to deal with missing data:

1. Limit yourself to only algorithms (e.g., decision trees) that can handle missing data
2. Remove any observations (rows) with missing data
3. Remove any features with missing data
4. Find a proxy feature to use instead of a feature with missing data
5. Replace missing feature values with a computed value

Typically, options one and two are not practical, while three is commonly used for low-quality features.

Option five is known as *imputation*.

We will use options three, four, and five in the labs.

When Numbers Are Not Numbers

Another powerful understanding produced by data profiling is where a feature is represented by misusing a numeric value.

Consider the *month* feature of the *flights* dataset:

```
── Variable type: numeric ──────────────────────────────────────────────────────
  skim_variable n_missing complete_rate    mean    sd   p0  p25  p50  p75 p100 hist
1 year                  0             1 2013       0 2013 2013 2013 2013 2013 ▁▁▇▁▁
2 month                 0             1    6.55 3.41    1    4    7   10   12 ▇▆▆▆▇
```

The profiling illustrates that *month* is encoded as a number when this is incorrect.

For example, December (i.e., 12) is not twelve times larger than January (i.e., 1).

Arithmetic operations (e.g., division) do not make sense for the *month* feature, so it should be transformed into a *factor*.

The best practice is to transform features into *factors* when applicable!

Factors

Categorical data is so important in statistics and predictive modeling that R has a specific way to represent categorical data – *factors*.

When using machine learning, transforming categorical data into *factors* is not only a best practice but often required

For example, some algorithms (e.g., logistic regression) will treat the *months* feature of the *flights* dataset as being numeric, producing predictive models that are not correct.

When wrangling data for machine learning, properly transforming categorical data is one of the most critical tasks.

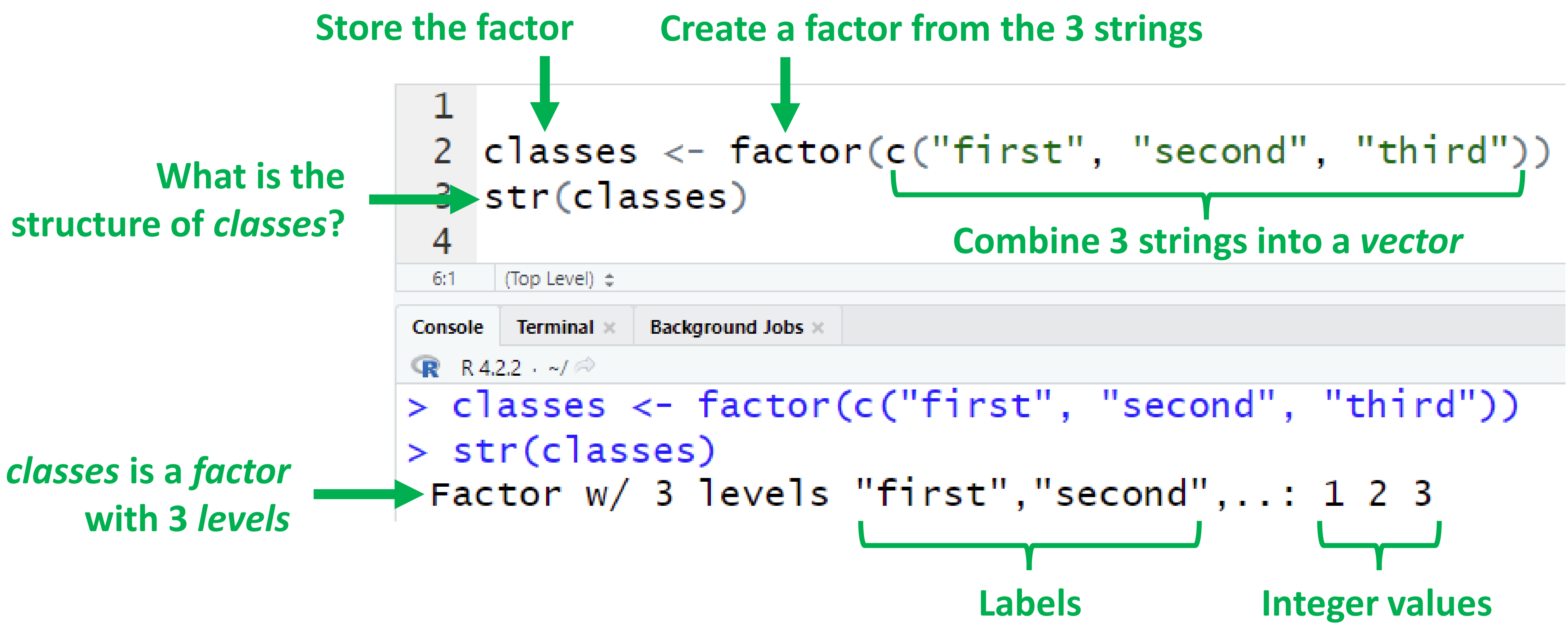
Both *character* and *numeric* data can be transformed into factors…

Creating Character Factors

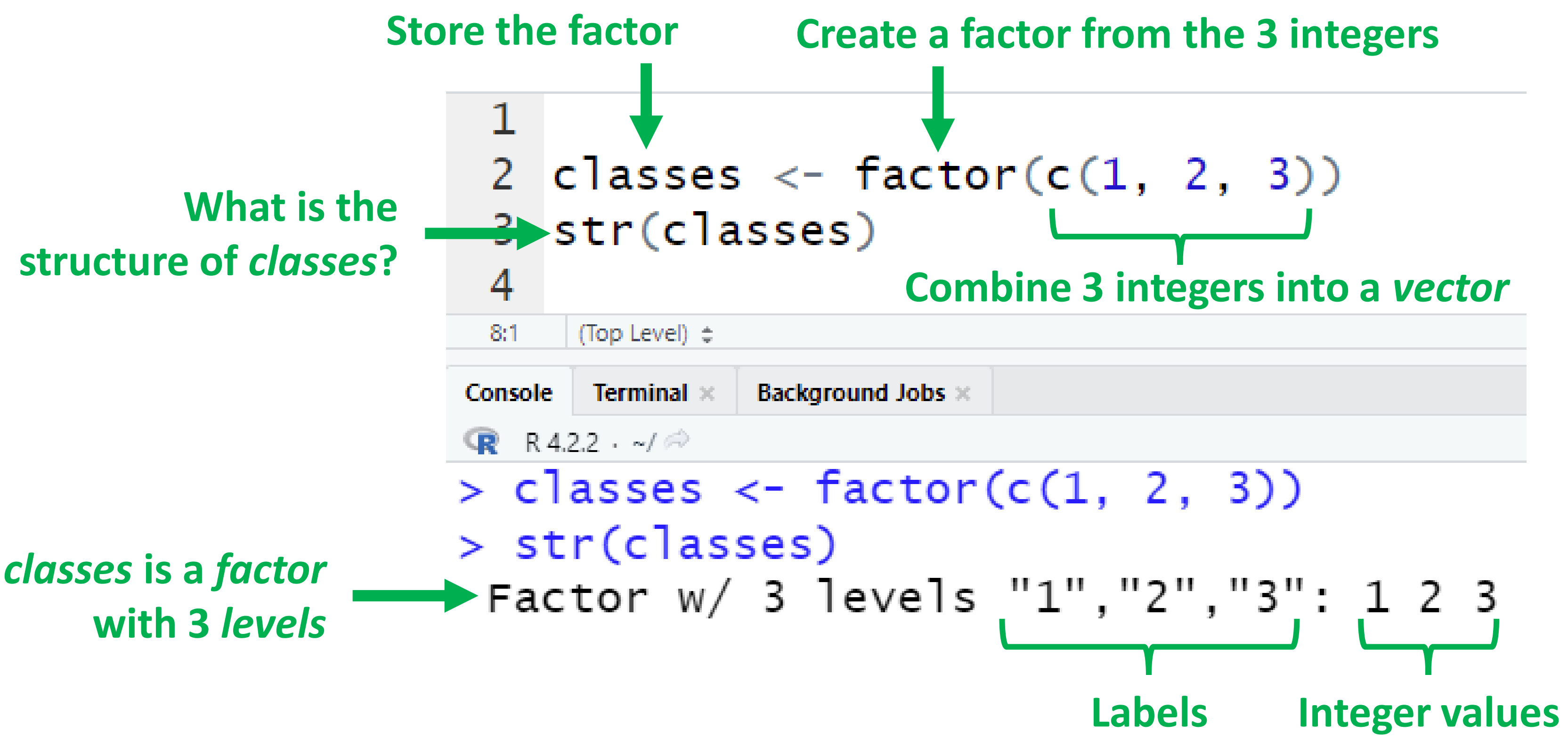
You create factors in R using the *factor* function.

Behind the scenes, the *factor* function creates an integer vector where each integer value receives a label.

Here's how it works when creating a factor from *character* data…



Creating Numeric Factors



Ordering Factors Levels

By default, the *factor* function orders *levels* using an alphanumeric sort.

The good news is that the most commonly used machine learning algorithms (e.g., random forest) do not care about level order…

What are the *factor levels* of origin?

```r

flights$origin <- factor(flights$origin)
str(flights$origin)
levels(flights$origin)

```

4:21 (Top Level)

Console Terminal Background Jobs

R 4.2.2 · ~/

```
> flights$origin <- factor(flights$origin)
> str(flights$origin)
 Factor w/ 3 levels "EWR","JFK","LGA": 1 3 2 2 3 1
> levels(flights$origin)
[1] "EWR" "JFK" "LGA"
```

Default - Labels in alphanumeric order

Ordering Factors Levels

However, it is common to order factor levels based on domain knowledge…

```r
flights$origin <- factor(flights$origin,
                         levels = c("LGA", "JFK", "EWR"))
str(flights$origin)
levels(flights$origin)
```

Specifying custom ordering

8:1 (Top Level)

Console Terminal Background Jobs

R 4.2.2 · ~/

```
> flights$origin <- factor(flights$origin,
+                          levels = c("LGA", "JFK", "EWR"))
> str(flights$origin)
 Factor w/ 3 levels "LGA","JFK","EWR": 3 1 2 2 1 3 3 1 2 1 ...
> levels(flights$origin)
[1] "LGA" "JFK" "EWR"
```

Factor level in descending order

Renaming Factors Levels

As discussed previously, categorical data is often misrepresented as numeric features.

In these cases, it is common to rename the factor levels to something more human-friendly….

List the unique values of *month*

Chronological list of month abbreviations

Use the *month* feature to grab the month abbreviation

Order abbreviations chronologically

```r

unique(flights$month)
month.abb

flights$month_abb <- factor(month.abb[flights$month],
                            levels = month.abb)
str(flights$month_abb)
levels(flights$month_abb)

```

```
11:1    (Top Level) ÷
Console   Terminal ×   Background Jobs ×
R 4.2.2 · ~/
> unique(flights$month)
 [1]  1 10 11 12  2  3  4  5  6  7  8  9
> month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
>
> flights$month_abb <- factor(month.abb[flights$month],
+                             levels = month.abb)
> str(flights$month_abb)
 Factor w/ 12 levels "Jan","Feb","Mar",..: 1 1 1 1 1 1 1 1 1 1 ...
> levels(flights$month_abb)
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

In chronological order

Syncing Factors Levels

A common cause of errors in machine learning with R is when the factor levels of the training dataset do not match the factor levels of the test dataset.

The solution to this problem is to define level ordering explicitly with the training dataset and ensure the test dataset uses this ordering…

```r

class_levels <- c("First", "Second", "Third", "Fourth")

train_classes <- factor(c("First", "Second", "Third", "Fourth"),
                        levels = class_levels)

test_classes <- factor(c("First", "Second", "Third"),
                       levels = class_levels)
str(test_classes)
levels(test_classes)

```

15:1 (Top Level)

Console Terminal Background Jobs

R 4.2.2 · ~/

```
> str(test_classes)
 Factor w/ 4 levels "First","Second",..: 1 2 3
> levels(test_classes)
[1] "First"  "Second" "Third"  "Fourth"
```

Define level ordering

Use level ordering

It's OK if the test dataset does not contain "Fourth"

TUESDAY February 14
Hands-On: Visual Data Analysis with R **NEW!**

WEDNESDAY February 15
Hands-On: Machine Learning Made Easy—No, Really!

THURSDAY February 16
Hands-On: Data Wrangling for Machine Learning **NEW!**

FRIDAY February 17
TDWI Data Visualization Principles and Practices

DAVID LANGER

Founder
Dave on Data

tdwi
LAS VEGAS