Quantitative Methods in Political Economy

Seminar 3: Transforming and combining data

Friday 13 November 2026

Wouter Leenders

wouter.leenders@kcl.ac.uk

3.1 Introduction

From visualisation to transformation

In the last seminar, we focused on visualising data. In practice, we rarely find data that is already in the exact shape required for plotting. Usually, we first need to transform data before we can analyse or visualise it.

Today we use `dplyr` and the `flights` data in `nycflights13`, following *R for Data Science*, second edition.

- Sections 3.1–3.5: rows, columns, pipes and groups.
- Sections 19.1–19.3: keys and joins.

Start a new script and load the packages

Let's open a new R script with **File** → **New File** → **R Script**. Save it as `seminar_3.R`.

```
# Install once, if needed:
# install.packages(c("dplyr", "nycflights13"))

# Load in each new R session:
library(dplyr)
library(nycflights13)
```

Recall: we can run a line by

- selecting the line, or
- placing the cursor in the line

then pressing Ctrl/Cmd+Enter or clicking "Run".

What appears when dplyr loads?

What does the Console show after `library(dplyr)` was run?

```
Attaching package:  'dplyr'
The following objects are masked from 'package:stats':
  filter, lag
The following objects are masked from 'package:base':
  intersect, setdiff, setequal, union
```

This is not an error. The message tells you that specific functions from other packages have been superseded by functions with the same name in `dplyr`: `filter()` now refers to `dplyr`'s function.

If you want to use a function from a specific package, we can write `dplyr::filter()` or `stats::filter()` to avoid any ambiguity.

What does flights look like?

```
flights
glimpse(flights)
# View(flights) opens the data viewer in RStudio.
```

336,776 rows and 19 columns. Each row corresponds to a flight scheduled to leave New York (JFK, LGA or EWR) in 2013.

month	day	dep_time	dep_delay	arr_delay	carrier
1	1	517	2	11	UA
1	1	533	4	20	UA
1	1	542	2	33	AA

How to use dplyr functions

The first argument of any dplyr function is a data frame. Subsequent arguments often describe which columns to apply the function to. The output is always a new data frame.

```
filter(flights, dest == "SFO")
flights |> filter(dest == "SFO")
```

These commands give the same 13,331 flights to San Francisco (SFO). The pipe |> is how functions are glued together. It takes what is to the left of the pipe and applies what is to its right. We can read the pipe as "then".

You can produce the pipe by typing Cmd/Ctrl+Shift+M.

Recall: if you do not assign an data frame to an object, it does not get saved in your environment.

3.2 Rows

filter()

The `filter()` function selects rows for which a specific condition holds.

```
late_departures <- flights |>
  filter(dep_delay > 120)
late_departures
```

month	day	dep_time	dep_delay	carrier	flight
1	1	848	853	MQ	3944
1	1	957	144	UA	856
1	1	1114	134	UA	1086

9,723 flights left more than two hours late.

`filter()` keeps rows where the condition is `TRUE`.

Comparisons: >, >=, <, <=, ==, !=.

Combine conditions: and, or, membership

```
jan1 <- flights |> filter(month == 1 & day == 1)
jan1_comma <- flights |> filter(month == 1, day == 1)
jan_feb <- flights |> filter(month == 1 | month == 2)
jan_feb_in <- flights |> filter(month %in% c(1, 2))
```

- `&` and commas both mean **and**.
- `|` means **or**.
- `%in%` tests membership in a set.

Missing values

```
missing_departures <- flights |> filter(is.na(dep_delay))
observed_departures <- flights |> filter(!is.na(dep_delay))
nrow(missing_departures)
nrow(observed_departures)
```

`is.na()` asks whether a value is missing; `!` means "not".

`dep_delay == NA` does not test if a value is missing.

arrange()

The `arrange()` function puts rows in order.

```
chronological <- flights |>
  arrange(year, month, day, dep_time)
longest_delays <- flights |>
  arrange(desc(dep_delay))
```

month	day	carrier	flight	dep_delay
1	9	HA	51	1301
6	15	MQ	3535	1137
1	10	MQ	3695	1126

`desc()` reverses the usual increasing order.

Additional sorting variables break ties. Missing values are placed last, including with `desc()`.

distinct()

The `distinct()` function selects all unique combinations. It selects the first occurrence of a unique combination.

```
unique_flights <- flights |> distinct()
routes <- flights |> distinct(origin, dest)
route_examples <- flights |>
  distinct(origin, dest, .keep_all = TRUE)
```

origin	dest
EWR	SFO
JFK	SFO
JFK	MIA
JFK	BQN

Using the argument `.keep_all = TRUE` retains all its columns.

count()

The `count()` function counts how often a combination occurs.

```
route_counts <- flights |>
  count(origin, dest, sort = TRUE)
route_counts
```

origin	dest	n
JFK	LAX	11262
LGA	ATL	10263
LGA	ORD	8857
JFK	SFO	8204

`sort = TRUE` puts the largest counts first.

Exercise 1: rows

1. Find flights arriving at least 120 minutes late.
2. Keep flights to either Bay Area airport: San Francisco (`SFO`) or Oakland (`OAK`).
3. Find flights arriving more than 120 minutes late despite leaving on time or early.
4. Sort flights by decreasing departure delay. Which flight is first?

3.3 Columns

mutate()

The `mutate()` function creates new variables.

```
flights |>
  mutate(gain = dep_delay - arr_delay,
         speed = distance / air_time * 60)

flights |>
  mutate(gain = dep_delay - arr_delay, .before = 1)

flights |>
  mutate(gain = dep_delay - arr_delay,
         hours = air_time / 60,
         gain_per_hour = gain / hours,
         .keep = "used")
```

By default new columns go at the end. `.before = 1` puts `gain` first; `.after = day` would put it after `day`.

`.keep = "used"` keeps the inputs and newly calculated columns only.

select()

The `select()` function selects columns

```
flights |> select(year, month, day)
flights |> select(year:day)
flights |> select(!year:day)
flights |> select(ends_with("delay"))
flights |> select(where(is.character))
```

- `:` selects a range of columns; `!` excludes it.
- `ends_with("delay")` selects columns that end with "delay".
- `where(is.character)` selects columns containing text.

Other name helpers: `starts_with()`, `contains()`, `num_range()`.

rename() and relocate()

```
flights |> rename(tail_num = tailnum)
flights |> select(tail_num = tailnum)

flights |> relocate(time_hour, air_time)
flights |> relocate(starts_with("arr"), .before = dep_time)
```

`rename(new = old)` changes a column's name and keeps all other columns.
`select(new = old)` renames while keeping only the selected columns.

`relocate()` changes column order. By default it moves columns to the front;
`.before` or `.after` sets another position.

Exercise 2: columns

1. Print the flights dataset, only showing `dep_time`, `dep_delay`, `arr_time` and `arr_delay`. Can you do this using name helpers?
2. Rename `air_time` to `air_time_min`, move it to the front, and save the result as `flights_minutes`.
3. Explain why selecting only `tailnum` and then trying to sort by `arr_delay` fails. How could you repair the order of the steps?

3.4 The pipe

Find the fastest flights to SFO

Suppose we want to find the fastest flights to SFO. We could do the following:

```
flights1 <- filter(flights, dest == "SFO")
flights2 <- mutate(flights1, speed = distance / air_time * 60)
flights3 <- select(flights2, year:day, dep_time,
                   carrier, flight, speed)
fastest_sfo_steps <- arrange(flights3, desc(speed))
```

We created four objects to obtain what we are interested in. Instead, we could use the pipe to "glue" functions consecutive actions:

```
fastest_sfo <- flights |>
  filter(dest == "SFO") |>
  mutate(speed = distance / air_time * 60) |>
  select(year:day, dep_time, carrier, flight, speed) |>
  arrange(desc(speed))
```

3.5 Groups

group_by()

The `group_by` function tells R which rows belong together

```
by_month <- flights |> group_by(month)
by_month
```

What changes in the printout?

```
# A tibble:  336,776 x 19
# Groups:  month [12]
```

There are still 336,776 flight rows. Grouping does not in itself alter any of the rows. It only adds instructions for later operations.

summarise()

The `summarise()` function calculates summary statistics, returning one row for each group.

```
flights |>
  summarise(avg_delay = mean(dep_delay))

flights |>
  summarise(avg_delay = mean(dep_delay, na.rm = TRUE))

flights |>
  group_by(month) |>
  summarise(avg_delay = mean(dep_delay, na.rm = TRUE))

flights |>
  group_by(month) |>
  summarise(avg_delay = mean(dep_delay, na.rm = TRUE,
            n_flights = n()))
```

slice()

The `slice()` function selects rows within each group

```
by_month |> slice(1:2)
by_month |> slice_head(n = 1)
by_month |> slice_tail(n = 1)
by_month |> slice_min(dep_delay, n = 1)
by_month |> slice_max(dep_delay, n = 1)
by_month |> slice_sample(n = 1)
```

`slice(1:2)` keeps the first two positions in each month.

`slice_head()` and `slice_tail()` take the first or last rows.

`slice_min()` and `slice_max()` choose rows by a variable's values.

`slice_sample(n = 1)` selects one random row per group.

ungroup()

The `ungroup()` function removes groupings from a data.frame. Any subsequent operations are applied to the dataset as a whole rather than separately to each group.

```
overall <- flights |>
  ungroup() |>
  summarize(avg_delay = mean(dep_delay, na.rm = TRUE),
            flights = n())
overall
```

.by

The `.by` argument in `mutate`, `summarise` and `filter` can be used to create groups for a single operation. The groups are removed automatically.

```
flights |>
 group_by(month) |>
 summarize(delay = mean(dep_delay, na.rm = TRUE),
           n = n(), .by = month)

flights |>
 summarize(delay = mean(dep_delay, na.rm = TRUE),
           n = n(), .by = month)
```

Exercise 3: groups

1. Calculate each carrier's mean departure delay and number of flights. Sort from largest to smallest mean. Which carrier comes first?
2. For each destination, select the flight with the greatest departure delay.
3. Calculate flight counts by `origin` and `dest` with `group_by()` and `summarize()`. How does this relate to `count()`?
4. Repeat the carrier summary using `.by`. Does anything change?

19.1 Introduction

Combining tables

The `flights` table gives carrier codes and aircraft identifiers. Their descriptions are stored in other tables, but we can connect those records through a join.

Table	One row represents	Examples of information
`flights`	A scheduled flight	Route, delay, aircraft
`airlines`	An airline	Carrier code and name
`airports`	An airport	Code, name, location
`planes`	An aircraft	Tail number, seats, year built
`weather`	An airport-hour	Temperature and wind

- **Mutating joins** add columns from matching records.
- **Filtering joins** keep or exclude rows based on matches.

Both families match records using **keys**.

19.2 Keys

Primary and compound keys

A **primary key** identifies one row uniquely. A **compound key** uses more than one column to identify rows uniquely.

Table	Primary key	Rows
`airlines`	`carrier`	16
`airports`	`faa`	1,458
`planes`	`tailnum`	3,322
`weather`	`origin`, `time_hour`	26,115

Foreign keys connect related tables

A **foreign key** is a variable that is a primary key in another table. For example, in the `flights` table, many flight rows may use the same airline or aircraft identifier, but in the `carrier` table, the carrier variable is the primary key.

Key in `flights`	Corresponding key
`carrier`	`airlines$carrier`
`tailnum`	`planes$tailnum`
`origin`	`airports$faa`
`dest`	`airports$faa`
`origin`, `time_hour`	`weather$origin`, `weather$time_hour`

19.3 Basic joins

Match flights to airlines: the inputs

```
flights2 <- flights |>
  select(year, time_hour, origin, dest, tailnum, carrier)
```

Suppose we want to attach the carrier name to our flights data.

Flight rows

tailnum	carrier
N24224	UA
N510UA	UA
N619AA	AA

Airline rows

carrier	name
AA	American Airlines Inc.
UA	United Air Lines Inc.

The common key is `carrier`.

left_join()

The `left_join()` function keeps all rows from the left table and adds columns from matching rows in the right table.

```
flights_airlines <- flights2 |>
  left_join(airlines, join_by(carrier))
```

tailnum	carrier	name
N24224	UA	United Air Lines Inc.
N510UA	UA	United Air Lines Inc.
N619AA	AA	American Airlines Inc.

`join_by(carrier)` matches equal carrier codes. Two UA flights use the same UA airline record.

Add weather or aircraft information

```
flight_weather <- flights2 |>
  left_join(select(weather, origin, time_hour, temp, wind_speed),
            join_by(origin, time_hour))

flight_planes <- flights2 |>
  left_join(select(planes, tailnum, type, engines, seats),
            join_by(tailnum))
```

Weather matching needs **both** origin and scheduled hour. Aircraft matching needs `tailnum`.

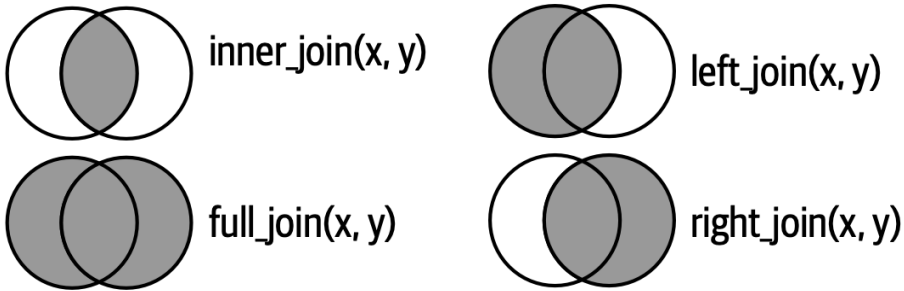
When the key names differ

```
destination_airports <- flights2 |>
  left_join(airports, join_by(dest == faa))

origin_airports <- flights2 |>
  left_join(airports, join_by(origin == faa))
```

origin	dest	name
EWR	SFO	San Francisco Intl
JFK	SFO	San Francisco Intl
JFK	MIA	Miami Intl

Four mutating joins: which rows remain?



Two small input tables

```
x <- tibble(key = c(1, 2, 3), val_x = c("x1", "x2", "x3"))
y <- tibble(key = c(1, 2, 4), val_y = c("y1", "y2", "y3"))
```

x

key	val_x
1	x1
2	x2
3	x3

y

key	val_y
1	y1
2	y2
4	y3

Keys 1 and 2 occur in both tables. Key 3 occurs only in `x`; key 4 only in `y`. Both tables have unique keys.

Compare left and inner joins

```
left_result <- left_join(x, y, join_by(key))
inner_result <- inner_join(x, y, join_by(key))
```

left_join: 3 rows

key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	*NA*

inner_join: 2 rows

key	val_x	val_y
1	x1	y1
2	x2	y2

The left join keeps key 3 and fills its missing `val_y` with `NA`. The inner join keeps only keys 1 and 2.

Compare right and full joins

```
right_result <- right_join(x, y, join_by(key))
full_result <- full_join(x, y, join_by(key))
```

right_join: 3 rows

key	val_x	val_y
1	x1	y1
2	x2	y2
4	*NA*	y3

full_join: 4 rows

key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	*NA*
4	*NA*	y3

The right join keeps key 4 from `y`; its `val_x` is missing. The full join keeps both unmatched keys, 3 and 4.

semi_join(): keep rows with a match

```
used_origins <- airports |>
  semi_join(flights2, join_by(faa == origin))
used_destinations <- airports |>
  semi_join(flights2, join_by(faa == dest))
```

faa	name
EWR	Newark Liberty Intl
JFK	John F Kennedy Intl
LGA	La Guardia

`used_origins` has three airport rows. `used_destinations` has 101. Only columns from `airports` remain.

anti_join(): find rows without a match

```
unmatched_dest <- flights2 |>
  anti_join(airports, join_by(dest == faa)) |>
  distinct(dest)
unmatched_tail <- flights2 |>
  anti_join(planes, join_by(tailnum)) |>
  distinct(tailnum)
```

dest
BQN
SJU
STT
PSE

Four destination codes have no airport record. The second result contains 722 distinct tail-number entries, including `NA`. Filtering joins retain only the left-hand columns.

Exercise 4: basic joins

1. Find the ten destinations with the most flight rows using `count(dest, sort = TRUE)` and `head(10)`. Use a filtering join to keep all flights to them.
2. Use an anti-join to check whether every flight has a matching airport-hour in `weather`. Which two columns form the key?
3. Calculate mean arrival delay by destination, keeping counts of rows and observed delays. Join the airport name using an explicit key. Check the row count.
4. Explain which flights an inner join to `planes` would remove, compared with a left join.