R Ladies NYC
November 2019

Simplified Data Quality Monitoring of ***Dynamic Longitudinal Data***

A Functional Programming Approach

Jacqueline Gutman, M.S.
Flatiron Health
Quantitative Sciences

flatiron

Outline

- Introduction to Flatiron
- Our data quality problem + how we handle it
- What is functional programming + how is it relevant in data science?
- 5 key benefits of functional approaches to data science
  - Readability
  - Compositionality
  - Reproducibility
  - Robustness
  - Efficiency

flatiron

Jacqueline Gutman

Data Scientist, Flatiron Health

```
Jackie %>%
    learn( Psych + Machine Learning)%>%
    study( NYU Center for Data Science)%>%
    intern( Data Science for Social Good)%>%
    work( NYU School of Medicine)%>%
    work( Plated/Albertsons)%>%
    work( Flatiron Health)
```

flatiron

OUR MISSION

To improve lives by learning from the experience of every cancer patient.

Founded	2012
Employees	~900 full-time employees 1,000+ part-time abstractors
Clinicians	12 medical oncologists 85+ total clinicians
Technical staff	350+ in software, IT, security, data insights & medical informatics
Quantitative scientists	~50 biostatisticians, epidemiologists & data scientists
Offices	New York (HQ) San Francisco

flatiron

Over 2.2 million

patient records available for research.

Over 15 of the top

oncology-pharma companies are members of our research network.

800 sites of care

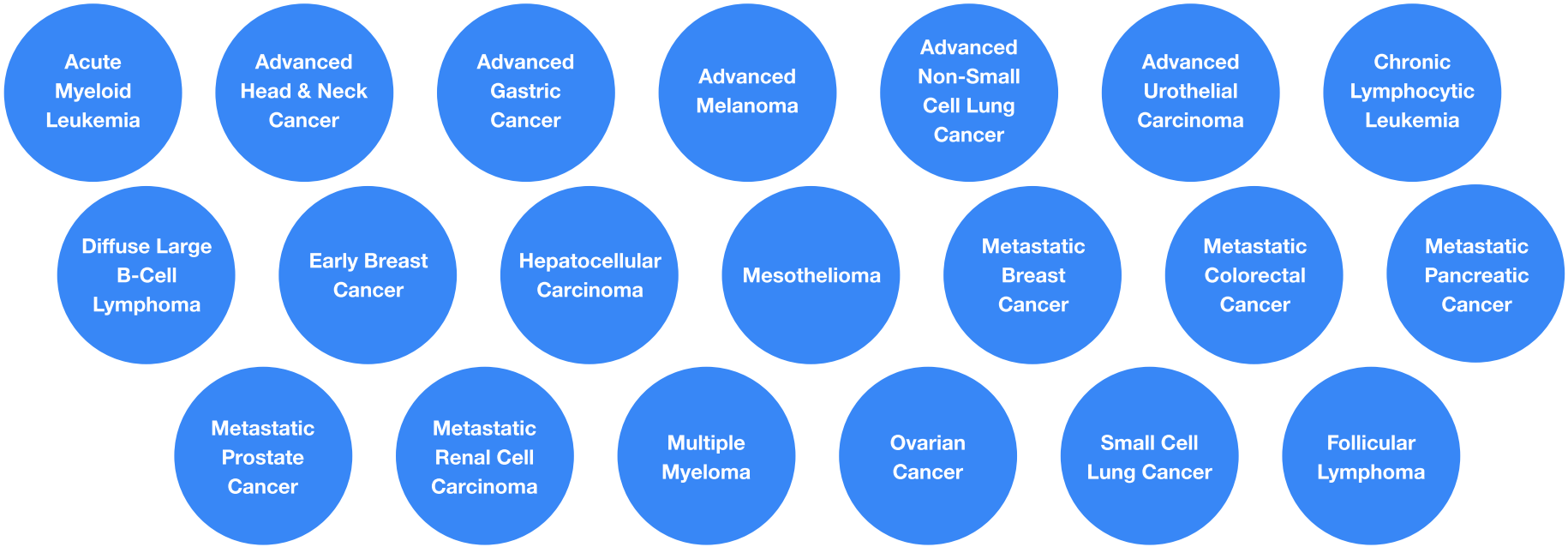
use Flatiron's OncoCloud™ software.

7 academic centers

partner with Flatiron on outcomes research and quality improvement.

flatiron

National, longitudinal real-world datasets refreshed monthly, with 30-day recency



flatiron

Our challenge:
Identify issues of data quality in ***dynamic longitudinal data***

What this entails

- New data elements collected and integrated each month from disparate sources
- Ensure quality by monitoring unusual changes within and across variables + patients
- Raise actionable and interpretable alerts when changes fall outside accepted bounds
- Harness methodological and statistical expertise with domain-specific clinical intuition to reason about many datasets simultaneously

flatiron

How we approach our problem

OBJECTIVES

- ❏ Codify statistical and domain knowledge in a library of composable, modular functions
- ❏ Decouple the high-level checks and reasoning from the low-level implementation details
- ❏ Extend reasoning about a single dataset to apply across an expanding number of datasets

BENEFITS

- ★ Reproducible and documented best practices are easily shared
- ★ Abstraction reduces the cognitive complexity of an analysis and makes reasoning more transparent
- ★ Easier to scale analysis not just within a dataset, but as the number of datasets and elements increase

Core principles of functional programming

- ➔ Functions!
- ➔ that can be composed into higher-order functions *(compositionality)*
- ➔ which abstract out what is being done *(declarative)* from how it will be carried out *(imperative)*
- ➔ and which avoid side-effects and external state dependencies *(immutability)*

Building blocks of our approach to data quality

NESTED DATA STRUCTURES

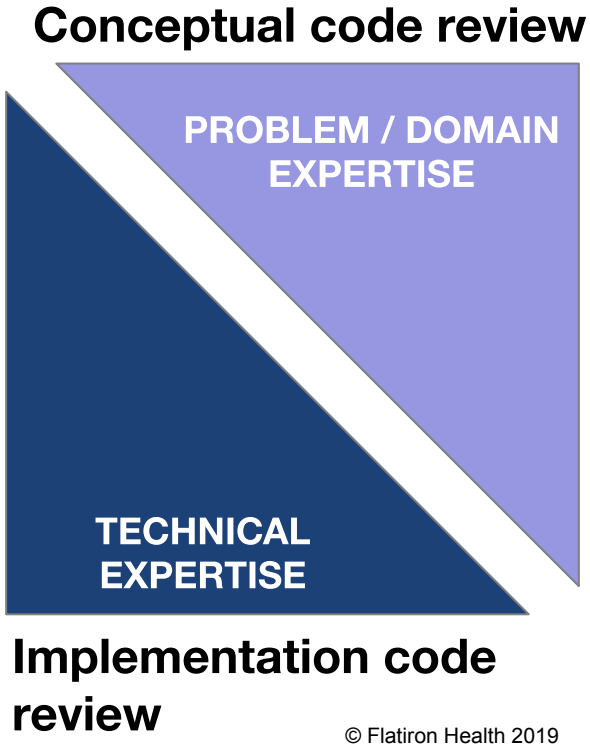
DataFrames for your DataFrames!

LIBRARY DEVELOPMENT

Promote visibility, unit testing and documentation for your functions

DIFFERENTIATED CODE REVIEW

For technical and non-technical project team members



Store tibbles, lists, and in-database lazy tbls as columns in a nested data_frame

Data + metadata + evaluation metrics + summary statistics = FRIENDS FOREVER

```
# A tibble: 40 x 6
   schema                  disease_prefix      table_name   lazy_tbl   raw_data             patient_list
   <chr>                   <chr>               <chr>        <list>     <list>               <list>
 1 market_tracking_20190131 met_breast_oral_lot demographics <tb_PSQLC> <tibble [100 × 9]>  <chr [18,754]>
 2 market_tracking_20190131 met_breast_oral_lot drugepisode  <tb_PSQLC> <tibble [100 × 14]> <chr [16,514]>
 3 market_tracking_20190131 nsclc_oral_lot      demographics <tb_PSQLC> <tibble [100 × 9]>  <chr [52,551]>
 4 market_tracking_20190131 nsclc_oral_lot      drugepisode  <tb_PSQLC> <tibble [100 × 14]> <chr [37,079]>
 5 market_tracking_20190228 met_breast_oral_lot demographics <tb_PSQLC> <tibble [100 × 9]>  <chr [18,961]>
 6 market_tracking_20190228 met_breast_oral_lot drugepisode  <tb_PSQLC> <tibble [100 × 14]> <chr [16,690]>
 7 market_tracking_20190228 nsclc_oral_lot      demographics <tb_PSQLC> <tibble [100 × 9]>  <chr [53,185]>
 8 market_tracking_20190228 nsclc_oral_lot      drugepisode  <tb_PSQLC> <tibble [100 × 14]> <chr [37,588]>
 9 market_tracking_20190331 met_breast_oral_lot demographics <tb_PSQLC> <tibble [100 × 9]>  <chr [19,156]>
10 market_tracking_20190331 met_breast_oral_lot drugepisode  <tb_PSQLC> <tibble [100 × 14]> <chr [16,867]>
# … with 30 more rows
```

Nested data structures for comparing and evaluating multiple datasets

database	delivery_date	data_cutoff	schema	disease_prefix	table_name	n_records	n_patients
monthly	2019-01-31	2019-01-01	market_tracking_20190131	met_breast_oral_lot	demographics	18754	18754
monthly	2019-01-31	2019-01-01	market_tracking_20190131	met_breast_oral_lot	drugepisode	438230	16514
monthly	2019-01-31	2019-01-01	market_tracking_20190131	nsclc_oral_lot	demographics	52551	52551
monthly	2019-01-31	2019-01-01	market_tracking_20190131	nsclc_oral_lot	drugepisode	698906	37079
monthly	2019-02-28	2019-02-01	market_tracking_20190228	met_breast_oral_lot	demographics	18961	18961
monthly	2019-02-28	2019-02-01	market_tracking_20190228	met_breast_oral_lot	drugepisode	446614	16690
monthly	2019-02-28	2019-02-01	market_tracking_20190228	nsclc_oral_lot	demographics	53185	53185
monthly	2019-02-28	2019-02-01	market_tracking_20190228	nsclc_oral_lot	drugepisode	711007	37588
monthly	2019-03-31	2019-03-01	market_tracking_20190331	met_breast_oral_lot	demographics	19156	19156
monthly	2019-03-31	2019-03-01	market_tracking_20190331	met_breast_oral_lot	drugepisode	455583	16867
monthly	2019-03-31	2019-03-01	market_tracking_20190331	nsclc_oral_lot	demographics	53591	53591
monthly	2019-03-31	2019-03-01	market_tracking_20190331	nsclc_oral_lot	drugepisode	723911	38088

Previous 1 2 3 4 Next

Treat individual datasets as single observations

Simple function:
1 dataset → 1 evaluation metric

Higher order functions:
Multiple datasets → composite evaluation metrics across several dimensions

Nested data structures for comparing and evaluating multiple datasets

```
count_records <- purrr::partial(execute_query,
                                select = "COUNT(*)")

count_patients <- purrr::partial(execute_query,
                                 select = "COUNT(distinct patientid)")

fetch_patients = purrr::partial(execute_query,
                                select = "DISTINCT patientid",
                                limit = 100)

data %<>%
    mutate(n_records = purrr::map(lazy_tbl, count_records, con = conn),
           n_patients = purrr::map(lazy_tbl, count_patients, con = conn),
           cohort = purrr::map(lazy_tbl, fetch_patients, con = conn))
```

What does all this get you?

- ★ Readability
- ★ Compositionality
- ★ Reproducibility
- ★ Robustness
- ★ Efficiency

Readability
Abstraction and Declarative intent

Analysis code can be **understood** and **sanity-checked** by non-technical staff

Clarity of intent: abstract declarative code is self-documenting

Docstrings and unit tests to align implementation with intention

```
compare_snapshots(
    january_data,
    february_data,
    fn = compute_time_to_dx)

find_added_patients(
  old_data =
    january_data$cohort,
  new_data =
    february_data$cohort)
```

Compositionality

Reasoning with higher level functions

```
filter_persistent_patients <-
    function(data) {
    ...
}
compute_change_from_prev <-
    function(data, var, prev_var) {
    ...
}

summarize_change_distribution <-
    function(data, change_var) {
    ...
}
```

```
process_drug_episodes <- . %>%
  filter_persistent_patients %>%
  compute_change_from_prev(
    var = "num_drug_episodes",
    prev_var = "prev_drug_episodes") %>%
  summarize_change_distribution(
    change_var = "change_num_drug_episodes")

data_processed <- process_drug_episodes(data)
```

Compositionality

Reasoning with higher level functions

```
lm(Sepal.Length ~ Species,
   data = iris) %>%
broom::tidy(
   conf.int = TRUE) %>%
filter(p.value < 0.05) %>%
arrange(desc(statistic))
```

```
tidy_lm <- purrr::compose(
    lm,
    ~ broom::tidy(.x,
        conf.int = TRUE),
    ~ filter(.x,
        p.value < 0.05),
    ~ arrange(.x,
        desc(statistic)),
    .dir = "forward")

tidy_lm(Sepal.Length ~ Species,
    data = iris)
```

flatiron

Compositionality

Reasoning with higher level functions

```
cohort %<>%
    left_join(demographics) %>%
    left_join(mortality) %>%
    left_join(drugepisode) %>%
    left_join(progression)

tbls <- list(demographics,
             mortality,
             drugepisode,
             progression)

cohort %>%
   purrr::reduce(tbls,
          left_join, .init = .) ->
     cohort_plus
```

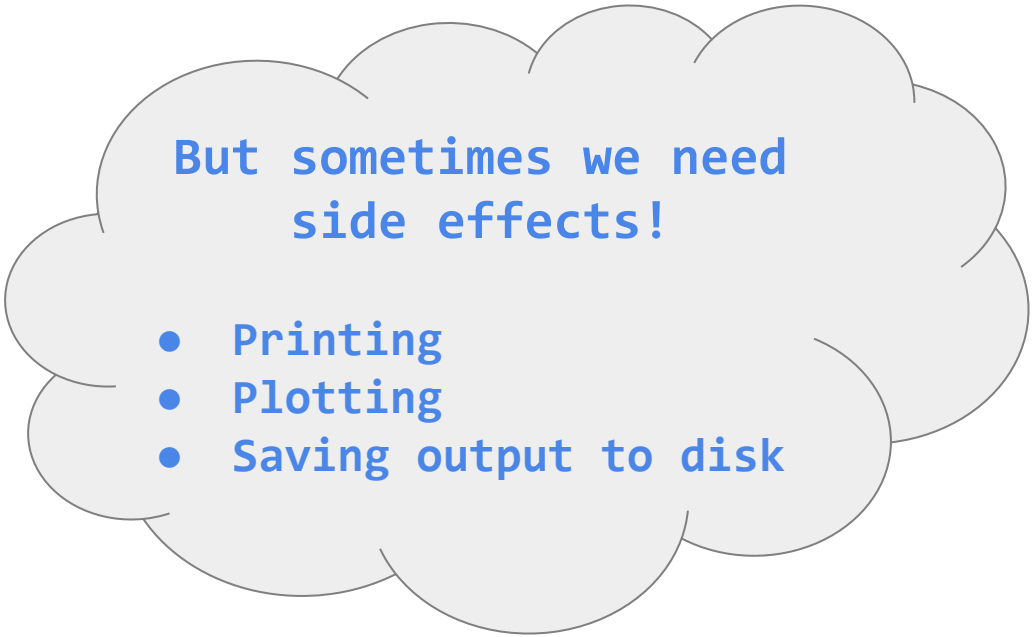
With a nested data frame:

```
all_data %>%
     mutate(cohort_plus =
          reduce(list(demographics,
                      mortality,
                      drugepisode,
                      progression),
                 .f = left_join,
                .init = cohort))
```

Reproducibility

Avoiding side effects and external states

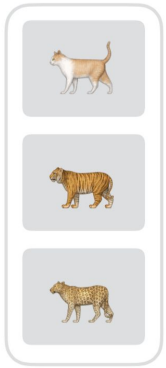
- Running the same function with the same input should always produce the same result
- Safeguard reproducibility with
  - Immutable inputs + outputs
  - Idempotent processes
  - Deterministic algorithms



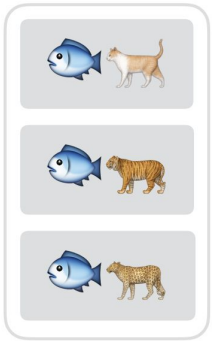
Reproducibility

Clearly separate pure functions from functions desired for their side effects

to each `cat` apply `give_fish`

`map(`  `, give_fish)`

OUTPUT



No side effects!

to each `cat` apply `love`

`walk(`  `, love )`

Side effects

Robustness
Testability and elegant error handling

- Functional modular code makes for happy unit testing
  - Avoid difficult to test code stemming from **Mutability, Side-Effects**, **Responsibility overload**, and **Procedural instructions**
- Avoid side effects and maintain your functional flow
  - try-catch exception blocks can be refactored as higher order functions

```
error_prone_fn <- function(data) {
    ...
}

safe_fn <- purrr::safely(
    .f = error_prone_fn,
    otherwise = c())

data %>%
    mutate(results = purrr::map(
        input, safe_fn))
```

Efficiency
Laziness and caching

```
parse_dateofdeath_chunked <- purrr::partial(do,
    data_modified = mutate(., dateofdeath = parse_dod(a, b, c)),
    .chunk_size = 1000L)

parse_dateofdeath_chunked(mortality)
```

Dbplyr backend for databases

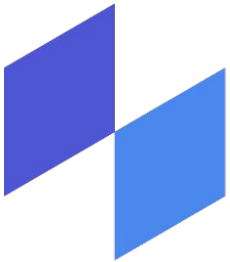
Dtplyr backend for data.tables

Efficiency
Laziness and caching

- Use when evaluating a function repeatedly over the rows (or larger chunks) of a dataset, and expect to regularly get the same input
- Performance gains depend on how often a function is being called with same argument(s)

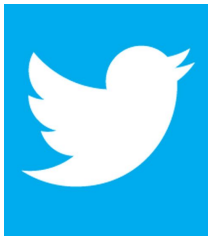
```
compute_biomarker_status <- function(
    biomarker_results){
    ...
}

compute_biomarker_status_cached <-
    memoise::memoise(compute_biomarker_status)

data %<>%
  mutate(biomarker_status =
    purrr::map(biomarker_results,
        compute_biomarker_status_cached)
```

Thanks!



flatiron.com/blog/

@jgutman



@dynamicdataduo