Empowering Data Science through Code Modernization: Bridging the Gap between Innovation and Efficiency

Nathan Barrett, Yunie Le, Ekaterina Levitskaya, Roy McKenzie

The Coleridge Initiative

COLERIDGE INITIATIVE


Survey Data
Analytical Pipeline
CODE
Research Products

Bad code is bad news for research

- At worst:
  - Data systems that misrepresent the raw data
  - Analyses that are wrong
- At best:
  - Work that is not transparent and difficult to reproduce
  - Data decisions which are unclear for end users
  - Data products that require huge amounts of manual labor to update
  - Legacy code that cannot be maintained

Three keys to addressing this issue

- Tools

  We must supply the tools analysts need to adopt reproducible principles.
  We must help analysts to re-use each other's work.
  Our technology platforms for analysis must enable reproducible analysis.

- Capability

  Analysts must have the right skills to implement high quality analysis.
  Managers and leaders of analysis must be confident managing analytical software.
  Organisations must be able to recruit the right people to develop and use reproducible analysis principles.

- Culture

  Analysts must feel encouraged and supported to develop analysis products with reproducible principles.
  Our culture must demand high-quality analysis. Our leaders and users must encourage continuous improvement.
  We must work in multidisciplinary teams to deliver the most valuable analysis.

Source: Reproducible Analytical Pipelines (RAP) strategy (2022)
https://analysisfunction.civilservice.gov.uk/policy-store/reproducible-analytical-pipelines-strategy/

A case study: ARMS survey data

- Annual agricultural survey dataset jointly produced by USDA ERS/NASS
  - Currently produced by a limited number of long SAS scripts
- USDA ERS partnered with Coleridge to update their processing code
  - Baseline: Switch from SAS to R
- The general approach: modular code

A case study: ARMS survey data

main.sas

We start with the current processing code - a long script with thousands of lines

We break this into functions for each variable

HH_SIZE.R

```
HH_SIZE <- function()
```

FARM_TYPOL.R

```
FARM_TYPOL  <- function()
```

These small scripts are combined in larger scripts

classification.R

```
output  <- input %>%
      HH_SIZE() %>%
      FARM_TYPOL()
```

Benefits of modular code: testing

Unit testing

- Variable function is the smallest unit of the code
- Incorporate tests that check the function performs as intended, i.e. confirm output of the function with the existing values if available
- Catches errors in code

Warning system

- Flag if manual review is needed
- Catches errors in underlying data

HH_SIZE.R

```
HH_SIZE <- function()
```

test_HH_SIZE.R

```
print (Output value == 2021 value)
TRUE, FALSE
```

Benefits of modular code: documentation

- `roxygen` style code documentation combined with modular code allows for variable documentation to be written within the code
- This leads to documentation that is:
  - More accurate
  - Easier to maintain
  - Decentralized

```
#' Value of production off farm
#'
#' @de
#' Thi
#'
#' @se
#' Thi
#' the
#'
#'
#' @se
#' -
#'
#' @ex
V1B <-
V1B
  df
}
```

V1B *Value of production off farm*

Description

This includes only the production that occurs not on the farm

Usage

```
V1B(df)
```

Formula

This variable captures the value of production that occurs on the farm (P1002).

Input variables

- P1002

Back to the three keys

- Tools
  - SAS to R: moving from proprietary, legacy software to open source tools undergoing active training and development
  - Modular code is easier to maintain than long, single-file scripts
- Capacity
  - Previously: faced a single point of knowledge
  - Decentralizing code base allows update responsibility to be shared
- Culture
  - Modular code allows for easier implementation of peer review
  - Shared ownership over code and documentation success

Thank you to Meaghan Smith, Brent Hueth, Carrie Jones, Daniel Milkove, Daniel Ayasse from ERS for the feedback and help with this project