Welcome to Statistics 750!

Christopher R. Genovese

Department of Statistics & Data Science

27 Aug 2024
Session #1

Plan

Introductions

Plan

Plan

Plan

Plan

Announcements

- Please fill out **office hours poll**. (Treat dates as generic days of week.) https://www.when2meet.com/?26114557-Bhj4w
- Special Office Hour times this week will be posted on Canvas. (Canvas is used mostly for announcements and some documents in special cases.)
- Email subject: [750]
- **Please bring your laptop to every class**
- **Reading**:
  - Thursday: System setup, F.9 intro and F.9.1
  - `https://36-750.github.io/course-info/system-setup/`
  - Thinking Languages document coming soon
- **Homework**: Get a github account; send me your username in email. Sliding Windows activity, part of HW #1, due Thursday 5 Sep.
- Github invitations coming shortly, look for them.

Plan

Plan

Motivation

- **Modern data analysis can be a complex business**
  Creating good software to manage this complexity has become an essential skill for statisticians.
- **Computing is taught *at the margins* in most statistics curricula**
  - Typical statistical computing courses focus on the details of methods and algorithms for various concrete problems.
  - Students are expected to learn the practice of computing and software engineering organically during their research.
  - Typical feedback and incentives can obscure the benefits of building good software.
- **The organic development cycle for statistical software can limit correctness, clarity, and reusability.**

Motivation (cont'd)

- **Building efficient, elegant, reusable software increases our productivity and effectiveness**

  Good software engineering emphasizes:
  - Managing complexity
  - Communicating clearly
  - Finding effective abstractions
  - Crafting well chosen solutions to problems
  - Obtaining good performance, reuse, and generalizability
- **Programming well is lots of fun**
- **Deep ideas can lead you to think about problems (math and stat not just computing) in a *new way*.**

  Exs: Algebraic Data Types, Categories, "Proof assistants"

Core Axioms

- A broad, firm foundation in computing will pay off throughout your career
- The way to get better at programming is to **practice** programming
- Software design and programming practice are skills every statistician needs
- Revision is a critical part of the development process
- Having (at least a passing) understanding of multiple languages will make you a better programmer
- At the intersection of computing/mathematics/statistics are deep ideas and perspectives that apply far beyond computing

Key Skills and Goals for *Practice*

- Good programming practice
- Efficient workflows
- Effective software design
- Choosing good representations (data structures, algorithms, designs)
- Establishing Efficiency and Correctness

By the end of this course, you should be able to:

- develop correct, well-structured, and readable code;
- design useful tests at all stages of development;
- effectively use development tools such as editors/IDEs, debuggers, profilers, testing frameworks, and a version control system;
- build a moderate scale software system that is well-designed and that facilitates code reuse and generalization;
- select algorithms and data structures for several common families of statistical and other problems;
- write small programs in a new language beyond R and Python. (Super-power languages: Clojure, Haskell, Rust, Racket; runners up: Ocaml, Scala, Julia).

Plan

Thinking Languages

We will emphasize more saliently the process of ***thinking about*** and ***designing*** our programs. To that end, we will develop in our discussions three **thinking languages** for talking about problems and designing solutions.

1. Thinking Language 1 is focused on using *types* to elegantly represent the concepts and entities that we are working with so and inform our design. It is spare and mathematical and allows us to express a variety of deep ideas. It is inspired by several beautiful (real) languages (Haskell, Idris, Unison) in current use.
2. Thinking Language 2 is an “imperative” pseudo-code for cleanly describing the steps in algorithms. It is relatively loose and informal putting clarity above syntax. It is inspired by scripting languages like Python and Julia.
3. Thinking Language 3 is a diagrammatic language that offers a surprisingly powerful tool for thinking about systems. Variants of this language have been used across a wide spectrum of fields, from quantum physics to operations research. We will develop this primarily later in the semester.

We will abbreviate these as TL1, TL2, and TL3.

Plan

Sliding Window Aggregation: A First Puzzle

Given a sequence of values of some type, we want to accumulate values in *sliding windows* using an associative binary operation and some initial value. The result is a new sequence of values, each of which is the aggregation over one window.

Specifically, we will implement a function `swag` that takes four inputs:

1. An initial value for the aggregation in each window
2. The associative operation
3. A window size
4. The sequence of values to be aggregated.

The function returns a new sequence consisting of successive (complete) windows.

Examples:

`swag "" text.concat 4 ["a", "b", "c", "d", "e", "f", "g", "h"]`
returns `["abcd", "bcde", "cdef", "defg", "efgh"]`

`swag 0 (+) 3 [0, 10, 20, 30, 40, 50]` returns `[30, 60, 90, 120]`

Key requirement: **use only $O(n)$ calls to the operation, independent of window size**, where n is the length of the sequence.

A Taste of TL1: Thinking with Types

How do we describe the function `swag`? We give it a *type*. A TL1 description:

```
swag : a -> (a -> a -> a) -> Nat -> [a] -> [a]
```

Here

- `:` is read “has type”
- `a` is a *type variable* that stands for an unspecified type. We sometimes need/want to put constraints on that type, which we can do.
- `x -> y` specifies a *function type*: the set of functions that take type `x` as input and return type `y` as output.
  `a -> a -> a` is the type of a function that takes two arguments of type `a` and returns a value of type `a`. We’ll see an explanation for this odd notation.
- `Nat` is a concrete type, the type of *natural numbers*.

Initial Design Work

Think about and discuss how you might tackle this problem. Feel free to start with a simple approach; it may be good enough, and it will certainly be revealing. Feel free to start with a special case (e.g., adding integers).

Ask questions, sketch ideas, pseudo-code, code, as you see fit.

It's better with a partner. . .

Some Questions (with Discussion)

- What are some examples of aggregation operations that might be useful in this task? Are they all interchangeable? Why or why not?
- What are some properties that aggregation operations may have? (Consider your examples.)
- How do the properties of the aggregation operation affect your approach?
- What are some ways that we might organize our data to support such aggregation algorithms? What data/entities do we need to represent and keep track of?
- What are some approaches you considered for the problem?
- ...

Some Answers

- Aggregation operations
  - Sum-like: sum, count, average, std dev, . . .
  - Collection: collect list, concatenate strings, *i*th occurrence, . . .
  - Median-like: median, *i*th smallest, quantiles, . . .
  - Max-like: max, min, argMax, max count or feature, . . .
  - Sketch-like: Bloom filter, hyperloglog, . . . (more on these later)
- Properties of aggregation operations
  - Associative
  - Commutative
  - Invertible
  - Result has fixed size
  - Deannotation requires no extra info (see below)

Some Answers (cont'd)

It might be useful to consider a slightly more general type for `swag`

```
swag : a -> (a -> a -> a) -> Nat -> [v] -> [v]
lift : v -> a
lower : a -> v
```

Here, the type variable `v` describes the values of the input data, and the type variable `a` describes *annotated values*. The operations `lift` and `lower` map a value to an annotated value and vice versa. (Note: `swag` actually needs access to `lift` and `lower`, so the type of the second argument is more properly a *tuple* `(a -> a -> a, v -> a, a -> v)` or similar object.)

As an example of annotation, think about how you might compute an average in this context.

The last property listed above says that we can do `lower` without any extra information.

Which of the operations from earlier have which of these properties?

Some Answers (cont'd)

- How do the properties affect our approach?

  For this, it is helpful to think about the *entities* that comprise this problem and the *operations* that act on them.

  For example, consider the "window" as an entity. What do we do to a window?

  - insert an annotated value into the window
  - remove an annotated value from the window

  We can also make a variety of *queries*, such as asking for aggregations from windows in a certain range of "times".

Towards Implementation

With your partners, build an initial implementation based on the ideas we discussed. Feel free to narrow the scope originally.

Does your implementation meet the key requirement? How do you know? Empirical data? Can you prove it?

Ask questions as you work.

Plan

The Benefits of Packaging

Modularity offers many benefits even for small projects.

A good basic practice is to put code for a task into its own package, reproducible and separate from the surrounding environment. It is often useful to have our application and computational code in *separate packages*

- R: `https://r-pkgs.org/`, `https://kbroman.org/pkg_primer/`, and `use_this`
- Python packages are centered around the `__init__.py` file `https://packaging.python.org/en/latest/tutorials/packaging-projects/`

Create a work directory for your SWAG code and set it up as a package. In Python, create an `__init__.py` file; it can be empty. In R:

```
library(usethis)
path <- file.path("pkgname")
create_package(path)
proj_activate(path)
# use_mit_license("My Name")
# use_package("ggplot2", "Suggests")
use_readme_md()
# use_news_md()
use_test("basic-test")
# use_data(x, y)  # saves data as dependencies
use_git()
```

The Benefits of Version Control

We will also make extensive use of version control to maintain the history of our project. If you have installed `git`, create a repository in your project directory.

```
# configuring git
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"

# Visual Studio Code:
git config --global core.editor "code --wait"

# Emacs:
# There are various options here
# A good choice: put (server-start) in your init file and use
git config --global core.editor emacsclient

# Vim:
git config --global core.editor vim

# Also: core.autocrlf and credential.helper  See setup
```

The Benefits of Version Control

```
# Create or move to a working directory
mkdir ~/s750
cd ~/s750

# Initializing a repository
mkdir swag
cd swag     # or   cd ~/s750-lecture/swag
git init
git status

# Workflow: change -> stage -> commit workflow
git status
git diff

git add __file1__ __file2__ ... __filen__
git add .

git status
git commit
git commit -m "A commit message"
```

THE END