

# STAT 215A Fall 2020

## Week 7

---

James Duncan, OH: M, Th 2-4pm

# Announcements

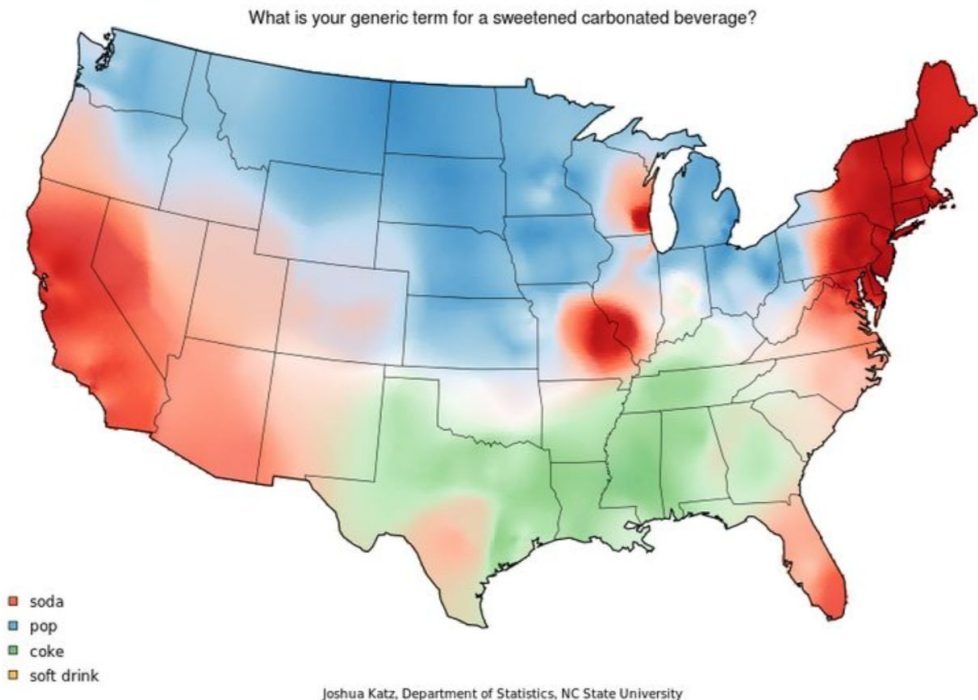
- Discussion section feedback survey results
- Lab 3 will be released on Tuesday 10/13
  - DUE: **10/22 at 11:59pm** (only 10 days!)
- Schedule change: discussion next week will be at usual day/time
- Midterm: 10/29
  - more info / practice midterm to come

# How did Lab 2 go?

- Challenges?
- Likes & dislikes?
- What clustering method did you use?

slido.com

#52734



<https://www.businessinsider.com/22-maps-that-show-the-deepest-linguistic-conflicts-in-america-2013-6#ok-th-is-one-is-crazy-everyone-pronounces-pecan-pie-differently-10>

# Outline for today

- Introduce **Lab 3: Stability and Computability**
- Parallelization
- Statistical Computing Facility (SCF)
- Rcpp

# Lab 3: Stability of K-means and computability

How to choose K using stability:

For each  $k = 2:k_{\max}$

For each  $b = 1:B$

Perturb the data (e.g. bootstrap, subsample)

Run K-means on the perturbed data

Get cluster memberships

Evaluate stability of the  $B$  cluster membership vectors

Choose  $k$  which gives the most stable clusters

**How do we quantify the stability of clusters?**

# Lab 3: Stability of K-means and computability

- Ben-Hur (2002): A stability-based method for discovering structure in clustered data:

---

**Algorithm 1** Calculation of clustering similarities in  $k$ -means

---

```
for  $k = 2$  to  $k_{max}$  do
  for  $i = 1$  to  $N$  do
     $sub_1 = \text{subsample}(X, m)$ , a subsample of fraction  $m$  of dataset  $X$ 
     $sub_2 = \text{subsample}(X, m)$ , a subsample of fraction  $m$  of dataset  $X$ 
     $L_1 = \text{cluster}(sub_1)$ 
     $L_2 = \text{cluster}(sub_2)$ 
     $intersect = sub_1 \cap sub_2$ 
     $S(i, k) = \text{similarity}(L_1(intersect), L_2(intersect))$ 
  end for
end for
```

---

- Similarity metrics:** correlation, Jaccard, matching

# Lab 3: Stability of K-means and computability

## Your objectives:

1. Write efficient code to implement Algorithm 1 and speed up the computations.
2. Evaluate the stability of K-means using the binary-encoded data from Lab 2.

I will look at your code *closely* in this lab, so please be sure to follow an appropriate R style guide:

- <https://style.tidyverse.org/>
- <https://google.github.io/styleguide/Rguide.html>

# How to speed up computation

- Easy:
  - Don't repeatedly re-compute object that only need to be computed once.
  - Don't define or store objects unnecessarily (intermediate variables)
- Other ways:
  - In R:
    - Base R: vectorize using the `apply()` and `Reduce()` family of functions
    - Tidyverse / `purrr`: use `map()` functions
  - Parallelize: use the multiple cores (or threads) on your laptop or the SCF cluster for larger jobs
  - Write functions in faster programming languages (e.g., C++) and read into R (using `Rcpp`)



# Key tools to speed up computation

- **Vectorized / functional programming**
- Parallelize
- SCF cluster
- C++ & Rcpp

# Vectorizing code with `apply()`

Functions like `apply()`, `lapply()`, `Reduce()`, `map()`, and `map_*()` are useful for applying a function to each element of the input:

- `apply()` - applies a function to the margins of your input array/matrix

```
apply(X = df, MARGIN = 1, FUN = mean) # same as rowMeans(df)
```

```
apply(X = df, MARGIN = 1, FUN = function(x) {X - mean(X)})
```

- `lapply()` - given vector or list input, applies a function to each element and returns a list:
- Also see `sapply()` and `mapply()`

# Vectorizing code with `apply()`

Functions like `apply()` and `lapply()` are useful for applying a function to each element of the input:

- `apply()` - applies a function to the margins of your input array/matrix

```
apply(X = df, MARGIN = 1, FUN = mean) # same as rowMeans(df)
```

```
apply(X = df, MARGIN = 1, FUN = function(x) {X - mean(X)})
```

- `lapply()` - given vector or list input, applies a function to each element and returns a list:
- Also see `sapply()` and `mapply()`

# Vectorizing code with `map()`

The `purrr` package provides the `map_*()` family of functions which provide similar utility with a few added niceties:

- `map()` - returns a list
- `map_dbl()` - returns a double vector
- `map_lgl()` - returns a logical vector
- See `?purrr::map`

# Key tools to speed up computation

- Vectorized / functional programming
- **Parallelize**
- SCF cluster
- C++ & Rcpp

# Parallelizing code

The Statistics Department has a resident expert in computation: **Chris Paciorek**

Useful resources prepared (mostly) by Chris:

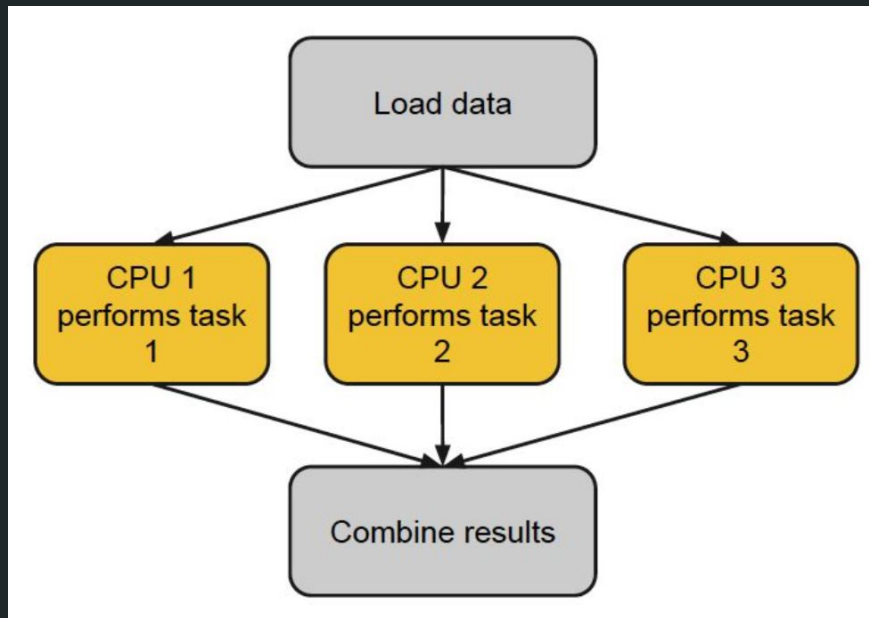
- <https://statistics.berkeley.edu/computing/training>
- <https://github.com/berkeley-scf/tutorial-parallel-basics>

Thanks to Rebecca Barter for her slides on this as well.



## Parallelizing code:

- Parallelization: doing things simultaneously
- However, parallel tasks cannot talk to one another
- Usually parallelize to speed up computation by
  - Doing loops simultaneously
  - Computing on multiple subsets of a large dataset at the same time
- Our focus: **embarrassingly parallel** tasks



# A simple example

- Imagine you have a for loop where each iteration of the for loop does not depend on any other iteration of the for loop, e.g.,

for each  $b = 1:B$

Take a subsample of your data matrix  $X$

Do something with that subsample

end for loop

- Rather than doing this for loop iteratively, can run each iteration of this for loop “in parallel” (i.e., simultaneously)
- This is a simple example of parallelization, but even here it is an incredibly powerful tool



# How to parallelize your R code?

**Option 1:** foreach and doParallel packages

```
library(foreach)
library(doParallel)

n_cores <- 4

registerDoParallel(n_cores)

B <- 10000

result <- foreach(i = 1:B) %dopar% {
  # stuff to run in each iteration
}
```

# How to parallelize your R code?

**Option 2:** `parallel` package

```
library(parallel)

n_cores <- 4
cl <- makeCluster(n_cores)

result <- parLapply(cl, X = data, FUN = fun)
```

# How to parallelize your R code?

**Option 3:** `future` / `future.apply` packages

```
library(future)
library(future.apply)

future::plan(
  multisession, workers = future::availableCores() - 1
)

future_lapply(1:B, function(b) {
  # stuff to run in each iteration
})
```

More info: <https://github.com/HenrikBengtsson/future.apply>

# How to parallelize your R code?

- See example in `parallel_example.R`
- To check how many cores your machine has

```
future::availableCores()
```

or

```
parallel::detectCores(all.tests = FALSE, logical = TRUE)
```

- If running on your home computer, good idea to leave at least one core free for your operating system (and you) to use.

# Key tools to speed up computation

- Vectorized / functional programming
- Parallelize
- **SCF cluster**
- C++ & Rcpp

# Using the SCF clusters

- If you haven't already, sign up for an SCF account at <https://scf.berkeley.edu/account>
- Information on submitting jobs to the cluster can be found here: <http://statistics.berkeley.edu/computing/servers/cluster>

# Using the SCF Clusters

1. **ssh into an SCF machine**
2. Copy your files to that computer
3. Set up a shell script that runs your job (e.g., `shell_example.sh`)
4. Submit your job using SLURM, e.g.

```
sbatch shell_example.sh
```

# Step 1: ssh into an SCF machine

- The SCF cluster contains the following LOTR-named machines that you can ssh into. Check <https://scf.berkeley.edu/ingrid>
- To SSH into a machine, type in your terminal:

```
ssh jpduncan@gimli.berkeley.edu
```

- Use your SCF username/password
- Once you ssh, you are logged in remotely to the SCF machine and can start using it.

Standalone Servers	<u>CPU</u> s
<a href="https://arwen.berkeley.edu">arwen.berkeley.edu</a>	32
<a href="https://bilbo.berkeley.edu">bilbo.berkeley.edu</a>	16
<a href="https://springer.berkeley.edu">springer.berkeley.edu</a>	16
<a href="https://legolas.berkeley.edu">legolas.berkeley.edu</a>	16
<a href="https://gimli.berkeley.edu">gimli.berkeley.edu</a>	16
<a href="https://hagrid.berkeley.edu">hagrid.berkeley.edu</a>	16
<a href="https://pooh.berkeley.edu">pooh.berkeley.edu</a>	16
<a href="https://boromir.berkeley.edu">boromir.berkeley.edu</a>	16
<a href="https://beren.berkeley.edu">beren.berkeley.edu</a>	8
<a href="https://gandalf.berkeley.edu">gandalf.berkeley.edu</a>	8
<a href="https://shelob.berkeley.edu">shelob.berkeley.edu</a>	8
<a href="https://roo.berkeley.edu">roo.berkeley.edu</a>	8
<a href="https://radagast.berkeley.edu">radagast.berkeley.edu</a>	8



# Using the SCF Clusters

1. ssh into an SCF machine
2. **Copy your files to that computer**
3. Set up a shell script that runs your job (e.g., `shell_example.sh`)
4. Submit your job using SLURM, e.g.

```
sbatch shell_example.sh
```

## Step 2: Copy your files to SCF

- Options:
  - Clone your GitHub repo on the remote machine:
    1. Change directories (cd) to where you want the copy of the repo
    2. `git clone https://github.com/USERNAME/stat-215-a`
- Another way: use **scp** to move files from your machine to the remote machine

On my computer:

```
james@james-HP-Spectre-x360 ~$ tar -czf stat-215-a.tar.gz stat-215-a
james@james-HP-Spectre-x360 ~$ scp stat-215-a.tar.gz jpduncan@gimli.berkeley.edu:~/
jpduncan@gimli.berkeley.edu's password:
stat-215-a.tar.gz 100% 20KB 1.8MB/s 00:00
```

On the SCF machine: `gimli.jpduncan$ tar -xzf stat-215-a.tar.gz`

# Using the SCF Clusters

1. ssh into an SCF machine
2. Copy your files to that computer
3. **Set up a shell script that runs your job (e.g., `shell_example.sh`)**
4. Submit your job using SLURM, e.g.

```
sbatch shell_example.sh
```

## Step 3: Write shell script to run your job

- See `shell_example.sh`

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --nodes=1
```

```
R CMD BATCH --no-save job.R job.out
```

- Make sure **cpus-per-task** is equal to the number of cores that you requested in your job.R script - typically, the number you used in **registerDoParallel()**, **makeCluster()**, or **future::plan()**

```
> library(future)
> future::availableCores()
Slurm
4
```

# Using the SCF Clusters

1. ssh into an SCF machine
2. Copy your files to that computer
3. Set up a shell script that runs your job (e.g., `shell_example.sh`)
4. **Submit your job using SLURM**, e.g.

```
sbatch shell_example.sh
```

## Step 4: Submitting your job

```
sbatch shell_example.sh
```

- To cancel your job if you made a mistake:

```
scancel [job_id], e.g. scancel 475567
```

- To check that your jobs are running as expected on the SCF cluster:

```
squeue
```

- To see only my jobs:

```
squeue -u jpduncan
```

# Demo

`week7/scf_example/`

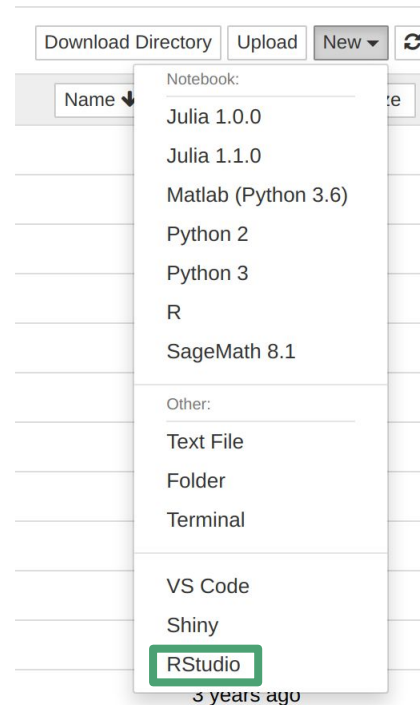
# Common mistakes

- If you are loading in data into R, set all file paths relative to the location of where you run your sbatch command
- Make sure **cpus-per-task** is equal to the number of cores that you requested in your job.R script – *typically*, the number inside **registerDoParallel()**, **makeCluster()**, or **future::plan()**
- Sometimes, functions that you call within your parallel loop are run in parallel by default. In this case, either request the appropriate number of cores or tell/force the function to use only one core. Ex. `ranger()`
- Don't forget to save or write out your results when running on the SCF clusters!



# Using the SCF jupyterhub

- <https://statistics.berkeley.edu/computing/jupyterhub>
- Easier for those not familiar with command line
- Go to <https://jupyter.stat.berkeley.edu> and log in
  - You can run Rstudio:
- Convenient when you need to interact with your code or to debug your code.



# Key tools to speed up computation

- Vectorized / functional programming
- Parallelize
- SCF cluster
- **C++ & Rcpp**

# Writing faster code with Rcpp

- Often times, C++ can be much faster than R
- Rcpp allows you to easily source C++ code into larger R functions

Rcpp\_demo.R

```
library('Rcpp')
sourceCpp('Rcpp_demo.cpp')

x <- rnorm(1e7)
y <- rnorm(1e7)
z <- cbind(x, y)

DistanceCPP(x, y)
```

Rcpp\_demo.cpp

```
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericVector DistanceCPP(Rcpp::NumericVector x, Rcpp::NumericVector y) {
  // Calculate the euclidian distance between <x> and <y>.

  // C++ requires initialization of variables.
  double result = 0.0;

  // This is the length of the x vector.
  int n = x.size();

  // Check that the size is the same and return NA if it is not.
  if (y.size() != n) {
    Rcpp::Rcout << "Error: the size of x and y must be the same.\n";
    return(Rcpp::NumericVector::create(NA_REAL));
  }

  for (int i = 0; i < n; i++) {
    result += pow(x[i] - y[i], 2.0);
  }

  // We need to convert between the double type and the R numeric vector type.
  return Rcpp::NumericVector::create(sqrt(result));
}
```

# Writing faster code with Rcpp

Some resources:

- <https://adv-r.hadley.nz/rcpp.html>
- <http://heather.cs.ucdavis.edu/~matloff/158/RcppTutorial.pdf>
- [https://teuder.github.io/rcpp4everyone\\_en/index.html](https://teuder.github.io/rcpp4everyone_en/index.html)
- Google and Stack Overflow

Go to `week7/lab_week7`