STAT 215A Fall 2019
Week 8a

Tiffany Tang

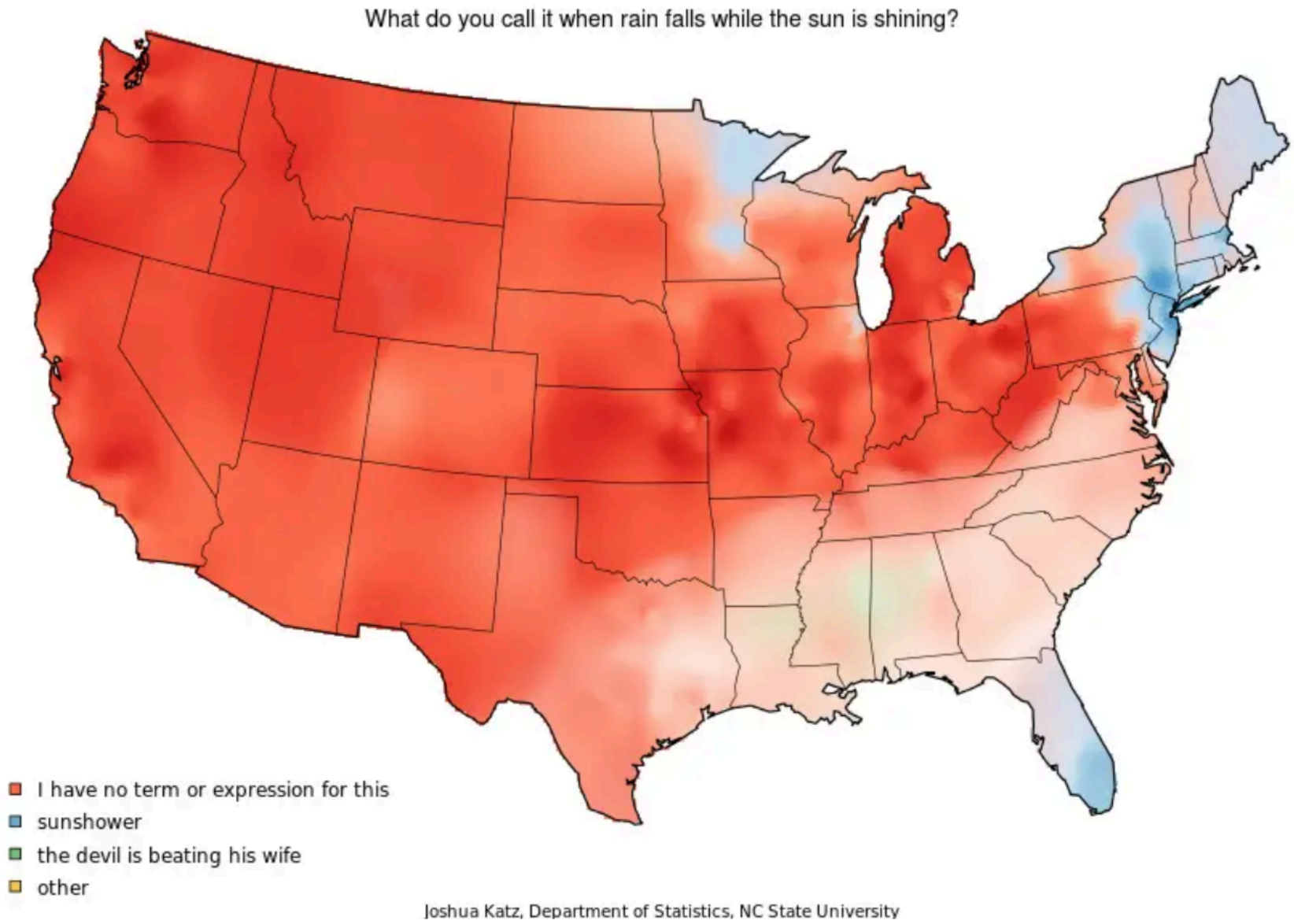
10/15/19

Announcements

- Bin is traveling this week
- No peer review
- **Today:** Lab 3 + Rcpp + SCF + everything computation
  - Lab 3 due ~1.5 weeks on **Thursday, October 24 at 11:59pm**
  - This is the shortest of all the labs, but still, please do not wait until the last minute
- **Thursday:** Discussion of Efron Bootstrap paper + Earthquake paper
- **Friday:** Regularized Regression – Lasso + ridge + many of its cousins

Lab 2

- How did it go?
- Questions?
- What did you like or didn't like?
- HW?



Plan for Today:

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

Lab 3: Stability of K-means and Computability

- **Last Time:** how to choose K in K-means using stability

  For each k = 2:kmax

    For each b = 1:B

      Perturb the data (i.e., 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 this procedure?**

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**
    $\text{sub}_1 = \text{subsample}\,(X, m)$, a subsample of fraction m of dataset X
    $\text{sub}_2 = \text{subsample}\,(X, m)$, a subsample of fraction m of dataset X
    $L_1 = \text{cluster}\,(\text{sub}_1)$
    $L_2 = \text{cluster}\,(\text{sub}_2)$
    $\text{intersect} = \text{sub}_1 \cap \text{sub}_2$
    $S\,(i, k) = \text{similarity}\,(L_1\,(\text{intersect})\,, L_2\,(\text{intersect}))$
  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 be looking at your code closely in this lab, so please be sure to follow the Google R style guide

How To Speed Up Computation

- Easy ways:
  - Don’t repeatedly re-compute objects that really only need to be computed once
  - Don’t define or store objects unnecessarily (e.g., intermediate variables)
- Other ways:
  - In R: vectorize using apply functions (base R) or map functions (purr) and avoid for loops as much as possible
  - Parallelize: using the SCF cluster or using multiple cores (or threads)
  - Write functions in faster languages (C++) and read those into R

Key Tools To Speed Up Computation

1. Vectorizing code
2. Parallelizing code in R
3. SCF clusters
4. Sourcing C++ code using R (i.e., Rcpp)

Key Tools To Speed Up Computation

1. **Vectorizing code**
2. Parallelizing code in R
3. SCF clusters
4. Sourcing C++ code using R (i.e., Rcpp)

Vectorizing Code with apply() and map()

- Functions like `apply()`, `lapply()`, `map()`, `map_*()` are useful for applying the same function to each "element" of your input
  - **`apply()`** – applies a function to the margins of your input array/matrix

```
apply(X = df, MARGIN = 1, FUN = mean) # equivalent to rowMeans(df)
apply(X = df, MARGIN = 1, FUN = function(X) {return(X - mean(X))})
```

  - **`lapply()`** – given vector or list input, applies a function to each element in your list

```
lapply(X = ls, FUN = mean)
```

  - See also **`sapply()`** and **`mapply()`**
  - **`map_*()`** – tidyverse equivalent of the apply() family of functions
    - **`map()`** – always returns a list
    - **`map_dbl()`** – always returns a double
    - **`map_lgl()`** – always returns a logical
    - See **`? map`**

Key Tools To Speed Up Computation

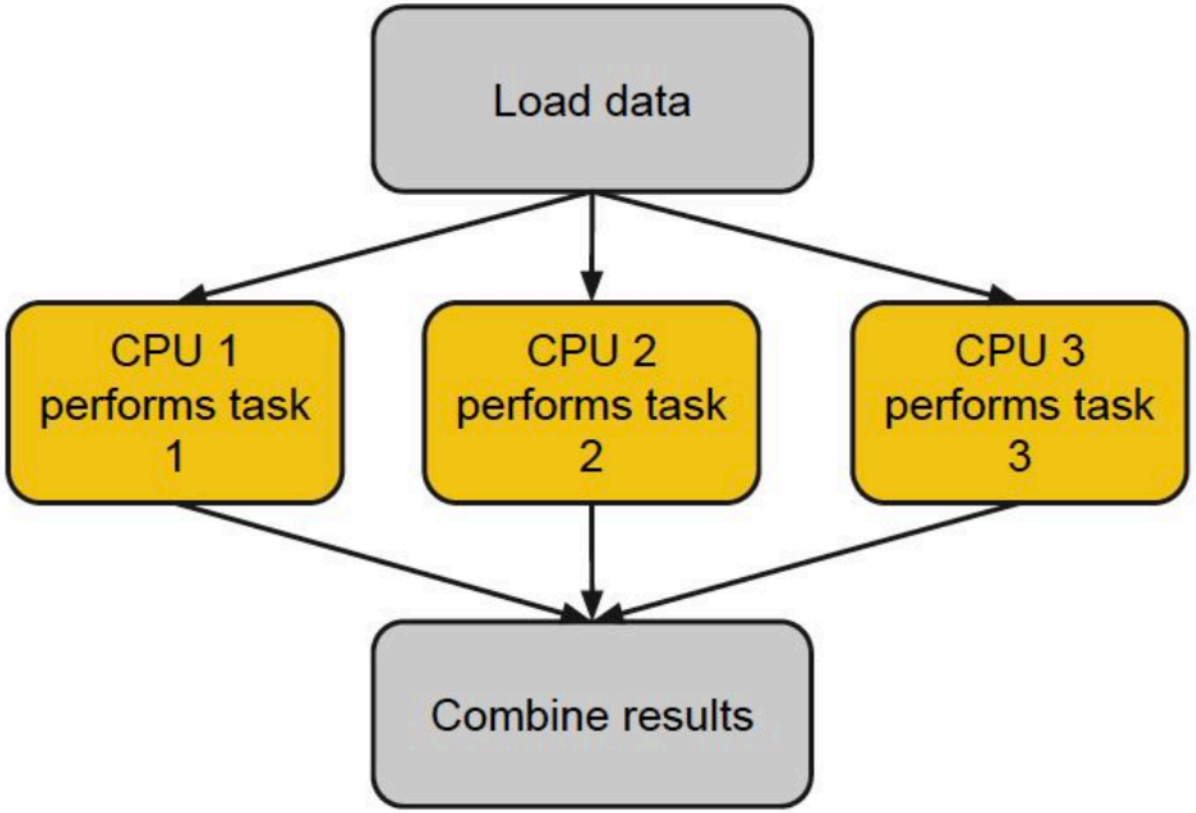
1. Vectorizing code
2. **Parallelizing code in R**
3. SCF clusters
4. Sourcing C++ code using R (i.e., Rcpp)

Parallelizing Code

- I am by no means an expert on parallelizing code
- But we have a resident expert: **Chris Paciorek**
- Useful resources prepared (mostly) by Chris:
  - http://statistics.berkeley.edu/computing/training
  - https://github.com/berkeley-scf/tutorial-parallel-basics
- Thanks to Rebecca Barter for her slides on this topic 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, or
  - Computing on multiple subsets of a large dataset simultaneously
- We will focus on **embarrassingly parallel** tasks



A Very 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 the 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 one of the most simple examples of parallelization, but it is a incredibly powerful tool

How to parallelize for loops in R?

- **Option 1:** foreach and doParallel packages

```
library(foreach)

library(doParallel)

nCores <- 4 # to set manually

registerDoParallel(nCores)

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

How to parallelize for loops in R?

- **Option 2:** parallel package

```
library(parallel)

nCores <- 4   # to set manually

cl <- makeCluster(nCores)

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

How to parallelize for loops in R?

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

```
library(parallel)

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

- If you ever run code in parallel on your computer, be sure to leave at least one core available for your operating system to use

Key Tools To Speed Up Computation

1. Vectorizing code
2. Parallelizing code in R
3. **SCF clusters**
4. Sourcing C++ code using R (i.e., 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
```

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:

```
arwen, beren, bilbo, gandalf, gimli, legolas, pooh, radagast, roo,
shelob, springer, treebeard
```

- To SSH into a machine, type in your terminal:

```
ssh tiffany.tang@bilbo.berkeley.edu
```

- Here, use your SCF username/password and pick your favorite LOTR character
- Once you ssh, you can start using that SCH machine from your own terminal

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

- There are several ways to do this
- Easiest way: **clone your GitHub repo** on the remote machine
  1. Change directories to where you would like to put your GitHub repo
  2. git clone https://github.com/USERNAME/stat-215-a
  3. cd ./stat-215-a
  4. git remote set-url origin https://github.com/USERNAME/stat- 215-a.git
- Another way: **scp** all necessary files from your machine to the remote machine
- Other options: http://statistics.berkeley.edu/computing/copying-files

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 inside **`registerDoParallel()`** or **`makeCluster()`**

- Make sure that the shell script is executable (change permissions)

```
chmod 755 shell_example.sh
```

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)
```

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

```
squeue
```

- To see only my jobs:

```
squeue –u tiffany.tang
```

A Quick Demonstration: scf_example/

Some 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()`** or **`makeCluster()`**
- Sometimes, functions that you call within your foreach() 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
- Convenient when you need to interact with your code or to debug your code

Key Tools To Speed Up Computation

1. Vectorizing code
2. Parallelizing code in R
3. SCF clusters
4. **Sourcing C++ code using R (i.e., 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:

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

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

DistanceCPP(x, y)
```

Rcpp_demo.cpp:

```cpp
#include <Rcpp.h>

Rcpp::NumericVector DistanceCPP(Rcpp::NumericVector x, Rcpp::NumericVector
  // 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 ty
  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
  - Google and Stack Overflow

Go to lab_week8a/