Big Data Analytics in R

Matthew J. Denny
University of Massachusetts Amherst

mdenny@polsci.umass.edu

May 28, 2015

www.mjdenny.com

INSTITUTE FOR SOCIAL SCIENCE RESEARCH
UNIVERSITY OF MASSACHUSETTS AMHERST

UMassAmherst

Overview

1. Programming Choices
2. Paralellization/Memory Management Example

1. Programming Choices

Efficient R programming

- Loops are slow in R, but fast enough for most things.
- Built-in functions are mostly written in C – much faster!
- Subset data before processing when possible.
- Avoid growing datastructures

Loops are "slow" in R

```
system.time({
    vect <- c(1:10000000)
    total <- 0
    #check using a loop
    for(i in 1:length(as.numeric(vect))){
        total <- total + vect[i]
    }
    print(total)
})
[1] 5e+13
   user  system elapsed
  7.641   0.062   7.701
```

And fast in C

```
system.time({
    vect <- c(1:10000000)
    #use the builtin R function
    total <- sum(as.numeric(vect))
    print(total)
})
[1] 5e+13
   user  system elapsed
  0.108   0.028   0.136
```

Summing over a sparse dataset

```
#number of observations
numobs <- 100000000

#observations we want to check
vec <- rep(0,numobs)

#only select 100 to check
vec[sample(1:numobs,100)] <- 1

#combine data
data <- cbind(c(1:numobs),vec)
```

Conditional checking

```
system.time({
    total <- 0
    for(i in 1:numobs){
        if(data[i,2] == 1)
        total <- total + data[i,1]
    }
    print(total)
})
[1] 5385484508
   user  system elapsed
199.917   0.289 200.350
```

Subsetting

```
system.time({
    dat <- subset(data, data[,2] ==1)
    total <- sum(dat[,1])
    print(total)
})
[1] 5385484508
   user  system elapsed
  5.474   1.497   8.245
```

1.a. Pre-Allocation

Adding to a vector vs. pre-allocation

```
system.time({
  vec <- NULL
  for (i in 1:(10^5)) vec <- c(vec,i)
})
   user  system elapsed
 18.495   7.401  25.935
system.time({
  vec <- rep(NA,10^5)
  for (i in 1:(10^5)) vec[i] <- i
})
   user  system elapsed
  0.144   0.002   0.145
```

Pre-allocated vector – bigger example

```
system.time({
  vec <- rep(NA,10^6)
  for (i in 1:(10^6)) vec[i] <- i
})
   user  system elapsed
  1.765   0.040   1.872
```

Adding to a vector – bigger example

```
system.time({
  vec <- NULL
  for (i in 1:(10^6)) vec <- c(vec,i)
})
```

```
Timing stopped at: 924.922 120.322 1872.294
```

I didn't feel like waiting...

Pre-Allocation

- Vectors in R can only hold about 2.1 Billion elements.
- Write to over-allocated vector then subset.
- Speedup is exponential in the vector size and number of additions.

1.b. Parallelization

Parallelization using `foreach`

- Works best when we need to calculate some complex statistic on each row/column of dataset.
- Works just like a regular `for( )` loop as long as operations are **independent**.
- Good for bootstrapping.

Parallelization using foreach

```
# Packages:
require(doMC)
require(foreach)

# Register number of cores
nCores <- 8
registerDoMC(nCores)

# iterations
N <- 100

# Run analysis in parallel
results <- foreach(i=1:N,.combine=rbind) %dopar% {
    result <- function(i)
}
```

Parallelization using a `snowfall` cluster

- Can run across multiple machines.
- Can run totally different jobs on each thread.
- Requires explicit management by researcher.

Parallelization using a snowfall cluster

```
# Package:
library(snowfall)

# Register cores
numcpus <- 4
sfInit(parallel=TRUE, cpus=numcpus )

# Check initialization
if(sfParallel()){
    cat( "Parallel on", sfCpus(), "nodes.\n" )
}else{
    cat( "Sequential mode.\n" )
}
```

Parallelization using a snowfall cluster

```
# Export all packages
for (i in 1:length(.packages())){
     eval(call("sfLibrary", (.packages()[i]),
     character.only=TRUE))
}

# Export a list of R data objects
sfExport("Object1","Object2","Object3")

# Apply a function across the cluster
result <- sfClusterApplyLB(indexes,Function)

# Stop the cluster
sfStop()
```

Parallelization using `mclapply( )`

- **Will not work with Windows machines.**
- Simple parallelization.
- Works well with functions written in Rcpp.

Parallelization using `mclapply( )`

```
# Packages:
library(parallel)

# Wrapper Function
run_on_cluster <- function(i){
    temp <- your_function(i,some other stuff)
    return(temp)
}

# Run analysis
indexes <- 1:Iterations
Result  <- mclapply(indexes,
                    run_on_cluster,
                    mc.cores = num_cpus)
```

1.c. Memory Efficient Regression

High memory regression using `biglm( )`

- Memory efficient implementation of `glm( )`
- Can also read in data in chunks from internet or from elational database.
- Will not take data in matrix form, only data.frame

High memory regression using `biglm( )`

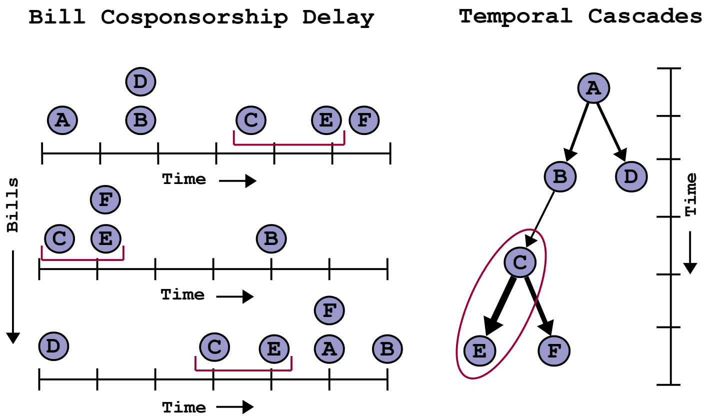
```
# Data must be of data.frame type
data <- as.data.frame(data)

# Use variable names in formula
str <- "V1 ~ V2 + V4"

# Run model
model<- bigglm(as.formula(str),
               data = full_data,
               family=binomial(),
               maxit = 20)
```

2.
Paralellization/Memory Management Example

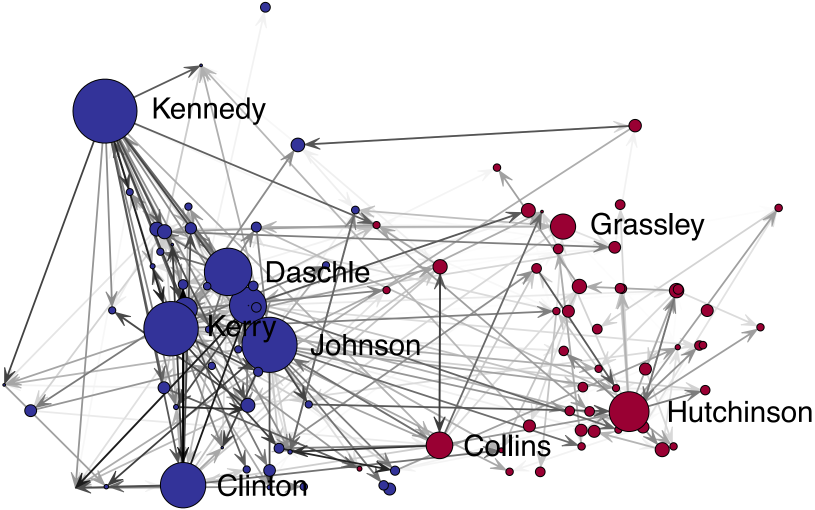
Latent Network Inference Example

- Want to measure the influence of legislators on each other.
- Use temporal patterns in bill cosponsorship as evidence.
- Gomez Rodriguez, M., Leskovec, J., & Krause, A. (2010). "**Inferring networks of diffusion and influence**". *KDD*

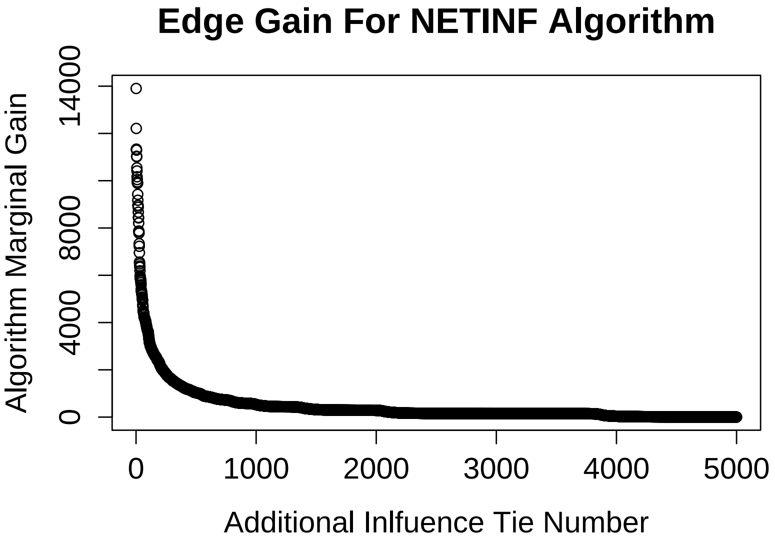
Inferring influecne



Measuring influence in the Senate



How many ties do I use?

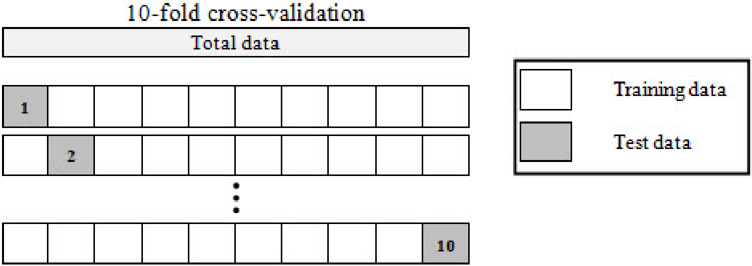


Strategy

- Predict when Senators will cosponsor in held-out sample.
- Fit event history models for model selection.
- Optimization over # edges and hyper-parameter (10 80/20 splits)
- Grid Search!

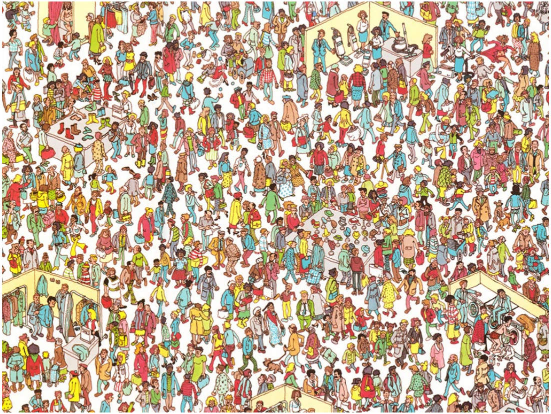
Cross validation

- Jensen, D. D., & Cohen, P. R. (2000). **Multiple Comparisons in Induction Algorithms**. *Machine Learning*, 309338.



Rare Events Logistic Regression

- King, G., & Zeng, L. (2001). **Logistic regression in rare events data**. *Political Analysis*, 9(2), 137163.



Use model log likelihood for selection.