Lab 0: Getting back to R

PBHLTH 250C (Spring 2024)

January 18, 2024

Guidelines for Assignments

- Submit homeworks by uploading knitted PDF files to Gradescope
  - The PDF you submit should be the result of knitting your Rmd or Qmd file to PDF
  - If knitting to PDF poses a challenge, knit to `docx` and edit the content in MS Word or other suitable Word processor
- Carefully select the pages so that they match the assignment outline
  - Use the `\newpage` macro to insert page breaks when appropriate
- This course will involve a lot of (pseudo)-random result; please take note of instances of `set.seed()`
  - The results you report and interpret should be the results from a clean run of your code chunks from top to bottom
- Please suppress warnings or other messages
- Please `echo = F`, but include code as an appendix by including the following chunk at the end of your document (you can find an example of use at the end of this document)

````
```{r ref.label=knitr::all_labels(), echo = T, eval = F}
```
````

- If you have long lines of R script, consider auto-wrapping the code by highlighting that script and using the keyboard shortcut `ctrl/cmd + shift + A`
- Here is an example setup chunk

````
```{r setup}
knitr::opts_chunk$set(
    echo = F, cache = TRUE,
    warning = F, message = F,
    fig.align = "center")
```
````

Built-in distributions in R

Density, distribution, and quantile functions for commonly used probability distributions are available with the basic installation of R:

- Normal
  `dnorm(x, mean = 0, sd = 1)`, `pnorm()`, `qnorm()`, `rnorm()`
- Binomial
  `dbinom(x, size, prob)`, `pbinom()`, `qbinom()`, `rbinom()`
- Uniform
  `dunif(x, min = 0, max = 1)`, `punif()`, `qunif()`, `runif()`
- Poisson
  `dpois(x, lambda)`, `ppois()`, `qpois()`, `rpois()`
- Beta
  `dbeta(x, shape1, shape2)`, `pbeta()`, `qbeta()`, `rbeta()`
- Chi-square
  `dchisq(x, df)`, `pchisq()`, `qchisq()`, `rchisq()`
- Gamma
  `dgamma(x, shape, rate = 1, scale = 1/rate)`, `pgamma()`, `qgamma()`, `rgamma()`

One distribution we will be working with that is not available in base R, is the multivariate normal distribution. For that, we can use the `MASS` package.

- Multivariate normal
  `MASS::mvrnorm(n = 1, mu, Sigma)` (random sampling)

The letter than comes before the abbreviated distribution name indicates whether your taking **r**andom samples, evaluating the **d**ensity, getting the **p**robability of observing that value or less (CDF), or finding the **q**uantile:

```r
# Draw 10 times from Bernoulli (p = 0.8)
rbinom(n = 10, size = 1, p = 0.8)
```

```
 [1] 1 1 1 1 1 1 1 1 1 1
```

```r
# Draw 10 times from binomial (p = 0.8, n = 10)
rbinom(n = 10, size = 10, p = 0.8)
```

```
 [1] 7 9 8 5 9 9 8 8 7 8
```

```r
# Density of standard normal at x = -1.96
dnorm(-1.96)
```

```
[1] 0.05844094
```

```r
# Cumulative probability at x = 0
pnorm(0)
```

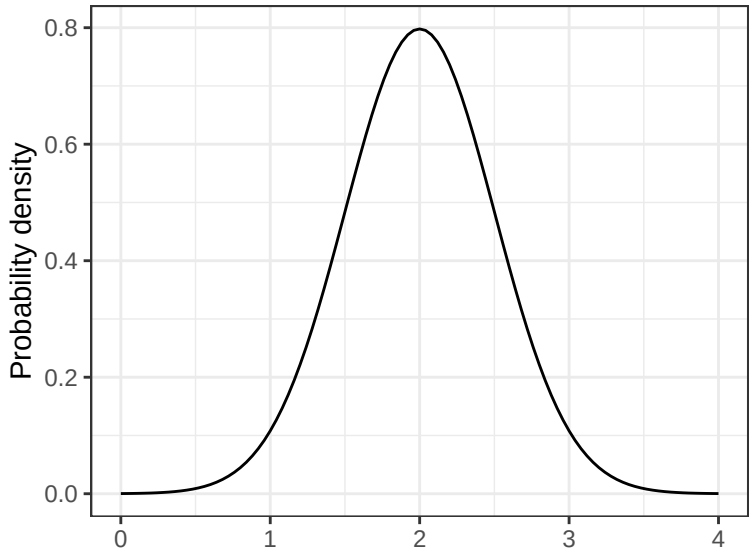
```
[1] 0.5
```

```r
# Z-score where area to the right is 0.025
qnorm(1 - 0.025)
```

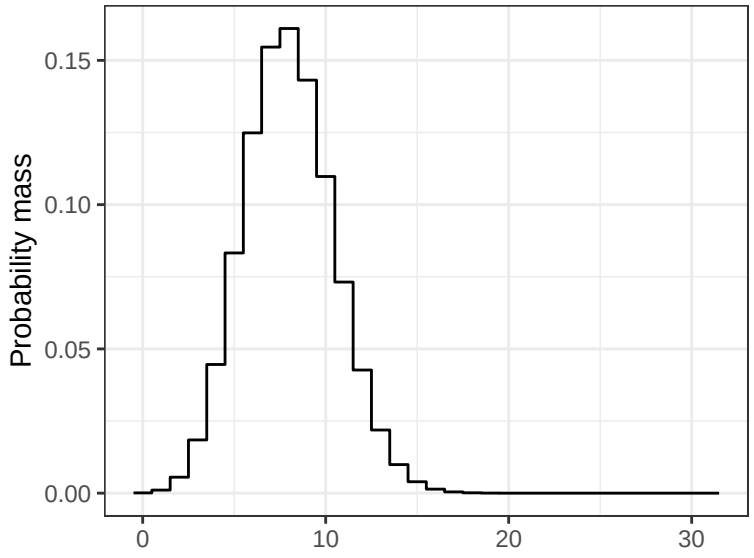
```
[1] 1.959964
```

Plotting probability density/mass functions is a way to develop some intuition with probability distributions.

```r
# Plot the density function of a normal with mean 2 and variance 0.25
ggplot(data.frame(bears = seq(0, 4, 0.1)), aes(x = bears)) +
  geom_function(fun = function(x) {dnorm(x, 2, 0.5)}) +
  labs(y = "Probability density") +
  theme_bw() + theme(
    axis.title.x = element_blank()
  )
```




```r
# Plot the mass function of binomial (p = 0.25, n = 32)
ggplot(data.frame(x = 0:32), aes(x = x, y = dbinom(x, 32, 0.25))) +
  geom_step(aes(x = x - 0.5)) +
  labs(y = "Probability mass") +
  theme_bw() + theme(
    axis.title.x = element_blank()
  )
```



The `sample()` function

The `sample()` function is used to generate a random samples and permutations from a user-specified "population" `x`. If `x` is a scalar (single number), it will be interpreted as `1:x`. By default, `sample()` returns random permutations i.e. the elements in `x` shuffled in a different order. The user may also specify the size of the result by specifying the `size`. Sample with replacement by indicating `replace = TRUE`. Sampling weights may be specified using the `prob` argument. The length of `prob` must be equal to the length of `x`, and will be scaled by the sum such that `sum(prob)` returns `1`. By default, we have `prob = NULL`, and `sample()` weighs each element of `x` equally during sampling.

```
set.seed(124)

# Random sample of size 8 from set {5, 6, ..., 15}
sample(5:15, 8)
```

```
[1]  5 11  9 12 10 13  6 15
```

```
# Random permutation of set {1, 2, ..., 10}
set.seed(250)
sample(10)
```

```
 [1]  2  3 10  8  7  6  1  5  9  4
```

```
set.seed(250)
sample(1:10)
```

```
 [1]  2  3 10  8  7  6  1  5  9  4
```

```
# Random sample with replacement
sample(10, replace = T)
```

```
 [1]  4 10  9  3  9  1  8  1  3  7
```

Exercises: Vectorized operatons and control flow[1]

1. Suppose we are interested in the Google popularity index of search term `candy`. Write a loop that returns a vector of stopping distances centered about their mean.

```
candy <- read.csv('multiTimeline.csv', skip = 2,
                  col.names = c("Month", "Popularity"))
```

2. Can the task above be accomplished without writing a loop? If so, provide the script below.

3. Now suppose we want to transform the vector of monthly popularity values X by applying the following formula:

$$D_{i-1} = X_i - X_{i-1}$$

for $i = 2, 3, \ldots, 225$. Write a loop to accomplish this task.

4. Can the task above be accomplished without writing a loop? If so, provide the script below.

[1] See the UC Berkeley Statistical Computing Facility's tutorial for additional information.

Convergence of large samples

In mathematics, the limit of a sequence is said to converge if the terms of that sequence "tend toward" a particular value. In statistics, we are not just interested in lists of numbers, but *random* lists of numbers. From these random lists of numbers, we calculate statistics in order to estimate some population parameter. The map from observed data to parameter estimate is called the estimator. Estimators are random variables often indexed by sample size n. Two convergence results of estimators are important:

- Law of Large Numbers
- Central Limit Theorem

Consider the simulated **sampling distributions** of the sample mean calculated from samples of different sizes $n = \{5, 50, 500, 1000\}$. We use the `replicate(n, expr)` function which evaluates an expression an arbitrary number of times.

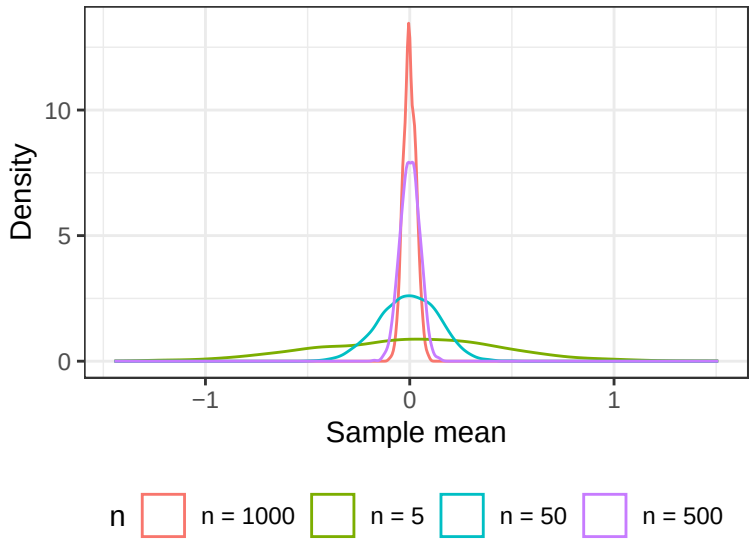
```
M <- 1000

# Run the simulations for the 4 different sample sizes
simulated_sampling_dist <- lapply(
  c(5, 50, 500, 1000), function(n) {
  replicate(M, mean(rnorm(n)))
})

# Reshape the data so that it's a data frame
simulated_sampling_dist <- data.frame(
  n = rep(paste0("n = ", c(5, 50, 500, 1000)), each = M),
  estimates = unlist(simulated_sampling_dist)
)

# Plot simulated sampling distributions
simulated_sampling_dist %>% ggplot(
  aes(x = estimates,
      color = n)
) + geom_density() +
    labs(y = "Density", x = "Sample mean") +
  theme_bw() +
```

```
  theme(legend.position = "bottom")
```



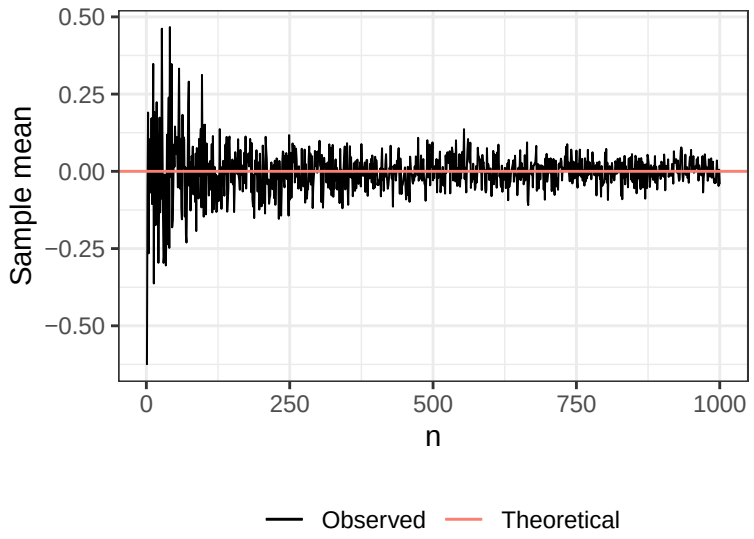
Now, let's consider sample statistics calculated form data of different sizes. First, consider the sample mean of random samples generated from the standard normal.

```
set.seed(250)
simulated_sample_means <- c()
N <- 1000
for (n in 1:N) {
  simulated_sample_means[n] <- mean(rnorm(n, mean = 0, sd = 1))
}

# plot(simulated_sample_means)

ggplot(data.frame(
  n = 1:N,
  "Sample mean" = simulated_sample_means,
  check.names = F
), aes(x = n, y = `Sample mean`)) +
  geom_line(linewidth = 1/.pt, aes(color = "Observed")) +
  # Expected value
  geom_hline(aes(yintercept = 0, color = "Theoretical")) +
  scale_color_manual(values = c("black", "salmon")) +
```

```
  theme_bw() +
  theme(legend.position = "bottom",
        legend.title = element_blank())
```

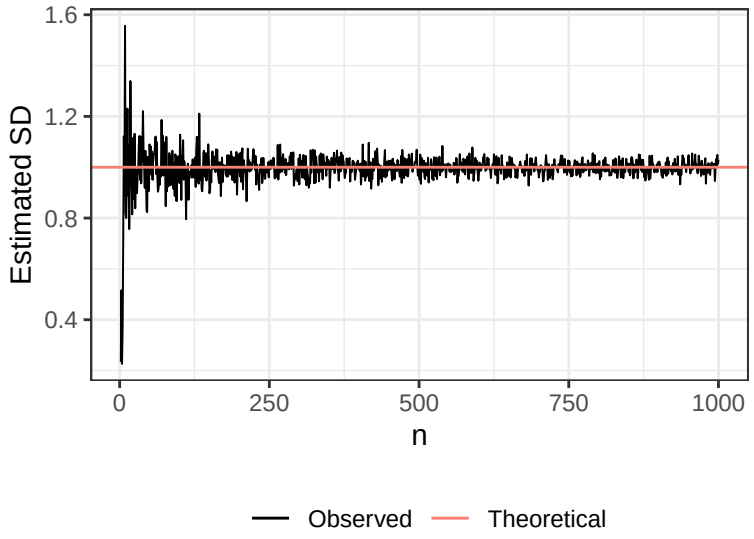


Now, let's consider the standard deviations estimated from random samples generated from the standard normal.

```
set.seed(250)
simulated_sample_sd <- c()
for (n in 2:N) {
  simulated_sample_sd[n - 1] <- sd(rnorm(n, mean = 0, sd = 1))
}

ggplot(data.frame(
  n = 2:N,
  "Estimated SD" = simulated_sample_sd,
  check.names = F
), aes(x = n, y = `Estimated SD`)) +
  geom_line(linewidth = 1/.pt, aes(color = "Observed")) +
  # Expected value
  geom_hline(aes(yintercept = 1, color = "Theoretical")) +
  scale_color_manual(values = c("black", "salmon")) +
  theme_bw() +
  theme(legend.position = "bottom",
```

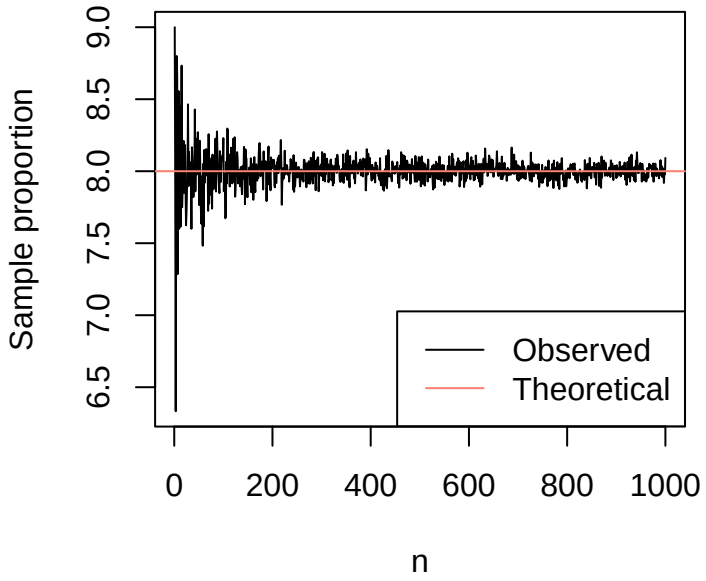
```
        legend.title = element_blank())
```



Here, we have an example using base R plotting tools.

```
set.seed(250)
simulated_sample_prop <- c()
for (n in 1:N) {
  simulated_sample_prop[n] <- mean(rbinom(n, size = 10, prob = 0.8))
}

plot(simulated_sample_prop,
     type = "l",
     xlab = "n",
     ylab = "Sample proportion")
abline(h = 8, col = "salmon") # Expected mean
legend("bottomright", legend = c("Observed", "Theoretical"),
             col = c("black", "salmon"), lty = c(1, 1))
```


Sample proportion
6.5
7.0
7.5
8.0
8.5
9.0
0
200
400
600
800
1000
n
Observed
Theoretical

Parallelization (optional)

To speed up sampling time, we can run loops in parallel by evaluating script simultaneously across cores and combining the results.

The `foreach` package depends on the `doParallel`. The `foreach` package must be used in conjunction with a package such as `doParallel` in order to execute code in parallel. The user must register a parallel "back-end" to use, otherwise `foreach` will execute tasks sequentially, even when the `%dopar%` operator is used.

Below is an example of a simple for loop, calculating the square root of a vector of numbers.

```
system.time({
    x <- 1:1e5
    y <- c()
    for (i in 1:1e5) {
        y[i] <- sqrt(x[i])
    }
})
```

```
   user  system elapsed
  0.012   0.001   0.013
```

We can parallelize this loop by changing `for` to `foreach` and using the `%dopar%` operator. The values generated within each iteration the `foreach` loop are not saved to the global environment, so we only save the result.

```
cl <- makeCluster(detectCores() - 1, type = "FORK")
registerDoParallel(cl)
system.time({
    y <- foreach(x = 1:1e5) %dopar% {
        sqrt(x)
    }
})
```

```
   user  system elapsed
  8.930   1.370  10.476
```

```
stopCluster(cl)
```

Sometimes, running code in parallel will not make a huge difference; sometimes it may even be slower. It takes time to set up the parallel operation and combine results, so parallel processing is not always the best solution. Bootstrapping is an example of when parallel processing may speed up run time significantly. Let's see how long it takes to run 10,000 bootstrap iterations in parallel on multiple cores:

```
data <- iris[which(iris[,5] != "setosa"), c(1,5)]
trials <- 1e4

cl <- makeCluster(detectCores() - 1)
registerDoParallel(cl)

# system.time({
    r <- foreach(i = 1:trials) %dopar% {
        index <- sample(100, 100, replace = TRUE)
        result1 <- glm(data[index, 2] ~ data[index, 1], family = binomial(logit))
        coefficients(result1)
    }
# })

stopCluster(cl)
```

By changing the `%dopar%` to `%do%`, we can run the same code sequentially to determine the performance improvement:

```
r <- list()
system.time({
    for (i in 1:trials) {
        index <- sample(100, 100, replace = TRUE)
        result1 <- glm(data[index, 2] ~ data[index, 1], family = binomial(logit))
        r[[i]] <- coefficients(result1)
    }
})
```

```
system.time({
    r <- lapply(1:trials, function(i) {
        index <- sample(100, 100, replace = TRUE)
        result1 <- glm(data[index, 2] ~ data[index, 1], family = binomial(logit))
        return(coefficients(result1))
    })
})
```

The `.combine` option

So far, all of our examples returned objects of class `list`. This is a good default class because it is one of the most general. But sometimes we'd like the results to be returned as an object of class `numeric`, for example. This can be done by using the `.combine` option in `foreach`:

```
cl <- makeCluster(detectCores() - 1)
registerDoParallel(cl)

#default
x <- foreach(i = 1:3) %dopar% exp(i)
x

#concatenate into a vector
x <- foreach(i = 1:3, .combine = 'c') %dopar% exp(i)
x



#combine columns
x <- foreach(i = 1:3, .combine = cbind) %dopar% exp(i)
x

#combine rows
x <- foreach(i = 1:3, .combine = rbind) %dopar% exp(i)
x

stopCluster(cl)
```

Appendix

```
knitr::opts_chunk$set(echo = T, cache = TRUE,
                      fig.align = "center",
                      fig.width = 4,
                      fig.height = 3)
library(tidyverse)
# Draw 10 times from Bernoulli (p = 0.8)
rbinom(n = 10, size = 1, p = 0.8)
# Draw 10 times from binomial (p = 0.8, n = 10)
rbinom(n = 10, size = 10, p = 0.8)
# Density of standard normal at x = -1.96
dnorm(-1.96)
# Cumulative probability at x = 0
pnorm(0)
# Z-score where area to the right is 0.025
qnorm(1 - 0.025)
# Plot the density function of a normal with mean 2 and variance 0.25
ggplot(data.frame(bears = seq(0, 4, 0.1)), aes(x = bears)) +
  geom_function(fun = function(x) {dnorm(x, 2, 0.5)}) +
  labs(y = "Probability density") +
  theme_bw() + theme(
    axis.title.x = element_blank()
  )
# Plot the mass function of binomial (p = 0.25, n = 32)
ggplot(data.frame(x = 0:32), aes(x = x, y = dbinom(x, 32, 0.25))) +
  geom_step(aes(x = x - 0.5)) +
  labs(y = "Probability mass") +
  theme_bw() + theme(
    axis.title.x = element_blank()
  )
set.seed(124)

# Random sample of size 8 from set {5, 6, ..., 15}
sample(5:15, 8)
```

```r
# Random permutation of set {1, 2, ..., 10}
set.seed(250)
sample(10)
set.seed(250)
sample(1:10)

# Random sample with replacement
sample(10, replace = T)
candy <- read.csv('multiTimeline.csv', skip = 2,
                  col.names = c("Month", "Popularity"))
M <- 1000

# Run the simulations for the 4 different sample sizes
simulated_sampling_dist <- lapply(
  c(5, 50, 500, 1000), function(n) {
  replicate(M, mean(rnorm(n)))
})

# Reshape the data so that it's a data frame
simulated_sampling_dist <- data.frame(
  n = rep(paste0("n = ", c(5, 50, 500, 1000)), each = M),
  estimates = unlist(simulated_sampling_dist)
)

# Plot simulated sampling distributions
simulated_sampling_dist %>% ggplot(
  aes(x = estimates,
      color = n)
) + geom_density() +
    labs(y = "Density", x = "Sample mean") +
  theme_bw() +
  theme(legend.position = "bottom")

set.seed(250)
simulated_sample_means <- c()
N <- 1000
```

```r
for (n in 1:N) {
  simulated_sample_means[n] <- mean(rnorm(n, mean = 0, sd = 1))
}

# plot(simulated_sample_means)

ggplot(data.frame(
  n = 1:N,
  "Sample mean" = simulated_sample_means,
  check.names = F
), aes(x = n, y = `Sample mean`)) +
  geom_line(linewidth = 1/.pt, aes(color = "Observed")) +
  # Expected value
  geom_hline(aes(yintercept = 0, color = "Theoretical")) +
  scale_color_manual(values = c("black", "salmon")) +
  theme_bw() +
  theme(legend.position = "bottom",
        legend.title = element_blank())
set.seed(250)
simulated_sample_sd <- c()
for (n in 2:N) {
  simulated_sample_sd[n - 1] <- sd(rnorm(n, mean = 0, sd = 1))
}

ggplot(data.frame(
  n = 2:N,
  "Estimated SD" = simulated_sample_sd,
  check.names = F
), aes(x = n, y = `Estimated SD`)) +
  geom_line(linewidth = 1/.pt, aes(color = "Observed")) +
  # Expected value
  geom_hline(aes(yintercept = 1, color = "Theoretical")) +
  scale_color_manual(values = c("black", "salmon")) +
  theme_bw() +
  theme(legend.position = "bottom",
        legend.title = element_blank())
```

```r
set.seed(250)
simulated_sample_prop <- c()
for (n in 1:N) {
  simulated_sample_prop[n] <- mean(rbinom(n, size = 10, prob = 0.8))
}

plot(simulated_sample_prop,
     type = "l",
     xlab = "n",
     ylab = "Sample proportion")
abline(h = 8, col = "salmon") # Expected mean
legend("bottomright", legend = c("Observed", "Theoretical"),
             col = c("black", "salmon"), lty = c(1, 1))
system.time({
    x <- 1:1e5
    y <- c()
    for (i in 1:1e5) {
        y[i] <- sqrt(x[i])
    }
})

library(foreach)
library(doParallel)
# install.packages("foreach")
# install.packages("doParallel")
cl <- makeCluster(detectCores() - 1, type = "FORK")
registerDoParallel(cl)
system.time({
    y <- foreach(x = 1:1e5) %dopar% {
        sqrt(x)
    }
})
stopCluster(cl)
data <- iris[which(iris[,5] != "setosa"), c(1,5)]
trials <- 1e4
```

```r
cl <- makeCluster(detectCores() - 1)
registerDoParallel(cl)

# system.time({
    r <- foreach(i = 1:trials) %dopar% {
        index <- sample(100, 100, replace = TRUE)
        result1 <- glm(data[index, 2] ~ data[index, 1], family = binomial(logit))
        coefficients(result1)
    }
# })

stopCluster(cl)

r <- list()
system.time({
    for (i in 1:trials) {
        index <- sample(100, 100, replace = TRUE)
        result1 <- glm(data[index, 2] ~ data[index, 1], family = binomial(logit))
        r[[i]] <- coefficients(result1)
    }
})

system.time({
    r <- lapply(1:trials, function(i) {
        index <- sample(100, 100, replace = TRUE)
        result1 <- glm(data[index, 2] ~ data[index, 1], family = binomial(logit))
        return(coefficients(result1))
    })
})

cl <- makeCluster(detectCores() - 1)
registerDoParallel(cl)

#default
x <- foreach(i = 1:3) %dopar% exp(i)
```

```r
x

#concatenate into a vector
x <- foreach(i = 1:3, .combine = 'c') %dopar% exp(i)
x


#combine columns
x <- foreach(i = 1:3, .combine = cbind) %dopar% exp(i)
x

#combine rows
x <- foreach(i = 1:3, .combine = rbind) %dopar% exp(i)
x

stopCluster(cl)
```