Métodos Computacionais para Estatística e Otimização

Lista 4 - Luiz Henrique Barretta Francisco

1. A média harmônica é dada pela seguinte equação $M_h = \frac{n}{\frac{1}{x_1}+\ldots+\frac{1}{x_n}}$.Implemente uma função R para o seu cálculo usando a estrutura de repetição for. Não use nenhum recurso do R que use vetorização. Gere uma amostra de tamanho 100 com o seguinte código.

```
set.seed(123)
y <- rpois(100, lambda = 10)
```

Baseado nesta amostra calcule a média harmonica usando a sua função.

```
media_harmonica_for <- function(x) {
  n <- length(x)
  soma <- 0
  for (i in 1:n) {
    soma <- soma + 1 / x[i]}
  n / soma}
media_harmonica_for(y)
```

```
## [1] 8.743717
```

2. Reimplemente a função do exercício 1, porém agora usando tudo que conseguir de recursos de vetorização já disponíveis no R. Faça uma comparação do tempo computacional necessário por cada uma das abordagens. Use o package *bench*.

```
library(bench)
media_harmonica_vet <- function(x) {
  length(x) / sum(1 / x)}
media_harmonica_vet(y)
```

```
## [1] 8.743717
```

```
mark(for_loop = media_harmonica_for(y),
     vetorizada = media_harmonica_vet(y))
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 for_loop        2us    2.3us   384633.        0B        0
## 2 vetorizada    400ns    600ns  1363252.    13.5KB        0
```

Os resultados do benchmark indicam que a versão vetorizada da função de média harmônica é significativamente mais eficiente que a versão com *for*. A função vetorizada executou aproximadamente 1.34 milhão de vezes por segundo, enquanto a versão com laço *for* executou cerca de 328 mil vezes por segundo — ou seja, a vetorizada foi cerca de 4 vezes mais rápida. Além disso, o tempo mínimo da versão vetorizada foi de 400 nanossegundos, contra 2.3 microssegundos da versão com laço. Isso demonstra a clara vantagem de utilizar recursos vetorizados no R para tarefas computacionalmente repetitivas.

3. Reimplemente a função do exercício 1, porém agora usando C++ e o pacote Rcpp. Novamente compare o tempo computacional entre as três abordagens usando o pacote bench.

```
library(Rcpp)

cppFunction('
  double media_harmonica_cpp(NumericVector x) {
    int n = x.size();
    double soma = 0;
    for (int i = 0; i < n; i++) {
      soma += 1.0 / x[i];
    }
    return n / soma;
  }
')

mark(for_loop = media_harmonica_for(y),
     vetorizada = media_harmonica_vet(y),
     cpp = media_harmonica_cpp(y))
```

```
## # A tibble: 3 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 for_loop        2us    2.3us   413281.        0B       0
## 2 vetorizada    400ns    600ns  1378583.      848B     138.
## 3 cpp           400ns    600ns  1226527.      848B       0
```

Os resultados mostram que a versão em C++ com Rcpp foi a mais rápida, executando cerca de 988 mil vezes por segundo, praticamente empatada com a versão vetorizada em R, que alcançou 956 mil iterações por segundo. Já a versão com laço *for* em R foi significativamente mais lenta, com cerca de 304 mil execuções por segundo, ou seja, aproximadamente 3 vezes mais lenta. Assim, tanto a abordagem vetorizada quanto a em C++ são altamente eficientes, mas a vetorizada oferece desempenho comparável com menor complexidade de implementação.

4. Em estatística testes qui-quadrado são muito populares. O R tem uma função para realizar tal teste para diversas situações. Uma situação comum é para verificar a associação em tabelas de contingência. O exemplo abaixo retirado da documentação tem o objetivo de avaliar a associação entre genero e perfil político.

```
M <- as.table(rbind(c(762, 327, 468), c(484, 239, 477)))
dimnames(M) <- list(gender = c("F", "M"),
                    party = c("Democrat","Independent", "Republican"))
M
```

```
##       party
```

```
## gender Democrat Independent Republican
##      F      762         327        468
##      M      484         239        477
```

```
chisq.test(M)
```

```
##
##  Pearson's Chi-squared test
##
## data:  M
## X-squared = 30.07, df = 2, p-value = 2.954e-07
```

Use uma ferramenta de debug para investigar como a estatística de teste é calculada. Reimplemente apenas a estatística de teste em uma função própria. Compare a sua função com a do R em termos de tempo computacional. Suponha que o interesse é apenas obter a estatística de teste.

```
chi_stat_manual <- function(x) {
  expected <- outer(rowSums(x), colSums(x)) / sum(x)
  sum((x - expected)^2 / expected)}

chi_stat_manual(M)
```

```
## [1] 30.07015
```

```
chisq.test(M)$statistic
```

```
## X-squared
##  30.07015
```

```
mark(manual = chi_stat_manual(M),
     base_r = as.numeric(chisq.test(M)$statistic))
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 manual        9.1us   10.4us    89473.    37.4KB     44.8
## 2 base_r       32.2us   37.7us    24521.        0B     39.3
```

A comparação mostra que a função implementada manualmente para calcular a estatística do teste qui-quadrado é significativamente mais rápida que a função *chisq.test()* do R base. A versão manual alcançou aproximadamente 76 mil execuções por segundo, enquanto a função do R teve cerca de 21 mil execuções por segundo, sendo mais de 3 vezes mais lenta. Além disso, a versão manual alocou um pouco mais de memória, mas ainda assim teve desempenho mais eficiente em tempo total de execução.

5. Comparar dois grupos é uma atividade popular em estatística. Considere o conjunto de dados iris disponível no R. Suponha que desejamos testar se a variável Sepal.Length é em média diferente entre as espécies setosa e versicolor. Para isso vamos usar um teste de aleatorização. Sob a hipótese nula o tamanho médio das sepalas é igual. Isso significa que tanto faz o grupo ao qual a flor pertence. Para simular desta situação é suficiente juntarmos os dados das duas espécies e sortear aleatóriamente a qual espécie a flor pertence. Para medir a diferença calculamos a média de cada espécie e fazemos

a diferença. Vamos repetir esse processo um grande número de vezes e ver como é a distribuição da estatística diferença. Após isso, basta calcular a diferença observada na amostra e identificar qual o percentual de vezes que ocorreu ela ou uma estatística mais extrema para obter o chamado p-valor. Implemente esse procedimento de forma sequencial e em paralelo. Compare o tempo computacional.

```
library(dplyr)
library(parallel)

dados <- iris %>% filter(Species %in% c("setosa", "versicolor"))
grupo <- dados$Species
valores <- dados$Sepal.Length

obs_diff <- mean(valores[grupo == "setosa"]) - mean(valores[grupo == "versicolor"])

teste_aleatorizacao_seq <- function(x, g, n_iter = 10000) {
  diffs <- numeric(n_iter)
  for (i in 1:n_iter) {
    g_perm <- sample(g)
    diffs[i] <- mean(x[g_perm == "setosa"]) - mean(x[g_perm == "versicolor"])
  }
  mean(abs(diffs) >= abs(obs_diff))
}

teste_aleatorizacao_par <- function(x, g, n_iter = 10000) {
  cl <- makeCluster(detectCores() - 1)
  clusterExport(cl, varlist = c("x", "g", "obs_diff"), envir = environment())
  diffs <- parSapply(cl, 1:n_iter, function(i) {
    g_perm <- sample(g)
    mean(x[g_perm == "setosa"]) - mean(x[g_perm == "versicolor"])
  })
  stopCluster(cl)
  mean(abs(diffs) >= abs(obs_diff))
}

mark(sequencial = teste_aleatorizacao_seq(valores, grupo),
     paralelo = teste_aleatorizacao_par(valores, grupo))
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 sequencial    344ms    344ms      2.91   43.67MB     10.2
## 2 paralelo      575ms    575ms      1.74    3.09MB      0
```

O resultado mostra que, apesar do uso de múltiplos núcleos, a versão paralela do teste de aleatorização foi mais lenta que a versão sequencial — levando cerca de 535 ms contra 308 ms da versão sequencial. Isso ocorre porque, para tarefas pequenas como essa (com apenas 100 observações e 10.000 permutações), o custo de inicialização e comunicação entre processos paralelos supera os ganhos de paralelização. A memória alocada pela versão sequencial foi maior, mas isso não compensou o tempo extra necessário na paralela. Em resumo, para tarefas leves, a versão sequencial é mais eficiente, enquanto a paralela tende a valer a pena apenas para volumes maiores de dados ou número de simulações significativamente maior.

```r
teste_aleatorizacao_seq <- function(x, g, n_iter = 100000) {
  diffs <- numeric(n_iter)
  for (i in 1:n_iter) {
    g_perm <- sample(g)
    diffs[i] <- mean(x[g_perm == "setosa"]) - mean(x[g_perm == "versicolor"])
  }
  mean(abs(diffs) >= abs(obs_diff))
}

teste_aleatorizacao_par <- function(x, g, n_iter = 100000) {
  cl <- makeCluster(detectCores() - 1)
  clusterExport(cl, varlist = c("x", "g", "obs_diff"), envir = environment())
  diffs <- parSapply(cl, 1:n_iter, function(i) {
    g_perm <- sample(g)
    mean(x[g_perm == "setosa"]) - mean(x[g_perm == "versicolor"])
  })
  stopCluster(cl)
  mean(abs(diffs) >= abs(obs_diff))
}

mark(sequencial = teste_aleatorizacao_seq(valores, grupo),
     paralelo = teste_aleatorizacao_par(valores, grupo))
```

```
## # A tibble: 2 x 6
##   expression      min   median `itr/sec` mem_alloc `gc/sec`
##   <bch:expr> <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
## 1 sequencial    3.43s    3.43s     0.291     436MB     3.50
## 2 paralelo   908.37ms 908.37ms     1.10     12.5MB     0
```

Com o aumento para 100.000 iterações, a versão paralela passou a ser mais eficiente, executando o teste em menos de 900 ms, contra 3 segundos da versão sequencial. Além de ser mais rápida, também consumiu menos memória (12.5 MB contra 436 MB), mostrando que o custo inicial da paralelização se dilui em tarefas maiores. Assim, para simulações mais extensas, o paralelismo se justifica e oferece ganhos claros de desempenho.