TestRandC

William Shih, Ricardo Simpao, Nilay Varshney, Luke Yee

3/17/2020

```
library(Rcpp)
library(RcppArmadillo)
library(rbenchmark)
set.seed(121)
sourceCpp("ctest.cpp")
source("rtest.R")
```

Benchmark calculating slope of p = 1, n = 200,000 with 100 replications

The C++ version is about 3 times faster.

	test	replications	elapsed	relative	user.self	sys.self
1	C++	100	0.18	1.000	0.12	0.03
4	R	100	0.76	4.222	0.59	0.13
2	lm.fit$coefficients[[2]]	100	2.83	15.722	2.13	0.47
3	lm$coefficients[[2]]	100	10.22	56.778	8.02	1.19

Benchmark multiplying 40*200000 by 200000*40 matrix with 5 replications

	test	replications	elapsed	relative	user.self	sys.self
4	C++ with RcppArmadillo	5	1.83	1.000	1.52	0.31
1	%*%	5	2.36	1.290	1.95	0.11
3	C++ with std::inner_product	5	2.86	1.563	2.70	0.14
2	C++ without std::inner_product	5	25.83	14.115	25.47	0.08

The C++ versions are slower than using %*% from base R when multiplying a 40*200000 matrix by 200000*40 matrix. The C++ version using std::inner_product is much faster than the C++ version without std::inner_product. Multiplying with RcppArmadillo is the fastest version for C++. Using std::inner_product multiplies a contiguous group of numbers by a contiguous group of numbers and sums them together. So we transpose the first matrix so that we can multiply a sequence of numbers by a sequence of numbers.

Benchmark multiplying 4*200000 by 200000*4 matrix

	test	replications	elapsed	relative	user.self	sys.self
1	%*%	100	1.01	1.000	0.84	0.16
3	C++ with std::inner_product	100	1.39	1.376	1.08	0.31
2	C++ without std::inner_product	100	1.56	1.545	1.39	0.17

	test	replications	elapsed	relative	user.self	sys.self
4	C++ with RcppArmadillo	100	1.81	1.792	0.91	0.89

The C++ versions are slower than using %*% from base R when multiplying a 4*200000 matrix by 200000*4 matrix. But C++ with std::inner_product is slightly slower than C++ without std::inner_product in this case. The sequence of numbers for std::inner_product is only 4 in this case, so the overhead of transposing the matrix is greater than the time save. The overhead from calling RcppArmadillo is also too much in this function as it turns out that C++ with RcppArmadillo is the slowest version.

Benchmark inverting 40*40 matrix with 100,000 replications

	test	replications	elapsed	relative	user.self	sys.self
2	C++ with RcppArmadillo	1e+05	6.06	1.000	5.86	0.08
1	solve	1e+05	13.44	2.218	13.15	0.05
3	C++ with Gauss-Jordan Elimination	1e+05	14.24	2.350	14.00	0.03

Inverting with RcppArmadillo is the fastest in this case. solve and C++ with Gauss-Jordan elimination is roughly the same.

Benchmark p = 4 and n = 200,000 Linear Regression with 100 replications

	test	replications	elapsed	relative	user.self	sys.self
7	R	100	3.17	1.000	2.86	0.25
6	C++ with RcppArmadillo	100	3.38	1.066	1.95	1.15
2	.lm.fit	100	3.62	1.142	3.31	0.17
3	lm.fit	100	4.25	1.341	3.82	0.31
4	C++ without std::inner_product	100	5.65	1.782	5.29	0.10
5	C++ with std::inner_product	100	7.69	2.426	6.92	0.21
1	lm	100	11.16	3.521	9.89	0.34

C++ with RcppArmadillo is the fastest with 4 variables and 200,000 rows for linear regression. C++ saves very little time compared to R for linear regression.

Benchmark p = 1 and n = 200,000 Linear Regression with 200 replications

	test	replications	elapsed	relative	user.self	sys.self
4	C++ without std::inner_product	200	2.00	1.000	1.86	0.11
2	.lm.fit	200	2.53	1.265	2.37	0.16
6	C++ with RcppArmadillo	200	2.60	1.300	1.81	0.78
7	R	200	3.09	1.545	2.79	0.30
3	lm.fit	200	3.14	1.570	2.93	0.17
5	C++ with std::inner_product	200	3.52	1.760	3.34	0.17
1	lm	200	10.54	5.270	10.12	0.40

The increased overhead with RcppArmadillo and std::inner_product makes C++ without std::inner_product the fastest.

Benchmark p = 40 and n = 200,000 Linear Regression with 5 replications

	test	replications	elapsed	relative	user.self	sys.self
6	C++ with RcppArmadillo	5	2.34	1.000	1.82	0.46
7	R	5	3.77	1.611	3.31	0.39
2	.lm.fit	5	4.82	2.060	4.48	0.19
3	lm.fit	5	5.67	2.423	4.91	0.36
1	lm	5	5.77	2.466	5.15	0.36
5	C++ with std::inner_product	5	8.14	3.479	7.53	0.31
4	C++ without std::inner_product	5	38.01	16.244	35.71	0.31

With more variables, C++ with RcppArmadillo saves relatively more time compared to R and other versions of C++.

Benchmark calling t distribution with 100,000 replications

	test	replications	elapsed	relative	user.self	sys.self
3	R	1e+05	0.72	1.000	0.70	0.00
1	C++ using Boost	1e+05	0.91	1.264	0.89	0.00
2	C++ calling R	1e+05	4.10	5.694	3.86	0.02

Calling the t distribution is fastest with R and slower calling C++ with boost. It is much slower to call C++, then call back R. So using boost is faster than calling back R in the C++ function.

Benchmark 95% confidence interval of linear regression (p = 20, n = 200,000) result with 10,000 replications

test	replications	elapsed	relative	user.self	sys.self
C++	10000	0.09	1.000	0.10	0.0
R	10000	0.30	3.333	0.29	0.0
confint.lm	10000	154.91	1721.222	136.48	13.9

C++ is 2.3 times faster than the R version and 1,300 times faster than confint.lm. Complex functions such as confint.lm takes a very long time to run.

Benchmark 95% prediction interval of linear regression (p = 20, n = 200,000) result with 10,000 replications

test	replications	elapsed	relative	user.self	sys.self
C++ with RcppArmadillo	10000	0.16	1.000	0.16	0.00
C++ with std::inner_product	10000	0.17	1.062	0.17	0.00
R	10000	0.31	1.937	0.29	0.00
predict.lm	10000	29.74	185.875	28.92	0.05

C++ with RcppArmadillo is slightly faster than C++ with std::inner_product for the prediction interval calculation. We get a small time save with C++ compared to R, about a 2 times difference. Again, C++

with RcppArmadillo is much faster than a complex function such as predict.lm, being 150 times faster.

Code Appendix

```
knitr::opts_chunk$set(echo = TRUE)
library(Rcpp)
library(RcppArmadillo)
library(rbenchmark)
set.seed(121)
sourceCpp("ctest.cpp")
source("rtest.R")

Random = runif(200000,0,100000)
AT = data.frame(x = 1:200000, y = 1:200000 + Random)


x1 = runif(200000,0,1000) + runif(200000, 500, 1000)
x2 = runif(200000,0,2000) + runif(200000, 1000, 2000)
x3 = runif(200000,2000,3000) + runif(200000, 1000, 2000)
x4 = runif(200000,2500,2750) + runif(200000, 1000, 2000)
y = x1 + x2 + x3 + x4 + Random^2 + runif(20000, 500, 777)
xFrame = as.matrix(data.frame(x1,x2,x3,x4))
x1 = runif(8000000,0,1000) + runif(8000000, 500, 1181)
FourtyX = matrix(x1, nrow = 200000, ncol = 40)
knitr::kable((benchmark("C++" = {calc_slopeC(AT)},
          "lm.fit$coefficients[[2]]" = {lm.fit(as.matrix(data.frame(1, AT[[1]])), AT[[2]])$coefficients
          "lm$coefficients[[2]]" = {lm(y ~ x, data = AT)$coefficients[[2]]},
          "R" = {calc_slope(AT)},
          replications = 100, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self"))))
knitr::kable(benchmark("%*%" = {t(FourtyX) %*% FourtyX},
          "C++ without std::inner_product" = {multiply(t(FourtyX), FourtyX)},
          "C++ with std::inner_product" = {multiply2(t(FourtyX), FourtyX)},
          "C++ with RcppArmadillo" = {armamultiply(t(FourtyX), FourtyX)},
          replications = 5, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self")))
knitr::kable(benchmark("%*%" = {t(xFrame) %*% xFrame},
          "C++ without std::inner_product" = {multiply(t(xFrame), xFrame)},
          "C++ with std::inner_product" = {multiply2(t(xFrame), xFrame)},
          "C++ with RcppArmadillo" = {armamultiply(t(xFrame), xFrame)},
          replications = 100, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self")))

InvertThis = t(FourtyX) %*% FourtyX
knitr::kable(benchmark("solve" = {solve(InvertThis)},
          "C++ with RcppArmadillo" = {armainverse(InvertThis)},
          "C++ with Gauss-Jordan Elimination" = {inverse(InvertThis)},
          replications = 100000, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self")))
#p = 4
knitr::kable((benchmark("lm" = {lm(y ~ xFrame)},
          ".lm.fit" = {.lm.fit(as.matrix(data.frame(1, xFrame)), y)},
          "lm.fit" = {lm.fit(as.matrix(data.frame(1, xFrame)), y)},
```

```
          "C++ without std::inner_product" = {linear_regC(xFrame, y)},
          "C++ with std::inner_product" = {linear_regC2(xFrame, y)},
          "C++ with RcppArmadillo" = {linear_regC3(xFrame, y)},
          "R" = {linear_reg(xFrame, y)},
            replications = 100, order = "elapsed",
            columns = c("test","replications","elapsed","relative","user.self","sys.self"))))
# p =1
knitr::kable((benchmark("lm" = {lm(AT[[2]] ~ AT[[1]])},
          ".lm.fit" = {.lm.fit(as.matrix(data.frame(1, AT[[1]])), as.vector(AT[[2]]))},
          "lm.fit" = {lm.fit(as.matrix(data.frame(1, AT[[1]])), as.vector(AT[[2]]))},
          "C++ without std::inner_product" = {linear_regC(as.matrix(AT[[1]]), as.vector(AT[[2]]))},
          "C++ with std::inner_product" = {linear_regC2(as.matrix(AT[[1]]), as.vector(AT[[2]]))},
          "C++ with RcppArmadillo" = {linear_regC3(as.matrix(AT[[1]]), as.vector(AT[[2]]))},
          "R" = {linear_reg(as.matrix(AT[[1]]), as.vector(AT[[2]]))},
          replications = 200, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self"))))
#p = 40
knitr::kable((benchmark("lm" = {lm(y ~ FourtyX)},
          ".lm.fit" = {.lm.fit(as.matrix(data.frame(1, FourtyX)), y)},
          "lm.fit" = {lm.fit(as.matrix(data.frame(1, FourtyX)), y)},
          "C++ without std::inner_product" = {linear_regC(FourtyX, y)},
          "C++ with std::inner_product" = {linear_regC2(FourtyX, y)},
          "C++ with RcppArmadillo" = {linear_regC3(FourtyX, y)},
          "R" = {linear_reg(FourtyX, y)},
          replications = 5, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self"))))
knitr::kable(benchmark("C++ using Boost" = {tc(0.99, 55)},
          "C++ calling R" = {tr(0.99,55)},
          "R" = {qt(0.99, 55)},
          replications = 100000, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self")))

ThirtyX = FourtyX[,1:20]
colnames(ThirtyX) = c(paste0("X",1:20))
l = lm(y ~ ., data = data.frame(cbind(ThirtyX,y)))
l$effects = NULL
z = linear_reg(ThirtyX, y)

knitr::kable(benchmark("C++" = {lr_coefficient_CI_C(z, 0.05)},
          "R" = {lr_coefficient_CI(z, 0.05)},
          "confint.lm" = {confint.lm(l)},
          replications = 10000, order = "elapsed",
          columns = c("test","replications","elapsed","relative","user.self","sys.self")))

data = runif(21,0,10)
data = data.frame(t(data))
datanum = as.numeric(data)[-1]
colnames(data) = names(l$coefficients)[-1]
knitr::kable(benchmark("C++ with RcppArmadillo" = {lr_prediction_interval_C(z, datanum, 0.05)},
          "C++ with std::inner_product" = {lr_prediction_interval_C2(z, datanum, 0.05)},
          "R" = {lr_prediction_interval(z, datanum, 0.05)},
          "predict.lm" = {predict(l, data, interval = "prediction", level = 0.95)},
          replications = 10000, order = "elapsed",
```

```
          columns = c("test","replications","elapsed","relative","user.self","sys.self")))
```