Techniki eksploracji danych

Krzysztof Gajowniczek

Rok akademicki: 2020/2021

Section 1

Pomiar czasu wykonania i profilowanie kodu

- Co to jest **kod wysokiej jakości**? Z pewnością musi robić dokładnie to, do czego powinien (np. jest zgodny ze specyfikacją techniczną).

Ponadto musi być:

- **czytelny** - czytelnik jest w stanie zrozumieć intencje programisty,
- **testowalny** - zorganizowany w taki sposób, że testy jednostkowe są możliwe,
- **możliwy do utrzymania** - poprawki i ulepszenia są stosunkowo łatwe do wykonania,
- **przenośny** - daje takie same wyniki na różnych platformach.

- Wysokiej jakości kod jest również **ekonomiczny**: charakteryzuje się dużą szybkością i niskim zużyciem pamięci.

Subsection 1

Pomiar czasu wykonania

- Rozważmy następujące implementacje tego samego algorytmu

```
ex1 <- sum

ex2 <- function(x) {
  res <- 0
  for (e in x) res <- res + e
  res
}
```

- Istnieje kilka sposobów mierzenia czasu wykonywania bloku kodu w „R”. `system.time()` „uruchamia” licznik czasu, ocenia fragment kodu i zwraca czas, który upłynął (w sekundach).

```
x <- runif(10000000)
system.time(ex1(x))
```

```
##    user  system elapsed
##   0.008   0.000   0.009
```

```
system.time(ex2(x))
```

```
##    user  system elapsed
##   0.220   0.001   0.220
```

- `user` - czas wykonania instrukcji użytkownika procesu wywołującego.
- `system` - czas wykonania wywołań systemowych w imieniu procesu wywołującego.
- `elapsed` - całkowity czas, który upłynął.

- `rbenchmark::benchmark()` idzie o poziom wyżej. Jesteśmy w stanie zbadać kilka fragmentów kodu naraz i poinstruować funkcję, aby rozważyła liczbę powtórzeń naszego eksperymentu

```
x <- runif(1000)
rbenchmark::benchmark(ex1(x), ex2(x), replications = 100)
```

```
##     test replications elapsed relative user.self sys.sel
## 1 ex1(x)          100   0.001        1     0.000
## 2 ex2(x)          100   0.002        2     0.002
```

- Mimo wszystko jest to proste opakowanie otaczające `system.time()`. Nawiasem mówiąc, rozkład czasu wykonywania jest najczęściej przekrzywiony w prawo:

- `elapsed` czas zależy od innych procesów w tle (np. Odtwarzacz MP3),
- `user` na czas użytkownika ma wpływ m.in. **garbage collector** „R".

- Z drugiej strony, `microbenchmark::microbenchmark()` opiera się na znacznie bardziej precyzyjnej metodzie pomiaru czasu.

```
x <- runif(100)
microbenchmark::microbenchmark(ex1(x), ex2(x), times=100)
```

```
## Unit: nanoseconds
##     expr  min   lq     mean median     uq  max neval cld
##  ex1(x)  207  244  282.12    262  283.5 1657   100  a
##  ex2(x) 2200 2248 2325.04   2271 2307.0 6003   100   b
```

```
microbenchmark::microbenchmark(1, {1}, (1), times=100000)
```

```
## Unit: nanoseconds
##       expr min lq     mean median uq   max neval cld
##          1   2 11 11.51097     12 12 26800 1e+05 a
##  {     1 }  30 33 37.32041     34 41 26455 1e+05  b
##       (1)  30 41 43.88963     42 44 13076 1e+05   c
```

Subsection 2

Profilowanie kodu

- Większość użytkowników R chciałaby, aby kod działał szybciej.
- Jednak nie zawsze jest jasne, jak to osiągnąć.
- Powszechnym podejściem jest poleganie na intuicji i mądrości szerszej społeczności „R" na temat przyspieszania kodu „R".
- Jedną z wad jest to, że może to prowadzić do skupienia się na optymalizacji rzeczy, które w rzeczywistości zajmują niewielką część całkowitego czasu pracy.

- Załóżmy, że pętla działa 5 razy szybciej.
- To brzmi jak ogromna poprawa, ale jeśli ta pętla zajmuje tylko 10% całkowitego czasu, to ogólnie przyspiesza to tylko 8%.
- Inną wadą jest to, że chociaż wiele powszechnie uznawanych przekonań jest prawdziwych (na przykład wstępne przydzielenie pamięci może przyspieszyć działanie), niektóre z nich nie są (np. funkcje `*apply` są z natury szybsze niż w przypadku pętli).
- Może to prowadzić do poświęcenia czasu na „optymalizacje", które tak naprawdę nie pomagają.
- Aby powolny kod był szybszy, potrzebujemy dokładnych informacji o tym, co spowalnia nasz kod.

- `Profvis` (Interactive Visualizations for ProfilingR Code) to narzędzie pomagające zrozumieć, jak `R` spędza swój czas. Zapewnia interaktywny interfejs graficzny do wizualizacji danych z `Rprof`, wbudowanego narzędziaR do zbierania danych profilowych.

- Poniżej znajduje się przykład użycia `profvis`.
- Kod tworzy wykres punktowy zestawu danych ‘diamonds’, który ma około 54 000 wierszy, estymuje model liniowy i rysuje linię dla modelu.

```
library(profvis)
profvis({
  data(diamonds, package = "ggplot2")

  plot(price ~ carat, data = diamonds)
  m <- lm(price ~ carat, data = diamonds)
  abline(m, col = "red")
})
```

- Na górze znajduje się kod, a na dole wykres płomienia.
- Na wykresie płomienia kierunek poziomy przedstawia czas w milisekundach, a kierunek pionowy reprezentuje stos wywołań.
- Patrząc na najniższe pozycje na stosie, większość czasu, około 2 sekund, spędza się w `plot`, a następnie znacznie mniej czasu w `lm` i prawie wcale nie jest spędzony w „abline" - nie pojawia się nawet na wykresie płomienia.

- `Memory`: przydzielona lub cofnięta pamięć (dla liczb ujemnych) dla danego stosu wywołań. Jest przedstawiane w megabajtach i zagregowane we wszystkich stosach wywołań w kodzie w danym wierszu.
- `Time`: czas spędzony w milisekundach. To pole jest również agregowane dla wszystkich stosów wywołań wykonanych na kodzie w danym wierszu.

Section 2

Integracja R z C++

- Biblioteka `Rcpp` może służyć do usuwania wąskich gardeł w wydajności R.
- Aby z niej skorzystać, będziemy potrzebować następujących narzędzi:

- Windows users: 'Rtools'
- OS X users: 'Xcode'
- Linux users: `yum install gcc gcc-c++ git` etc.

Subsection 1

Język programowania C++

- C++ to **kompilowany** (nie: interpretowany) uniwersalny język programowania.
- Jest przenośny, zorientowany obiektowo, ogólny i zapewnia narzędzia do manipulacji pamięcią niskiego poziomu.
- Jest rozwijany przez Bjarne Stroustrupa od 1979 roku jako rozszerzenie języka programowania C.
- Początkowo został znormalizowany w 1998 roku jako ISO/IEC 14882:1998. C++ charakteryzuje się wysoką wydajnością i elastycznością.
- Można całkiem bezpiecznie założyć, że C/C ++ generuje najszybszy kod maszynowy, oczywiście o ile nie dotyczy to niektórych zaawansowanych poprawek na poziomie maszyny, równoległości itp.
- Należy pamiętać, że różne optymalizacje specyficzne dla kompilatora mogą wpływać na wydajność kodu.

Subsection 2

Rcpp - motywacja

- R jest zaimplementowany w C (R oraz Fortranie). W rezultacie wszystko, co możemy zrobić w R, można zaimplementować w C/C ++. Na przykład następujące funkcje bezpośrednio wywołują skompilowany kod:

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

```
c
```

```
## function (...)  .Primitive("c")
```

- *R/C API* zapewnia najszybszy sposób (pod względem wydajności) komunikowania się z „R" ze skompilowanego kodu.
- Jednak zdecydowanie nie jest to najwygodniejsze. Wszystkie obiekty `R` są obsługiwane za pomocą typu `SEXP`, który jest wskaźnikiem do struktury `SEXPREC`.

Table 1: R/C API mapping struktur danych.

SEXPTYPE	**R equivalent**
REALSXP	numeric with storage mode double
INTSXP	integer
LGLSXP	logical
STRSXP	character
VECSXP	list (generic vector)
NILSXP	NULL
SYMSXP	name/symbol
CLOSXP	function or function closure
ENVSXP	environment

- Na przykład to jest kod C/C ++ odpowiadający wywołaniu R polecenia c(123.45, 67.89):

```
SEXP createVectorOfLength2(){ // R / C API
  SEXP result ;
  result = PROTECT(allocVector(REALSXP, 2));
  REAL (result)[0] = 123.45;
  REAL (result)[1] = 67.89;
  UNPROTECT(1);
  return result;
}
```

- Interfejs API R/C może być trudny do nauczenia się i używania dla wielu użytkowników R. Dlatego w tym module omówimy pakiet Rcpp. Upraszcza pisanie skompilowanego kodu. Wystarczy porównać powyższe z:

```
NumericVector createVectorOfLength2  { // Rcpp
  return NumericVector::create(123.45, 67.89);
}
```

- Rcpp to zestaw wygodnych opakowań (ang. wrappers) C++ dla całego interfejsu API R/C.
- Możemy go użyć do usunięcia wąskich gardeł wydajnościowych w naszym kodzie R, zaimplementować kod, który jest trudny do wektoryzacji, lub gdy potrzebujemy zaawansowanych algorytmów, rekurencji lub struktur danych.

Subsection 3

Rcpp - przykłady

- Obliczmy n-tą liczbę Fibonacciego. Sekwencja wyjściowa: $(1, 1, 2, 3, 5, 8, 13, 21, ...)$.

```
fib1 <- function(n) {
  if (n <= 1) return(1)
  last12 <- c(1, 1)
  for (i in 2:n)
    last12 <- c(last12[1]+last12[2], last12[1])
  last12[1]
}
sapply(0:7, fib1)
```

```
## [1]  1  1  2  3  5  8 13 21
```

- A tak możemy to zaimplementować w Rcpp:

```
Rcpp::cppFunction("
  int fib2(int n) {
  if (n <= 1) return 1;
  int last1 = 1;
  int last2 = 1;
  for (int i=2; i<=n; ++i) {
    int last3 = last2;
    last2 = last1;
    last1 = last2+last3;
  }
  return last1;
}
")
```

- Kilka benchmarków:

```
microbenchmark::microbenchmark(fib1(25), fib2(25))
```

```
## Unit: microseconds
##       expr   min    lq     mean median     uq     max neva
##  fib1(25) 4.838 5.117 5.68801  5.553 6.0045  11.010   10
##  fib2(25) 1.018 1.090 8.10859  1.120 1.2045 669.424   10
```

- Oprócz użycia "inline" lub `Rcpp`, możemy umieścić nasz kod C++ w oddzielnych plikach źródłowych.
- Oto zawartość pliku `test.cpp`:

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int fib2 (int n) {
  if (n <= 1) return 1;
  int last1 = 1;
  int last2 = 1;
  for (int i = 2; i <= n; ++i) {
    int last3 = last2;
    last2 = last1;
    last1 = last2 + last3;
  }
  return last1;
}
```

- Aby skompilować plik źródłowy, wywołujemy:

```
Rcpp::sourceCpp("test.cpp")
```

Subsection 4

Podstawowe typy atomowe R w Rcpp

- W R, *wektoryzacja* jest oczywista. Na przykład macierze, szeregi czasowe, factors, ramki danych bazują na wektorach.

- Poniżej znajdują się typy wektorów atomowych R porównane z ich odpowiednikami w Rcpp.

typeof()	**mode()**	**Rcpp class**	**elem type**
logical	logical	LogicalVector	int
integer	numeric	IntegerVector	int
double	numeric	NumericVector	double
character	character	CharacterVector	Rcpp::String

- Powinniśmy pamiętać, że w C/C ++ pierwszy element wektora ma indeks 0. Nawiasem mówiąc, interfejs R `cpp::Vector` jest z grubsza zgodny ze standardem STL `std::vector`.

- Napiszmy funkcję obliczającą sumę elementów w `NumericVector`.

```
Rcpp::cppFunction('
  double sum2(NumericVector x) {
    int n = x.size(); // a method
    double result = 0.0;
    for (int i=0; i<n; ++i)
    result += x[i]; // or x(i)
    return result;
}
')
```

Subsection 5

Rcpp obsługuje typy niestandardowe

Listy

- Lista R to sekwencja (wektor) składająca się z dowolnego rodzaju obiektów R (dowolny SEXP w R/C API, dowolny RObject w Rcpp).

```
Rcpp::cppFunction('
  List list_test() {
    List out(2);
    out[0] = CharacterVector::create("one", "two");
    out[1] = List::create(1, 2.0);
    return out;
  }
')
str(list_test())
```

Funkcje

- Funkcja R działa na niektóre RObject i generuje RObject.

```
Rcpp::cppFunction('
  RObject some_call(Function f, RObject x) {
    return f(x);
  }
')
some_call(runif, 5)
```

Typy złożone

- `factor`
- `matrix`
- `data.frame`

Section 3

Literatura

- R Core Team, *Writing 'R' extensions*, 2014, Sec. 3
- Matloff N., *The art of R programming*, No Starch Press, 2011, Chap. 14
- Cormen T.H., Leiserson C.E., Rivest R.L., Stein C., *Introduction to Algorithms*, MIT Press and McGrawHill, 2009
- Knuth D.E., *The art of computer programming*, Vols. 1 and 3, Addison-Wesley, 1997

- Eddelbuettel D., *Seamless R and C++ Integration with Rcpp*, Springer, 2013
- *en.cppreference.com* – C++ reference
- Stroustrup B., *A Tour of C++*, Addison-Wesley, 2013
- Stroustrup B., *The C++ Programming Language*, Addison-Wesley, 2000
- Josuttis N.M., *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley, 2012
- *rcpp.org* – Rcpp homepage
- *Rcpp API documentation (Doxygen)*
- *Using Rcpp with RStudio*
- *High performance functions with Rcpp* by H. Wickham