STAT 33B Homework 6 Solutions

Yuanrui Zhu (3034615728)

Nov 19, 2020

This homework is due **Nov 19, 2020** by 11:59pm PT.

Homeworks are graded for correctness.

As you work, write your answers in this notebook. Answer questions with complete sentences, and put code in code chunks. You can make as many new code chunks as you like.

Please do not delete the exercises already in this notebook, because it may interfere with our grading tools.

You need to submit your work in two places:

- Submit this Rmd file with your edits on bCourses.
- Knit and submit the generated PDF file on Gradescope.

If you have any last-minute trouble knitting, **DON'T PANIC**. Submit your Rmd file on time and follow up in office hours or on Piazza to sort out the PDF.

Profiling

The purpose of this homework is to practice profiling and optimizing code.

The function in the next section is quite slow. The exercises will walk you through various performance improvements to the function.

Note 1: When benchmarking or profiling code, other programs running on your computer can affect the results. So when you benchmark or profile code, first make sure you don't have lots of other programs running. Close your web browser, word processor, media player, etc.

Note 2: Each time you profile code the results will vary slightly, even if you don't change the code. Thus if you are investigating an optimization that appears to have a small effect, you may want to profile the code several times to determine whether the effect is real or due to random variation.

Note 3: The code cells in this notebook are all set up with `eval=FALSE` so that they don't actually run the code they contain. Your notebook may fail to knit if you put profiler code in a cell that doesn't have `eval=FALSE`. You do not need to show us the output from profvis; just tell us about what you observed.

The Function

The `sample_markov` function, shown below, generates a vector that represents the state (A or B) of a system over time. Each element corresponds to one time step.

At each time step, the system transitions to a new state. If the system is in state A, then at the next time step it can either stay in state A, or switch to state B. If the system is in state B, then at the next time step it can either switch to state A, or stay in state B. The transition is random, with a fixed probability for each of the possible transitions.

For instance, suppose the probability of A to B is 0.3, and the probability of B to B is 0.4. We can use these to compute the other probabilties. The probability of A to A must be $1 - 0.3 = 0.7$, and the probability of B to A must be $1 - 0.4 = 0.6$.

This random system is called a 2-state Markov chain.

The parameters of the function are:

- `n`, the number of time steps to run the system, which equals the length of the returned vector.
- `p`, a length-2 vector that contains the probability of A to B and B to B, respectively.
- `init`, the initial state of the system.

The function is defined as:

```r
sample_markov = function(n, p = c(0.6, 0.7), init = "A") {
  chain = init

  for (i in 1:n) {
    # Check the previous state, chain[i].
    if (chain[i] == "A") {
      # Get the probabilities for {A to A} and {A to B}.
      prob = c(1 - p[1], p[1])
      # Randomly sample new state using the probabilities.
      x = sample(c("A", "B"), 1, prob = prob)
      chain = c(chain, x)

    } else {
      # Get the probabilities for {B to A} and {B to B}.
      prob = c(1 - p[2], p[2])
      # Randomly sample new state using the probabilities.
      x = sample(c("A", "B"), 1, prob = prob)
      chain = c(chain, x)
    }
  }

  chain[-1]
}
```

Round 1

Use profvis to profile the `sample_markov` function. Make sure the `n` argument in your call to `sample_markov` is large enough that the function runs for at least 30 seconds. You can use the default arguments for the other parameters.

The `sample_markov` function uses a loop, but doesn't preallocate the vector that stores the results. Record the total time it takes for the function to run, and describe evidence you can see in the profile that failure to preallocate hurts the performance of the function.

Finally, edit the `sample_markov` function so that it preallocates the vector that stores the results. Use profvis to profile the edited version (with the same arguments). Record the total time it takes for the function to run, and comment on how the performance changed.

YOUR ANSWER GOES HERE:

```r
# Your code to profile the original function goes here:
library(profvis)
profvis({
  sample_markov = function(n, p = c(0.6, 0.7), init = "A") {
    chain = init
    for (i in 1:n) {
      # Check the previous state, chain[i].
      if (chain[i] == "A") {
        # Get the probabilities for {A to A} and {A to B}.
        prob = c(1 - p[1], p[1])
        # Randomly sample new state using the probabilities.
        x = sample(c("A", "B"), 1, prob = prob)
        chain = c(chain, x)

      } else {
        # Get the probabilities for {B to A} and {B to B}.
        prob = c(1 - p[2], p[2])
        # Randomly sample new state using the probabilities.
        x = sample(c("A", "B"), 1, prob = prob)
        chain = c(chain, x)
      }
    }

    chain[-1]
  }
  sample_markov(1e5)
})
```

```r
# Your edited function and the code to profile it go here:

profvis({
  sample_markov = function(n, p = c(0.6, 0.7), init = "A") {
    chain = character(n + 1)
    chain[1] = init

    for (i in 1:n) {
      # Check the previous state, chain[i].
      if (chain[i] == "A") {
        # Get the probabilities for {A to A} and {A to B}.
        prob = c(1 - p[1], p[1])
        # Randomly sample new state using the probabilities.
        x = sample(c("A", "B"), 1, prob = prob)
        chain[i + 1] = x

      } else {
        # Get the probabilities for {B to A} and {B to B}.
        prob = c(1 - p[2], p[2])
        # Randomly sample new state using the probabilities.
        x = sample(c("A", "B"), 1, prob = prob)
        chain[i + 1] = x
      }
    }
    chain[-1]
  }
  sample_markov(1e5)
})
```

Round 2

In your new version of the `sample_markov` function, the most time-consuming lines should be `x = sample(c("A", "B"), 1, prob = prob)`.

In some cases, using numbers instead of strings can make a computation significantly faster, and here it is possible to represent the two states in the chain as 0 and 1 instead of A and B.

Edit your `sample_markov` function from round 1 so that it uses the numbers 0 and 1 to represent the states instead of the strings "A" and "B". Use profvis to profile the edited version (with the same arguments). Record the total time it takes for the function to run, and comment on how the performance changed.

YOUR ANSWER GOES HERE:

```r
# Your code goes here.
library(profvis)
profvis({
  sample_markov = function(n, p = c(0.6, 0.7), init = 0) {
    chain = numeric(n + 1)
    chain[1] = init

    for (i in 1:n) {
      # Check the previous state, chain[i].
      if (chain[i] == 0) {
        # Get the probabilities for {A to A} and {A to B}.
        prob = c(1 - p[1], p[1])
        # Randomly sample new state using the probabilities.
        x = sample(c(0, 1), 1, prob = prob)
        chain[i + 1] = x

      } else {
        # Get the probabilities for {B to A} and {B to B}.
        prob = c(1 - p[2], p[2])
        # Randomly sample new state using the probabilities.
        x = sample(c(0, 1), 1, prob = prob)
        chain[i + 1] = x
      }
    }

    chain[-1]
  }
  sample_markov(1e5)
})
```

Round 3

Switching from strings to numbers to represent the states should give your function a small but measurable speed boost.

Among experienced R users, it's well known that the general-purpose `sample` function is slower than R's functions for sampling from specific distributions. If you want to sample 0s and 1s, the `rbinom` function is significantly faster.

The `rbinom` function samples values from the "binomial distribution", which models the number of successes in a series of identical, random trials. Each trial randomly gives a failure (0) or a success (1). You can sample a single 0 or 1 with `rbinom` by setting both the number of observations and the number of trials to 1. See the documentation for details.

Edit your `sample_markov` function from round 2 so that it uses `rbinom` to sample the 0s and 1s instead of `sample`. In doing so, it should be possible to eliminate the variables `prob` and `x`.

Use profvis to profile the edited version (with the same arguments). Record the total time it takes for the function to run, and comment on how the performance changed.

YOUR ANSWER GOES HERE:

```r
# Your code goes here.
library(profvis)
profvis({
  sample_markov = function(n, p = c(0.6, 0.7), init = 0) {
    chain = numeric(n + 1)
    chain[1] = init

    for (i in 1:n) {
      # Check the previous state, chain[i].
      if (chain[i] == 0) {
        # p[1] correspond to probability of 0 -> 1
        chain[i + 1] = rbinom(1, 1, prob = p[1])

      } else {
        # p[2] correspond to probability of 1 -> 1
        chain[i + 1] = rbinom(1, 1, prob = p[2])
      }
    }

    chain[-1]
  }
  sample_markov(1e5)
})
```

Round 4

Switching from `sample` to `rbinom` should give your function a signifcant speed boost.

Another strategy for improving performance is to move code out of loops, either because it doesn't need to be done multiple times, or because it can be vectorized instead. Generally the code to start with is whatever takes the longest to run in the loop.

In your `sample_markov` function, the calls to `rbinom` take up most of the time in the loop. Code that samples random values is an especially good candidate for vectorization, because sampling is time-consuming, and all of R's sampling functions are vectorized.

However, we can't just sample `n` values with `rbinom` outside the loop, because the probability of success (1) varies. At each time step, the probability of transitioning to 1 depends on the state in the previous time step.

Fortunately, we can solve this problem by using different strategy for sampling 0s and 1s. Suppose we want the probabilty of getting a 1 to be `p`. Then one sampling strategy is:

1. Uniformly sample a decimal value from the interval 0 to 1. In R, we can do this with the `runif` function.
2. Compare the step 1 value to `p`. If it is less than `p`, count it as 1; otherwise, count it as 0.

The advantage of this strategy is that the sampling happens in step 1, but we don't need to know the value of `p` until step 2.

Edit your `sample_markov` function from round 3 so that it uses the sampling strategy described above to sample 0s and 1s. In particular, use a vectorized call to `runif` *outside the loop* to generate a step 1 value for all `n` time steps. In the loop, use `<` to compare the appropriate step 1 value to the appropriate probability from `p`, and thereby determine whether the step 2 value is 0 or 1. Note that you can avoid adding an extra if-statement by using the fact that `as.integer` converts `FALSE` to 0 and `TRUE` to 1.

Use profvis to profile the edited version (with the same arguments). Record the total time it takes for the function to run, and comment on how the performance changed.

Also comment on how the performance of this final version of the function compares to the version at the beginning of the homework.

YOUR ANSWER GOES HERE:

```r
# Your code goes here.
profvis({
  sample_markov = function(n, p = c(0.6, 0.7), init = 0) {
    chain = numeric(n + 1)
    chain[1] = init
    step_one = runif(n)
    for (i in 1:n) {
      # Check the previous state, chain[i].
      if (chain[i] == 0) {
        # p[1] correspond to probability of 0 -> 1
        chain[i + 1] = as.integer(step_one[i] < p[1])

      } else {
        # p[2] correspond to probability of 1 -> 1
        chain[i + 1] = as.integer(step_one[i] < p[2])
      }
    }

    chain[-1]
  }
  sample_markov(1e5)
})
```