STAT 447 Assignment 3

Caden Hewlett

2024-01-24

Question 1 : Functions on the Unit Interval

For this question, use Simple Monte Carlo. The main twist compared to week one is that you will use a continuous random variable.

Part 1

Write a function called `mc_estimate` that takes a function $f : [0, 1] \to \mathbb{R}$ and outputs the Monte Carlo estimator of $\int_0^1 f(x)dx$ using $n = 100,000$ independent samples from unif$(0, 1)$.

NOTE: Because my computer could handle it, I used $M = 100,000$ rather than $M = 10,000$ just for fun. The results are similar (but obviously more precise for larger M.)

The function is created using the code below:

```
mc_estimate <- function(f){
  # declare the number of iterations
  M <- 100000
  # randomly generate M total observations in [0, 1]
  m_vals <- runif(M)
  # then, for all m in random generations, evaluate f(m)
  G_m <- f(m_vals)
  # then compute the average
  G_hat_m <- (1/M)*sum(G_m)
  # and return
  return(G_hat_m)
}
```

Part 2

Consider the function $f : [0, 1] \to [0, \infty)$ given by:

$$f(x) = \frac{1}{\sqrt[3]{x^2(1-x)}}$$

Note, importantly, that

$$\mathcal{I}_1 = \int_0^1 f(x)\mathrm{d}x = \frac{\pi}{\sin\left(\frac{\pi}{3}\right)}$$

Test your implementation of `mc_estimate` by checking that it produces an answer close to the value above.

We'll start by computing the actual result:

```
expected = pi / (sin(pi/3))
expected
```

```
## [1] 3.627599
```

Now, we implement `mc_estimate`.

```
# Apollo 11 moon landing, as an integer
set.seed(19690720)

f <- function(x){
  1 / ( ((x^2)*(1 - x))^(1/3) )
}

observed <- mc_estimate(f)
observed
```

```
## [1] 3.629373
```

It seems we got pretty close! Let's calculate the percent difference.

```
(abs(observed - expected) / expected)*100
```

```
## [1] 0.04892034
```

It looks like there's about a 0.05% difference between $\hat{G}_M$ for $M = 100,000$.

For completeness, I also ran the code with $M = 10,000$ and observed approximately a 2.52% difference.

Part 3

The following integral, known as the sine integral, does not admit a closed-form expression.

$$\mathcal{I}_2 = \int_0^1 \frac{\sin(t)}{t} \mathrm{d}t$$

It does not admit a closed-form expression. Estimate its value using `mc_estimate(f)`.

To test our Monte Carlo Approximation, we will evaluate $\mathrm{Si}(1)$ using the `pracma` package.

```
Si(1)
```

```
## [1] 0.9460831
```

Now, we can see how close our `mc_estimate` gets. We let $g(x) = \frac{\sin(x)}{x}$.

```r
set.seed(19690720)

g <- function(x){
  sin(x) / x
}
observed = mc_estimate(g)
observed
```

```
## [1] 0.9464033
```

It looks like we got really close! Let's find the percent difference:

```r
expected = Si(1)
# we overwrite our old variables for convenience (and memory)
(abs(observed - expected) / expected)*100
```

```
## [1] 0.03384555
```

It looks like there's about a 0.03% difference between $\hat{G}_M$ for $M = 100,000$.

For completeness, I also ran the code with $M = 10,000$ and observed approximately a 0.09% difference.

Question 2 : Implementing SNIS for simPPLe

Part 1

First, install the package `distr`.

Since this is a `.Rmd` file, all packages are loaded at the beginning in a hidden cell with suppressed warnings.

```r
"distr" %in% installed.packages()
```

```
## [1] TRUE
```

Part 2

Read this short tutorial on `distr`. Nothing to submit for this item.

For completeness, I will work through the tutorial on my own and place the results here.

```r
# Commented out to avoid spamming outputs
# distPoisson <- Pois(lambda = 3.2)
#
# d(distPoisson)(2)
#
# sum(r(distPoisson)(10))
#
# d(sqrt(distPoisson))(2)
```

Part 3

Read the "scaffold code", and use `distr` and two of the functions in the example to create a fair coin, flip it, and to compute the probability of that flip:

The scaffold code is given below:

```
## Utilities to make the distr library a bit nicer to use

p <- function(distribution, realization) {
  d(distribution)(realization) # return the PMF or density
}

Bern = function(probability_to_get_one) {
  DiscreteDistribution(supp = 0:1, prob = c(1-probability_to_get_one, probability_to_get_one))
}

## Key functions called by simPPLe programs

# Use simulate(distribution) for unobserved random variables
simulate <- function(distribution) {
  r(distribution)(1) # sample once from the given distribution
}

# Use observe(realization, distribution) for observed random variables
observe = function(realization, distribution) {
  # `<<-` lets us modify variables that live in the global scope from inside a function
  weight <<- weight * p(distribution, realization)
}
```

The documented code to complete this task is shown below:

```
# create the coin (Bernoulli distribution)
coin <- Bern(1/2)
# flip the coin once
flip <- simulate(coin)
# compute the probability of the flip
prob <- p(coin, flip)
# report the findings
print(paste("The coin flipped a ", flip,
      ". The probability of this flip is ",
       prob, sep = ""))
```

```
## [1] "The coin flipped a 1. The probability of this flip is 0.5"
```

Part 4

Complete the implementation of the function `posterior`:

To do this, let's consider what inputs we are given, relative to the Importance Sampling procedure we discussed in lecture.

For starters, we are given M with the `number_of_iterations` parameter. We're also given the proposal function $q(x)$ with the `ppl_function`.

Notably, we aren't given a test function $g(x)$ so we will assume it is the indicator function.

We will use:

$$g(x) = \mathbb{1}(X = 1)$$

So we can approximate $P(X = 1 \mid Y = y)$.

Following the Algorithm, the first thing we have to do is find $(X^{(m)})$ for iteration $m \in [1, \ldots, M] \subseteq \mathbb{N}$.

We'll do this by adding the line defining the `m` variable.

Further, we'll define the function $g(x)$ prior to the `for` loop, explicitly as a function. This means we can return to this code later and improve it (perhaps allowing $g(x)$ to be passed as a parameter.) This allows us to compute $G^{(m)} = g(X^{(m)})$.

```r
posterior = function(ppl_function, number_of_iterations) {
  numerator = 0.0
  denominator = 0.0
  # add simple indicator function
  g <- function(x){x == 1}
  for (i in 1:number_of_iterations) {
    # Step 1: Simulate Iteration m
    m <- simulate(ppl_function)
    # Step 2: Compute G^m = g(X^m)
    G <- g(m)
    # Step 3: Compute gamma(X^m) p(X)L(y|x)
    gamma <- p(ppl_function, m)
    # Step 4: Compute w(X^m) = gamma(X^m)/q(X^m)
    weight <<- 1.0
    observe(m, ppl_function)
    # Step 5: Add this to our rolling weight total
    denominator <<- denominator + weight
    # Step 6: Update the rolling numerator
    numerator <<- numerator + (w*G)
  }
  return(numerator/denominator)
}
```

Part 5

Test your program by checking that you can approximate the posterior probability of the fair coin obtained in exercise 1, Q.2.