Estimation of Tumor Growth Rate

Zijia Cao, Entong Li, Tom Sweeney

May 6, 2020

Note on work distribution:

Zijia, Entong, and Tom fisrt worked seperately to find a criterion used to compare two sample distributions. Then we worked together to determine the best criterion, which is the Cramer-von Mises Test. We worked on the `search strategies` part together. We wrote the basic description of each part together, and then Tom added more detials and polish to final report.

1. Introduction

Description of the problem

In this project, we want to figure out the growth rate of tumors, including the rate of originating from a single cell, the rate of systemic tumors occurrence etc. To estimate the tumor growth rate, we need to known the time p between detection of the initial tumor and detection of the first recurrent tumor. Since time p is unknown to us at this stage, simulation is a good way to generate time p.

Our first step is to estimate only one growth rate, which is the growth rate of tumors originate from a single cell. Then estimate more than one tumor growth rate at the same time. To estimate the growth rate of tumors originate from a single cell, a data set is given by the cancer client, which has 116 obersevations totally. Each observation represents the time p from detection of an initial tumor to the time of detection of a recurrent tumor for one breast cancer patient. It's already known that the time p is generated by the following process with given parameters $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$.

In our model, tumors grow exponentially at rate θ_1 from one cell. Therefore, the number of cells in the tumor at time t is given by $Y_j(t) = \exp(\theta_1 t)$. New systemic tumors occur in a Poisson process with rate θ_2. In particular, the probability of zero new systemic tumors by time t is $P[N(t) = 0] = \exp(-\theta_2 t)$. The detection of each tumor j follows a nonhomogenous Poisson process with rate $\theta_3 Y_j(t)$, where is $Y_j(t)$ the tumor size at time t as before. In particular, the probability of no detection by time t is $P[N(t) = 0] = \exp(-\int_0^t \theta_3 Y_j(s)\, ds) = \exp\left(-\theta_3 \left(\exp(\theta_1 t) - 1\right)/\theta_1\right)$. Finally, the initial tumor metastasizes following a nonhomogenous Poisson process with rate $\theta_4 Y_0(t)$, where $Y_0(t)$ is the size of the initial tumor at time t. In particular, the probability of no metastasis by time t is $P[N(t) = 0] = \exp(-\int_0^t \theta_4 Y_0(s)\, ds) = \exp\left(-\theta_4 \left(\exp(\theta_1 t) - 1\right)/\theta_1\right)$.

The pseudocode displayed below shows how p can be simulated in practice.

The time p elapsed between detection of the initial tumor and detection of the first recurrent tumor is equal to the difference between the time of the second detection and the time of the first detection (`Detect1`). The time of the second detection is equal to the sum of the time the second tumor appears (`Second`) and the additional time required to detect it (`Detect2`). The time the second tumor appears is either the time that the first tumor metastasizes (`Metastasis`) or the time that a new systemic tumor appears (`NewSystemic`), whichever occurs first (`min(Metastasis,NewSystemic)`). By construction, the initial tumor cannot metastasize after it has been detected because the initial tumor will have been treated. Therefore, `Second` = `Metastasis` only if `Metastasis` $\leq$ `Detect1`.

Let U_1(`Detect1`) be the probability that an initial tumor is not detected in time `Detect1` (one minus the probability there is indeed an initial tumor detected in time `Detect1`). Let U_2(`Metastasis`) be the probability that the inital tumor does not metastasize in time `Metastasis` (one minus the probability the inital tumor does indeed metastasize in time `Metastasis`). Let U_3(`NewSystemic`) be the probability that no new systemic tumors appear in time `NewSystemic` (again, defined in the negative). Let U_4(`Detect2`) be the probability

that a second tumor is not detected in time `Detect2`, starting after the appearance of the second tumor (once again, the negative case). With these constructions, we can invert the relationships between the U_1, U_2, U_3, U_4 and the times to make nice code.

Suppose $\theta_1, \theta_2, \theta_3, \theta_4$ are given. We randomly draw U_1, U_2, U_3, U_4 from a standard uniform distribution. The method is then given by:

Repeat until $p > 0$:
  Generate $U_1, U_2, U_3, U_4 \sim \text{Uniform}(0, 1)$
  `Detect1` $\leftarrow \log(1 - \theta_1/\theta_3) \log U_1)/\theta_1$
  `Metastasis` $\leftarrow \log(1 - \theta_1/\theta_4) \log U_2)/\theta_1$
  `NewSystemic` $\leftarrow \log(-\log U_3)/\theta_2$
  If `Metastasis` $>$ `Detect1` then
    `Second` $\leftarrow$ `NewSystemic`
  Else
    `Second` $\leftarrow$ `min(Metastasis,NewSystemic)`
  `Detect1` $\leftarrow \log(1 - \theta_1/\theta_3) \log U_4)/\theta_1$
  $p \leftarrow$ `Second` $+$ `Detect2` $-$ `Detect1`

Simulation function

We present an R function below that performs the simulation described in the previous subsection.

```
cancer.dat <- scan("cancer data.txt")
simulate.fuc <- function(the, sample.size = 1) {
  time.p = rep(NA, sample.size)
  for (i in 1:sample.size) {
    repeat {
      u <- runif(4,0,1)
      Detect1 <- log(1-(the[1]/the[3])*log(u[1]))/the[1]
      Metastasis <- log(1-(the[1]/the[4])*log(u[2]))/the[1]
      NewSystemic <- (-log(u[3]))/the[2]
      if (Metastasis > Detect1){
        Second <- NewSystemic
      } else {
        Second <- min(Metastasis,NewSystemic)
      }
      Detect2 <- log(1-(the[1]/the[3])*log(u[4]))/the[1]
      p <- Second + Detect2 - Detect1
      if (p>0) {
        time.p[i] = round(p) ; break
      }
    }
  }
  return(time.p)
}
```

However, while other three θ are given, the θ_1 used to generated the given dataset now is missing. The main purpose of this report is to estimate the value of θ_1, with given $\theta_2 = 1/730$, $\theta_3 = \theta_4 = 1/2^{24}$.

Estimating θ_1

If the data were drawn from a known distribution with a closed-form equation for the density, we could derive maximum likelihood or method of moments. Instead we will pursue a strategy by which we vary θ_1 and we

generate simulated data for various values of θ_1. Then we tune θ_1 so simulated data is as close as possible to actual data.

First, we will need to choose an objective function.

2. Choosing an objective function

A good objective function will allow for comparing two distributions and will be equal to zero if the two distributions are equal. The objective function should penalize differences in location, shape, and scale.

Kolmogorov–Smirnov test

Kolmogorov–Smirnov test is one of the nonparametric test used to compare a sample with given probability distribution. The null and alternative hypothesis of the test are:

$$H_0 : F = F_0 \quad vs. \quad H_1 : F \neq F_0,$$

where F is the cumulative distribution function of the given dataset, and F_0 is the given empirical cumulative distribution function.

The test statistic is the supremum distance between F and F_0, which use the formula:

$$D_n := \sqrt{n} \sup_{x \in \mathbb{R}} |F_n(x) - F_0(x)|.$$

If $F = F_0$, then the test statistic should be small, otherwise, it will reject the null hypothesis.

Kullback-Leibler Divergence

A non-parametric test method to directly compare the difference between the original distribution and the generated distribution.

The reason to choose **Kullback-Leibler Divergence** is because, when people use another probability distribution to approximate the original data distribution, the information must be missing, KL divergence is used to measure this lack of information.

The formula of **Kullback-Leibler Divergence** is: $D_{KL}(p||q) = \sum_{i=1}^{N} p(x_i) log \frac{p(x_i)}{q(x_i)}$

p is the original probability dsitribution and q is the simulated probability distribution in this project.

If differeces between two distributions are very small, then the test statistic, or KL divergence, will be very small.

Cramer-von Mises Test

Cramer-von mises test is one of the nonparameteric goodness-of-fit test for distribution models. The null and alternative hypothesis of the test are:

$$H_0 : F = F_0 \quad vs. \quad H_1 : F \neq F_0,$$

where F is the cumulative distribution function of the given dataset, and F_0 is the given empirical cumulative distribution function.

The test statistic is the quadratic distance between F and F_0, which use the formula:

$$W_n^2 = n \int (F_n(x) - F_0(x))^2 dF_0(x).$$

Thus, if the null hypothesis $F = F_0$ is true, then the test statistic W_n^2 should be very small, otherwise it will reject the null hypothesis.

After comparing these three methods, we find that Cramer-von Mises test has higher accuracy in predicting the θ_1 than other two methods. Thus we decide to use Cramer-von Mises test to continue the rest part of this report.

The optimization function

We begin by creating a function that takes a proposed value of θ_1 and returns the Cramer-von Mises test statistic from a comparison between the cancer data and simulated data using the given θ_1.

```
the2 <- 1/730
the3 <- 1/2^24
the4 <- 1/2^24

#Function to get the statistic from Cramer-von Mises test for a specific theta1

get.M2 <- function(the1, sample.size = 5000) {
  the <- c(the1, 1/730, 1/2^24,1/2^24)
  fit <- simulate.fuc(the, sample.size)
  F0 <- ecdf(fit)
  test <- cvm.test(x = cancer.dat, null = F0)
  M2 <- test$statistic
  return(M2)
}
```

3. Analyzing the objective function

Now that we have chosen to use the Cramer-von Mises test statistic as our objective criterion, we want to analyze the objective function. In particular, we want to consider what a reasonable upper bound for θ_1 might be and to explore how we might estimate the function.

Narrowing down the range of θ_1 values

We don't know the range of possible θ_1 values. We are told that θ_1 is the exponential rate at which a tumor grows, starting from a single cell. Therefore, we know that θ_1 will be greater than zero, but we don't know what a reasonable upper bound would be.

Our goal now is to choose an upper bound for θ_1.

Let the initial size for tumor j be equal to one cell. Then, the unit size of tumor j at time t is:

$$Y_j(t) = e^{\theta_1 t}$$

The data provided to us are the observed times from detection of an initial tumor to the time of detection of a recurrent tumor. Therefore, we can get a sense of the observed range of t by looking at the range of the data provided. The maximum value of the observed times is:

```
max(cancer.dat)
```

```
## [1] 2419
```

Therefore, a reasonable value of θ_1 would therefore be able to provide a reasonable value of $e^{\theta_1 \cdot 2419}$ for the number of cells in a tumor. Then we ask: How many cells can a large tumor have?

According to a Physician Data Query (PDQ) (2018) text published by the U.S. government's National Cancer Institute (2018), a tumor the size of a grapefruit would have a diameter of 10cm. A cube with length, width, and height of 10cm would have volume 1000cm^3. According to Del Monte (2009), a tumor reaching the size of 1cm^3 (approximately 1g wet weight) is "commonly assumed" to contain 1×10^9 cells (but most tumors of this volume actually contain far fewer cells).

Putting these facts together, a good upper estimate of the number of cells in a large tumor is $1000 \times 10^9 = 1 \times 10^{12}$, or one trillion cells. Indeed, the entire human body consists of about 40 trillion cells, according to Bianconi et al. (2013). So we can have confidence in this upper limit.

A value of θ_1 greater than or equal to 0.2 would therefore be very unreasonable:

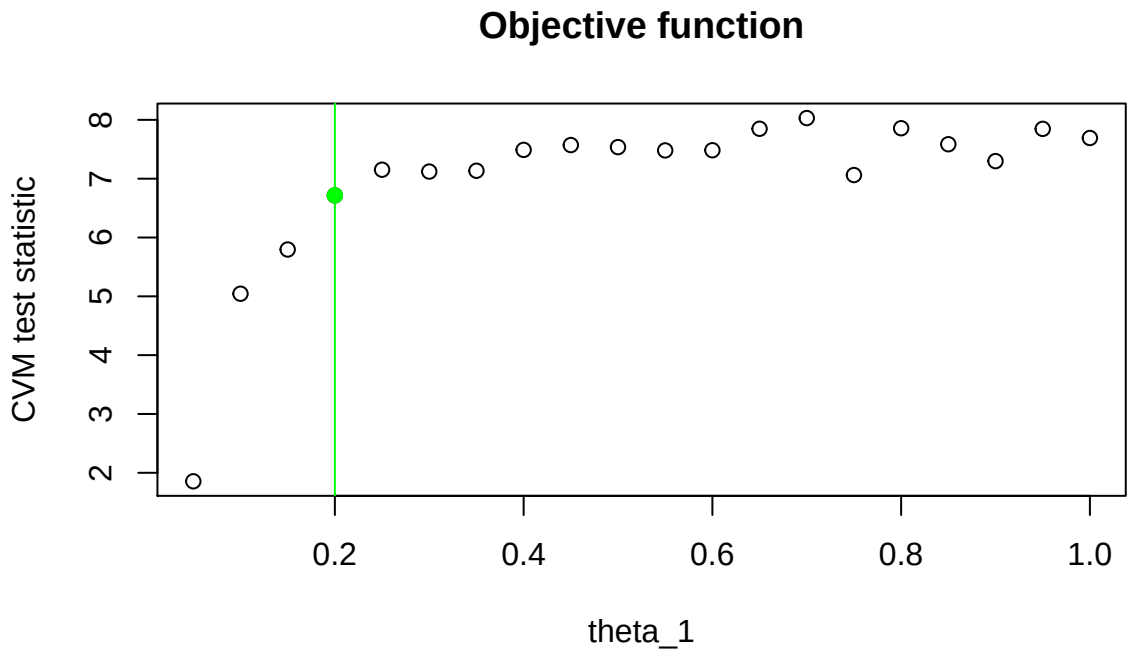
```
proposed.theta1 <- c(0.2, 0.5, 1, 2)
exp(proposed.theta1*2419)
```

```
## [1] 1.293214e+210           Inf           Inf           Inf
```

Therefore, we restrict our attention to values of θ_1 between 0 and 0.2.

To confirm that this is reasonable for our data, we check Cramer-von Mises test statistics using our function for 20 values of θ_1 between 0 and 1. We print a plot of the CVM statistics associated with each value of θ_1. The CVM statistic associated with 0.2 is shown with a green dot:

```
theta1 <- seq(0.05,1, by = 0.05)

cvm = rep(NA, length(theta1))
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1)
}
```



The plot shows that a value of $\theta_1 = 0.2$ clearly produces a distribution that differs from the distribution of the provided data much more than values less than 0.2. At values greater than 0.2, the CVM statistic plateaus. Therefore, we should be satisifed with an upper bound of 0.2. For the remainder of this investigation, we will restrict our search to values between zero and 0.2.

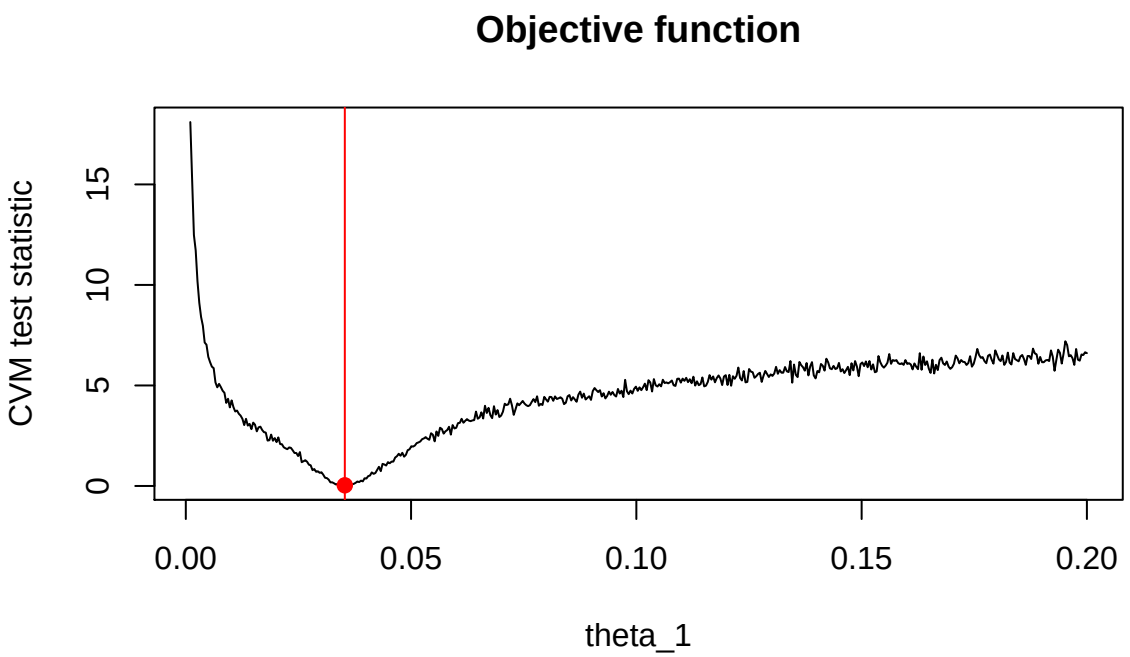
```
# setting upper bound of theta1
upper <- 0.2
```

Estimating the objective function

Higher values of the sample size parameter (`sample.size`) in `simulate.fuc()` will reduce variation caused by the randomness in the simulation. Ideally, the empirical distribution of the simulated data would be exactly the same with each iteration of `get.M2()` as long as the value of `the1` does not change. However, the simulation randomness makes iterative usage of `get.M2()` noisy. We set `sample.size` equal to 5000, but slower machines may face long computation times with high values. This is illustrated in the appendix.

Now that we've developed a method for simulating distributions for each proposed value of θ_1, let's look at the CVM statistics. We plot the objective function for 500 values of θ_1. The CVM statistic associated with the optimal value of θ_1 is shown with a red dot.

```
leng <- 500
theta1 <- seq(0.001, upper, length.out = leng)
cvm = rep(NA, length(theta1))
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1)
}
mini <- theta1[which.min(cvm)]
```



We see that the optimal value sits in a relatively deep basin, and there are no other basins. There are no other extrema or anything close to another extremum. We should have high confidence in our estimate.

The objective function looks nice and seems well-behaved, with one exception: It is noisy at larger values of θ_1. This noise is caused by the randomness in the simulation run for each iteration.

In the vicinity of the minimizer, the noise is much smaller. Some iterative optimization techniques assume that noise decreases as the search value approaches the optimizer. This is a controversial assumption, but it is actually the case here.

3. Choosing a search strategy

First, we consider what values of θ_1 are plausible. Then, we try a naive strategy of exhaustive search. Using the values of the Cramer-von Mises test statistic from the exhaustive search, we plot the objective function. Finally, we consider more efficient search methods.

In all, we consider the following search strategies:

(1) Exhaustive search,
(2) R function `optimize()`,
(3) Fibonacci search, and
(4) Golden-section search.

In the following section, we consider a different method that is easily generalizable for multiple dimension: stochastic approximation.

```
# creating empty vectors for method results
time.taken <- rep(NA, 5)
time.taken <- rep(NA, 5)
```

Naive strategy: exhaustive search

First we tried an exhaustive search method. This was the least efficient of the four approaches. This method finds an optimal value by testing a large number of values between 0 and 0.2 using a for-loop. This method

requires simulating a new distribution for each iteration of the for loop. The number of iterations is equal to the number of tested values of θ_1.

The estimated optimal value is printed below, along with a plot displaying the distribution of the given data and the distribution of the optimal simulated data. We also print the time taken to run the optimization code.

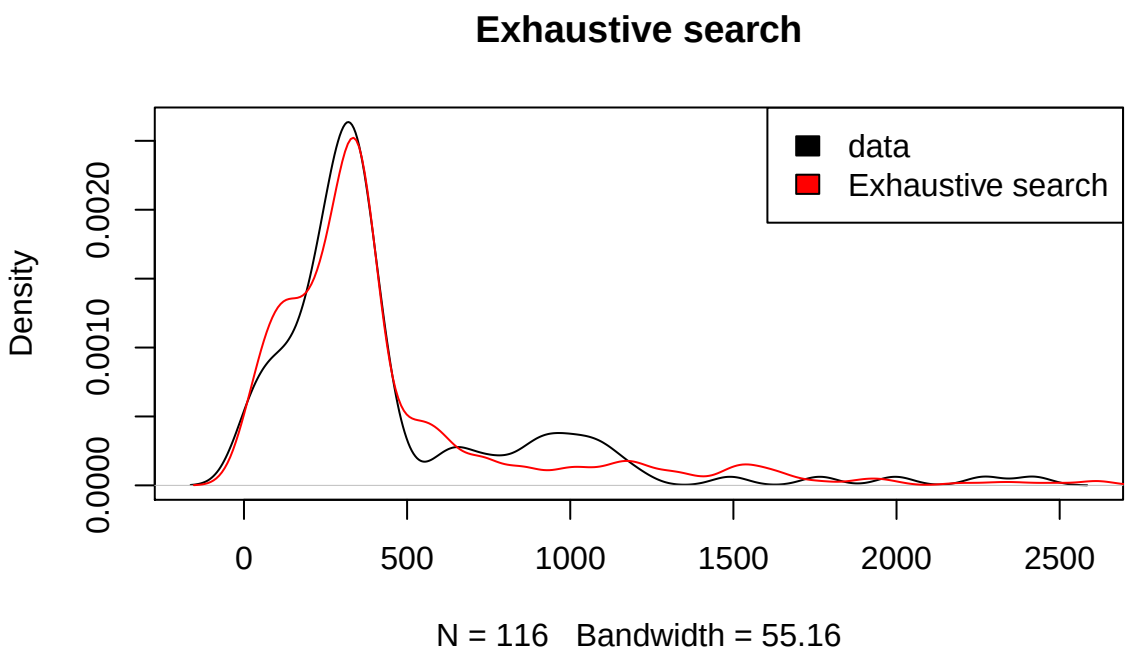
```
mini <- rep(NA, 5)

# method implementation
start.time <- Sys.time()
cvm = rep(NA, length(theta1))
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1)
}
mini[1] <- theta1[which.min(cvm)]
time.taken[1] <- Sys.time() - start.time
mini[1]
```

```
## [1] 0.034499
```

```
# computation time
time.taken[1]
```

```
## [1] 8.374716
```

```
# comparing distributions
simu1 <- simulate.fuc(c(mini[1],the2, the3, the4),leng)
```



This method required a computation time of: 8.3747158 seconds.

Recall that the optimizer sits in a basin and that there are no other basins.

The computation suggests we should consider more intelligent search strategies. A less expensive method would be particularly helpful if we want to optimize over multiple parameters. Adding additional parameters θ_j would increase the complexity and computation time exponentially.

However, more efficient search strategies might be more sensitive to the noise caused by the simulation randomness. These trade-offs will be considered below.

4. Numerical search strategies in one dimension

The following methods are examined: 1-dimensional `optimize()`, Fibonacci search, and Golden-section search.

The results provide the same answers as the exhaustive search. However, these search methods are much faster. All three methods perform relatively equally.

1-dimensional `optimize()`

The defualt function in `R` used to find the minimum or maximum of the given function in the ginve interval, but the accuracy of this function is not high. Although this method has more noise than the rest of two methods, it could provide a good first guess of θ_1, or use its output to shrink the inverval used to search for the optimize value. For example, here the `optimize` function will return a value close to the estimated value with the naive search method.

The estimated optimal value is printed below, along with a plot displaying the distribution of the given data and the distribution of the optimal simulated data. We also print the time taken to run the optimization code.

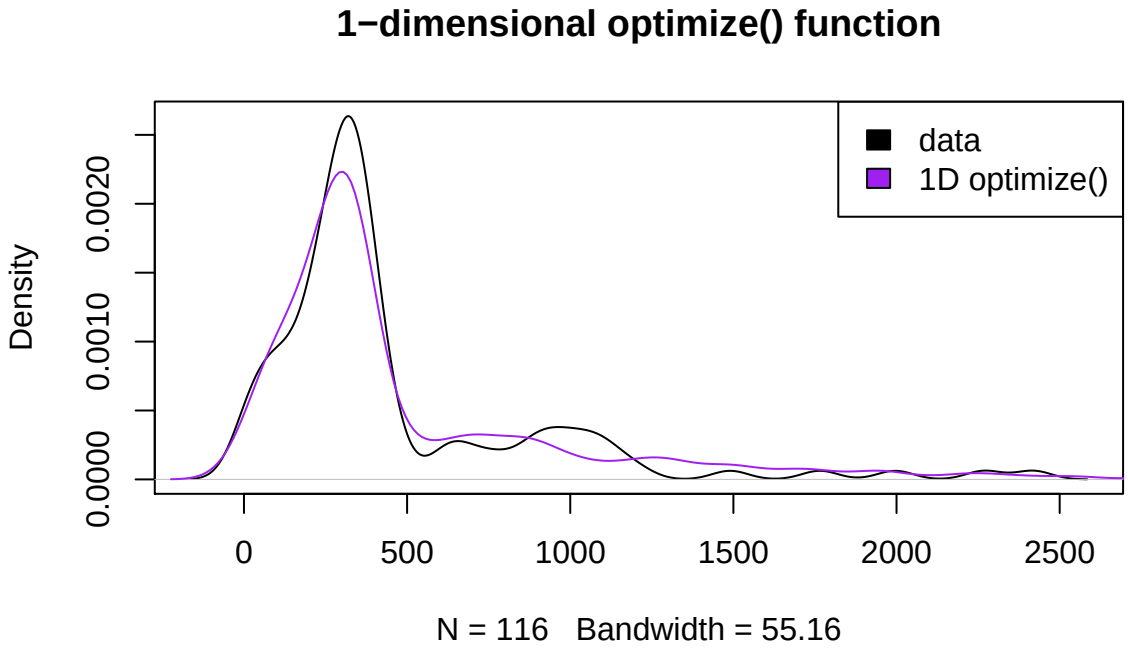
```
# method implementation
start.time <- Sys.time()
opt.1d <- optimize(get.M2, c(0, upper), tol = 10^(-5))
mini[2] <- opt.1d$minimum
time.taken[2] <- Sys.time() - start.time
mini[2]
```

```
## [1] 0.03584035
```

```
# computation time
time.taken[2]
```

```
## [1] 0.3480229
```

```
# comparing distributions
simu2 <- simulate.fuc(c(mini[2],the2, the3, the4),leng)
```



This method required a computation time of: 0.3480229 seconds.

Fibonacci search

Fibonacci search is an optimal search method with the least number of function estimates. The reason is that Fibonacci search divides the array into unequal parts and operations involved in this search are addition and subtraction only, which means it reduces the work load of the computing machine.

The formula for Fibonacci search:

$$F[k] = F[k-1] + F[k-2]$$

$$F[k] - 1 = (F[k-1] - 1) + 1 + (F[k-2] - 1)$$

1. key = mid, Success 2. key < mid, the new search interval [low, F[k-1]-1] 3. key > mid, the new search interval [F[k-2]-1, high]

The difference between bisection and Fibonacci is that bisection take the midpoint of two numbers, the lower part and the upper part are even while Fibonacci has uneven lower part and upper part, based on the Fibonacci sequence. This saves computation time by avoiding calculation of midpoints.

We use the function `fibsearch` in library `pramca`.

The estimated optimal value is printed below, along with a plot displaying the distribution of the given data and the distribution of the optimal simulated data. We also print the time taken to run the optimization code.

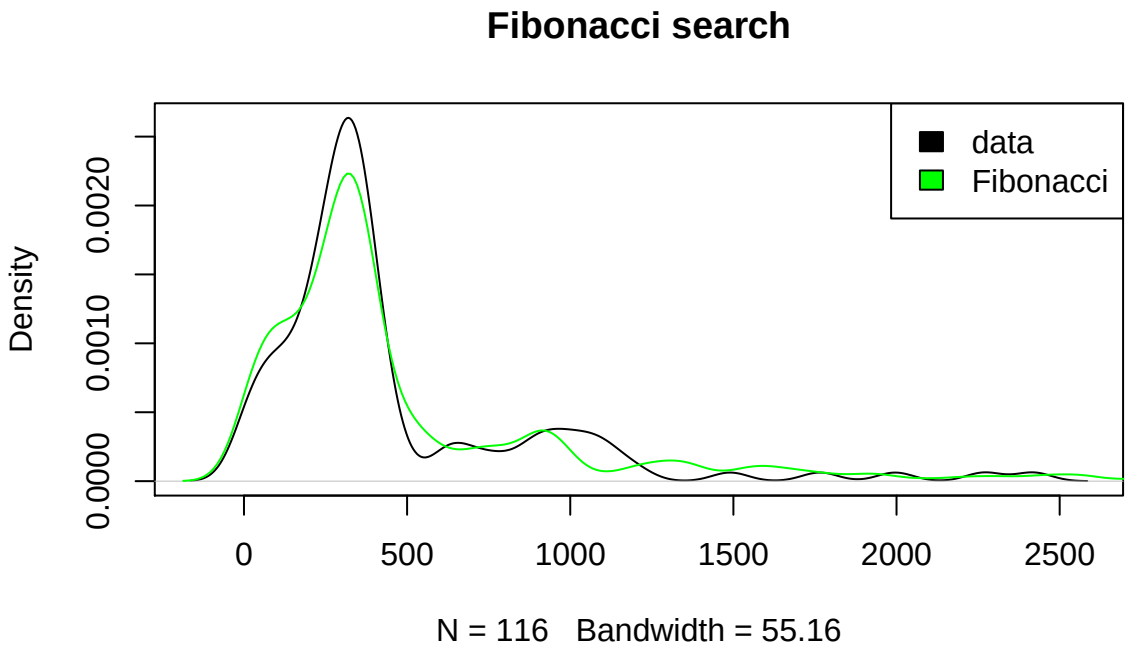
```
# method implementation
start.time <- Sys.time()
f.search <- fibsearch(get.M2, 0, upper, endp = FALSE, tol=10^(-5))
mini[3] <- f.search$xmin
time.taken[3] <- Sys.time() - start.time
mini[3]
```

```
## [1] 0.0354447
```

```
# computation time
time.taken[3]
```

```
## [1] 0.398782
```

```
# comparing distributions
simu3 <- simulate.fuc(c(mini[3], the2, the3, the4), leng)
```



This method required a computation time of: 0.398782 seconds.

Golden-section search

Golden-section search is a search method to find the maximum or minimum of a unimodal function by narrowing down the range of values, the range is determined by Golden ratio. The difference between bisection and golden-section is bisection takes the midpoint of two numbers on each iteration, and the golden-section take the golden section of two numbers. This saves computation time by avoiding calculation of midpoints.

The golden-section method is related to Fibonacci search because the golden ratio is the limit of ratio of successive terms in the Fibonacci sequence.

We use the function `golden_ratio` in library `pramca`.

The estimated optimal value is printed below, along with a plot displaying the distribution of the given data and the distribution of the optimal simulated data. We also print the time taken to run the optimization code.

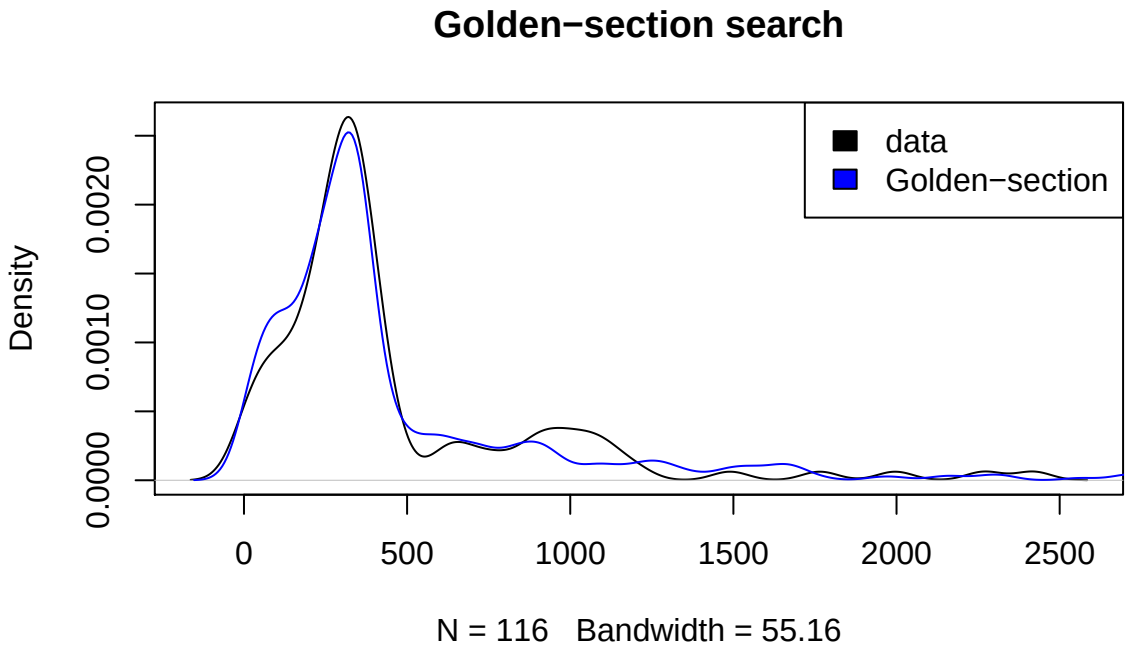
```
# method implementation
start.time <- Sys.time()
g.search <- golden_ratio(get.M2, 0, upper, maxiter = 500, tol =10^(-5))
mini[4] <- g.search$xmin
time.taken[4] <- Sys.time() - start.time
mini[4]
```

```
## [1] 0.03568819
```

```
# computation time
time.taken[4]
```

```
## [1] 0.3741081
```

```r
# comparing distributions
simu4 <- simulate.fuc(c(mini[4], the2, the3, the4), leng)
```



This method required a computation time of: 0.3741081 seconds.

5. Stochastic approximation

Theory and function

Stochastic approximation is group of methods that allows otpimization of an underlying function when noisy data are present. This is relevant for our problem because our simulated data are very noisy. In particular, the objective function for θ_1 is quite noisy for values of θ_1 close to zero and greater than 0.05.

The algorithm for finding a function's roots with stochastic approximation was proposed by Robbins and Monro (1951). The algorithm was modified for local optimization by Kiefer and Wolfowitz (1952). Kiefer and Wolfowitz approximate the function's gradient with central differences. We follow their proposed procedure.

The function that we want to optimize is $f(\theta_1)$, the objective function of θ_1 using the Cramer-von Mises test statistic. This function is represented by `get.M2()` in our code.

The optimization algorithm is as follows:

(1) Choose an initial guess for the optimal θ_1. Call this $\theta_{1,1}$.
(2) Then:

$$\theta_{1,n+1} = \theta_{1,n} - a_n \frac{f(\theta_{1,n} + c_n) - f(\theta_{1,n} - c_n)}{2c_n}$$

where $n = 1, 2, \ldots$, $c_n \to 0$, $\sum a_n = \infty$, $\sum a_n c_n < \infty$, and $\sum (a_n/c_n)^2 < \infty$. For example, $a_n = bn^{-1}$ and $c_n = bn^{-1/3}$ satisfy these requirements, where b is some constant.

As shown in Kiefer and Wolfowitz (1952), $\theta_{1,n}$ will converge in probability to the local optimizer.

This approximates the derivative with a steepest descent method. The method is easily generalizable for multiple dimensions. The Kiefer and Wolfowitz (1952) algorithm was extended for multiple dimensions by J.R.

Blum (1954). Because our problem is optimization in a single variable, we implement the one-dimensional method.

Part of the problem is choosing a_n and c_n. We choose c_n small enough that $\theta_{1,n} \pm c_n$ is likely to remain between the lower and upper bounds of the search range, even for small n (including $n = 1$). We achieve this by using $a_n = bn^{-1}$ and $c_n = bn^{-1/3}$ with b equal to the difference between the upper bound and lower bound divided by the number of iterations. In our example, this means:

$$b = \frac{\text{upper} - \text{lower}}{\text{iterations}} = \frac{0.2 - 0}{500} = 0.0004$$

This choice is designed to avoid cases in which c_n for some n is such that $\theta_{1,n} \pm c_n$ is greater than the upper search bound or less than the lower search bound. When n is small and b is large, $bn^{-1/3}$ will be large. In particular, the problem is serious for the first iteration, which involves computing $\theta_{1,2} = \theta_{1,1} - a_1 \left(f(\theta_{1,1} + c_1) - f(\theta_{1,1} - c_1)\right) / (2c_1)$. In this iteration, $bn^{-1/3} = b$. Obviously, if b is larger than the difference between the search bounds, then $\theta_{1,1} \pm c_1 = \theta_{1,1} \pm b$ will necessarily be outside of the search bounds. Then, $f(\theta_{1,1} \pm c_1)$ may be nonexistent or problematic. In our example, if $\theta_{1,1} - c_1 < 0$, then $f(\theta_{1,1} - c_1)$ will not exist. And if $\theta_{1,1} + c_1 > 0.2$, then $f(\theta_{1,1} + c_1)$ is much likelier to lead to an incorrect solution due to noise in f at high values of θ_1. However, it is not possible to guaruntee that $\theta_{1,1} \pm c_1$ will fall within the search bounds. This is because $\theta_{1,1}$ is a random draw from the uniform distribution, $\theta_{1,1} \sim \text{Uniform}(\text{lower}, \text{upper})$. By the properties of the uniform distribution, there is a nonzero probability that $|\theta_{1,1} - \text{lower}| < c_n$ or $|\text{upper} - \theta_{1,1}| < c_n$ for any value of $c_n > 0$. A good upper limit for c_n would be a value such that not both $|\theta_{1,1} - \text{lower}| < c_n$ and $|\text{upper} - \theta_{1,1}| < c_n$. This occurs when $c_n < (\text{upper} - \text{lower})/2$. Therefore, if $c = bn^{-1/3}$ as in our example, values of b much less than $(\text{upper} - \text{lower})/2$ are desirable.

Note that our algorithm will, by construction, force $\theta_{1,n+1}$ to be within the search bounds if the value of $\theta_{1,n} - a_n \left(f(\theta_{1,n} + c_n) - f(\theta_{1,n} - c_n)\right) / (2c_1)$ is outside the bounds. It does this by drawing again from a uniform distribution between the bounds. That is, when $\theta_{1,n} - a_n \left(f(\theta_{1,n} + c_n) - f(\theta_{1,n} - c_n)\right) / (2c_1)$ is less than lower or greater than upper, the algorithm will sample $\theta_{1,n+1} \sim \text{Uniform}(\text{lower}, \text{upper})$. However, each random redrawing of $\theta_{1,n+1}$ restarts the stochastic approximation method, meaning that the search algorithm makes no progress. Each random redrawing of $\theta_{1,n+1}$ in this sense is waste of an iteration and adds to the method's computation cost. Therefore, smaller values of c_n are desirable.

However, there is also a problem when c_n are too small. As c_n get smaller, the method becomes increasingly sensitive to noise in f. This is because $f(\theta_{1,1} + c_1) - f(\theta_{1,1} - c_1)$ depends on f, which is a noisy function. Therefore, when c_n are very small, the estimated direction of the gradient of f, which is given by the sign of $f(\theta_{1,1} + c_1) - f(\theta_{1,1} - c_1)$, will become unreliable when using a small number of iterations. When c_n are sufficiently large, the sign of $f(\theta_{1,1} + c_1) - f(\theta_{1,1} - c_1)$ is likely to indicate the true direction of the gradient of f even with few iterations. On the other hand, when f is not noisy, $f(\theta_{1,1} + c_1) - f(\theta_{1,1} - c_1)$ will become very small as c_n approach zero, as long as f is smooth. In this low-noise case, $|\theta_{1,n+1} - \theta_{1,n}| = |a_n \left(f(\theta_{1,n} + c_n) - f(\theta_{1,n} - c_n)\right) / (2c_1)|$ will also become very small (as long as a_n/c_n also decreases to zero, but this is true by the initial restrictions). As a consequence, the proposed value of θ_1 will move by only small increments on each iteration. This increases the number of iterations required to adequately search the whole range. Therefore, c_n should not be too small.

A final constraint is that the value of a_n should not be too large or too small. When the a_n are large (holding c_n constant), a_n/c_n will be large and will amplify the noise in $f(\theta_{1,1} + c_1) - f(\theta_{1,1} - c_1)$. This increases the likelihood of leaving the search bounds, which is problematic for reasons discussed above. When the a_n are small, a_n/c_n will be small and will cause $|\theta_{1,n+1} - \theta_{1,n}| = |a_n \left(f(\theta_{1,n} + c_n) - f(\theta_{1,n} - c_n)\right) / (2c_1)|$ to be small. In this case, the proposed value of θ_1 will move by only small increments on each iteration. As discussed above, this increases the number of iterations required to adequately search the whole range. Additionally, the $|\theta_{1,n+1} - \theta_{1,n}|$ increments will converge in probability to zero as n goes to infinity. This is especially problematic if a_n and c_n are such that $|\theta_{1,n+1} - \theta_{1,n}|$ becomes smaller than the machine's smallest number. In this case, the algorithm might fail to complete its search. A good proposal for the a_n is setting a_n such that $a_n/c_n = n^{-1}/n^{-1/3} = n^{-2/3}$. This is the case in the recommended choices from Kiefer and Wolfowitz (1952). Note that at $n = 1000$, this gives $a_n/c_n = 0.01$, which is an appropriate scaling factor for

our example and is much larger than our machine's smallest number. Therefore, we choose $a_n = bn^{-1}$ with $c_n = bn^{-1/3}$ so that the factor of b cancels out.

It yet remains to choose a value of b such that $b < (\text{upper} - \text{lower})/2$ but is not too small. We prefer $b = (\text{upper} - \text{lower}) \,/\, (\text{iterations})$ because it is helpful to make the c_n inversely proportional to the number of iterations. When the number of iterations is small (close to 2), a large value of c_n is good so that $|\theta_{1,n+1} - \theta_{1,n}| = |a_n \left(f(\theta_{1,n} + c_n) - f(\theta_{1,n} - c_n)\right) / \left(2c_1\right)|$ will be large enough to adequately search the whole range with only the small number of iterations. When the number of iterations is large, using small c_n becomes less problematic for two reasons. First, the noise in f is compensated with more iterations, so that $\left(f(\theta_{1,1} + c_1) - f(\theta_{1,1} - c_1)\right)/(2c_n)$ will converge in probability to the correct gradient. Second, a large number of iterations compensates for small values of $|\theta_{1,n+1} - \theta_{1,n}| = |a_n \left(f(\theta_{1,n} + c_n) - f(\theta_{1,n} - c_n)\right) / \left(2c_1\right)|$, so that the algorithm can adequately search the whole range. Because the problems of choosing small c_n are lessened, the benefits of choosing small c_n take greater priority when the number of iterations is large.

In practice, the choices of $a_n = bn^{-1}$, $c_n = bn^{-1/3}$, $b = (\text{upper} - \text{lower}) \,/\, (\text{iterations})$, and 500 iterations have been very successful for our examples.

We create a function based on the algorithm proposed by Kiefer and Wolfowitz (1952). The function finds a local optimizer for given function, lower and upper bounds, and tolerance level.

```
kw.app <- function(fun, lower, upper, iter.max = 10000, tol) {
  if (upper - lower <= 0) {
    stop("Upper must be greater than lower")
  }
  as <- ((upper-lower)/iter.max)/1:iter.max
  cs <- ((upper-lower)/iter.max)*(1/1:iter.max)^(1/3)
  init <- runif(1, lower, upper)
  theta <- c(init,rep(NA,iter.max))
  for (i in 1:iter.max) {
    theta[i+1] <- theta[i] -
      as[i] * ( fun(theta[i] + cs[i]) - fun(theta[i] - cs[i]) )/ (2*cs[i])
    if(theta[i+1] < lower | theta[i+1] > upper) {
      theta[i+1] <- runif(1, lower, upper)
    }
    if (abs(theta[i+1] - theta[i]) < tol) {
      break
    }
  }
  return(list(theta.min = theta[i+1], f.min = fun(theta[i+1]), iters = i))
}
```

The preceding discussion should make it evident that the difficulty of choosing suitable a_n and c_n is a major disadvantage of the stochastic approximation method. Even if our stochastic approximation algorithm finds the optimal θ_1 efficiently, the time and labor required to find suitable a_n and c_n should count as a major strike against this method.

Implementation

The estimated optimal value is printed below, along with a plot displaying the distribution of the given data and the distribution of the optimal simulated data. We also print the time taken to run the optimization code.

```
start.time <- Sys.time()
kw <- kw.app(get.M2, 0, upper, iter.max = 500, tol = 10^(-5))
mini[5] <- kw$theta.min
```

```
time.taken[5] <- Sys.time() - start.time
mini[5]
```

```
## [1] 0.03580902
```

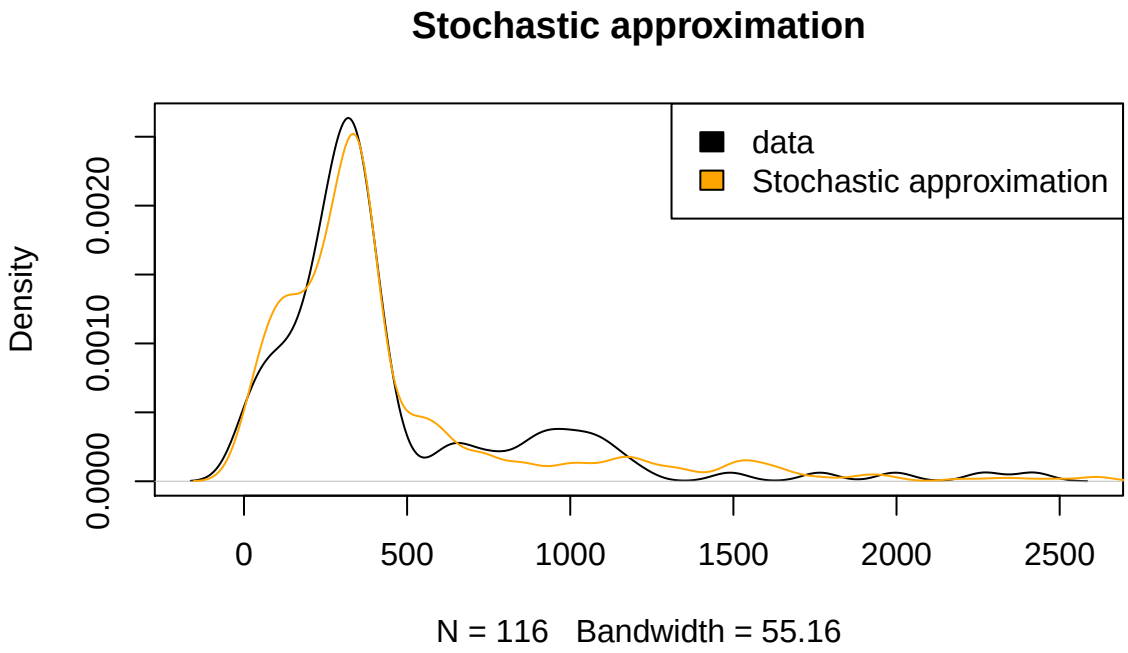
```
# computation time
time.taken[5]
```

```
## [1] 1.619234
```

```
kw$iters
```

```
## [1] 47
```

```
# comparing distributions
simu5 <- simulate.fuc(c(mini[5],the2, the3, the4),leng)
```



This method required a computation time of: 1.6192341 seconds. It required 47 iterations.

We see that the approximated optimizer is very close to the optimizer found with other methods. However, this method is much slower than the three efficient search methods proposed previously.

Note that the time displayed for this method does not include the time required to tune the values of a_n and c_n.

6. Conclusion

We estimate that the value of θ_1 for the given data is around 0.035. For efficient estimation, we recommend using our function `get.M2()`, generated from Cramer-von Mises Test, along with any of the following methods: R function `optimize()`, Fibonacci search, Golden-section search, or Stochastic approximation. All four methods were significantly faster than a naive search method, but the naive search method could always provide the accurate prediction but these four methods will have noise. R function `optimize()` is easiest to access among these three methods, but it will have more noise than other two methods. Fibonacci search and Golden-section search are efficient, both of them work fast, and by increasing the simulation sample size or the tolerance, their noise could be reduced. The Stochastic approximation likes the compromise of the naive

search method and the other three methods, it could provide a more accurate prediction than those three mothod, and takes less time than the naive rearch method.

For how to deal with the difficulty that the efficient results are still sensitive to noise. We discuss this is in the appendix below.

Table 1: Optimization method comparison

	minimizer estimate	computation time (s)
Exhaustive search	0.0344990	8.3747158
1-dim. optimize()	0.0358403	0.3480229
Fibonacci search	0.0354447	0.3987820
Golden-section search	0.0356882	0.3741081
Stochastic approximation	0.0358090	1.6192341

Currently, we are mainly focused on the one-dimensional methods. All three methods mentioned above provides us a clear idea about how the variable θ_1 is obtained. Among all the methods, exhaustive search has the longest search time since it is the most direct method whether dealing with one-dimensional or multi-dimensional problems. It calculates almost everything to find the appropriate variable. Compared to exhaustive search, 1-dim. optimize, Fibonacci search, Golden-section search reduce the burden of computing. Instead of trying each possible answers, these three methods use different strategies to narrow down the range of the final result step by step. One limitation is that the three methods we recommend above are one-dimensional methods. If we wanted to perform multidimensional optimization, we could not use these three search methods. We also would not want to use the exhaustive search due to the limitation of computational capabilities.

The stochastic approximation is generalizable and is good to deal with multi-dimensional problems. Compared to methods mentioned in the above paragraph, stochastic approximation involves tuning the sequences, which reflects on the choice of a_n and c_n.

To delve deeper, what comes next is to estimate both θ_1 and θ_2. Exhaustive search might work, but our computer does not allow us to do so. Our first choice right now is the 2-dimensional stochastic approximation.

It is very likely that tuning the initial parameters for multidemnsional stochastic approximation will be more laborious and time-consuming than for the one-dimensional case. Although stochastic approximation is our preferred multidemnsional method of the options discussed here, we should hope to find another efficient method that is easier to tune. This will require additional research.

Appendix: Noise in the objective function

Reducing the sample size for the simulated distributions, which are created for each iteration of the optimization algorithms, will increase the noise the objective function for θ_1. This is a problem because non-exhaustive search methods may select incorrect values for the optimizer. Efficient search algorithms are likelier find wrong local minimizers in the noise when the sample size is small.

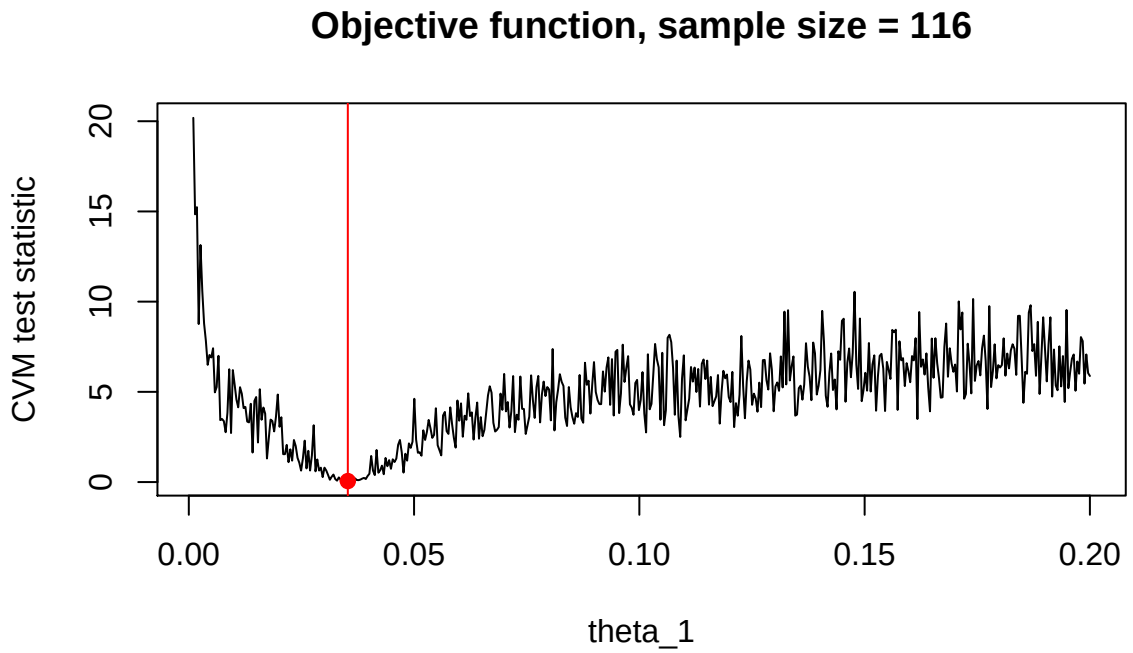
We illustrate this problem by estimating the objective function for θ_1 using the exhaustive search method. We print three estimated objective funcctions using different sample sizes: 116 (the length of the given data), 500, and 1000. The methods shown above use a sample size of 5000.

```
# settining initial parameters
leng <- 500
theta1 <- seq(0.001, upper, length.out = leng)
cvm <- rep(NA, length(theta1))
time.taken <- rep(NA, 5)
```

The noise is most obvious in simulations with a sample size of 116 and values of θ_1 greater than 0.05.

```
sample.size = 116

start.time <- Sys.time()
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1, sample.size)
}
time.taken[1] <- Sys.time() - start.time

mini <- theta1[which.min(cvm)]

# computation time
time.taken[1]
```

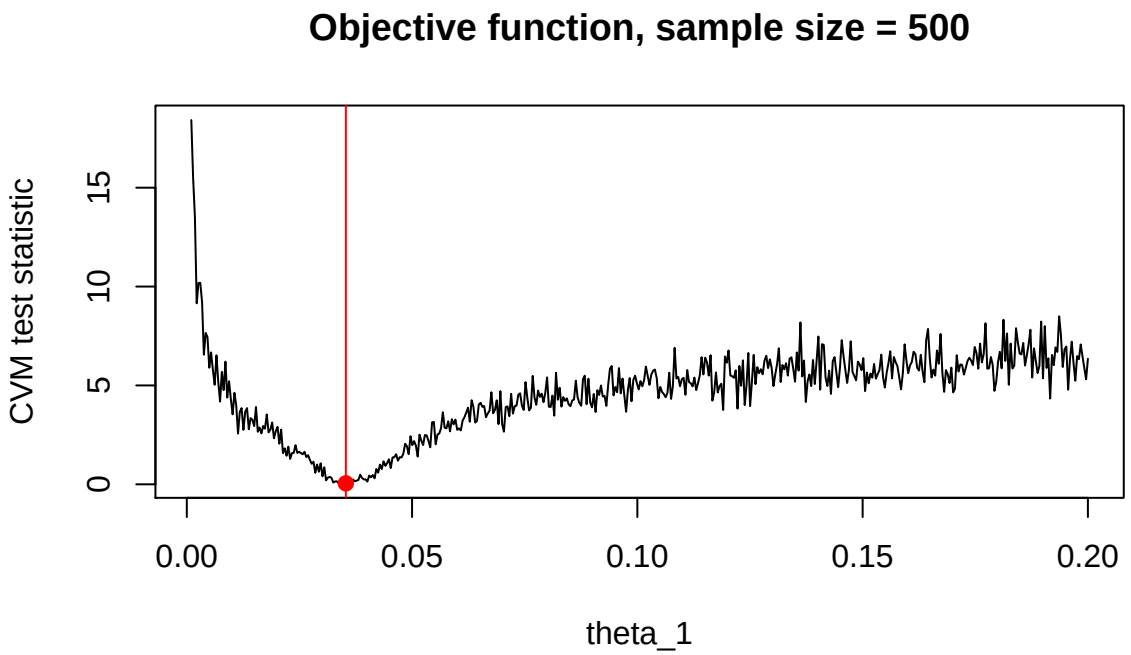
```
## [1] 0.806987
```



There is significant improvement with a sample size of 500.

```
sample.size = 500

start.time <- Sys.time()
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1, sample.size)
}
time.taken[2] <- Sys.time() - start.time

mini <- theta1[which.min(cvm)]

# computation time
time.taken[2]
```

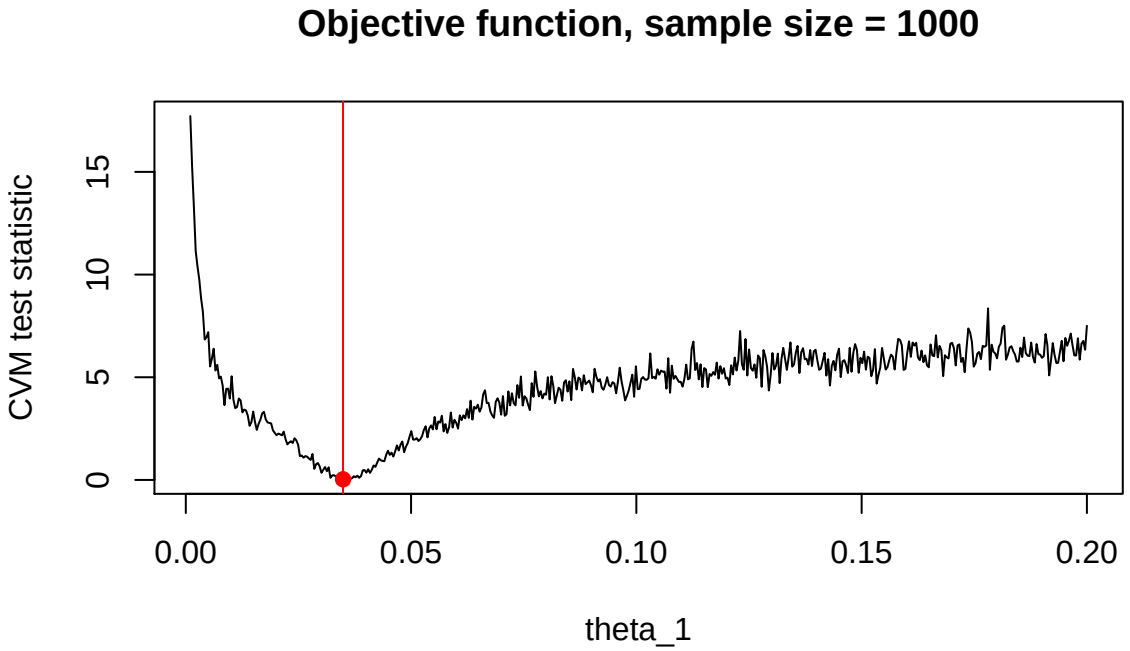
```
## [1] 1.312432
```



For one-dimensional optimization, a sample size of 1000 probably suffices.

```
sample.size = 1000

start.time <- Sys.time()
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1, sample.size)
}
time.taken[3] <- Sys.time() - start.time

mini <- theta1[which.min(cvm)]

# computation time
time.taken[3]
```

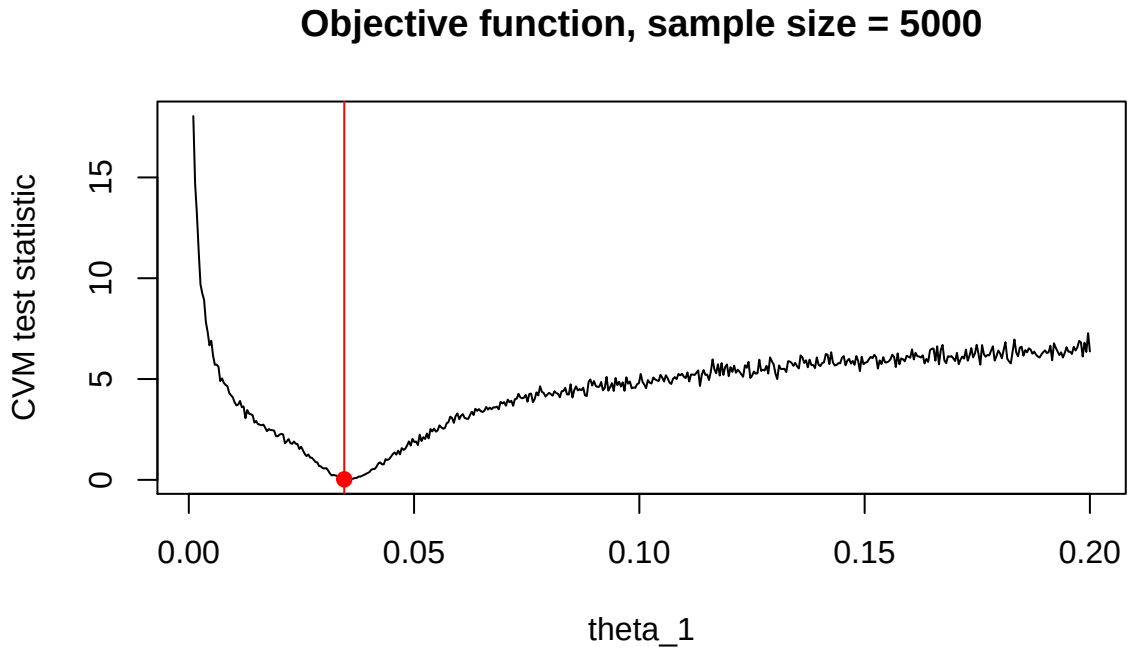
```
## [1] 2.124975
```



The noise in the objective function will decrease as the sample size approaches infinity.

```
sample.size = 5000

start.time <- Sys.time()
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1, sample.size)
}
time.taken[4] <- Sys.time() - start.time

mini <- theta1[which.min(cvm)]

# computation time
time.taken[4]
```

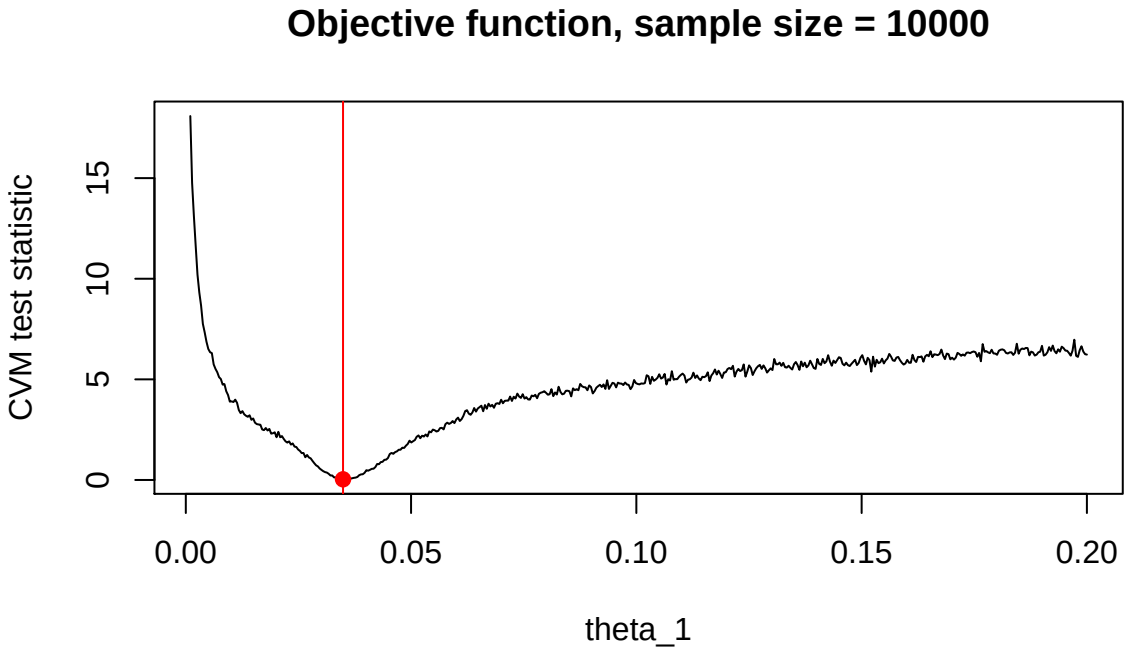
```
## [1] 8.185906
```



There is a trade-off with computation time.

```
sample.size = 10000

time.taken[5] <- Sys.time()
for (i in 1:length(theta1)) {
  the1 <- theta1[i]
  cvm[i] <- get.M2(the1, sample.size)
}
time.taken[5] <- Sys.time() - start.time

mini <- theta1[which.min(cvm)]

# computation time
time.taken[5]
```

```
## [1] 23.98286
```



We see that the estimation using a sample size of 10,000 is not much better than the estimation using 5,000. However, it took significantly longer to create.

Note that for all of the sample sizes, the noise in the objective function is small around the vicinity of the minimizer. Some iterative optimization methods assume that noise decreases as the search value approaches the optimizer. This is a controversial assumption, but it is actually the case here. Therefore, these methods might work for this problem with even small sample sizes such as 100. However, larger sample sizes will still provide more confidence in the result.

The sample sizes and computation times are displayed below.

Table 2: Sample size trade-off

sample size	computation time (s)
116	0.806987
500	1.312432
1000	2.124975
5000	8.185906
10000	23.982861

Based on this trade-off, we settled for a sample size of 5000.

References

Bianconi E., Piovesan A., Facchin F., Beraudi A., Casadei R., Frabetti F., Vitale L., Pelleri M.C., Tassani S., Piva F., Perez-Amodio S., Strippoli P., Canaider S. (2013). An estimation of the number of cells in the human body. Annuals of Human Biology, 40(6): 463-471. doi: 10.3109/03014460.2013.807878

Blum, J. R. (1954). Multidimensional stochastic approximation methods. Annuals of Mathematical Statistics, 25(4): 737-744. doi: 10.1214/aoms/1177728659

Del Monte, U. (2009). Does the cell number 10(9) still really fit one gram of tumor tissue? Cell Cycle, 8(3): 505–506. doi: 10.4161/cc.8.3.7608

Kiefer, J. and Wolfowitz, J. (1952). Stochastic estimation of the maximum of a regression function. Annuals of Mathematical Statistics, 23(3): 462-466. doi: 10.1214/aoms/1177729392

PDQ Adult Treatment Editorial Board. (2018). Ovarian germ cell tumors treatment (PDQ): Patient version. PDQ Cancer Information Summaries, U.S. National Cancer Institute, 2002-. https://www.ncbi.nlm.nih.gov/books/NBK65925/

Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. Annuals of Mathematical Statistics, 22(3), 400-407. doi: 10.1214/aoms/1177729586