498818_SDS475_Summary_1

2024-09-05

Statistical Computation Summary 1

Introduction to Statistical Methods

Monte Carlo

In the first lesson of the course, we briefly covered an overview of several widely used, and very popular statistical methods that allow us to tackle complex problems, simulate data, and make inferences when traditional methods fall short.
The first statistical method we touched on was Monte Carlo, a powerful computational tool used to estimate or approximate integrals using random sampling. **Monte Carlo is particularly important because many real-world problems involve complex models for which integrals or expectations cannot be directly computed using standard calculus or closed-form formulas.** Unlike traditional methods, Monte Carlo does not rely on a closed-form formula to compute results. Instead, it simulates many random scenarios to get a good approximation of the desired quantity, typically by random sampling from a specific probability distribution.
In simpler terms, Monte Carlo involves simulating many possible outcomes or scenarios to approximate the desired result.
One of the key takeaways from the lecture was to understand why we want to estimate/approximate integrals using Monte Carlo. **The reason estimating/approximating integrals can be so useful is because foundational concepts in probability theory tell us that we can approximate Expected Value from the approximation of complex integrals.**
Expected Value (EV) of a random variable X, is equal to the integral of

$$\int xf(x)dx$$

where $f(x)$ is the pdf of X. Approximating integrals using Monte Carlo allows us to estimate expected values for complex models, which leads to additional insights in their behavior.
Monte Carlo Process can be broken down into 3 key steps.
1. **Choose a Probability Distribution,** $p(x)$: This is the distribution from which we will sample
2. **Generate Samples** $X_1, X_2, ..., X_n$: These are random variables drawn independently from the distribution $p(x)$
3. **Approximate the Integral**: For integral approximation of $f(x)dx$, need to use a mathematical technique. Multiply and divide by $p(x)$, which allows us to use random samples from the distribution $p(x)$. Using probability theory, one can conclude

$$E(f(x)/p(x)) = \frac{1}{n}\sum_{i=1}^{n}\frac{f(X_i)}{p(X_i)}$$

Here the idea is that $p(x)$ is easier to sample from, and we are still able to estimate properties of $f(x)$ by using this weighted average $f(X_i)/p(X_i)$ of the samples. This process helps simplify the evaluation of difficult integrals and provides an approximation of expected values using a series of random samples. It is particularly useful for solving high-dimensional problems and is widely used in fields where direct computation of integrals is impractical or impossible.

Bootstrap

The second statistical method we briefly covered was Bootstrapping. The key idea behind bootstrapping is that, from a small original sample size, we can generate many new samples by randomly selecting data points with replacement. This allows us to estimate the variability of a statistic, such as the sample mean or median, even when the original dataset is small.
Bootstrapping involves randomly selecting data points with replacement from the original sample to create new bootstrap samples, each of the same size as the original dataset. Since each bootstrap sample is created with replacement, some data points may appear multiple times in a simple sample, while others may not appear at all. By generating many bootstrap samples from the original dataset, we can estimate the sampling distribution of a statistic. **The sampling distribution of a statistic is essentially how a statistic would behave mapped onto a specific probability distribution.** This is particularly useful for constructing confidence intervals and making inferences abut the population, especially when the dataset is small.
Bootstrapping significantly helps in situations where traditional methods of statistical inference may not be applicable, as it makes very few assumptions about the underlying distribution of the data. It is widely used in small sample scenarios for confidence interval estimation and bias correction.

Bayesian Statistics

In the first lesson, we touched on Bayesian statistics. The key idea in Bayesian statistics is that we update our knowledge about an unknown parameter based on observed data. This process combines our prior beliefs with the information provided by the data, leading to an updated belief called the posterior distribution. **The goal in Bayesian statistics is to use observed data to update our understanding of an unknown parameter theta or even missing data.**
The process works as follows:
1. **Prior** $P(\theta)$: The prior represents our belief about the parameter theta before observing any data. It is like a starting point or assumption about the possible values of theta, based on previous knowledge or intuition.
2. **Likelihood** $P(\textbf{data}|\theta)$: The likelihood reflects how likely the observed data is, given a specific value of theta. In other words, it tells us how well our assumption about theta (from the prior) matches the observed data. Basically, How good is your guess?
3. **Posterior** $P(\theta|\textbf{data})$: After observing the data, the posterior represents our updated belief about theta. The posterior distribution gives us a more refined understanding of the parameter theta, taking into account both our prior beliefs and the observed data.

Why do we care about Bayesian?
In many real-world problems, the posterior distribution is complex and doesn't have a closed form solution. This makes it difficult to compute directly. **Markov Chain Monte Carlo is a powerful tool that helps us sample from this posterior distribution when it's too complex to calculate exactly.**

Markov Chain Monte Carlo (MCMC)

The third statistical method we briefly covered in the first lesson was Markov Chain Monte Carlo Method. MCMC is a powerful tool used for sampling from complex probability distributions, particularly when direct sampling is difficult or impossible. **In simple terms, MCMC is a method used to generate random samples from complex probability distributions when other methods don't work.**
MCMC builds a chain of samples where each new sample is based on the previous one (the Markov Property), and over time, the samples will resemble the target distribution we're interested in. **The key idea behind MCMC is to generate random samples from a target distribution such as a posterior distribution in Bayesian statistics by constructing a Markov Chain.** A Markov Chain is a sequence of random variables where the next value in the sequence only depends on the current value, not on any previous values. This is called the Markov property. **MCMC combines the idea of a Markov chain with Monte Carlo sampling to approximate complex distributions.**
MCMC allows us to generate a sequence of samples that are approximately distributed according to the target

distribution (the posterior). By generating enough samples, we can then estimate expectations, probabilities, and other properties of the distribution (the posterior).

Process for Handling Missing Data with MCMC:

1. **Define a prior**: Start by assuming the missing data comes from a certain distribution. This is your prior belief.

2. **Use the data**: Observed data helps you understand what the missing values might be. You calculate the likelihood of the observed data for different guesses about the missing values.

3. **Posterior distribution**: Combine your prior belief with the data to form the posterior distribution of the missing values. This posterior distribution reflects your updated beliefs.

4. **Use MCMC**: Since the posterior is difficult to compute directly, use MCMC to sample from it. Over time, the Markov Chain generates samples that give you a good approximation of the missing values.

5. **Impute the Missing Data**: Finally, use the samples from the posterior distribution to either fill in the missing data or to understand the range of possible values for the missing data.

MCMC is the tool we use to sample from the posterior distribution when it's too complex to calculate directly. Using MCMC, we can estimate missing values while accounting for both the observed data and the uncertainty in our estimates. The beauty of MCMC is that it inherently accounts for mising data in the iterative process.

Implementation in R

Monte Carlo in R

```
# Set seed for reproducibility
# Reproducibility is important because of the randomization process of Monte Carlo
set.seed(123)
# Common Base R functions for Monte Carlo:
# rnorm(): For generating random numbers from a normal distribution
# runif(): For generating random numbers from a uniform distribution
# sample(): For generating random samples from a set of values

# Example
# number of samples
n <- 1000
samples<- rnorm(n,mean=0,sd=1) # Generate n random samples from N(0,1)
expected_value <- mean(samples) # Estimate the expected value (mean)
expected_value
```

```
## [1] 0.01612787
```

```
# This is a very simple example of Monte Carlo, where the
# chosen probability distribution was standard normal
```

Bootstrap in R

```
# Set seed for reproducibility
# Reproducibility is important because of the randomization process of Bootstrapping
set.seed(123)
```

```r
# Bootstrapping package in R
library(boot)
# Example data using 100 random samples from standard normal distribution
data <- rnorm(100)
# Create a function to calculate the statistic (mean in this case)
boot_mean <- function(data,indices){
 return(mean(data[indices]))
 }
boot_results<-boot(data,statistic=boot_mean, R=1000)
boot_results
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = data, statistic = boot_mean, R = 1000)
##
##
## Bootstrap Statistics :
##       original      bias    std. error
## t1* 0.09040591 0.002019936  0.08894492
```

Bayesian in R (Not Covered in Lecture Yet)

```r
# We did not cover this in lecture yet
# I wanted to know what packages are typically used so I looked them up
# Most widely used packages in R are rjags and brms
# install.packages("rjags")
# install.packages("brms")
# library(rjags)
# library(brms)
# These packages allow you to define models, run Bayesian inference
# and obtain posterior distributions using MCMC sampling methods
```

MCMC in R (Not Covered in Lecture Yet)

```r
# We did not cover this in lecture yet
# I wanted to know what packages are typically used so I looked them up
# rjags is also used for MCMC
# install.packages("MCMCpack")
# library(MCMCpack)
```

Vectorization

Vectorization refers to the process of applying operations to entire vectors (or arrays) of data at once, rather than iterating through elements one by one using loops. **Vectorization is important to data scientists because it saves run time/computational power. Furthermore, data scientists are often interested in optimizing the efficiency and speed of execution of code, which often can be achieved through vectorization.** Vectorization is used in scientific computing where huge chunks of data need to be processed efficiently. **A key point to understand is that in general, a vectorized approach will almost always be faster than a loop-based approach in R.** This is because vectorized operations are already optimized in R and avoid overhead from R's interpreter.

So do Data Scientists use loops?

Data Scientists concerned with optimizing run time and data processing efficiency will always use a vectorized approach if possible. **The key here is that a vectorized approach may not always be possible as emphasized in the lecture.** There are several instances/examples in which vectorization is not always feasible. These examples include:
1. Non-Vectorizable Logic - When the computation has a dependency between iteratons. For example, recursive relationships or problems where the value of the current iteration depends on the previous one.

```
x <- numeric(100)
x[1] <- 1
for (i in 2:100) {
  x[i] <- x[i-1] + i  # Dependency between iterations
}
```

2. Stateful Processes - Algorithms like MCMC or random walks depend on a sequential process where the current state is informed by the previous state. These are inherently iterative and can't be fully vectorized.
3. Dynamic Programming - Algorithms like dynamic programming often build solutions step by step, and loops are the most natural way to handle this.
4. Memory Constraints - In some cases, working with extremely large datasets, a loop might be more efficient than trying to vectorize as vectorization could force the creation of intermediate objects, consuming additional memory.
Rare, Special Cases - If a vectorized approach involves creating lots of intermediary objects or copying large datasets, a well-optimized loop that handles the memory better could actually be faster!

In summary, Data Scientists are sometimes forced to use loops for specific algorithms or processes that interfere with vectorization logic. Whenever plausible, Data Scientists will use vectorization to improve code efficiency and cleanliness.

How to Measure Time in R

In R, the system.time() function is used to determine the time it takes the computer to run the code and get an output. The system.time() function output gives us several insights. **The user time is the time it takes for the computer to actually carry out operations and calculations in your code. System time is the time the computer spends managing system-related tasks for your code, like handling memory or accessing files. Elapsed time is the total real time it takes for your code to finish.** If your process is computationally heavy, you'll see more user time. If your process involves a lot of file access or memory handling, you'll see more system time. This is an important thing to understand for future, computationally taxing tasks.

Example Showcasing Speed of Vectorization vs Loop

```
vec <- seq(1,1e6,by=1)
result <- numeric(length(vec))
time_loop <- system.time({
  for (i in 1:length(vec)){
    result[i]<- vec[i] + 10
  }
})
print(time_loop)
```

```
##    user  system elapsed
##   0.077   0.000   0.077
```

```
vec2<- seq(1,1e6,by=1)
result2<- numeric(length(vec))
time_vectorized<- system.time({
```

```
  result2<- vec2 + 10
})
print(time_vectorized)
```

```
##    user  system elapsed
##   0.001   0.000   0.001
```

As we can clearly see and expect, time_vectorized is much faster than time_loop.

Problems with Loops

There are instances where a fully non-vectorized approach will be extremely slow or cause time-limit errors. To try and avoid some these problems, Data Scientists may use built-in functions in R. Some examples of built in functions that are commonly used include:

sum(): sum of all elements in the vector
prod(): product of all elements in the vector
log(): log of all elements in the vector
cumsum(): cumulative sum of all elements in the vector
cumprod(): cumulative product of all elements in the vector
dist(): distance matrix between rows of a data matrix, can specify method, default method is Euclidean

Pros and Cons to Vectorization

Pros
1. Most built in functions are already vectorized
2. Vectorization makes your code look more succinct and run faster
3. Avoids using for loops, which can signficantly slow down operations after a large number of iterations

Cons
1. Vectorization is limited - Vectorization is limited by certain algorithmic dependencies, such as when each iteration relies on the result of the previous one, making recursive functions or stateful algorithms not suitable for vectorization.

Dichotomization

Dichotomization is the process of converting a continuous variable into a binary variable. For example, you might dichotomize income into "high" and "low" based on a certain threshold. Another common example is categorizing blood pressure measurements into high and low based on certain thresholds.

Pros and Cons

Pros - Dichotomization is often done for simplicity or when working with certain statistical models that require binary variables, like logistic regression or classification tasks. Dichotomization can make results easier to interpret, allows for the fitting of specific statistical models, and can handle non-linearity (if the relationship between the variable and the outcome is non-linear).

Cons- Dichotomization of a continous variable can lead to significant loss of information and statistical power. Collapsing a range of values into two categories removes the specific details, and nuances of the data. Intuitively, this creates a reduction in variability that is lost from the original dataset. Furthermore, deciding the threshold for dichotomization can be subjective and arbitrary, which is a problem because it can lead to different conclusions based on the cutoff placement.

Vectorized Approach to Dichotomization

```
# the use of the ifelse() function achieves the goal of dichotomization
# For loops can perform dichotomization but need if() .... else()
# Additional techniques for Dichotomization include:
# cut(): particularly useful for when you need to divide a continuous variable into intervals, which ca
# findInterval(): more flexible alternative to cut(),
# used to categorize data based on intervals
# Logical operations: >, < , ==, != are ways to dichotomize continous data
# into binary outcomes
# factor(): directly convert a variable into binary factor
```

Subscripting

Subscripting is the process of accessing specific elements, rows, columns, or subsets of a data structure, such as a vector, matrix, or data frame, using indices or conditions. R provides flexible ways to use subscripting for data manipulation and analysis.

Review of Subscripting Functions in R

[] : Basic subscripting with indices or conditions
$: Acessing columns in a dataframe
subset(): Simplifies filtering rows and selecting columns
which(): Find the indices of TRUE values in a logical condition
Logical operators: >, <, ==, != : To filter rows or select elements

Subscripting with Commas and Blanks

For Vectors, No Commas Needed because a vector has only one dimension. It is just a list of elements, so you can access elements using indices or conditions directly

For Matrices and Data Frames, a comma is used to separate rows and columns in the subscript: before the comma refers to rows, after the comma refers to columns.
1. **Accessing Specific Elements**: When accessing specific elements in a matrix or data frame, you need to provide both the row and column index.
2. **Leaving One Side Blank to Select All Rows/Columns**: You can leave the row or column index blank if you want to select all rows or all columns. This is very powerful when you need an entire row or column without selecting individual elements. **Blank before the comma, selects all rows. Blank after the comma, selects all columns**

```
# Example using matrices
m<- matrix(1:9, nrow=3)
# Access element in 2nd row, 3rd column
m[2,3]
```

```
## [1] 8
```

```
# All elements from the 2nd row
m[2,]
```

```
## [1] 2 5 8
```

```
# All elements from the 3rd column
m[,3]
```

```
## [1] 7 8 9
```

```r
# Example using dataframe
# Example data frame
df <- data.frame(Name = c("Alice", "Bob", "Charlie"), Age = c(25, 30, 22), Score = c(90, 85, 88))
# Access element in 1st row, 2nd column
df[1,2]
```

```
## [1] 25
```

```r
# Select all rows of the "Score" column
df[, "Score"]
```

```
## [1] 90 85 88
```

```r
# Select all columns for the 1st row
df[1, ]
```

```
##    Name Age Score
## 1 Alice  25    90
```

3. **Leaving Both Blanks to Select All Elements**: When you leave both the row and column indices blank, it selects all elements from the data structure, useful when you want to reference the entire matrix, or dataframe.

```r
m[,]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```r
df[,]
```

```
##      Name Age Score
## 1   Alice  25    90
## 2     Bob  30    85
## 3 Charlie  22    88
```

4. **Using Negative Indices to Exclude Rows or Columns**: Negative indices are used to exclude specific rows or columns. If you want all rows or columns except one, you can use negative indexing.

```r
# Exclude the 2nd row
m[-2,]
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    3    6    9
```

```r
# Exclude the 3rd column
m[,-3]
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```r
# Exclude the "Age" column from the dataframe
df[,-2]
```

```
##      Name Score
```

```
## 1   Alice    90
## 2     Bob    85
## 3 Charlie    88
```

5. **Using Logical Conditions for Rows and Columns**: You can use logical statements within subscripting to select rows or columns that satisfy certain criteria. This is particularly useful for filtering dataframes.

```
# Select rows where Age is greater than 25
df[df$Age>25,]
```

```
##   Name Age Score
## 2  Bob  30    85
```

As a result of understanding both dichotomization and subscripting, it is clear to me that dichotomization is basically just a specific form of subscripting that involves using binary categorization in its filtering process for the data based on specific conditions or criteria. This is an interesting insight to obtain from the lectures.

Family of Apply Functions

Before getting into the family of apply functions that are available in R, we briefly discussed how to find the sum of each column in a matrix.

```
# Two ways to sum each column in a matrix
data <- matrix(1:16,nrow=4,ncol=4,byrow=TRUE)
sum_col<- colSums(data)
sum_col
```

```
## [1] 28 32 36 40
```

```
sum_x_col<- rep(1,nrow(data)) %*% data
sum_x_col
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   28   32   36   40
```

Alternatively, for a large matrix, writing a loop to calculate the sum of each column can be slow and cumbersome. Luckily, we have a family of built-in functions in R that can we can use to help us.

Apply Function

Apply() can apply the same operation across the rows or columns of a matrix or array, simplifying the code. Apply() applies a function over the margins (rows or columns) of an array or matrix. Parameters include: **X**: Array or Matrix, **MARGIN**: Dimension to apply the function over (1 = rows, 2 = columns) , **FUN**: Function to apply, sum, mean, etc., **...**)
apply(X,MARGIN,FUN,...)

```
# Example in R
# Sum over rows
# data is a 4 x 4 matrix, from 1:16 from above
apply(data,1,sum)
```

```
## [1] 10 26 42 58
```

```
# Sum over columns
apply(data,2,sum)
```

```
## [1] 28 32 36 40
```

Lapply Function

lapply() can apply a function to each element of a list and return a list of results. Parameters: **X**: The list, **FUN**: The function to apply
lapply(X,FUN,...)

```
# Example in R
# Sum each element of the list
my_list<- list(a=1:5, b=6:10)
lapply(my_list, sum)
```

```
## $a
## [1] 15
##
## $b
## [1] 40
```

Sapply Function

sapply() simplified version of lapply() that tries to simplify the result into a vector, matrix, or array when possible. Parameters: **X**: The list or vector, **FUN**: The function to apply.
sapply(X,FUN,...)

```
# Example in R
# Sum each element and simplify it to a vector
my_list<- list(a=1:5, b=6:10)
sapply(my_list, sum)
```

```
##  a  b
## 15 40
```

Vapply Function

vapply() similar to sapply(), but you explicitly specify the type of output. **This makes vapply() safer and often faster**. Parameters: **X**: The array or matrix, **FUN**: The function to apply, **FUN.VALUE**: The expected type of output.

```
# Example in R
# Sum each element and simplify it to a vector
my_list<- list(a=1:5, b=6:10)
vapply(my_list, sum, numeric(1))
```

```
##  a  b
## 15 40
```

Common Misconception

Using apply(), lapply(), sapply(), etc., is not true vectorization. In reality, these functions just hide the loops, they do not offer the same efficiency gains as vectorized operations

Example where apply is slower than for loop

```
# Example in R
# Create a large matrix with 10,000 rows and 10,000 columns
set.seed(123)
large_matrix<- matrix(rnorm(1e7),nrow=10000,ncol=10000)
# Use apply to sum the rows
system.time({
```

```
  rows_sums_apply<- apply(large_matrix,1,FUN = sum)
})
```

```
##    user  system elapsed
##   2.678   0.801   3.480
```

```
# Initialize a vector to store row sums
rows_sums_loop<- numeric(nrow(large_matrix))
# Use a for loop to sum the rows
system.time({
 for (i in 1:nrow(large_matrix)){
   rows_sums_loop[i]<- sum(large_matrix[i,])
 }
})
```

```
##    user  system elapsed
##   2.071   0.438   2.510
```

rowSums() is optimal

Using rowSums() instead of the apply() function or a for loop is optimal because rowSums() is a vectorized process

```
system.time({
  row_sums_vectorized <- rowSums(large_matrix)
})
```

```
##    user  system elapsed
##   0.219   0.000   0.219
```

As we can see, rowSums() is significantly faster than the apply function or a for loop.

Other functions

Vectorize(): Vectorize creates a function wrapper that vectorizes the action of its argument FUN. Parameters: **FUN**: The function you want to vectorize. **vectorize.args**: The arguments you want to vectorize (i.e., treat as vectors). **SIMPLIFY**: If TRUE, the result will be simplified to an array or matrix if possible. **USE.NAMES**: If TRUE, the result will use the names of the input vectors

In simpler terms, Vectorize() is a tool that helps you apply a function to many items at once (like a whole list) instead of doing it one by one, making the process faster and easier.

```
# Vectorize(FUN,vectorize.args=arg.names, SIMPLIFY=TRUE,USE.NAMES=TRUE)
f<- function(x,m,FUNV=NULL){
  combn(x,m,FUN=FUNV)
}
combnV<- Vectorize(f,vectorize.args = c("x","m"))
result<- combnV(4,1:4)
print(result)
```

```
## [[1]]
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
##
## [[2]]
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    1    1    2    2    3
```

```
## [2,]    2    3    4    3    4    4
##
## [[3]]
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    2
## [2,]    2    2    3    3
## [3,]    3    4    4    4
##
## [[4]]
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

Introduction to Numerical Methods in R

Positional Numbers System

A positional number system is a numeral system in which the value of a digit is determined by its position within a number and the base of the system. The most common system is base 10 (decimal), but computers often use base 2 (binary).

```
.Machine$double.base
```

```
## [1] 2
```

This tells us that in R, the positional number system used is base 2. Mathematically, a positional number system with base 2 looks like this:

$$d_1 \cdot 2^n + d_2 \cdot 2^{n-1} + d_3 \cdot 2^{n-2} + \cdots$$

with decimal point separating d_0 and d_{-1}, where $d_j \in [0, 1]$. A key point to consider from lecture: **Understanding the binary system is important for evaluating how computer store numbers, as this can lead to computational limitations such as precision and rounding errors.**

The function strtoi() returns the vector of digits that represent a given integer in another base.

To Convert Binary to Decimal

```
binary_string<- "1101"
decimal_number<- strtoi(binary_string,base=2)
print(decimal_number)
```

```
## [1] 13
```

Machine Variable in R

Machine variables refer to how numbers are stored in memory, typically as integer, double or logical types. Each type has its own limitations regarding precision, which depends on the number of bits allocated for storing values. This is crucial for understanding how the computer conducts floating-point arithmetic. The precision is limited by how many bits the computer allocates to store these values. Integers are 32-bits, floats are 32 bits, and doubles are 64 bits.

```
# Checking type and machine precision
num <- 1.5
typeof(num)  # "double"
```

```
## [1] "double"
```

```r
.Machine$double.eps  # Smallest positive number such that 1 + eps != 1
```

```
## [1] 2.220446e-16
```

```r
.Machine$integer.max # Largest integer that can be represented in R
```

```
## [1] 2147483647
```

```r
.Machine$double.xmin # The smallest possible double-precision number that can be represented in R
```

```
## [1] 2.225074e-308
```

```r
.Machine$double.xmax # The largest possible double-precision number that can be represented in R
```

```
## [1] 1.797693e+308
```

```r
.Machine$double.neg.eps # Smallest representable negative number
```

```
## [1] 1.110223e-16
```

```r
.Machine$double.digits # Number of base-10 digits that can be represented
```

```
## [1] 53
```

Fixed and Floating-Point Numbers

Fixed-point numbers represent numbers with a finite sequence of digits before and after the decimal point. The position of the decimal point is constant. For example, positive integers. Fixed-point numbers are less flexible when it comes to representing large range of values.
Floating-point numbers (more common in R) can "float", meaning the decimal point can move to accomodate a wider range of values by using a exponent and mantissa (fractional part). Can represent a much broader range of values, from very large to very small, but with limited precision. **Floating point numbers can represent larger values but with a tradeoff: they may lose precision when dealing with very large or very small numbers.**

```r
fixed_point <- sprintf("%.2f", 123.456)  # Display 2 decimal places
fixed_point
```

```
## [1] "123.46"
```

```r
floating_point <- 1.23456e2  # Equivalent to 123.456 in scientific notation
floating_point
```

```
## [1] 123.456
```

```r
# Broken mathematical identity due to floating-point precision
result <- (0.3 - 0.1) == 0.2
result
```

```
## [1] FALSE
```

This returned false as expected.

How to Deal with Floating Point Precision

The function, **all.equal()** uses machine tolerance The function **identical()** and == are equivalent and does not properly deal with floating point.

```r
# Use all.equal to handle floating-point precision tolerance
all.equal(0.3 - 0.1, 0.2)
```

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

```r
identical(0.3-0.1,0.2)
```

```
## [1] FALSE
```

As we can see identical() is equivalent to ==, showing false which is contradictory to mathematical identities.

Overflow and Underflow

Overflow occurs when a number exceeds the largest value that can be stored in memory (causing it to return Inf in R). Underflow occurs when a number is too small to be represented accurately and the result is effectively rounded down to 0.

```r
# Largest double value in R
largest_double <- .Machine$double.xmax
largest_double
```

```
## [1] 1.797693e+308
```

```r
# Largest integer value in R
largest_integer <- .Machine$integer.max
largest_integer
```

```
## [1] 2147483647
```

```r
# Overflow for double: multiplying the largest double results in Inf (Infinity)
overflow_example <- largest_double * 2
overflow_example  # Returns Inf
```

```
## [1] Inf
```

```r
# Incrementing the largest double by 1 does not cause overflow (because of float precision)
incremented_double <- largest_double + 1
incremented_double  # Still a valid double, but no precision change due to large value
```

```
## [1] 1.797693e+308
```

```r
# Overflow for integers: integers are exact, so exceeding the largest integer causes an issue
integer_overflow <- largest_integer + 1
integer_overflow  # Returns NA because integers overflow
```

```
## [1] 2147483648
```

```r
# Smallest double precision value in R (approaches 0)
smallest_double <- .Machine$double.xmin
smallest_double
```

```
## [1] 2.225074e-308
```

```r
# Underflow for doubles: dividing the smallest double by a large number results in underflow
underflow_example <- smallest_double / 100000
underflow_example  # Does not cause underflow
```

```
## [1] 2.225074e-313
```

```r
# Smallest normalized number in R
smallest_normalized <- .Machine$double.xmin
print(smallest_normalized)
```

```
## [1] 2.225074e-308
```

```r
# Divide by a large number to cause true underflow
underflow_example <- smallest_normalized / 1e20
print(underflow_example)  # Now this returns 0 due to underflow
```

```
## [1] 0
```

In practical applications, it's important to be mindful of operations that might lead to overflow or underflow, as these can introduce inaccuracies in numerical computations through loss of information and statistical power. Essentially, **this was emphasized to understand the computer has a floor and ceiling in terms of numerical storage and rounding precision which can potentially lead to some mathematical identities appearing to be contradicted by software.**

Why it Matters?

Binary Representation: All data in computers is ultimately binary, including numbers, text, and instructions. The desire for paramount precision can be everpresent in financial calculations, as even tiny errors can have significant impacts. Furthermore, this precision is needed in scientific computing and data analysis, where comparisons involving floating point numbers can lead to unexpected outcomes if not handled carefully.

Approximation of a Function

In numerical methods, approximating a function often involves interpolation or polynomial approximation when exact solutions are impossible or computationally expensive.

Taylor Series Expansion

The Taylor series expansion of a function $f(x)$ around a point x_0 is given by:

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{f''(x_0)}{2!}(x - x_0)^2 + \frac{f^{(3)}(x_0)}{3!}(x - x_0)^3 + \cdots$$

This series allows us to approximate functions by polynomials in the vicinity of x_0.

Computers cannot handle infinite series, and must truncate it to obtain a numerical approximation.

In the lecture, the key takeaway was that computers cannot handle infinite series, so we need to truncate or "cut" the series when performing calculations. This process of truncation isn't necessarily bad, as it still gives us useful approximations. For numerical methods, especially in optimization (minimizing or maximizing a function), we often approximate derivatives because exact evaluation of the function at certain points can be complex. By approximating the derivative and setting it equal to zero, we can find points that either minimize or maximize the function and this can be very useful in real-world problems.

Implementation of Taylor Series in R

```r
#install.packages('pracma')
library(pracma)
```

```
##
## Attaching package: 'pracma'
```

```
## The following object is masked from 'package:boot':
##
##     logit
```

```r
p <- taylor(f=exp,x0=0,n=3)
p
```

```
## [1] 0.1666667 0.5000000 1.0000000 1.0000000
```

```
approx_values<- polyval(p,1:5)
print(approx_values)
```

```
## [1]  2.666667  6.333334 13.000002 23.666670 39.333341
```

Numerical Derivatives

Numerical derivatives approximate the derivative of a function using finite differences. A common method is the central difference formula we covered in class and went over the theory to obtain this result:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Critical Reflection

For the first two weeks of lecture, up to Tuesday, I am going to include Thursday's lecture in the Summary 2, I would say the key takeaways were to get a basic understanding of more complex statistical methods like MC, Bootstrapping, and MCMC and when we would want to potentially use them. In addition, I think the professor emphasized the importance of vectorization to optimize run time and efficiency and cleanliness of code, as many professional roles actually care about computational speed. Finally, I found it very interesting to learn about the computer, and how it processes information, how we can get contradictions in mathematical identities, and that using numerical methods, we can approximate derivatives, which are most useful for optimization problems.