Simulation - Cross validation - Monte Carlo methods

Simulation (a.k.a Monte Carlo simulation) is about analyzing any theoretical property using repeated random samplings or synthetic data. Sometimes, we even learn about a theoretical feature via simulation results. Simulation based methods are also often called Monte Carlo methods.

Why do we need simulation studies in statistics? Simulation studies are designed to understand the mechanistic features of any statistical model. We often identify default settings where the model works the best. One can however justify by deriving theoretical properties. Nevertheless, such derivations are usually limited to simpler models. Hence, simulations are cheaper and somewhat easier alternatives.

Here are some theoretical results that we know. We will validate these results via simulation.

1 Proof of theoretical concept

If you understand a theory, you should be able to verify the result using simulated datasets. So, running a simulation experiment is also good to verify if you understand the concept accurately.

• If $x_1, \ldots, x_n \sim \text{Normal}(\mu, \sigma^2)$, then the $\alpha 100\%$ confidence interval can be given by $(\bar{x} - Z_{1-\alpha/2}\sigma/\sqrt{n}, \bar{x} + Z_{1-\alpha/2}\sigma/\sqrt{n})$ is σ is known. Otherwise $(\bar{x} - t_{1-\alpha/2,n-1}\hat{\sigma}/\sqrt{n}, \bar{x} + t_{1-\alpha/2,n-1}\hat{\sigma}/\sqrt{n})$.

Take true $\mu = 0$ and $\sigma = 1$. Let's verify the above theoretical confidence interval using simulation for $n = 30, 50, 100$.

```
n <- 50 #Change this to 30 and 100
alpha <- 0.05
mu0 <- 1
ind <- 0
for(rep in 1:10000){
  sam <- rnorm(n, mu0, 1)
  CIlower <- mean(sam) - qnorm(1-alpha/2)*1/sqrt(n)
  CIupper <- mean(sam) + qnorm(1-alpha/2)*1/sqrt(n)

  #Checking wheather CI includes mu0
  ind[rep] <- (CIlower <= mu0) * (CIupper >= mu0)
}

mean(ind) #Check if it matches theoretical value of 1-alpha
```

- Testing type 1 error is indeed α for a α significant test.
- Test $H_0 : \mu = \mu_0$ against $H_A : \mu \neq \mu_0$ To evaluate whether size/level of test achieves advertised α under H_0 Generate data under $H_0 : \mu = \mu_0$ and calculate the proportion of rejections of H_0. Proportion of rejecting H_0 approximately equals to α.

Let,
n = 20 (number of samples)
B = 1000 (number of simulations)
alpha = 0.05
two.sided test

```
alpha <- 0.05; mu=0; n <- 20; B <- 1000

counts <- 0
for(b in 1:B){
  set.seed(b)

  x <- rnorm(n, mean=mu)
  atest <- t.test(x)

  #Rejection criteria is atest$p.value < alpha
  if(atest$p.value < alpha){
    counts <- counts + 1
  }
}

counts/B

alpha <- 0.05; mu=0; n <- 20; B <- 1000
```

```
counts <- 0
for(b in 1:B){
  set.seed(b)

  x <- rnorm(n, mean=mu)
  atest <- wilcox.test(x)

  if(atest$p.value < alpha){
    counts <- counts + 1
  }
}

counts/B
```

If several tests has the same size, how to compare which test is better?
Ans: We can compare power.

To evaluate power: Generate data under some alternatives, $\mu \neq \mu_0$ (i.e.,$\mu = \mu_1$) and calculate the proportion of rejections of H_0.
Test $H_0 : \mu = 0$ against $H_1 : \mu = 1$

2 Cross-validation (CV)

This is also a Monte Carlo method. There are several varieties of this method. 1) It helps to compare and select an appropriate model for the specific predictive modeling problem. 2) It also often helps to tune a hyperparameter.

In the first case, we use CV to compare across different models, like comparing the least square regression with the least absolute deviation regression or ridge penalty based regression with LASSO penalty based regression. Here, the candidate models are 1) a) the least square regression and b) the least absolute deviation and in the second case, 2) a) the ridge penalty based regression and b) the LASSO penalty based regression.

In case of hyperparameter tuning, our candidate models are under the same class with different hyperparameters. For example, if we want to learn the appropriate penalty parameter λ for a ridge regression, the candidate models are a) ridge regression with penalty $\lambda = 0.1$, b) ridge regression with penalty $\lambda = 0.2$, c) ridge regression with penalty $\lambda = 0.3$, etc.

Hold-out cross-validation is the simplest and most common technique. The algorithm of hold-out technique:

- Divide the dataset into two parts: the training set and the test set.
- Usually, 80% of the dataset goes to the training set and 20% to the test set, but you may choose any splitting that you might prefer.
- Train the model on the training set.
- Validate on the test set as in computing prediction error.
- Pick the model with the best validation performance, like with the least prediction error.

However, hold-out has a major disadvantage. If the dataset is not evenly distributed, validation based on just one test-train split may not be very robust. For example, the training and test sets may differ a lot. Thus, the following alternative methods are usually considered.

- K-fold cross-validation.
- Leave-p-out cross-validation.

The choice often depends on computational time as well as accuracy in selection. However, they all depend on multiple test-train splits.

K-fold cross-validation

1. Pick a number of folds – k. Usually, k is set to 5 or 10.
2. Split the dataset into k equal (if possible) parts (they are called folds). Thus each section will have approximately n/k data points.
3. Choose $k-1$ folds as the training set. The remaining fold will be the test set
4. Train the model on the training set. On each iteration of cross-validation, you must train a new model independently of the model trained on the previous iteration
5. Validate on the test set Save the result of the validation

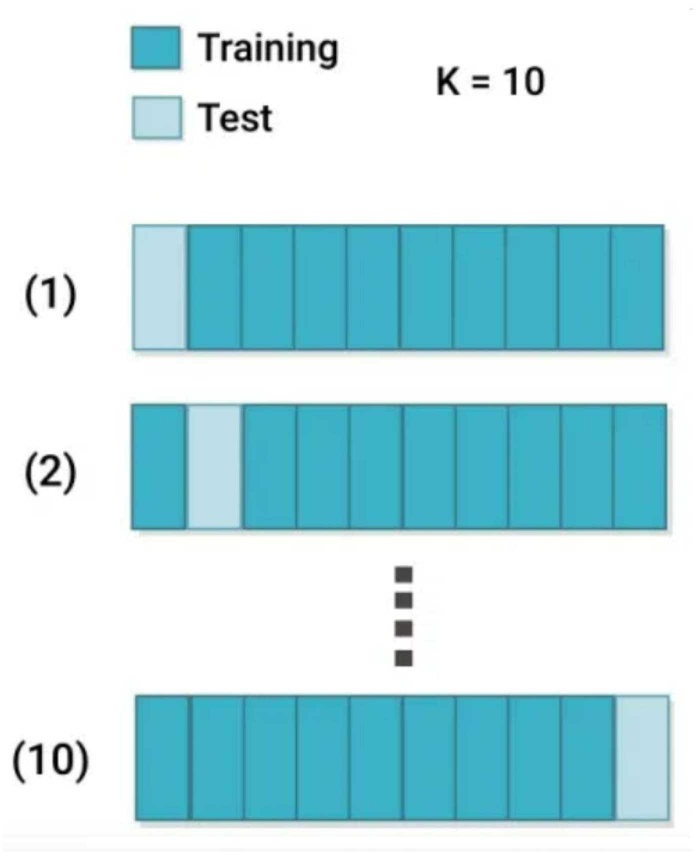

Figure 1: In case 10-fold CV, we partition the data of equal size. Then we have 10 possible choices for the test dataset. We thus will (parallelly) fit our model 10 times for each case.

6. Repeat steps 3 – 6 k times. Each time use the remaining fold as the test set. In the end, you should have validated the model on every fold that you have.

7. To get the final score average the results that you got on step 6.

Figure 1 illustrates this technique.

Leave-p-out cross-validation: Under this strategy, we randomly pick p observations to form our test set. The rest of the $n - p$ observations are used for training. There are in total $\binom{n}{p}$ possible test-train splits under this scheme. Ideally, we should consider all of them and average out to get the final score.

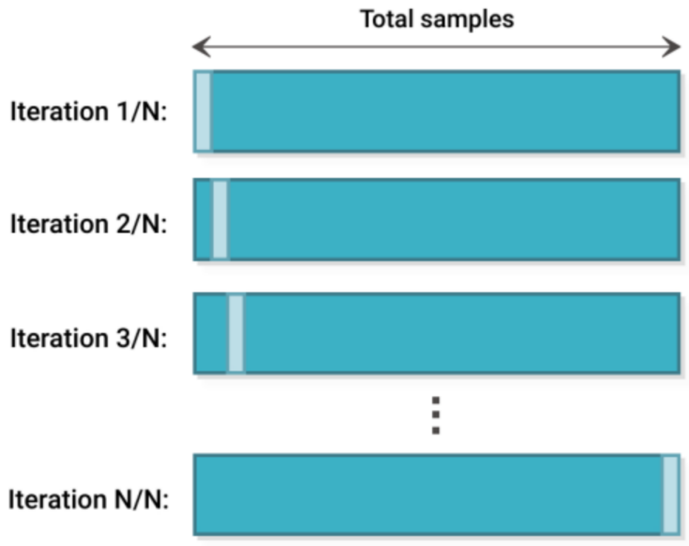

Figure 2: In case leave-one-out CV, we run the test-train splitting N times. In each case, the test dataset only contains one datapoint.

Since, estimation of the parameters will always depend on training data. Thus, we usually let $n - p \geq p$ and must have $p \geq 1$. Under this constraint, $\binom{n}{p}$ is the minimum for $p = 1$. The $p = 1$ case is commonly referred as 'Leave-one-out cross-validation'. Figure 2 illustrates this technique.

Repeated random sub-sampling CV: Under this technique, we run Leave-p-out cross-validation for a pre-specified number of times instead of all possible $\binom{n}{p}$ choices. This pre-specified number should be a bit large enough, like 20 or 30.

3 Establishing computation time

We know that the computation time to invert an $n \times n$ matrix is of order $O(n^3)$ (this is called Big O notation). This means that $f(n) \leq Cn^3$ for all $n > n_0$ with some constant $C > 0$ and $n_0 > 0$. Here, $f(n) =$ Theoretical computation time to invert an $n \times n$ matrix.

```
nvec <- c(100, 300, 600, 900, 1200, 1500, 1800, 2100, 2400)

timemat <- matrix(0, length(nvec), 6)

for(i in 1:length(nvec)){
  n <- nvec[i]
  for(j in 1:6){
    #Generate a nXn matrix
    mat <- matrix(rnorm(n^2), n, n)
    t1 <- Sys.time()
    B <- solve(mat)
    t2 <- Sys.time()

    timemat[i, j] <- t2 - t1
  }
}
plot(nvec, rowMeans(timemat), type='l')

time <- rowMeans(timemat)
plot(time/nvec)
plot(time/nvec^2)
plot(time/nvec^3)
plot(time/nvec^4)
```

In the above example, `plot(time/nvec)` and `plot(time/nvec`^{2}`)` are increasing. So, `time` is definitely increasing at a faster rate than `nvec`^{2}. Then `plot(time/nvec`^{3}`)` sort of becomes constant and plot(time/nvec^4) is decreasing. Decreasing means that `time` is increasing at a slower rate than `nvec`^{3}. Hence, `time` is increasing at a rate similar to `nvec`^{3}. Hence, our theoretical result is verified.

Another way could be just regress `log(time)` on `log(nvec)` if we expect that the order will be polynomial [Specifically polynomial order means time $\approx C$ nvec^a or equivalently log(time) $\approx \log(C) + a\log(\text{nvec})$. We need to estimate a].

```
    lm(log(time)~log(nvec))$coefficients[2]
```

This solution will be around 2.86, may not be exactly 3 due to computational error. This type of sampling-based computational error is also Monte Carlo error.

Computation time is always calculated as a function of some parameters (in the above case, it is the dimension n). For a general case, let that parameter be x and then $f(x)$ is the computation time x as the parameter. Here $f(x)$ is always unknown or very difficult to compute. We thus often try to find a simple function $g(x)$ such that $f(x) = O(g(x))$, meaning $\frac{f(x)}{g(x)} \leq C$, for some constant C. In the above example $g(x) = x^3$. Using simulation, we can in general compute the computation time of any method.