

1 Bootstrapping Functions

1.1 Basic Algorithm

According to Wasserman 2004 we can define $T_n = g(X_1, \dots, X_n)$ as an statistic that depends on the data, and apply bootstrapping for estimating the standard error and confidence intervals of T_n for statistical inference. First, Wasserman 2004 proves that by using the law of large numbers, and assuming that we draw an IID sample of Y_1, \dots, Y_B , we can conclude that as $B \rightarrow \infty$

$$\bar{Y} \xrightarrow{p} E(Y) \quad (1)$$

$$\frac{1}{B} \sum_{j=1}^B (Y_j - \bar{Y})^2 \xrightarrow{p} V(Y) \quad (2)$$

Therefore, we can use the sample mean and the sample variance as an approximation for the population mean and variance. Following the procedures stated by Wasserman 2004, we first assume that the data obtained initially follows a distribution \hat{F}_n , from where we can take samples X_1^*, \dots, X_n^* and compute $T_n^* = g(X_1^*, \dots, X_n^*)$ B times. From this step we will get a vector of $T_{n,1}^*, \dots, T_{n,B}^*$, from where we can compute the variance (Wasserman 2004):

$$v_{boot} = \frac{1}{B} \sum_{b=1}^B \left(T_{n,b}^* - \frac{1}{B} \sum_{r=1}^B T_{n,r}^* \right)^2 \quad (3)$$

The standard error of the statistic T_n is the square root of the variance, $SE = \sqrt{v_{boot}}$. Although there are several methods to estimate the confidence interval of the statistics, we are going to use the percentiles of the statistic's distribution. Consequently, the interval is defined as $C_n = (T_{\alpha/2}^*, T_{1-\alpha/2}^*)$.

The previous can be resumed in Algorithm 1 shows the steps followed more generally. One can observe that it divides the process into three main parts: the first part generates B samples, of n_b size, from the original data using replacement, the second part estimates the statistic T_n for each b sample, and, finally, the algorithm estimates the standard error and the confidence interval. The sampling phase is a loop of size B that applies a sampling function¹, therefore we can assume that the time complexity

of this process should be linear: $O(n)$, where n is equal to B . (Analysis based on Cormen et al. 2009) Similarly, the second phase, is a process that estimates the statistic for each sample b , which should be bounded by the same time complexity as the first phase if the statistic is simple enough, $O(n)$. Finally, the last step estimates the variance and the confidence interval, which, assuming as given steps without their own time complexity, have a constant time complexity $O(1)$. Under these assumptions, we can conclude that we expect for the algorithm to have a linear time complexity $O(n)$, where the slope is determined by the number of samples taken from the original data and the improvements made by using tools like parallel computing or optimized functions.

Algorithm 1: Bootstrapping

Data: A sample of a random variable X

Result: A standard error and confidence interval

```

1 /* Sampling phase */
2 for  $b$  in range( $B$ ):
3     | select  $X_{1,b}^*, \dots, X_{n,b}^*$  elements of the original data using
      | replacement;
4 /* Estimation phase */
5 for  $b$  in range( $B$ ):
6     | estimate  $T_{n,b}^*$  for the  $b$  sample;
7 /* Results phase */
8 estimate the square root of the variance of the statistics and the
   confidence interval

```

This algorithm could be translated into the following Python code following the steps mentioned by Wasserman 2004. As an example, we are going to assume that the original data comes from a normal distribution with $\mu = 5$ and $\sigma^2 = 1$ and we want to estimate the standard error and the confidence interval for the statistic, which in this case is the mean.

```

import numpy as np
from scipy.stats import norm
# Assume we have a random sample from a normal distribution
np.random.seed(11)
sample_normal = np.random.normal(5, 1, 10000)

```

¹We are assuming that the sampling function has a linear time complexity, since it can be understood as a loop that generates a random number, under some specific conditions, to select an index of the input.

```

# 1. Generate 1000 random samples with replacement
samples = np.array([np.random.choice(sample_normal,
                                     size=100, replace=True) for _ in range(1000)])

# 2. Estimate the mean (statistic) for each random sample
mean_dist = np.array([np.mean(x) for x in samples])

# 3. Estimate the standard error and the confidence interval
se_mean = np.sqrt(np.var(mean_dist))
confint_mean = np.percentile(mean_dist, [2.5, 97.5])
print("Standard Error: ", round(se_mean, 3),
      "\n95% Confidence Interval: ", round(confint_mean[0], 3),
      " - ", round(confint_mean[1], 3))

Standard Error:  0.099
95% Confidence Interval:  4.8 - 5.201

```

The main results from the algorithm are the standard error and the confidence interval of the statistic (i.e. mean), additionally we can observe the behavior of the bootstrap results in a histogram which, in the case of estimating the mean, illustrates the central limit theorem. Figure 1 shows the histogram of the bootstrap samples for the mean with the density kernel of the observed data (blue line), and the normal distribution (black line). Theory tells us that (Wasserman 2004), by the central limit theorem, the distribution of the sample mean converges in distribution to a $N(\mu, \sigma^2)$. With this, one can establish probability statements about the mean of the random variable X .

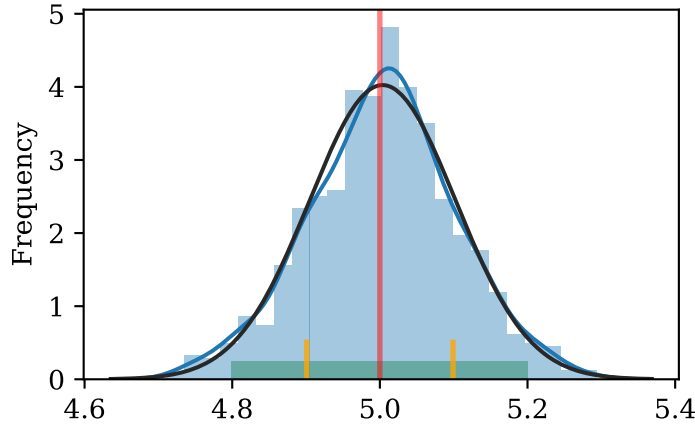


Figure 1: Histogram and density plot of the bootstrap samples. *Note:* confidence interval in green area, standard errors in orange and normal distribution in black

The algorithm implemented shows the expected results with respect to the statistical objectives, however, the main focus of this study is to analyze the performance of bootstrapping when using parallel computing. The next sections will divide the problem between a non-parallel version (i.e. serial) and a parallel implementation of the bootstrapping algorithm.

1.2 Serial Algorithms

The serial version of bootstrapping replicates the steps followed by the Algorithm 1 with only one processor of the computer. At this point, to construct a function that executes the algorithm, a main aspect is the generation of random numbers for the sampling phase. We decided to use two types of generators to understand how the performance of the algorithm can change based on specific changes on each step. The two options are: the `random` package of python core libraries (Algorithm 2) and `Numpy` package (Algorithm 3). It is noteworthy that the first implementation uses a nested loop (line 4), incrementing the time complexity to a polynomial form $O(n^2)$. Therefore, we can expect that the time complexity of the function that uses `numpy` is bounded by the function without `numpy`.

Algorithm 2: Serial Bootstrapping without Numpy

Input : A sample of a random variable X

Output: A object with the standard error and confidence interval

```
1 def bootstrap(input, statistic function, number of samples, size of
  sample):
2     /* Sampling phase */
3     for b in range(B):
4         for j in range(size of samples):
5             // Using random.randint module
6             index = create a random integer  $\in [0, \text{len}(\text{data}[n]) - 1]$ ;
7             sampleb[j] = data[index];
8     /* Estimation phase */
9     for b in range(B):
10        | result_array[b] = stat_function(array_of_samples[b]);
11    /* Results phase */
12    estimate the square root of the variance of the statistics and the
    confidence interval;
```

On the other hand, on Algorithm 3 we replicated the Algorithm 2 using the Numpy package to generate the samples from the original data. Since Numpy is a scientific package made to optimize processes like bootstrapping, we expect a improvement in terms of performance when both functions are compared. The Algorithm 3 enumerates the steps taken by the function to execute the bootstrapping. The main difference with the previous function resides in the **for** loop at line 3 in Algorithm 3, since we avoided the introduction of an additional **for** loop. Therefore, we could assume that the time complexity of this algorithm is $O(n)$, since it just performs the sample and estimation in one **for** loop.

Algorithm 3: Serial Bootstrapping with Numpy

Input : A sample of a random variable X

Output: A object with the standard error and confidence interval

```
1 def bootstrap_np(input, statistic function, number of samples, size
  of sample):
2     /* Sampling phase and Estimation phase */
3     for b in range(B):
4         // Using numpy.random.choice
5         array_of_samples[b] = create a random sample of determined
          size from data;
6         result_array[b] = stat_function(array_of_samples[b]);
7     /* Results phase */
8     estimate the square root of the variance of the statistics and the
      confidence interval;
```

To further analyze the performance of the algorithms, we first implemented a test to generate a range of B samples, since this is the main argument of the bootstrap function that could affect the performance of the functions. The range of the test has a minimum of $B = 1,000$ samples up to $B = 1,000,000$ samples by steps of 10,000. For each implementation of the algorithm, we time the performance using the `timeit` module of Python libraries, which returns an average time for each run.

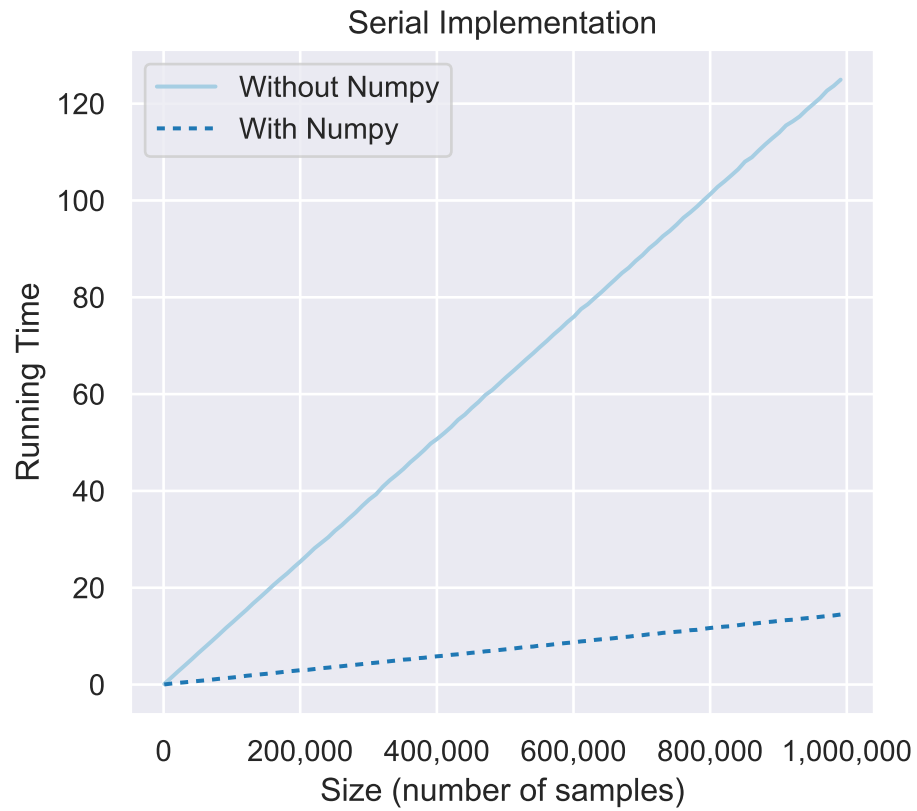


Figure 2: Bootstrapping Test for Serial Functions

References

- Cormen, Thomas H. et al. (2009). *Algorithms*. MIT Press.
- Wasserman, Larry (2004). *All of Statistics*. Springer Science+Business Media.