Its very exciting to announce first (public) package (and the second package I’ve ever written, the first being unannounced until the accompanying paper is accepted). That package is **MCHT**, a package for bootstrap and Monte Carlo hypothesis testing.

**MCHT** is a package implementing an interface for creating and using Monte Carlo tests. The primary function of the package is MCHTest(), which creates functions with S3 class MCHTest that perform a Monte Carlo test.

**Installation**

**Library**("MCHT").

**Monte Carlo Hypothesis Testing**

Monte Carlo testing is a form of hypothesis testing where the p-values are computed using the empirical distribution of the test statistic computed from data simulated under the null hypothesis. These tests are used when the distribution of the test statistic under the null hypothesis is intractable or difficult to compute, or as an exact test (that is, a test where the distribution used to compute $p$-values is appropriate for any sample size, not just large sample sizes).

Suppose that s_nis the observed value of the test statistic and large values of s_nare evidence against the null hypothesis; normally, p-values would be computed as 1 - F(s_n), where F(s_n) = F(S_n \leq s_n)is the cumulative distribution functions and S_nis the random variable version of s_n. We cannot use Ffor some reason; it’s intractable, or the Fprovided is only appropriate for large sample sizes.

Instead of using Fwe will use \hat{F}_N, which is the empirical CDF of the same test statistic computed from simulated data following the distribution prescribed by the null hypothesis of the test. For the sake of simplicity in this presentation, assume that Sis a continuous random variable. Now our p-value is 1 - \hat{F}_N(s_n), where \hat{F}_N(s_n) = \frac{1}{N}\sum_{j = 1}^{N}I_{\{\tilde{S}_{n,j} \leq s_n\}}where Iis the indicator function and \tilde{S}_{n,j}is an independent random copy of S_ncomputed from simulated data with a sample size of n.

The power of these tests increase with N(see [1]) but modern computers are able to simulate large Nquickly, so this is rarely an issue. The procedure above also assumes that there are no nuisance parameters and the distribution of S_ncan effectively be known precisely when the null hypothesis is true (and all other conditions of the test are met, such as distributional assumptions). A different procedure needs to be applied when nuisance parameters are not explicitly stated under the null hypothesis. [2] suggests a procedure using optimization techniques (recommending simulated annealing specifically) to adversarially select values for nuisance parameters valid under the null hypothesis that maximize the p-value computed from the simulated data. This procedure is often called *maximized Monte Carlo* (MMC) testing. That is the procedure employed here. (In fact, the tests created by MCHTest() are the tests described in [2].) Unfortunately, MMC, while conservative and exact, has much less power than if the unknown parameters were known, perhaps due to the behavior of samples under distributions with parameter values distant from the true parameter values (see [3]).

**Bootstrap Hypothesis Testing**

Bootstrap statistical testing is very similar to Monte Carlo testing; the key difference is that bootstrap testing uses information from the sample. For example a parametric bootstrap test would estimate the parameters of the distribution the data is assumed to follow and generate datasets from that distribution using those estimates as the actual parameter values. A permutation test (like Fisher’s permutation test; see [4]) would use the original dataset values but randomly shuffle the labeles (stating which sample an observation belongs to) to generate new data sets and thus new simulated test statistics. p-values are essentially computed the same way.

Unlike Monte Carlo tests and MMC, these tests are not exact tests. That said, they often have good finite sample properties. (See [3].) See the documentation mentioned above for more details and references.

**Motivation**

Why write a package for these types of tests? This is not the only package that facilitates bootstrapping or Monte Carlo testing. The website RDocumentation includes documentation for the package **MChtest**, by Michael Fay which exists for Monte Carlo testing, too. The package **MaxMC** by Julien Neves is devoted to MMC specifically, as described by [2]. Then there’s the package **boot**, which is intended to facilitate bootstrapping. (If I’m missing anything, please let me know in the comments.)

**MChtest** is no longer on CRAN and implements a particular form of Monte Carlo testing and thus does not work for MMC. **MaxMC** appears to be in a very raw state. **boot** seems general enough that it could be used for bootstrap testing but still seems more geared towards constructing bootstrap confidence intervals and standard errors rather than hypothesis testing. All of these have a very different architecture from **MCHT**, which is primarily for creating a function like t.test() that performs a hypothesis test that was described when the function was created.

Additionally, this was good practice in practicing package development and more advanced R programming. This is the first time I made serious use of closures, S3 classes and R’s flavor of object-oriented programming, and environments. So far the result seems to be an flexible and robust tool for performing tests based on randomization.

**Getting Started**

Let’s start with a “Hello, world!”-esque example for a Monte Carlo test: a Monte Carlo version of the t-test.

The one-sample t-test, one of the oldest statistical tests used today, is used to test for the location of the population mean \mu. It decides between the set of hypotheses:

H_0: \mu = \mu_0

H_A: \mu \neq \mu_0

(The alternative could also be one-sided, perhaps instead stating \mu  \mu_0.) The t-test is an exact, most-powerful test for any sample size if the data generating process (DGP) that was used to produce the sample is a [Gaussian distribution](https://en.wikipedia.org/wiki/Normal_distribution). If we believe this assumption then the Monte Carlo version of the test is a contrived example as we could not do better than to use t.test(), but the moment we drop this assumption there is an opening for Monte Carlo testing to be useful.

Let’s load up the package.

library(MCHT)

## .------..------..------..------.

## |M.--. ||C.--. ||H.--. ||T.--. |

## | (\/) || :/\: || :/\: || :/\: |

## | :\/: || :\/: || (\_\_) || (\_\_) |

## | '--'M|| '--'C|| '--'H|| '--'T|

## `------'`------'`------'`------' v. 0.1.0

## Type citation("MCHT") for citing this R package in publications

The star function of the pacakge is the MCHTest() function.

args(MCHTest)

## function (test\_stat, stat\_gen, rand\_gen = function(n) {

## stats::runif(n)

## }, N = 10000, seed = NULL, memoise\_sample = TRUE, pval\_func = MCHT::pval,

## method = "Monte Carlo Test", test\_params = NULL, fixed\_params = NULL,

## nuisance\_params = NULL, optim\_control = NULL, tiebreaking = FALSE,

## lock\_alternative = TRUE, threshold\_pval = 1, suppress\_threshold\_warning = FALSE,

## localize\_functions = FALSE, imported\_objects = NULL)

## NULL

The documentation for this function is the majority of the manual and I’ve written multiple examples demonstrating its use. In short, a single call to MCHTest() will create an MCHTest-S3-class object (which is just a function) that can be use for hypothesis testing. Three arguments (all of which are functions) passed to the call will characterize the resulting test:

* test\_stat: A function with an argument x that computes the test statistic, with x being the argument that accepts the dataset from which to compute the test statistic.
* rand\_gen: A function generating random datasets, and must have either an argument x that would accept the original dataset or an argument n that represents the size of the dataset.
* stat\_gen: A function with an argument x that will take the random numbers generated by rand\_gen and turn them into a simulated test statistic. Sometimes stat\_gen is the same as test\_stat, but it is better to write separate functions, as will be seen later.

The functions passed to these arguments can accept other parameters, particularly parameters describing test parameters (that is, the parameter values we are testing, such as the population mean \mu), fixed parameters (parameter values the test assumes, like \sigma, the population standard deviation, whose value is assumed by the z-test often taught in introductory statistics courses;), and nuisance parameters (parameter values we don’t know, are not directly investigating, and may be needed to know the distribution of the test statistic). For the cases mentioned above, there are MCHTest() parameters that can be used for recognizing them: test\_params, fixed\_params, and nuisance\_params, respectively. While one could in principle ignore these parameters and pass functions to test\_stat, stat\_gen, and rand\_gen that use them anyway, I would recommend not doing so. First, there’s no guarantee that MCHTest-class objects would handle the extra parameters correctly. Second, when MCHTest() is made aware of these special cases, it can check that the functions passed to test\_stat, stat\_gen, and rand\_gen handle these types of parameters correctly and will throw an error when it appears they do not. This safety measure helps you use **MCHT** correctly.

Carrying on, let’s create our first MCHTest object for a t-test.

ts <- function(x) {

sqrt(length(x)) \* mean(x)/sd(x)

}

mc.t.test.1 <- MCHTest(ts, ts, rnorm, N = 10000, seed = 123)

Above, both test\_stat and stat\_gen are ts() (they're the first and second arguments, respectively) and the random number generator rand\_gen is rnorm(). Two other parameters are:

* N: the number of simulated test statistics to generate.
* seed: the seed of the random number generator, which makes test results consistent and reproducible.

MCHTest-class objects have a print() method that summarize how the object was defined. We see it in action here:

mc.t.test.1

##

## Details for Monte Carlo Test

##

## Seed: 123

## Replications: 10000

##

## Memoisation enabled

## Argument "alternative" is locked

This will tell us the seed being used and the number of replicates used for hypothesis testing, along with other messages. I want to draw attention to the message Argument "alternative" is locked. This means that the test we just created will ignore anything passed to the parameter alternative (similar to the parameter of the same name t.test() has). We can enable that parameter by setting the MCHTest() parameter lock\_alternative to FALSE.

(mc.t.test.1 <- MCHTest(ts, ts, rnorm, N = 10000, seed = 123,

lock\_alternative = FALSE))

##

## Details for Monte Carlo Test

##

## Seed: 123

## Replications: 10000

##

## Memoisation enabled

Let's now try this function out on data.

dat <- c(0.27, 0.04, 1.37, 0.23, 0.34, 1.44, 0.34, 4.05, 1.59, 1.54)

mc.t.test.1(dat)

##

## Monte Carlo Test

##

## data: dat

## S = 2.9445, p-value = 0.0072

If you run the above code you may see a complaint about %dopar% being run sequentially. This complaint appears when we don't register CPU cores for parallelization. **MCHT** uses **foreach**, **doParallel**, and **doRNG** to parallelize simulations and thus hopefully speed them up. Simulations can take a long time and parallelization can help make the process faster. If we were to continue we would not see the complaint again; R accepts that there's only one core visible and thus doesn't parallelize. But we can register the other cores on our system with the following:

library(doParallel)

registerDoParallel(detectCores())

Not only do we have parallelization enabled, MCHTest() automatically enables [memoization](https://en.wikipedia.org/wiki/Memoization) so that it doesn't redo simulations if the data (or at least the data's sample size) hasn't changed. (This can be turned off by setting the MCHTest() parameter memoise-sample to FALSE.) Again, this is so that we save time and don't have to fear repeat usage of our MCHTest-class function.

The above test effectively checked whether the population mean was zero against the alternative that the population mean is greater than zero (due to the default behaviour when alternative is not specified). By changing the alternative parameter we can test against other alternative hypotheses.

mc.t.test.1(dat, alternative = "less")

##

## Monte Carlo Test

##

## data: dat

## S = 2.9445, p-value = 0.9928

## alternative hypothesis: less

mc.t.test.1(dat, alternative = "two.sided")

##

## Monte Carlo Test

##

## data: dat

## S = 2.9445, p-value = 0.0144

## alternative hypothesis: two.sided

Compare this to t.test().

t.test(dat, alternative = "two.sided")

##

## One Sample t-test

##

## data: dat

## t = 2.9445, df = 9, p-value = 0.01637

## alternative hypothesis: true mean is not equal to 0

## 95 percent confidence interval:

## 0.2597649 1.9822351

## sample estimates:

## mean of x

## 1.121

The two tests reach similar conclusions.

However, t.test() is an exact and most-powerful test at any sample size under the assumptions we made. But all we need to do is not assume the data was drawn from a Gaussian distribution to throw the t-test for a loop. t-test will often do well even when the Gaussian assumption is violated but those statements hold for large sample sizes; at no sample size will the test be an exact test. Monte Carlo tests, though, can be exact tests for any sample size under different (often strong) distributional assumptions, without having to compute the distribution of the test statistic under the null hypothesis.

I know for a fact that dat was generated using an exponential distribution, so let's write a new version of the t-test that uses this information. While we're at it, let's add a parameter so that we know we're teseting for the mean of the data and that mean can be specified by the user.

ts <- function(x, mu = 1) {

# Throw an error if mu is not positive; exponential random variables have only

# positive mu

if (mu <= 0) stop("mu must be positive")

sqrt(length(x)) \* (mean(x) - mu)/sd(x)

}

sg <- function(x, mu = 1) {

x <- mu \* x

sqrt(length(x)) \* (mean(x) - mu)/sd(x)

}

(mc.t.test.2 <- MCHTest(ts, sg, rexp, seed = 123,

method = "One-Sample Monte Carlo Exponential t-Test",

test\_params = "mu", lock\_alternative = FALSE))

##

## Details for One-Sample Monte Carlo Exponential t-Test

##

## Seed: 123

## Replications: 10000

## Tested Parameters: mu

## Default mu: 1

##

## Memoisation enabled

Using this new function works the same, only now we can specify the \muwe want to test.

mc.t.test.2(dat, mu = 2, alternative = "two.sided")

##

## One-Sample Monte Carlo Exponential t-Test

##

## data: dat

## S = -2.3088, p-value = 0.181

## alternative hypothesis: true mu is not equal to 2

mc.t.test.2(dat, mu = 1, alternative = "two.sided")

##

## One-Sample Monte Carlo Exponential t-Test

##

## data: dat

## S = 0.31782, p-value = 0.6888

## alternative hypothesis: true mu is not equal to 1

t.test(dat, mu = 1, alternative = "two.sided")

##

## One Sample t-test

##

## data: dat

## t = 0.31782, df = 9, p-value = 0.7579

## alternative hypothesis: true mean is not equal to 1

## 95 percent confidence interval:

## 0.2597649 1.9822351

## sample estimates:

## mean of x

## 1.121

Now the t-test and the Monte Carlo test produce p-values that are not similar, and the Monte Carlo t-test will in general be more accurate. (It appears that the regular t-test is more conservative than the Monte Carlo test and thus is less powerful.)

**Conclusion**

I would consider the current release of **MCHT** to be early beta; it is usable but it's not yet able to be considered "stable". Keep that in mind if you plan to use it.

I'm very excited about this package and look forward to writing more about it. Stay tuned for future blog posts explaining its functionality. It's highly likely it has strange and mysterious behavior so I hope that if anyone encounters strange behavior, they report it and help push **MCHT** closer to a "stable" state.