Lab 5

Creating and Using Our Own Statistical Functions
Psychology 310

Instructions. Work through the lab, saving the output as you go. If you work in Microsoft Word, you can easily copy any graph to Word via the clipboard. Numerical output may also be copied easily by highlighting, moving it to the clipboard, then copying into Word. However, you should format R output in TrueType Courier New font so that it is *monospaced.* Your output file should be named `LAST_FIRST_LAB5.DOC`, where `LAST` is your last name, and `FIRST` is your first name. Any additional files should have the same naming scheme, except the file extension should be correct. You may add any description text you wish after `LAB5` in the file name.

Preamble. Today's assignment involves creating and using your own statistical functions.

1 Introduction

As I've tried to impress on you throughout the course, the face of statistical practice has changed very rapidly in the last decade, and promises to continue changing even more rapidly. The presence of R on the scene offers you some tremendous opportunities to save time and effort, beginning with Psychology 310 itself. R has a rather steep learning curve at first, but it offers several huge advantages for people just starting graduate school.

First of all, it is free, and will continue to be free. Nobody will ever take it away from you, charge you an "upgrade fee" for a product that has actually gotten worse, or terminate your license in the middle of an August night.

Second, it is constantly being worked on by experts. Problems, once discovered, can be fixed in a few days. Commercial statistical software producers simply cannot react that quickly to errors in their products.

Third, you can personalize it. Once you discover how to do things that *you need to do*, you can construct your own library that will continue to do things *just that way.*

2 Statistical Functions

Many complex statistical functions are already implemented in R. They may or may not be implemented in precisely the form that you need (or want) them. In developing a statistical function, you need to make decisions in three fundamental categories:

1. What you will be given in terms of

(a) Data, its form and delivery pattern. Will the data be in raw form or in summary form?

(b) Subclass of problem. For example, is it a one-sided test or a two-sided test? If it is one sided, what direction?

2. How to process the problem, given the input. (In commercial software, you also need to incorporate elaborate protections against erroneous or impossible input.)

3. How to deliver the output. Should it be bare-bones, or should it have a nice tabular format with headings, etc.?

In some situations (such as this course) you may have these decisions made for you. A course instructor might demand a function that requires input in a particular format, for example. In others, you'll make the decisions yourself. Of course, in the real world, your decisions may evolve over time, and your personal library functions will evolve right along with them.

3 A Simple Example – the 1-Sample t Test

Let's take a really simple example, the 1-sample t test. The test statistic, as you know, is

$$t_{n-1} = \frac{\overline{X}_\bullet - \mu_0}{S/\sqrt{n}} \tag{1}$$

where n is the sample size, $\overline{X}_\bullet$ is the sample mean, S is the sample standard deviation, and μ_0 is the null-hypothesized value of the population mean.

This statistic is referred to the Student t distribution for its critical values. This distribution, of course, is built into R.

Suppose we decide to write our own 1-sample t function. Processing a few examples in class involving the 1-sample t-test, we realize that there are two fundamental classes of hypothesis (1-tailed and 2-tailed) and two fundamental types of data, raw (you are given a list of numbers for the sample) and aggregated (summary), in which you are given the mean and either the standard deviation or variance.

Here is a bare-bones function that processes only raw data. It returns the t statistic and the degrees of freedom in a vector. It is up to you to decide what to do from there.

```
> t.from.raw <- function(data,mu_0)
+ {
+ ## compute n and df
+ n <- length(data)
+ df <- n - 1
+ ## compute t statistic
+ t <- sqrt(n)*(mean(data) - mu_0)/sd(data)
```

```
+ return(c(t,df))
+ }
```

Once you've defined such a function, you can try it out on some data. For example, suppose you were testing the hypothesis that $\mu = 100$, and you had the data 109,95,113,98,121,94,123,112. The t-statistic is

```
> data <- c(109,95,113,98,121,94,123,112)
> results <- t.from.raw(data,100)
> results
[1] 2.025747 7.000000
```

This function has several shortcomings. Once you get the t-statistic, you have to decide whether it is significant, and at what level. It might be easier, in the long run, to have the program do that for you. One compromise would be to have the program return a p-value. A p-value is the α at which the present data would have barely rejected, given the specific type of test. The same t-statistic value calculated from your data will have different p-values, depending on whether the null hypothesis is one-sided or two-sided, and whether, if the hypothesis is one-sided, the result is in a direction opposite the null hypothesized region. Someone examining a p-value can immediately tell if the null hypothesis is rejected with a Type I error rate of α. The rule is: if the p-value is less than α, reject, otherwise, do not reject.

Here is an example: With the above data, we had a t-statistic of 2.0257 based on 7 degrees of freedom. This value of t is at the following quantile point of the t distribution:

```
> pt(results[1],results[2])
[1] 0.9587858
```

This is the probability of getting a value *less than or equal to* the observed value of t. The probability of getting a value greater than the observed value is therefore

```
> 1 - pt(results[1],results[2])
[1] 0.04121424
```

What is the p-value? Well, we have to remember that, if the test is two-sided with $\alpha = .05$, the rejection points are placed so that there is a probability of $\alpha/2 = .025$ outside them. So if the test is two-sided, you need to take the probability of exceeding the observed t and *double it* in order to ascertain the value of α at which the test would have barely rejected. In other words, if the null hypothesis was that $\mu = 100$, then the p-value is

```
> 2*(1 - pt(results[1],results[2]))
[1] 0.08242848
```

What if our t value had been -2.0257? Since there is a symmetric rejection area on the negative side of 100, the p-value could be reported as

```
> 2*(pt(-results[1],results[2]))
[1] 0.08242848
```

but this is equivalent to

```
> 2*(1-pt(results[1],results[2]))
[1] 0.08242848
```

In other words, if the hypothesis is two-sided, you can apply one rule of computation to the *absolute value* of the t-statistic.

Since the p-value is not less than .05, the hypothesis would not be rejected with $\alpha = .05$. On the other hand, because it is less than .10, the null hypothesis would be rejected with $\alpha = .10$.

What if the null hypothesis had been that $\mu \leq 100$ and the t-statistic was 2.0257? In that case, all the α would be in the upper tail, and so there would be no need to double the probability of the observed t. In that case, the "one-tailed p-value" would be reported as

```
> 1 - pt(results[1],results[2])
[1] 0.04121424
```

What if the null hypothesis had been that $\mu \geq 100$ and the t-statistic was 2.0257? In this case, the t value is high, but in the wrong direction. The p-value would be reported as

```
> pt(results[1],results[2])
[1] 0.9587858
```

Since this value is much higher than .05, we would, of course, not reject the null hypothesis.

Taking all the above into consideration, what are "the rules" for computing p-values when you observe a value of `t.obs` with degrees of freedom equal to `df`? Here they are.

1. If test is 2-sided, compute `2*(1 - pt(abs(t.obs,df)))`
2. If test is 1-sided
   (a) If the null hypothesis is $\mu \leq \mu_0$, compute `1-pt(t.obs,df)`
   (b) If the null hypothesis is $\mu \geq \mu_0$, compute `pt(t.obs,df)`

Now that we know what to do, we need to upgrade our function. Here is an example:

```
> t.from.raw.2 <- function(data,mu_0,null.hypothesis="equals")
+ {
+ ## compute n and df
+ n <- length(data)
+ df <- n - 1
+ ## compute t statistic
+ t <- sqrt(n)*(mean(data) - mu_0)/sd(data)
+ ## compute appropriate p-value
+ if(null.hypothesis == "equals")p.value <- 2*(1-pt(abs(t),df))
+ if(null.hypothesis == "less")p.value <- 1-pt(t,df)
+ if(null.hypothesis == "greater")p.value <- pt(t,df)
+ return(c(t,df,p.value))
+ }
```

Let's try it out. First, we test the hypothesis that $\mu = 100$

```
> t.from.raw.2(data,100,"equals")
[1] 2.02574659 7.00000000 0.08242848
```

The p-value exceeds .05, so we would not reject the null hypothesis.
Next, we test the hypothesis that $mu \leq 100$.

```
> t.from.raw.2(data,100,"less")
[1] 2.02574659 7.00000000 0.04121424
```

Now, the p-value is less than .05, and we do reject the null hypothesis.
On the other hand, if our hypothesis had been that $mu \geq 100$, the t-statistic would be "on the wrong side" of 100, and we would get a very high p-value, indicating non-rejection.

```
> t.from.raw.2(data,100,"greater")
[1] 2.0257466 7.0000000 0.9587858
```

We could make this function more elaborate in several ways. One way would be to have the function label all its output. This might reduce the probability of some kind of error. We could also have the function tell us whether the null is rejected (but then we would have to provide a value of α). I'll specify a default value of .05.

```
> t.from.raw.3 <- function(data,mu_0,alpha = .05,
+ null.hypothesis="equals",digits=4)
+ {
+ ## compute n and df
+ n <- length(data)
+ df <- n - 1
+ ## compute t statistic
```

```
+ t <- sqrt(n)*(mean(data) - mu_0)/sd(data)
+ ## compute appropriate p-value
+ if(null.hypothesis == "equals")p.value <- 2*(1-pt(abs(t),df))
+ if(null.hypothesis == "less")p.value <- 1-pt(t,df)
+ if(null.hypothesis == "greater")p.value <- pt(t,df)
+ if(p.value < alpha)reject=1 else reject=0
+ results <- list(t.observed = round(t,digits),
+    df=as.integer(df),p.value = round(p.value,digits),
+    alpha = round(alpha,digits), reject=as.integer(reject))
+ return((results))
+ }
```

Notice that I also added a parameter to allow you to specify the number of rounded decimal places in the output.

Let's try it:

```
> t.from.raw.3(data,100)

$t.observed
[1] 2.0257

$df
[1] 7

$p.value
[1] 0.0824

$alpha
[1] 0.05

$reject
[1] 0
```

For more compact output, you can use

```
> unlist(t.from.raw.3(data,100,digits=3))

t.observed         df    p.value      alpha     reject
     2.026      7.000      0.082      0.050      0.000
```

As a general-purpose function, `t.from.raw.3` has a significant problem — it requires raw data. What we want is a capability to can handle either raw data *or* data input as means and standard deviations, or data input as means and variances! Keep in mind that we might want to expand this t statistic to a more general statistic later.

Here is one approach to modifying our `t.from.raw.3` function.

```
> t.one.sample <- function(data,mu_0,data.type="raw",
+ alpha = .05,null.hypothesis="equals",digits=4)
+ {
+
+ if(data.type=="raw")
+ {
+  n <- length(data)
+  xbar <- mean(data)
+  S <- sd(data)
+ }
+ if(data.type=="mean.sd")
+ {
+ xbar<-data[1]
+ S <- data[2]
+ n <- data[3]
+ }
+ if(data.type=="mean.var")
+ {
+ xbar <- data[1]
+ S <- sqrt(data[2])
+ n <- data[3]
+ }
+ ## compute df
+ df <- n-1
+ ## compute t statistic
+ t <- sqrt(n)*(xbar - mu_0)/S
+ ## compute appropriate p-value
+ if(null.hypothesis == "equals")p.value <- 2*(1-pt(abs(t),df))
+ if(null.hypothesis == "less")p.value <- 1-pt(t,df)
+ if(null.hypothesis == "greater")p.value <- pt(t,df)
+ if(p.value < alpha)reject=1 else reject=0
+ results <- list(t.observed = round(t,digits),
+ df=as.integer(df),p.value = round(p.value,digits),
+ alpha = round(alpha,digits), reject=as.integer(reject))
+ return((results))
+ }
```

Let's try it.

```
> t.one.sample(data,100)

$t.observed
[1] 2.0257

$df
[1] 7
```

```
$p.value
[1] 0.0824

$alpha
[1] 0.05

$reject
[1] 0

> t.one.sample(c(108,12.4,32),100,data.type = "mean.sd")

$t.observed
[1] 3.6496

$df
[1] 31

$p.value
[1] 0.001

$alpha
[1] 0.05

$reject
[1] 1
```

So now you have an all-purpose 1-sample t-test function.

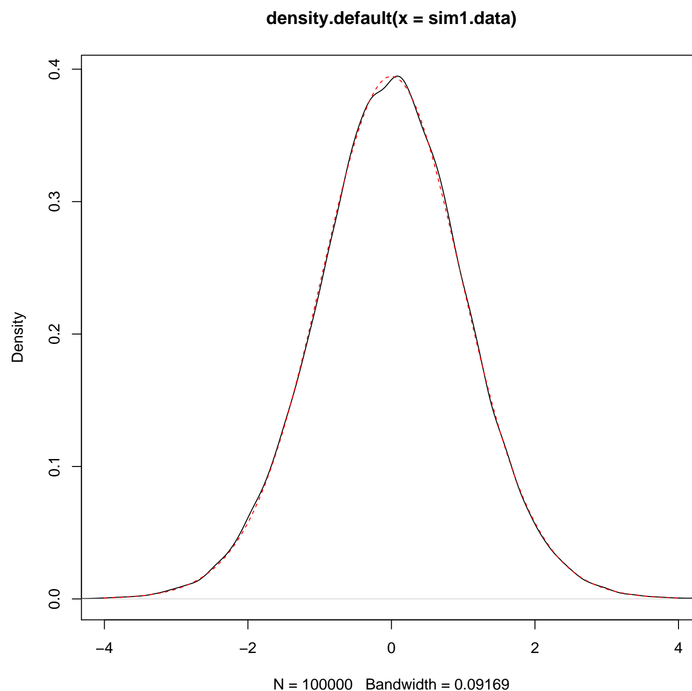
Of course, if you are doing a large simulation and speed is of the essence, you might want to employ `t.from.raw` and avoid all the frills.

4 Simulating t-Test Performance

All statistics have assumptions, and the 1-sample t-test is no exception. The key assumptions are independence and normality. As I mentioned in class, the test can go completely haywire when observations are not independent. However, a standard view in many textbooks is that the test is robust to violation of the normality assumption, because the central limit theorem guarantees that in most practical circumstances, as n becomes large, the distribution of the sample mean becomes increasingly normal.

Let's examine the effect of violating the normality assumption by using our `t.from.raw()` function to examine the performance of the t-statistic. Let's start by examining the performance of the statistic when $n = 25$, and the population is normal. We'll do 100,000 simulations, testing the hypothesis that $\mu = 0$, when the population mean is 0 and the standard deviation is 1. Below I perform the simulation, adding the theoretical density curve in dotted red. By the way, the mean of a Student's t variable is 0, and the standard deviation with degrees of freedom equal to ν is $\sqrt{\nu/(\nu-2)}$.

```
> set.seed(12345)
> sim1.data <- replicate(100000,t.from.raw(rnorm(25),0)[1])
> mean(sim1.data)
[1] -0.001993339
> sd(sim1.data)
[1] 1.045006
> plot.density(density(sim1.data),xlim=c(-4,4))
> curve(dt(x,24),add=T,col="red",lty=2)
```



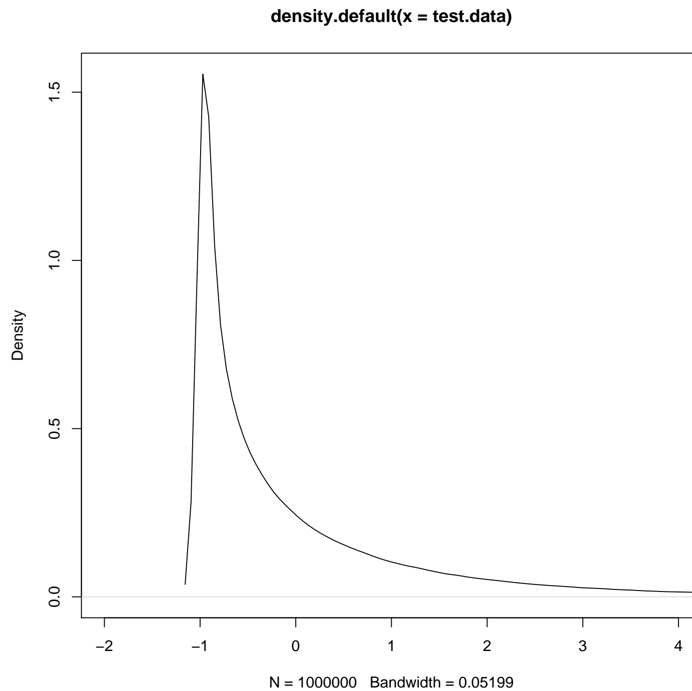
Of course, when the population is normal, and the observations independent, the t statistic behaves exactly as it should.

Now suppose we use a the square of a standard normal random variable to generate our population. The square of a standard normal variable Z has a chi-square distribution with one degree of freedom. This distribution is highly skewed, as we can see from the plot below. It has a mean of 1 and a variance of 2, so if we subtract 1 from every value of Z^2, we should get a random variable with a mean of 0, a variance of 2, and very substantial skew.

Let's simulate 1,000,000 draws from this distribution, and plot the shape.

```
> set.seed(12345)
> test.data <- rnorm(1000000)^2 -1
> mean(test.data)
```

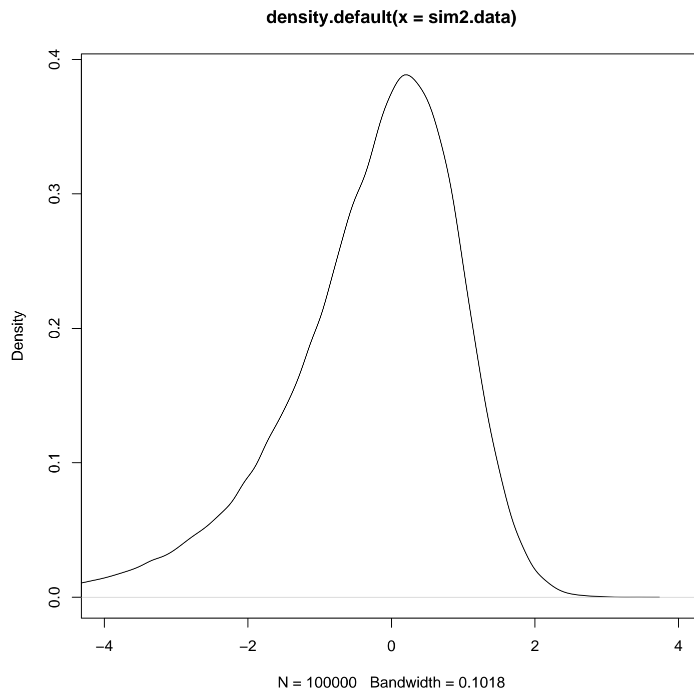
```
[1] 0.002630861
> var(test.data)
[1] 2.008753
> plot.density(density(test.data),xlim=c(-2,4))
```



Clearly, this distribution is *very* non-normal in the sense of being highly skewed. We can use simulation to see how the t test performs when the population is shaped like this.

Let's simulate 100,000 tests.

```
> set.seed(12345)
> sim2.data <- replicate(100000,t.from.raw(rnorm(25)^2-1,0)[1])
> mean(sim2.data)
[1] -0.3325753
> var(sim2.data)
[1] 1.744434
> plot.density(density(sim2.data),xlim=c(-4,4))
```



As you can see, the distribution is far from symmetric. What is the rejection probability when $\alpha = .05$, and the test is 2-sided? We can use a trick to calculate this. A rejection occurs if the observed value of the t-statistic exceeds the 97.5 percentile point in absolute value. So we can use a line of code to compute the "empirical rejection probability."

```
> total.rejection.probability <- mean(abs(sim2.data)>qt(.975,24))
> upper.rejection.probability <- mean(sim2.data>qt(.975,24))
> lower.rejection.probability <- mean(sim2.data < qt(.025,24))
> total.rejection.probability

[1] 0.09726

> upper.rejection.probability

[1] 0.00386

> lower.rejection.probability

[1] 0.0934
```

How did that work? Look at the first line carefully. The expression `abs(sim2.data)>qt(.975,24)` is a logical expression. If TRUE, it takes on the value 1, otherwise the value 0. So the mean of the expression, applied to all the data, is simply the proportion of times the expression was true, i.e., the proportion of rejections.

As you can see, the total rejection probability is nearly twice what it should be. Moreover, the rejections are very unbalanced. The vast majority occur on the low side.

For you to answer. How much will things improve with sample size of 50? What does the distribution of the t statistic look like? What is the empirical rejection rate? With $n = 50$, what does the distribution of the sample mean look like?

For you to answer. About how large does the sample size have to be before the t-statistic has an overall rejection rate within .01 of the nominal value? What does the rejection balance look like?

For you to answer. About how large does the sample size have to be before the t statistic has an overall rejection rate with .01 of the nominal value, *and* both the upper and lower rejection probabilities are greater than .01 and less than .03?

For you to answer. Compare and contrast the shape of the distribution of the sample mean at $n = 50$ with the distribution of the $t - statistic$.

For you to answer. The t-distribution assumes that the sample mean and variance are independent. When the population distribution is normal, the mean and variance are independent. Check the correlation between the sample mean and sample variance across 100,000 samples of size 50 from a normal distribution, using a starting seed of 12345. Remember, if you make two runs (one to get the variances, one to get the means), reset the seed! Otherwise you will be calculating means and variances from different data. What is the correlation between the means and variances? Create a scatterplot. What does it look like? Now, do the same thing for 100,000 samples of size 50 from a chi-square distribution. What is the correlation between the sample mean and sample variance. What does the scatterplot look like?

5 Creating a Generalized t-Statistic Routine

With minimal help from me, I want you to create your own generalized t routine for use with independent samples. It can be a bare-bones routine that works on aggregated data (i.e., means, standard deviations, and sample sizes for each group).

The opening and closing lines of your function should read like this.

```
GeneralizedT<-function(means,sds,ns,wts,k0=0)
{

your function here....

return(c(t,df))
}
```

Your input to the function will be vectors of means, sds, ns, and the linear weights that define the linear combination. k0 is the constant that the linear

combination is hypothesized to be equal to. The default is 0.

For example, suppose you have two groups, and the means are 20 and 10, the sds are 12 and 14, and the sample sizes are 20 and 30. Compute the two sample test statistic that the mean difference is zero.

Here is what your input and output should look like

```
> GeneralizedT(c(20,10),c(12,14),c(20,30),c(1,-1))
[1]  2.615503 48.000000
```