

STAT4CI3/6CI3 Computational Methods for Inference

Assignment 1 Solutions

R code for this solution in a plain text file is also available separately

Note that, Questions 1-4, the first number in the marks is for STATS 4CI students (25 marks per question) whereas the second is for STATS 6CI students (20 marks per question). Question 5 is only for STATS 6CI students so there is only one mark out of a total of 20. _____

- Q. 1** a) Using the formulae given for the various cases we get the following function. There was one situation not covered in the question which is an infinite sum with $r \leq -1$ which is undefined since the sum oscillates between increasing positive and negative numbers without bound. Since I didn't mention it, no marks will be deducted for omitting that part but I do include it in my solution for completeness

```
sum.geom <- function(x,r,n) {  
  #  
  # Function to give finite or infinite sums of  
  # a geometric series.  
  # Arguments:  
  # x: the first term in the series  
  # r: the ratio between successive terms  
  # n: the number of terms can be a positive integer for  
  #     finite sums or Inf for infinite sums.  
  #  
  if (n<=0) stop("n must be postive")  
  if (x==0) return(0)  
  if (r==0) return(x)  
  if (is.finite(n)) {  
    if (n!=floor(n)) {  
      n <- floor(n)  
      warning(paste("Non-integer n rounded down to",n))  
    }  
    if (r != 1) return(x*(1-r^n)/(1-r))  
    else return(n*x)  
  }  
  else {  
    if (r >= 1)  
      return(Inf)  
    else if (r <= -1)  
      return(NaN)  
    else  
      return(x/(1-r))  
  }  
}
```

[15/10 marks]

- b) As explained in the question, a recursive function is one that calls itself unless some condition is met. Such a function must have an `if` statement to end recursion. It is not always very efficient but can be useful in some situations.

For the choose function we will stop recursion if $n = r$ or if $r = 0$, both of which cases give us a value of 1. Otherwise we call the function twice for the two parts on the right-hand side of the equation in the question.

```
choose.rec <- function(n,r) {  
  #  
  # Recursive function to calculate the number of ways to  
  # choose r objects from n  
  #  
    if (n != floor(n)) {  
      warning("non-integer n rounded down")  
      n <- floor(n)  
    }  
    if (r != floor(r)) {  
      warning("non-integer r rounded down")  
      r <- floor(r)  
    }  
    if (n==0) stop("n must be greater than 0")  
    if (n<r) stop("n must be greater than or equal to r")  
    if ((n==r)|| (r==0)) return(1)  
    else  
      choose.rec(n-1,r-1)+choose.rec(n-1,r)  
}
```

[10/10 marks]

- Q. 2 a) Here is my code for the pooled t -test of equality of means. For full marks your code needs to consider the three possible alternatives, allow for different size of test, print out a summary of the test and return some of the important aspects of the test.

```
pooled.t.test <- function(x,y, alpha=0.05, alternative="two-sided") {  
  # Function to do a pooled t-test of equality of means based  
  # on two independent samples  
  # Arguments:  
  # x,y - two numeric vectors corresponding to the 2 samples  
  # alpha - significance level (default 0.05)  
  # alternative - alternative hypothesis, default is "two-sided"  
  # other possible values are "less" and "greater"  
  n1 <- length(x)  
  xbar <- mean(x)  
  s2x <- var(x)  
  n2 <- length(y)  
  ybar <- mean(y)  
  s2y <- var(y)  
  # Calculate the pooled estimate of the common variance and test statistic  
  s2p <- ((n1-1)*s2x+(n2-1)*s2y)/(n1+n2-2)  
  stat <- (xbar-ybar)/sqrt(s2p*(1/n1+1/n2))  
  # Next find the p-value (which depends on the alternative)  
  if (alternative=="two-sided")  
    pv <- 2*pt(abs(stat), n1+n2-2, lower=FALSE)  
  else if (alternative=="less") pv <- pt(stat, n1+n2-2)  
    else if (alternative=="greater") pv <- pt(stat, n1+n2-2, lower=FALSE)  
  else pv <- NA  
  reject <- pv<alpha  
  # Print out some results  
  cat(paste("Test Statistic =", round(stat,4), "\n"))  
  if (!is.na(pv)) {  
    if (alternative=="two-sided")  
      cat("Alternative hypothesis: mu(X) does not equal mu(Y)\n")  
    else if (alternative=="greater")  
      cat("Alternative hypothesis: mu(X) is greater than mu(Y)\n")  
    else cat("Alternative hypothesis: mu(X) is greater than mu(Y)\n")  
    cat(paste("P-value =", format(pv,digits=4), "\n"))  
    cat(paste(ifelse(reject, "Reject", "Do not reject"),  
      "the null hypothesis at the alpha =",alpha,"level\n"))  
  }  
  else cat(paste("Cannot calculate p-value, alternative was",  
    alternative, "but should be \"two-sided\",  
    \"less\" or \"greater\"\n"))  
  # Return results invisibly because we have already printed out a summary.  
  output <- list(means=c(xbar,ybar), vars=c(s2x,s2y),  
    pooled.s2=s2p, stat=stat, pv=pv)  
  invisible(output)  
}
```

[10/8 marks]

Now we test the code with the given data

```
> x <- c(6.35, 3.47, 6.54, 6.21, 8.72, 10.43)
> y <- c(8.99, 10.94, 9.88, 11.50, 9.90, 11.89, 12.10, 9.33)
> pooled.t.test(x,y)
Test Statistic = -3.7362
Alternative hypothesis: mu(X) does not equal mu(Y)
P-value = 0.002842
Reject the null hypothesis at the alpha = 0.05 level
```

[2/2 marks]

- b) To do forward selection we need to loop over the possible models at each stage of the analysis. There are a number of ways to do this and all are equally acceptable. I find it easiest to deal with the first stage separately and then loop over subsequent stages but that is not necessary. In my output I returned a summary of the variables added at each stage plus a list of all the results at each stage. You will get full marks as long as you return something along the lines of one of these as well as the final fitted model from `lm`. Here is my code:

```
forward.select <- function(y, X, alpha=0.05) {
  # Function to do forward selection
  # Arguments:
  #   y - a quantitative response vector
  #   X - a matrix (or data.frame) of covariates
  #   alpha - a maximum significance level for inclusion

  Stage <- 1
  p <- ncol(X)
  Vars <- 1:p
  if (is.null(names(X)))
    names <- paste("X",1:p,sep="")
  else
    names <- names(X)
  names.all <- names
  Data.model <- data.frame(y=y)
  Results <- list(Stage1=matrix(NA, ncol=5, nrow=p))
  # The first Stage; Examine each column of X
  for (i in 1:p){
    fit <- coef(summary(lm(y~X[,i])))
    Results$Stage1[i,] <- c(i,fit[2,])
  }
  # Sort the Results by p-value
  Results$Stage1 <- Results$Stage1[order(Results$Stage1[,5]),]
  if (Results$Stage1[1,5]>=alpha)
    return(model=lm(y~1), Results=Results$Stage1)
  # Now the first is the added variable provided we did
  # not exit the function at the last line.
  v.add <- Results$Stage1[1,1]
  k <- 1
```

```

continue <- T
# Now we use a while loop for the remaining stages
while(continue){
  # The current data for the model is set up
  # Also we update the potential variables to be added
  names.mod <- c(names(Data.model), names.all[v.add])
  Data.model <- cbind(Data.model,X[,v.add])
  names(Data.model) <- names.mod
  names <- names[Vars!=v.add]
  Vars <- Vars[Vars!=v.add]
  k <- k+1 # The Stage Number
  j <- 0 # A counter for the variables tried at this stage
  Stage <- paste("Stage",k,sep="")
  Results[[Stage]] <- matrix(NA, ncol=5,
                                nrow=length(Vars))

  for (i in Vars){
    # Loop over the potential variables
    j <- j+1
    Data.fit <- cbind(Data.model,X[,i])
    fit <- coef(summary(lm(y~., data=Data.fit)))
    Results[[Stage]][j,] <- c(i,fit[k+1,])
  }
  # Order the results for this stage and select
  # the variable with the smallest p-value
  Results[[Stage]] <- Results[[Stage]][order(Results[[Stage]][,5]),]
  v.add <- Results[[Stage]][1,1]
  # Decide if any variable will be added
  continue <- Results[[Stage]][1,5] < alpha
}
# Now summarize all of the results
Res <- t(sapply(Results, function(x) x[1,])[,,-k])
v.add <- names.all[Res[,1]]
Summary <- data.frame(v.add, Res)
names(Summary) <- c("Variable", "Index", "Estimate",
                    "Std Error", "t stat", "p-value")
row.names(Summary) <- paste("Stage",1:nrow(Res))
# Get the final fitted model and print it out.
fit <- lm(y~., data=Data.model)
cat("Final Fitted Model is:\n")
print(fit)
# Return the model summary of stages and full results.
invisible(list(model=fit, Summary=Summary,
               Results=Results))
}

```

[10/8 marks]

When I fit this to the nuclear data I get the following

```
> nuclear.forward <- forward.select(nuclear$cost, nuclear[,-1])
```

Final Fitted Model is:

Call:

```
lm(formula = y ~ ., data = Data.model)
```

Coefficients:

(Intercept)	date	cap	pt	ne
-4756.2145	71.0193	0.4198	-128.9438	99.3988

A summary of the fitting is as follows:

```
> nuclear.forward$Summary
```

	Variable	Index	Estimate	Std Error	t stat
Stage 1	date	1	102.289283	24.2309152	4.221437
Stage 2	cap	4	0.413225	0.1076052	3.840195
Stage 3	pt	10	-162.763622	52.2900431	-3.112708
Stage 4	ne	6	99.398780	38.6361959	2.572685

p-value

Stage 1 0.0002070933

Stage 2 0.0006163248

Stage 3 0.0042425132

Stage 4 0.0159076839

[3/2 marks]

- Q. 3 a)** First we need to find the values of $f(x)$ and hence the cumulative distribution function $F(x)$.

From the question we see that the probability mass function is

x	1	2	3	4	5
$f(x)$	0.10	0.05	0.10	0.25	0.50

Thus the cumulative distribution function is

$$F(x) = \begin{cases} 0 & x < 1 \\ 0.10 & 1 \leq x < 2 \\ 0.15 & 2 \leq x < 3 \\ 0.25 & 3 \leq x < 4 \\ 0.50 & 4 \leq x < 5 \\ 1.00 & x \geq 5 \end{cases}$$

[5/4 marks]

Using the method described in class we get the following algorithm

1. Generate $U \sim \text{Uniform}(0, 1)$.
2. If $U \leq 0.1$ then return $X = 1$
 If $0.10 < U \leq 0.15$ then return $X = 2$
 If $0.15 < U \leq 0.25$ then return $X = 3$
 If $0.25 < U \leq 0.50$ then return $X = 4$
 If $U > 0.50$ then return $X = 5$.

I will accept this but it is actually more efficient to consider the reverse order of comparisons in Step 2. so we return faster as in

2. If $U > 0.5$ then return $X = 5$
 If $0.25 < U \leq 0.5$ then return $X = 4$
 If $0.15 < U \leq 0.25$ then return $X = 3$
 If $0.10 < U \leq 0.15$ then return $X = 2$
 If $U \leq 0.10$ then return $X = 1$.

[5/4 marks]

Applying this algorithm to the given uniform values we get

U	0.5197	0.1790	0.9994	0.6873	0.7294	0.5791	0.0361	0.2581	0.0026	0.8213
X	5	3	5	5	5	5	1	4	1	5

[2/2 marks]

- b)** The probability mass function for the sum of the numbers shown on a pair of independent dice is

x	2	3	4	5	6	7	8	9	10	11	12
$p(x)$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Hence the cumulative distribution function for this random variable is

$$F(x) = \begin{cases} 0 & x < 2 \\ \frac{1}{36} & 2 \leq x < 3 \\ \frac{3}{36} & 3 \leq x < 4 \\ \frac{6}{36} & 4 \leq x < 5 \\ \frac{10}{36} & 5 \leq x < 6 \\ \frac{15}{36} & 6 \leq x < 7 \\ \frac{21}{36} & 7 \leq x < 8 \\ \frac{26}{36} & 8 \leq x < 9 \\ \frac{30}{36} & 9 \leq x < 10 \\ \frac{33}{36} & 10 \leq x < 11 \\ \frac{35}{36} & 11 \leq x < 12 \\ 1 & x \geq 12 \end{cases}$$

[5/4 marks]

Hence we can generate a Uniform(0,1) random variable and compare it to this cumulative distribution function. Here is an efficient way to do this.

```
dice.gen <- function(n) {
  # Function to generate n random observations of the total in a
  # roll of two fair dice.
  u <- runif(n)
  x <- rep(7,n)
  x[u<=15/36] <- 6
  x[u<=10/36] <- 5
  x[u<=6/36] <- 4
  x[u<=3/36] <- 3
  x[u<=1/36] <- 2
  x[u>21/36] <- 8
  x[u>26/36] <- 9
  x[u>30/36] <- 10
  x[u>33/36] <- 11
  x[u>35/36] <- 12
  x
}
```

[5/4 marks]

In the following code I will generate 360,000 random numbers from using this function and compare the theoretical expected counts at each outcome with the observed counts. I have set the seed so that you can exactly reproduce this if desired.

```
> set.seed(1022019)
> dice.out <- dice.gen(360000)
> rbind(table(dice.out),10000*c(1:6,5:1))
```

	2	3	4	5	6	7	8	9	10	11	12
[1,]	10062	20030	30055	39863	50022	59825	50056	39991	29947	20115	10034
[2,]	10000	20000	30000	40000	50000	60000	50000	40000	30000	20000	10000

We see that the observed counts are generally very close to the expected counts. The biggest relative difference is only about 0.6% which is very close. **[3/2 marks]**

Q. 4 a) U_1 and U_2 are independent $\text{Unif}[0, 1]$ random variables.

$$Z_1 = \sqrt{-2 \ln U_1} \cos(2\pi U_2) \quad Z_2 = \sqrt{-2 \ln U_1} \sin(2\pi U_2)$$

So the inverse transformation is

$$\begin{aligned} Z_1^2 + Z_2^2 = -2 \ln U_1 &\Rightarrow U_1 = \exp \left\{ -\frac{1}{2}(Z_1^2 + Z_2^2) \right\} \\ \frac{Z_2}{Z_1} = \tan(2\pi U_2) &\Rightarrow U_2 = \frac{1}{2\pi} \tan^{-1} \left(\frac{Z_2}{Z_1} \right) \end{aligned}$$

[2/1 marks]

Hence the Jacobian of the transformation is

$$|J| = \left| \begin{array}{cc} -z_1 \exp \left\{ -\frac{1}{2}(z_1^2 + z_2^2) \right\} & -z_2 \exp \left\{ -\frac{1}{2}(z_1^2 + z_2^2) \right\} \\ \frac{-z_2/z_1^2}{\frac{1}{2\pi} \left(1 + \frac{z_2^2}{z_1^2} \right)} & \frac{1/z_1^2}{\frac{1}{2\pi} \left(1 + \frac{z_2^2}{z_1^2} \right)} \end{array} \right| = \frac{1}{2\pi} \exp \left\{ -\frac{1}{2}(z_1^2 + z_2^2) \right\}$$

[3/2 marks]

Now note that $\sqrt{-2 \ln U_i} \in (0, \infty)$ which in each case is multiplied by something which lies in the interval $[-1, 1]$ so Z_1 and Z_2 can both take on any real values. [3/2 marks]

Hence we have

$$f_{Z_1, Z_2}(z_1, z_2) = \frac{1}{2\pi} \exp \left\{ -\frac{1}{2}(z_1^2 + z_2^2) \right\} = \left(\frac{1}{\sqrt{2\pi}} e^{-z_1^2/2} \right) \left(\frac{1}{\sqrt{2\pi}} e^{-z_2^2/2} \right)$$

So Z_1 and Z_2 are independent $N(0, 1)$ random variables.

[3/2 marks]

Here is one possible implementation in R.

```
box.muller <- function(n, mu=0, sigma=1) {
  # Implementation of the Box-Muller algorithm to
  # generate normal random variates.
  # Arguments:
  #   n - required sample size
  #   mu - mean (default 0)
  #   sigma - the standard deviation (default 1)

  if ((n>0) && (n%%2==0)) m <- n/2
  else if ((n>0) && (n%%2==1)) m <- (n+1)/2
  else stop("n must be a positive integer")
  U1 <- runif(m)
  U2 <- runif(m)
  temp <- sqrt(-2*log(U1))
  Y1 <- temp*sin(2*pi*U2)
  Y2 <- temp*cos(2*pi*U2)
  mu + sigma*c(Y1,Y2)[1:n]
}
```

[4/3 marks]

- b) The easiest way to do this is to explicitly find $P(Y = y)$ for any non-negative integer y . For such $y = 0, 1, \dots$ we have

$$\begin{aligned}
 P(Y = y) &= P(\lfloor X \rfloor = y) \\
 &= P(y \leq X < y+1) \\
 &= \int_y^{y+1} \lambda e^{-\lambda x} dx \\
 &= -e^{-\lambda x} \Big|_{x=y}^{x=y+1} \\
 &= e^{-\lambda y} - e^{-\lambda(y+1)} \\
 &= (1 - e^{-\lambda}) e^{-\lambda y} \\
 &= p(1 - p)^y \quad \text{for } y = 0, 1, \dots
 \end{aligned}$$

which is the probability mass function of the geometric random variable with success probability $p = 1 - e^{-\lambda}$. [5/5 marks]

If we want to generate from a geometric distribution with given p then we need to find λ in terms of p .

$$p = 1 - e^{-\lambda} \Rightarrow \lambda = -\log(1 - p)$$

[1/1 marks]

We saw in class that we can generate from the exponential(λ) distribution by setting $X = -\log(U)/\lambda$ where $U \sim \text{Uniform}(0, 1)$. Hence we get the following algorithm to generate from the geometric(p) distribution:

1. Generate $U \sim \text{Uniform}(0, 1)$.
2. Set $X = \log(U)/\log(1 - p)$.
3. Return the integer part of X .

In R we get the following implementation of this algorithm

```

rand.geom <- function(n, p) {
#
# Function to generate a random sample from the geometric distribution.
# Arguments:
#   n: the required sample size
#   p: the success probability of the geometric distribution
#
  if ((p<=0) | (p>=1))
    stop("Success probability must be in the interval (0,1)")
  if (n<1)
    stop("n must be a positive integer")
  U <- runif(n)
  floor(log(U)/log(1-p))
}

```

[4/4 marks]

Q. 5 Only required and marked for STATS 6CI3 students but it is recommended that STATS 4CI3 students examine this solution also and ask me if there is anything you do not understand

- a) A χ_2^2 random variable is a gamma random variable with $\alpha = 1$ and $\beta = 2$. We recall that $\Gamma(1) = 1$ by definition and so the density of the χ_2^2 random variable is

$$f(x) = \begin{cases} \frac{1}{2}e^{-x/2} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

We recognize that this is simply an exponential distribution with mean equal to 2 and so we can generate such a random variable as

$$X = -2\log(U) \quad \text{where } U \sim \text{Uniform}(0, 1)$$

[3 marks]

For any even degrees of freedom r we then generate $r/2$ independent χ_2^2 random variables and sum them together

$$Y = -2 \sum_{k=1}^{r/2} \log(U_k)$$

and return Y . Here is code to do this

```
rchi.1 <- function(n, r) {
  # Function to generate chi-squared random variables with
  # even degrees of freedom.
  # The arguments are the required sample size and the
  # degrees of freedom.
  if (r%%2 != 0)
    stop("Only even degrees of freedom allowed")
  U <- matrix(runif(r/2*n), nrow=n)
  X <- -2*log(U)
  rowSums(X)
}
```

[4 marks]

- b) Here is the required algorithm to generate a χ_r^2 random variable

1. Generate r standard normal random variables Z_1, \dots, Z_r using the Box-Muller algorithm,

2. Return $X = \sum_{i=1}^r Z_i^2$

[3 marks]

We can reuse the `box.muller` function from Exercise 4(a) to generate n standard normal random variables. and then take sums of their squares. With this method we can generate any degrees of freedom, not just even (although to generate an odd number of random variables with an odd degrees of freedom we generate and discard one standard normal).

Here is the required code

```
rchi.2 <- function(n, r){
  #Function to generate chi-squared random variables with
  # any degrees of freedom.
  # The arguments are the required sample size and the
  # degrees of freedom.

  Z <- matrix(box.muller(n*r), nrow=n)
  rowSums(Z^2)
}
```

[2 marks]

The second algorithm will work for any degrees of freedom but it requires $\lceil nr \rceil$ trigonometric calculations. These calculations are not actually necessary since, from the proof of the Box-Muller algorithm, we see that

$$Z_1^2 + Z_2^2 = -2 \log(U_1) \cos^2(2\pi U_2) - 2 \log(U_1) \sin^2(2\pi U_2) = -2 \log(U_1)$$

This is precisely the calculation we used in Part (a) to get a χ_2^2 random variable. Trigonometric functions can be time consuming so avoiding them as we did in Part (a) can be more efficient. Of course the downside is that we cannot generate random variates coming from distributions with odd degrees of freedom. An efficient compromise would be to use the method in Part (a) to generate the n random variables with $2\lfloor r/2 \rfloor$ degrees of freedom. If r is even we are done and if r is odd we can then use the Box-Muller algorithm to generate n standard normal random variables. We could then square the standard normals and add them to the χ^2 random variables to get random variables with r degrees of freedom.

[2 marks]

- c) This is a case of a mixture distribution with an infinite number of components. The probability of each component is given by a Poisson probability. Hence we can generalize from my notes to say that we first generate the component based on a Poisson distribution and then generate from the appropriate central chi-squared distribution whose degrees of freedom will be $k + 2Y$ where Y is the Poisson random variate generated. Here is a function to do that

```
rchisq.nc <- function(n, k, lambda) {
  #
  # Function to generate from a non-central chi-squared
  # distribution with degrees of freedom k and non-centrality
  # parameter lambda.
  #
  # Arguments
  # n: number of random variates to generate
  # k: degrees of freedom
  # lambda: non-centrality parameter
  #

  # Ensure that k is positive
  if (k<=0) error("Degrees of Freedom must be positive")
  # Make sure the non-centrality parameter is non-negative.
  if (lambda < 0)
```

```
    error("Non-centrality parameter cannot be negative")
# Generate the Poisson random variates
Y <- rpois(n, lambda/2)
# Now generate from the appropriate central chi-squared
# distributions. Note that here we are using the fact that
# rchisq is vectorized in its parameters.
rchisq(n, k+2*Y)
}
```

[6 marks]