***Assignment 1: Likelihood optimization***

**The data set *wind.xls* represents wind speed records for twelve Meridian sites distributed around the New Zealand. It has long been known that the wind speed matches well with the Weibull distribution. Your task is to make some inference about the parameter values for the city Otago**

1. **Load necessary information concerning Otago city to R. Plot the histogram of the wind speed and comment whether it looks like Weibull (you may also compare it with the shape of the related Rayleigh distribution)**

We convert our xls file to csv and import it into R using the following code.

dsOtago<- read.csv( "G:\\LIU\\Computational Statistics\\Labs\\Final lab\\wind.csv", header=TRUE, sep=",");

otago <- as.numeric(dsOtago$Otago);

hist(otago);

We get the following graph which looks very much like the Weibull distribution (it has a long tail). **\*0.5p**

**\*0.5p**

1. **Write your own function as a parameter of the data that finds the maximum log-likelihood estimates of the parameters *shape* and *scale* of the Weibull distribution by using an optimization method such as BFGS or CG. Compute the parameter estimates (if you get warnings, ignore them). What kind of problems may one encounter when these methods are used for the search of the maximum log-likelihood estimates? What other methods can you see as alternatives that do not have these problems?**

Now we try to estimate the parameters shape and scale using the following code (We use BFGS here).

> loglikelihoodfn <- function (params, mydata)

+ {

+ shapeParam <- params[1];

+ scaleParam <- params[2];

+

+ ll<- dweibull(mydata, shapeParam, scaleParam, log=TRUE);

+ -sum(ll);

+ }

>

> intitalParams <- c(1, 1);

> optim (intitalParams, loglikelihoodfn, gr=NULL, method="BFGS", mydata=otago);

$par

[1] 2.163650 9.063086

$value

[1] 18899.50

$counts

function gradient

65 14

$convergence

[1] 0

$message

NULL

There were 22 warnings (use warnings() to see them)

>

We find that for our data, a shape value of 2.163650 and scale value of 9.063086. We find that our optimization algorithm actually succeeds in finding the maximum value of our log-likelihood (note we used negative likelihood minimization). However we find that we also generate warnings. We write the following code to look at our warnings. **\*2p**

> warnings()

Warning messages:

1: In dweibull(x, shape, scale, log) : NaNs produced

2: In dweibull(x, shape, scale, log) : NaNs produced

3: In dweibull(x, shape, scale, log) : NaNs produced

4: In dweibull(x, shape, scale, log) : NaNs produced

5: In dweibull(x, shape, scale, log) : NaNs produced

6: In dweibull(x, shape, scale, log) : NaNs produced

7: In dweibull(x, shape, scale, log) : NaNs produced

8: In dweibull(x, shape, scale, log) : NaNs produced

9: In dweibull(x, shape, scale, log) : NaNs produced

10: In dweibull(x, shape, scale, log) : NaNs produced

11: In dweibull(x, shape, scale, log) : NaNs produced

12: In dweibull(x, shape, scale, log) : NaNs produced

13: In dweibull(x, shape, scale, log) : NaNs produced

14: In dweibull(x, shape, scale, log) : NaNs produced

15: In dweibull(x, shape, scale, log) : NaNs produced

16: In dweibull(x, shape, scale, log) : NaNs produced

17: In dweibull(x, shape, scale, log) : NaNs produced

18: In dweibull(x, shape, scale, log) : NaNs produced

19: In dweibull(x, shape, scale, log) : NaNs produced

20: In dweibull(x, shape, scale, log) : NaNs produced

21: In dweibull(x, shape, scale, log) : NaNs produced

22: In dweibull(x, shape, scale, log) : NaNs produced

We find that we produce several undefined value for 22 points. This means that the algorithm tried non-positive (zero or negative) values for our shape and scale parameters. We note that for Weibull distribution our shape and scale parameters take on values greater than one. Therefore, Weibull distribution or its log-likelihood value is undefined for negative or zero scale or shape parameters. We conclude that our domain (for scale and shape) is restricted to take only positive values.

Optimization methods such as BFGS and CG can encounter problems in computing maximum log-likelihood. One problem is that they provide only an approximation to the parameters (for our data). Analytical methods are more accurate and recommended (if possible). Another problem is that these methods may take a lot of time (or are unable to converge). This may happen because of higher dimensionality (lot of parameters) or bad starting values.

In our case, our domain is restricted to only positive values (greater than zero). Optimization methods such as BFGS and CG assume a continuous domain and may also try negative values because of that. Hence these algorithms may fail to converge (for e.g. if they get struck in negative values). **\*1p**

There are special methods for constraint based optimization which optimize with respect to a constraint. We could also use methods such as simulated annealing. However we will need to discretize our domain to use those methods. Here we use one technique to overcome this problem by returning (dummy) log-likelihood values for non-positive values in order to push our optimization algorithm to use positive parameters. We use the following code in R and note that now we don’t generate any warnings and are successfully able to find the best values for our parameters every time.

> loglikelihoodfn <- function (params, mydata)

+ {

+ shapeParam <- params[1];

+ scaleParam <- params[2];

+

+ if (shapeParam <= 0 || scaleParam <= 0)

+ {

+ return(999999);

+ }

+

+ ll<- dweibull(mydata, shapeParam, scaleParam, log=TRUE);

+ -sum(ll);

+ }

>

> intitalParams <- c(3, 3);

>

> optim (intitalParams, loglikelihoodfn, gr=NULL, method="BFGS", mydata=otago);

$par

[1] 2.163650 9.063087

$value

[1] 18899.50

$counts

function gradient

64 14

$convergence

[1] 0

$message

NULL

**\*0.5p**

1. **Modify your function in such way that it returns only *shape* parameter. Consider this function as an estimator of *shape* and generate 100 bootstrap estimates of this parameter. Produce an appropriate plot and make comments. Compute 95% confidence bounds (using percentile method only!) for the parameter *shape* and answer whether the data can in principle come from Rayleigh distribution.**

Now we modify our function to use a bootstrap sample instead of the whole data. This will obviously give us different parameter values. We modify our code as given below.

loglikelihoodfn(c(1,1), mydata=adata );

fnStatistic <- function (thedata, index)

{

thisdata<-thedata[index];

shapeVal<-optim (c(1,1), loglikelihoodfn, gr=NULL, method="BFGS", mydata=thisdata)$par[1];

shapeVal

}

bootout <- boot (otago, fnStatistic, R=100);

**\*1.5p**

We now plot a histogram for our parameter values using the following code.

> hist(bootout$t)



We find that our distribution of shape parameter is not exactly normally distributed as can be seen from the mean and median values below. **\*0.5p**

> mean(bootout$t)

[1] 2.161114

> median(bootout$t)

[1] 2.158483

> r<-range(bootout$t)

> r[1]

[1] 2.122475

> r[2]

[1] 2.217852

> r[2]-r[1]

[1] 0.09537716

> sd(bootout$t)

[1] 0.01950275

We also find that we have significant standard deviation (or standard error) in our parameter estimate. We would therefore like to find the 95% confidence interval in order to find where our parameter value will most likely be in. We can simply use boot.ci function in R to do our job (we already have 100% intervals computed from range function i.e. 2.122475 < shape < 2.217852).

We use the following code in R.

> boot.ci (bootout, conf=0.95, type="perc");

BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS

Based on 100 bootstrap replicates

CALL :

boot.ci(boot.out = bootout, conf = 0.95, type = "perc")

Intervals :

Level Percentile

95% ( 2.125, 2.207 ) **\*0.5p**

Calculations and Intervals on Original Scale

Some percentile intervals may be unstable

We find that we can say with 95% confidence (basing on our data) that our shape parameter has a value between 2.125 and 2.207. The shape value of Rayleigh distribution has a value of 2. This value is not in our 95% confidence interval of (2.125, 2.207). Hence we can conclude that this data doesn’t come from Rayleigh distribution. **\*1p**

***Assignment 2: Computing an integral***

Consider the distribution with density

1. **Write your own function that can generate sample of size *n* from the distribution above using the acceptance-rejection method with uniform distribution as the majorizing density. The function should also compute what percent *R* of the totally generated random numbers was rejected in this method. What was the value of the scaling constant *c* you used in deriving the majorizing function and why you have chosen this value?**

For using acceptance-rejection method we need our majorizing density to dominate our target density at each point. i.e. c\*g(X) >= p(X) for all X. For uniform distribution the value of c is simply the maximum value of our function. We can find the maximum value of our function by computing its first derivative which comes out to be pi\*pi/2\*cos(pi\*x) = 0. Therefore our final equation is x =

We find that this value is 0.5 (we also know that same from our sin function as our function is just scaling of sin function and sin has maxima at 0.5)

> acos(0)

[1] 1.570796

> acos(0)/pi

[1] 0.5

We therefore find that highest value by computing the value of our function at this point.

> ourfn(0.5)

[1] 1.570796

Hence we will use this value as our value of c. We use the following code to plot this scenario.

y1 <- c();

y2 <- c();

constant<-1.570796;

xvals <- seq(0, 1, by=0.001);

for (x in xvals)

{

y1 <- c(y1, constant \* majorfn (x));

y2 <- c(y2, ourfn (x));

}

plot (xvals, y1, type="l", col="blue" );

points(xvals, y2, col="red", pch=15, cex=0.1);

**\*1p**

Now we have confirmed that this makes our product c\*majorizingfn(x) greater than our target function targetfn(x) at each point. The two function also have the same support. Hence we use this value of c for our implementation given below.

rfnAcceptReject <- function (n)

{

count <- 0;

rejects <- 0;

nums <- c();

while (TRUE)

{

constant<-1.570796;

y <- runif(1); #generate a random number from our majorizing function

u <- runif(1);

#now we need to test to see if it is okay to accept

if (u <= ourfn(y)/(constant\*majorfn(y)) )

{

count <- count + 1;

nums <- c(nums, y);

if (count >= n)

{

break;

}

}

else

{

rejects <- rejects + 1; #we have a reject

}

}

list (result=nums, RejectPerc=(rejects/n))

}

Now we do some sample runs, we find that our value of reject percentage goes towards the value of c-1 (expected value of number of rejects) with higher number of points generated. **\*1.5p The rejection rate of the assignment was defined as rejected/total. You define it as rejected/accepted. Points are not reduced.**

> rfnAcceptReject (100)

$result

[1] 0.25661719 0.23228547 0.28407436 0.46599829 0.78325584 0.53665918 0.64160312

[8] 0.53317788 0.60235119 0.06209967 0.59701574 0.37510657 0.14471955 0.42176445

[15] 0.50150050 0.45834170 0.20377791 0.21112751 0.18848217 0.91675582 0.62874007

[22] 0.06694432 0.53544504 0.86899525 0.16588414 0.29304602 0.25101036 0.52968988

[29] 0.47201464 0.34618667 0.30829015 0.08134321 0.43589649 0.55320798 0.21971202

[36] 0.50511895 0.50265299 0.65652215 0.95653418 0.19588826 0.57729627 0.48161527

[43] 0.56298773 0.68614509 0.21546191 0.47320299 0.43297834 0.54730879 0.67493865

[50] 0.67902830 0.61431723 0.46740020 0.48646469 0.25863192 0.56194697 0.85049700

[57] 0.58664134 0.44114901 0.61155407 0.56647157 0.44722911 0.55211986 0.48742943

[64] 0.18381598 0.60437755 0.29834802 0.61153715 0.23287799 0.80011230 0.88482435

[71] 0.49052123 0.42486408 0.79257076 0.87814412 0.50710876 0.37162244 0.63596553

[78] 0.33873118 0.73265773 0.68628793 0.35905949 0.86711923 0.45939772 0.52258072

[85] 0.29717733 0.55954471 0.51307886 0.25800319 0.63246502 0.58684404 0.24883421

[92] 0.43709032 0.50589330 0.38172150 0.23457880 0.39144574 0.36630419 0.29957937

[99] 0.17046822 0.67868726

$RejectPerc

[1] 0.7

> rfnAcceptReject (1000)$RejectPerc

[1] 0.576

> rfnAcceptReject (10000)$RejectPerc

[1] 0.5719

>

1. **Generate a sample of size 1000 by using your function and plot the histogram. Does the sample look like it should?**

We also plot the histogram to further test our function.

> vals<-rfnAcceptReject (1000)$result

> hist(vals)



We find that this histogram very nearly approximates our target function. We use the following code in R to overlay our probability values from target function on our histogram. **\*1p**

vals<-rfnAcceptReject (10000)$result

hist(vals, freq = FALSE)

y1 <- c();

xvals <- seq(0, 1, by=0.001);

for (x in xvals)

{

y1 <- c(y1, ourfn (x));

}

points(xvals, y1, col="red", pch=15, cex=0.1);



Hence we are satisfied from the generated distribution. We find that as we generate more points our distribution is more close to our target distribution.



Hence we are really satisfied with our result.

1. **Print out the computed rejection rate *R*. Find out without using computer what would be the rejection rate *R* if the sample size would be very large (i.e., the expected value of *R*).**

The rejection rate would simply be c-1 (expected number of steps – 1). Hence it should be close to (we have variation but the average is) 1.570796 -1 or 0.570796% for very large n (or n -> infinity). This is the reason why we need c to be as small as possible but enought to majorize. We can also see this phenomena to occur by using the following code. **\*1.5p**

> rfnAcceptReject (100)$RejectPerc

[1] 0.52

> rfnAcceptReject (1000)$RejectPerc

[1] 0.569

> rfnAcceptReject (10000)$RejectPerc

[1] 0.5737

1. **Use the obtained sample to compute the value of the integral to derive the value of the integral by applying the importance sampling. Why using the introduced above as importance function is better than using the density of the uniform distribution?**

Importance sampling reduces the variance in the estimation of our integral by finding an optimal decomposition of our function f(x) i.e. g(x)\*p(x). We sample more values for values of x where f(x) is large (or give more importance to those points). We decompose our function f(x) by formula

f(x) = f(x)/p(x) \* p(x) or f(x) = g(x)\*p(x) where g(x) = f(x)/p(x).

For our function we can use the following decomposition.

Now our p(x) =

And g(x) =

We know from assignment 1 that p(x) is a probability density function. Now we need to choose p(x) such that our variance (of estimating integral) is minimized. We find that best decomposition (and so p(x) function) is such that our g(x) is nearly constant. We find that it is not constant but still a good approximation (value change little slowly). We plot g(x) as well to see this change.

g <- function (x)

{

res <- 2/(pi\*(5+(x^2)))

res;

}

y1 <- c();

xvals <- seq(0, 1, by=0.001);

for (x in xvals)

{

y1 <- c(y1, g(x));

}

plot (xvals, y1, type="l", col="blue" );

Our plot for g(x) is given below.



Using the uniform distribution (0, 1) (which as identity probability function) for our p(x) gives us the following result.

This makes our g(x) as

Now we plot the two functions and find that our choosen g(x) is almost constant as compared to using a uniform density function (for decomposition).

> g2 <- function (x)

+ {

+ res <- sin(pi\*x)/(5+(x^2))

+ res

+ }

>

> y1 <- c();

> y2 <- c();

> xvals <- seq(0, 1, by=0.001);

>

> for (x in xvals)

+ {

+ y1 <- c(y1, g(x));

+ y2 <- c(y2, g2(x));

+ }

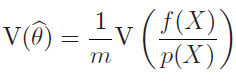
>

> plot (xvals, y2, type="l", col="red" );

> points (xvals, y1, type="l", col="blue" );

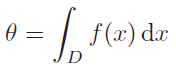


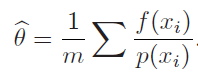
We also know that the variance of our parameter estimation from monte carlo can be given by the formula.



Hence dividing f(x) by p(x) makes the above value small. Also if our f(x) is proportional (i.e. multiple of) to p(x), we get a constant variance. However we have a factor of x (it is not exactly proportional ) . However it is good enough (we could use perform decomposition by finding the integral of f(x) and using it as p(x) but that doesn’t give us any benefit).

Now we use code to compute the value of our integral by the formula .





We run the following code in R.

> g <- function (x)

+ {

+ res <- 2/(pi\*(5+(x^2)))

+ res;

+ }

> mean(g(otago))

[1] 0.01610407

Hence we find that the value of our integral is 0.01610407. This value will have less variance than for e.g. using p(x) as uniform density. We use the following code to find that variance of our estimation. **\*1p**

gsample <- function (data, index)

+ {

+ mydata <- data[index]

+ res <- mean(g(mydata))

+ res

+ }

>

> bootout <- boot (otago, gsample, R=10000);

> hist(bootout$t)

We get the following graph in R.



The corresponding graph using uniform distribution has a very large variance as can be seen from the R code below. **\*1p**

> gsample <- function (data, index)

+ {

+ mydata <- data[index]

+ res <- mean(g(mydata))

+ res

+ }

>

> bootout <- boot (otago, gsample, R=10000);

> hist(bootout$t)

> gsample <- function (data, index)

+ {

+ mydata <- data[index]

+ res <- mean(g2(mydata))

+ res

+ }

>

> bootout <- boot (otago, gsample, R=10000);

> hist(bootout$t)



1. **Compute the same integral by using the function *integrate* in R and comment on the result.**

> f <- function (x)

+ {

+ res <- sin(pi\*x)/(5+(x^2))

+ res

+ }

>

> integrate(f, 0, 1)

0.1203855 with absolute error < 1.3e-15

We know that the integral function in R calculates the integral of a function by using analytical methods. Hence it is the real value of our integral.

We find that our value calculated using importance sample is very much dependent on our data. It is just an approximation calculated from our data. As the size of our data increase we get better accuracy.

We can conclude that when it is easy to compute integral directly. It is best to do so (analytical) rather than relying on an approximate. Here we find that the value we get was actually far from the real integral value. **\*1p**