From Wikipedia, the free encyclopedia

Incomplete Beta Function – Used to obtain t and F Probabilities

The Beta Function:

$$\mathrm{B}(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$$

The beta function is symmetric, meaning that

$$\mathrm{B}(x,y) = \mathrm{B}(y,x).$$

The Beta function is related to the Gamma Function by:

$$\mathrm{B}(x,y) = \frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)}$$

The **incomplete beta function**, a generalization of the beta function, is defined as

$$\mathrm{B}(x;\,a,b) = \int_0^x t^{a-1}\,(1-t)^{b-1}\,dt.$$

For $x = 1$, the incomplete beta function coincides with the complete beta function. The relationship between the two functions is like that between the gamma function and its generalization the incomplete gamma function.

The **regularized incomplete beta function** (or **regularized beta function** for short) is defined in terms of the incomplete beta function and the complete beta function:

$$I_x(a,b) = \frac{\mathrm{B}(x;\,a,b)}{\mathrm{B}(a,b)}.$$

Working out the integral (one can use integration by parts to do that) for integer values of a and b, one finds:

$$I_x(a,b) = \sum_{j=a}^{a+b-1} \frac{(a+b-1)!}{j!(a+b-1-j)!} x^j (1-x)^{a+b-1-j}.$$

The regularized incomplete beta function can be used to evaluate the cumulative density function of a random variable *X* from a binomial distribution, where the "probability of success" is *p* and the sample size is *n*:

$$F(k;n,p) = \Pr(X \leq k) = I_{1-p}(n-k, k+1).$$

Properties

$$I_0(a,b) = 0$$

$$I_1(a,b) = 1$$

$$I_x(a,b) = 1 - I_{1-x}(b,a)$$

From *Numerical Recipes in C*

The incomplete beta function is defined by

$$I_x(a,b) \equiv \frac{B_x(a,b)}{B(a,b)} \equiv \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1} dt \qquad (a,b>0) \qquad (6.4.1)$$

It has the limiting values

$$I_0(a,b) = 0 \qquad I_1(a,b) = 1 \qquad (6.4.2)$$

and the symmetry relation

$$I_x(a,b) = 1 - I_{1-x}(b,a) \qquad (6.4.3)$$

If a and b are both rather greater than one, then $I_x(a,b)$ rises from "near-zero" to "near-unity" quite sharply at about $x = a/(a+b)$. Figure 6.4.1 plots the function for several pairs (a,b).

The incomplete beta function has a series expansion

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)}\left[1 + \sum_{n=0}^{\infty}\frac{B(a+1,n+1)}{B(a+b,n+1)}x^{n+1}\right], \tag{6.4.4}$$

but this does not prove to be very useful in its numerical evaluation. (Note, however, that the beta functions in the coefficients can be evaluated for each value of n with just the previous value and a few multiplies, using equations 6.1.9 and 6.1.3.)

The continued fraction representation proves to be much more useful,

$$I_x(a,b) = \frac{x^a(1-x)^b}{aB(a,b)}\left[\frac{1}{1+}\frac{d_1}{1+}\frac{d_2}{1+}\cdots\right] \tag{6.4.5}$$

where

$$\begin{aligned} d_{2m+1} &= -\frac{(a+m)(a+b+m)x}{(a+2m)(a+2m+1)} \\ d_{2m} &= \frac{m(b-m)x}{(a+2m-1)(a+2m)} \end{aligned} \tag{6.4.6}$$

This continued fraction converges rapidly for $x < (a+1)/(a+b+2)$, taking in the worst case $O(\sqrt{\max(a,b)})$ iterations. But for $x > (a+1)/(a+b+2)$ we can just use the symmetry relation (6.4.3) to obtain an equivalent computation where the continued fraction will also converge rapidly. Hence we have

```
#include <math.h>

float betai(float a, float b, float x)
Returns the incomplete beta function Ix(a,b).
{
    float betacf(float a, float b, float x);
    float gammln(float xx);
    void nrerror(char error_text[]);
    float bt;

    if (x < 0.0 || x > 1.0) nrerror("Bad x in routine betai");
    if (x == 0.0 || x == 1.0) bt=0.0;
    else                                    Factors in front of the continued fraction.
        bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log(1.0-x));
    if (x < (a+1.0)/(a+b+2.0))              Use continued fraction directly.
        return bt*betacf(a,b,x)/a;
    else                                    Use continued fraction after making the sym-
        return 1.0-bt*betacf(b,a,1.0-x)/b;      metry transformation.
}
```

The gamma function is defined by the integral

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt \tag{6.1.1}$$

When the argument z is an integer, the gamma function is just the familiar factorial function, but offset by one,

$$n! = \Gamma(n+1) \tag{6.1.2}$$

The gamma function satisfies the recurrence relation

$$\Gamma(z+1) = z\Gamma(z) \tag{6.1.3}$$

If the function is known for arguments $z > 1$ or, more generally, in the half complex plane $\mathrm{Re}(z) > 1$ it can be obtained for $z < 1$ or $\mathrm{Re}\,(z) < 1$ by the reflection formula

$$\Gamma(1-z) = \frac{\pi}{\Gamma(z)\sin(\pi z)} = \frac{\pi z}{\Gamma(1+z)\sin(\pi z)} \tag{6.1.4}$$

Notice that $\Gamma(z)$ has a pole at $z = 0$, and at all negative integer values of z.

There are a variety of methods in use for calculating the function $\Gamma(z)$ numerically, but none is quite as neat as the approximation derived by Lanczos [1]. This scheme is entirely specific to the gamma function, seemingly plucked from thin air. We will not attempt to derive the approximation, but only state the resulting formula: For certain integer choices of γ and N, and for certain coefficients $c_1, c_2, \ldots, c_N$, the gamma function is given by

$$\begin{aligned}\Gamma(z+1) &= (z+\gamma+\tfrac{1}{2})^{z+\frac{1}{2}} e^{-(z+\gamma+\frac{1}{2})} \\ &\times \sqrt{2\pi}\left[c_0 + \frac{c_1}{z+1} + \frac{c_2}{z+2} + \cdots + \frac{c_N}{z+N} + \epsilon\right] \quad (z > 0)\end{aligned} \tag{6.1.5}$$

ols_read_svd_general.c (30 August 2009)

```
/*
c:/mingw/bin/gcc -I"c:/program files/R/R-2.9.0/include" -L"C:/Program
Files/R/R-2.9.0/bin" -Wall %1.c -o %1.exe -lRlapack -lRblas

General OLS program.  Reads matrix created by writedata_ols.c

The only parameters the user has to set are the number of rows and
columns -- nrow and ncol below.  The number of columns counts a column
of "1"s used for the intercept term.  All memory is then dynamically
allocated using nrow and ncol.

This version does a Singular Value Decomposition on the input matrix
*/
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <R_ext/Lapack.h>
#include <R_ext/BLAS.h>
#define MAXIT 100
#define EPS 3.0e-7
#define FPMIN 1.0e-30
/* Prototypes for Incomplete Beta Distribution – use to get probabilities for
t and F values */
double betai(double a, double b, double x);
double betacf(double a, double b, double x);
double gammln(double xx);

void xsvd(int kpnp, int kpnq, double *, double *, double *, double *);
void xrsquare(double *sse, double *tss, int nrow, int ncol, double *Y, double
*X, double *coef);

FILE *fp;
FILE *jp;

int main(){

        int i, j, info, errno;
        char trans = 't', notrans ='n';
        double alpha = 1.0, beta=0.0;
        int nrow=1000, ncol=25;
        int one=1;
        double *X, *Y, *XprimeX, *XXinv, *XXinvX, *coef, *xxinvdiag;
        double *u, *lambda, *vt;
        int *ipiv;
        double time1, time2, timedif;
        double sse, tss, stderrest=0.0, pearsonrsquare, betastderror;
        double vvv, aaa, bbb, ttt, xxx, tinvprob, vvv1, vvv2, fff, finvprob,
totallnl;

        X = (double *) malloc (nrow*ncol*sizeof(double));
        Y = (double *) malloc (nrow*sizeof(double));
```

```
        XprimeX = (double *) malloc (ncol*ncol*sizeof(double));
        XXinv = (double *) malloc (ncol*ncol*sizeof(double));
        XXinvX = (double *) malloc (nrow*ncol*sizeof(double));
        coef = (double *) malloc (ncol*sizeof(double));
        xxinvdiag = (double *) malloc (ncol*sizeof(double));
        ipiv = (int *) malloc (ncol*sizeof(int));
        u      = (double *) malloc (nrow*nrow*sizeof(double));
        lambda = (double *) malloc (ncol*ncol*sizeof(double));
        vt     = (double *) malloc (ncol*ncol*sizeof(double));
/* clock() is part of time.h -- returns the implementation's
 * best approximationto the processor time elapsed since the
 * program was invoke, divide by CLOCKS_PER_SEC to get the time
 * in seconds           */
        time1 = (double) clock();            /* get initial time */
        time1 = time1 / CLOCKS_PER_SEC;      /*    in seconds    */

        printf("\nnumber of rows = %d  number of columns = %d\n\n",nrow,ncol);

        jp = fopen("c:/docs_c_summer_course/data_ols_svd.txt","w");

        if((fp = fopen("c:/docs_c_summer_course/data_ols.txt","r"))==NULL)
        {
                printf("\nUnable to open file OLS_DATA.TXT: %s\n",
strerror(errno));
                exit(EXIT_FAILURE);
        }
        else {

           fprintf(jp," Y and X = \n");
           for(i=0;i<nrow;i++)
           {
                  fscanf(fp,"%lf",&Y[i]);
                  for(j=0;j<ncol;j++)
                  {
                         fscanf(fp,"%lf",&X[i+j*nrow]);
                  }
                  fprintf(jp,"%10d %12.6f", i,Y[i]);
                  for(j=0;j<ncol;j++)
                  {
                         fprintf(jp,"%12.6f",X[i+j*nrow]);
                  }
                  fprintf(jp,"\n");
           }
        }
/* Call Singular Value Decomposition Routine to look at the colinearity
 * in X */
/*  Clock the SVD Routine*/

        time2 = (double) clock();            /* get initial time */
        time2 = time2 / CLOCKS_PER_SEC;      /*    in seconds    */
        xsvd(nrow,ncol,X,u,lambda,vt);
        printf("Singular Values\n");
        for(j=0;j<ncol;j++)
        {
                printf("%d %f\n",j,lambda[j]);
```

```
        }
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time2;
        printf("SVD took %12.3f seconds\n", timedif);
        fprintf(jp,"SVD took %12.3f seconds\n", timedif);

        dgemm_(&trans,&notrans,&ncol,&ncol,&nrow,&alpha,X,&nrow,X,&nrow,&beta,
XprimeX,&ncol);
        fprintf(jp,"\n\nX'X = \n");
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        fprintf(jp,"%12.6f",XprimeX[i+j*ncol]);
                }
                fprintf(jp,"\n");
        }
/* Initialize the Identity matrix */
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        XXinv[i+j*ncol]=0.0;
                        if(i == j)XXinv[i+j*ncol]=1.0;
                }
        }
        dgesv_(&ncol,&ncol,XprimeX,&ncol,ipiv,XXinv,&ncol,&info);
        fprintf(jp,"\n\n(X'X)-1 = \n");
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        fprintf(jp,"%12.6f",XXinv[i+j*ncol]);
/* Save Diagonal of (X'X)-1 */
                        if(i == j)xxinvdiag[i] = XXinv[i+j*ncol];
                }
                fprintf(jp,"\n");
        }

//XXinv is 2x2
//X' is 2x5
//X is 5x2

        dgemm_(&notrans,&trans,&ncol,&nrow,&ncol,&alpha,XXinv,&ncol,X,&nrow,&b
eta,XXinvX,&ncol);

//XXinvX is 2x5
//Y is 5x1

        dgemm_(&notrans,&notrans,&ncol,&one,&nrow,&alpha,XXinvX,&ncol,Y,&nrow,
&beta,coef,&ncol);
/* Get Sum of Squared Error */
        xrsquare(&sse, &tss, nrow, ncol, Y, X, coef);
        /* Standard Error of the Estimate */
        stderrest = sqrt(sse/(double)(nrow-ncol));
/* Write out Coefficient Vector and Standard Errors */
```

```
        fprintf(jp,"\n\nCoefficient Vector = ");
        printf("\n\nCoefficient Vector = ");
        vvv = (double)(nrow-ncol);
        aaa = vvv/2.0;
        bbb = 0.5;
        for(i=0;i<ncol;i++)
        {
                betastderror = stderrest*sqrt(xxinvdiag[i]);
                ttt = coef[i]/betastderror;
                xxx = vvv/(vvv+ttt*ttt);
                tinvprob = betai(aaa, bbb, xxx);
                printf("\n%d %12.6f %12.6f %12.6f %12.6f", i, coef[i],
betastderror, ttt, tinvprob);
                fprintf(jp,"\n%d %12.6f %12.6f %12.6f %12.6f", i, coef[i],
betastderror, ttt, tinvprob);
        }
        printf("\n\n");
        fprintf(jp,"\n\n");

        fprintf(jp,"SSE = %12.7g\n",sse);
        printf("SSE = %12.7g\n",sse);
        fprintf(jp,"TSS = %12.7g\n",tss);
        printf("TSS = %12.7g\n",tss);
         fprintf(jp,"Standard Error of the Estimate = %12.7g\n",stderrest);
        printf("Standard Error of the Estimate = %12.7g\n",stderrest);
// Compute F(ncol-1, nrow-ncol) Using the Incomplete Beta Function
        vvv1 = ncol - 1;
        vvv2 = nrow - ncol;
        aaa = vvv2/2.0;
        bbb = vvv1/2,0;
// Overall F = {[TSS - SSE]/(ncol-1)}/{SSE/(nrow-ncol)}
        fff = ((tss - sse)/(double)(ncol -1))/(sse/(double)(nrow-ncol));
        xxx = vvv2/(vvv2 + vvv1*fff);
        finvprob = betai(aaa, bbb, xxx);
        fprintf(jp," F,ncol-1,nrow-ncol = %12.7g, Prob > F = %12.7g\n",fff,
finvprob);
        printf(" F,ncol-1,nrow-ncol = %12.7g, Prob > F = %12.7g\n",fff,
finvprob);
// Log-Likelihood
        totallnl = -(nrow/2.0)*log(2.0*3.1415926536) - nrow*log(stderrest) -
sse/(2*stderrest*stderrest);
        fprintf(jp,"Log-Likelihood = %12.7g\n",totallnl);
        printf("Log-Likelihood = %12.7g\n",totallnl);
// Pearson r-square
        pearsonrsquare = 1.0 - sse/tss;
        fprintf(jp,"Pearson R Squared = %12.7g\n",pearsonrsquare);
        printf("Pearson R Squared = %12.7g\n",pearsonrsquare);

        free(X);
        free(Y);
        free(XprimeX);
        free(XXinvX);
        free(coef);
        free(xxinvdiag);
        free(ipiv);
```

```
        free(u);
        free(lambda);
        free(vt);
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
        printf("The total elapsed time of the program is %12.3f seconds\n",
timedif);
        fprintf(jp,"\nThe total elapsed time of the program is %12.3f
seconds\n", timedif);

        fclose(jp);
        fclose(fp);
        return(0);

}
/*
 *  Pearson R-Square Subroutine -- Computes r-square for simple OLS
 */
void xrsquare(double *sse, double *tss, int nrow, int ncol, double *Y, double
*X, double *coef)
{
        int i, j;
        double sum, sum2, sum3, ymean;
        /* Calculate Y */
        sum2=0.0;
        ymean=0.0;
        for(i=0;i<nrow;i++)
        {
                ymean=ymean+Y[i];
                sum=0.0;
                for(j=0;j<ncol;j++)
                {
                        sum=sum+coef[j]*X[i+j*nrow];
                }
/*  Calculate the SSE here*/
                sum2=sum2+(sum - Y[i])*(sum - Y[i]);
        }
        ymean=ymean/(double)(nrow);
        sum3=0.0;
        for(i=0;i<nrow;i++)
        {
/*  Calculate the TSS here*/
                sum3=sum3+(Y[i]-ymean)*(Y[i]-ymean);
        }
        *sse = sum2;
        *tss = sum3;
//return(sse);

}
/*

Singular Value Decomposition Subroutine

*/
void xsvd(int kpnp, int kpnq, double *y, double *u, double *lambda, double
*vt) {
```

```
/*
*/

        double *a, *work;
        double sumulv, svd_error_sum, svd_error_sum_2;
        int i, j, jj;
        int  info = 12;
        int  lwork= kpnp*kpnp+kpnq*kpnq;
        int  lda,ldu,ldvt;

        a     = calloc( kpnp*kpnq, sizeof(double));
        work  = calloc( lwork, sizeof(double));


        lda  = kpnp;
        ldu  = kpnp;
        ldvt = kpnq;
        for (i=0;i<lwork;i++){
                work[i] = 0;
        }

        fprintf(jp,"lwork=%i\n",lwork);

        for (j=0;j<kpnq;j++) {
                for (i=0;i<kpnp;i++) {
                        a[(j*kpnp)+i] = y[(j*kpnp)+i];
                }
        }

        dgesvd_("A","A", &kpnp, &kpnq, a, &lda, lambda,
                  u, &ldu, vt, &ldvt, work, &lwork, &info);

        fprintf(jp,"Info = %i\n",info);
        printf("Info = %i\n",info);
        fprintf(jp,"Singular Values\n");
        printf("Singular Values\n");
        for(jj=0;jj<kpnq;jj++)
        {
                fprintf(jp,"%d %f\n",jj,lambda[jj]);
                printf("%d %f\n",jj,lambda[jj]);
        }
/*
  Do simple check of SVD

*/
        svd_error_sum=0.0;
        svd_error_sum_2=0.0;
        for (i=0;i<kpnp;i++)
        {
                for (jj=0;jj<kpnq;jj++)
                {
                        sumulv=0.0;
                        for (j=0;j<kpnq;j++)
                        {
```

```
        sumulv+=u[(j*kpnp)+i]*lambda[j]*vt[j+(jj*kpnq)];
                        }
                        svd_error_sum+=(y[i+(jj*kpnp)]-sumulv)*(y[i+(jj*kpnp)]-
sumulv);
                        svd_error_sum_2+=fabs(y[i+(jj*kpnp)]-sumulv);
                }
        }
        fprintf(jp,"SVD Error Check = %12.7g
%12.7g\n",svd_error_sum,svd_error_sum_2);
        printf("SVD Error Check = %12.7g
%12.7g\n",svd_error_sum,svd_error_sum_2);
        free(work);
        free(a);
}
/* Incomplete Beta Distribution Function -- Use to obtain t and F
 * distribution Probabilities (p-values) */
double betai(double a, double b, double x)
{
        double betacf(double a, double b, double x);
        double gammln(double xx);
//      void nrerror(char error_text[]);
        double bt;

        if (x < 0.0 || x > 1.0)
        {
                printf("Bad x in routine betai");
                exit(EXIT_FAILURE);
        }
        if (x == 0.0 || x == 1.0) bt=0.0;
        else
                bt=exp(gammln(a+b)-gammln(a)-gammln(b)+a*log(x)+b*log(1.0-x));
        if (x < (a+1.0)/(a+b+2.0))
                return bt*betacf(a,b,x)/a;
        else
                return 1.0-bt*betacf(b,a,1.0-x)/b;
}

double betacf(double a, double b, double x)
{
//      void nrerror(char error_text[]);
        int m,m2;
        double aa,c,d,del,h,qab,qam,qap;

        qab=a+b;
        qap=a+1.0;
        qam=a-1.0;
        c=1.0;
        d=1.0-qab*x/qap;
        if (fabs(d) < FPMIN) d=FPMIN;
        d=1.0/d;
        h=d;
        for (m=1;m<=MAXIT;m++) {
                m2=2*m;
                aa=m*(b-m)*x/((qam+m2)*(a+m2));
```

```
                d=1.0+aa*d;
                if (fabs(d) < FPMIN) d=FPMIN;
                c=1.0+aa/c;
                if (fabs(c) < FPMIN) c=FPMIN;
                d=1.0/d;
                h *= d*c;
                aa = -(a+m)*(qab+m)*x/((a+m2)*(qap+m2));
                d=1.0+aa*d;
                if (fabs(d) < FPMIN) d=FPMIN;
                c=1.0+aa/c;
                if (fabs(c) < FPMIN) c=FPMIN;
                d=1.0/d;
                del=d*c;
                h *= del;
                if (fabs(del-1.0) < EPS) break;
        }
        if (m > MAXIT)
        {
                printf("a or b too big, or MAXIT too small in betacf");
                exit(EXIT_FAILURE);
        }
        return h;
}
double gammln(double xx)
{
        double x,y,tmp,ser;
        static double cof[6]={76.18009172947146,-86.50532032941677,
        24.01409824083091,-1.231739572450155,
        0.1208650973866179e-2,-0.5395239384953e-5};
        int j;

        y=x=xx;
        tmp=x+5.5;
        tmp -= (x+0.5)*log(tmp);
        ser=1.000000000190015;
        for (j=0;j<=5;j++) ser += cof[j]/++y;
        return -tmp+log(2.5066282746310005*ser/x);
}
```