5.12 モンテカルロシミュレーション Bootstrap の他方法　ver.0.1

ここでは、前章で取り上げたもの以外の代表的な Bootstrap の方法と統計学での基本理論を示します。ここでは、単純化のため、前章のような回帰ではなくて平均などの推定量に絞って話を進めていきます。

Balanced Bootstrap

この方法は、通常の Bootstrap の際のサンプリングの方法を違った方法に置き換えたものです。手もとに 100 枚程度のカードかなにかがあるとします。10 人分のテストの点数があるとして、最初の 10 枚に１つずつ書きこみます。次の 10 枚にも同じ点数を書きこみます。これと同じことを、例えば、10 回分書きこみます。都合 10 人分の点数が 10 回ずつ「均等に」書き込んだことになります。これをここで十分にシャッフルします。そして、最初の 10 枚を取り出して、推定量(例えば平均)を計算して、どこかに書きとめておきます。次に、最初の 10 枚は捨ててしまって、次の 10 枚を取り出して、同様に推定量を計算して、どこかに書きとめておきます。この作業をカードがなくなるまで繰り返します。最後に、書きとめた 10 回分の推定量を平均します。これが、Balanced Bootstrap です。１変数であって推定量を平均にとってやれば、均等にデータがあるわけですから、シャッフルしたとしても平均の平均は、もとのサンプルの平均に等しくなるはずです。この点は、１つずつ除いた平均を平均しても、もとのサンプルの平均に等しくなる Jackknife と同じような結果になります。

プログラム

```
new; cls;
data={78,68,89,92,60,42,74,80,78};
times=200;
call balancedboot(data,times);

proc(3)=balancedboot(data,times);
    local n,thetahat,vartheta,temp,i,thetahat_i;
    local resample,bsthetahat,bias,corrected,bsvar;
    n=rows(data);
    /* Theta here is mean. */
    thetahat=meanc(data);
    vartheta=stdc(data)^2/n;
    temp=data;
    i=1;
    do while i<=times-1;
```

```
        data=temp|data;
        i=i+1;
    endo;
    data=sortc(data~rndu(n*times,1),2);
    data=data[.,1];
    thetahat_i=zeros(times,1);
    i=1;
    do while i<=times;
        resample=data[n*(i-1)+1:n*i];
        thetahat_i[i]=meanc(resample);
        i=i+1;
    endo;
    bsthetahat=meanc(thetahat_i);
    bias=bsthetahat-thetahat;
    corrected=thetahat-bias;
    bsvar=stdc(thetahat_i)^2;
    print/lz "# of sample=" n;
    print/lz "# of repetition=" times;
    print/lz "     sample mean=" thetahat;
    print/lz "     sample var =" vartheta;
    print/lz "bootstrap mean=" bsthetahat;
    print/lz "bootstrap var =" bsvar;
    print/ld "                bias=" bias;
    print/lz "bias-corrected=" corrected;
    retp(bsthetahat,bias,bsvar);
endp;
```

プログラムでは、data を垂直方向に｜でもって、最初に temp に data の列を設定しておいて times-1 回分マージして、あらためて data として置きなおす。そして、シャッフルは簡易的に一様乱数を発生させて、その大きさの順位によって data を並べなおす。すなわち、

data=sortc(data~rndu(n*times,1),2);によって、n×times 列分の一様乱数を発生させ、これを data とマージさせ、２列目を基準に小さいものから大きいものへとソートする。そして、再び、data を 1 列目だけを取り出して、１列だけのものにする。この data を用いて resample=data[n*(i-1)+1:n*i];によって、i=1 から times 回まで resample を取り出してその推定量を計算して、あらかじめゼロで領域確保されている thetahat_i に１つずつ格納していく。最終的に、この thetahat_i を計算して、平均とその平均の分散を求めている。

画面表示

```
# of sample= 9
# of repetition= 200
   sample mean= 73.444444
   sample var = 25.975309
bootstrap mean= 73.444444
bootstrap var = 26.494199
          bias= 0.00000000
bias-corrected= 73.444444
```

上の結果でわかるように、Jackknife のように１変数の平均を推定量にした場合、サンプリングの方法からも予想できるように、Bias は０(割りきれない場合の端数のまるめを含んでいるので完全には０にはならない)で、もともとのサンプルの平均と、この場合の Balanced Bootstrap の平均は等しい。なお、上の sample var は、これまでのほかのリサンプリングの分析と同じように sample mean の分散のことで、Bootstrap Variance と比べるために計算しています。なお、以下の様々な方法では、上のデータが均一に Balanced されたものではなくて、再び通常の Bootstrap の考え方を採用することにします。

Jackknife after Bootstrap

この方法は、先に Bootstrap を行ない、times 回分のサンプルを作る。これを計算すれば推定量とその SE(または分散)が得られる。これを、今回は直接に計算せずに、times 回分のサンプルのそれぞれから、（それぞれの Bootstrap Sample の j 番目ではなくて）もとのサンプルの j 番目に相当するものを切り落として、それをもとに Bootstrap SE(j)を求める。したがって、切り落とすデータが１個の場合もあれば、複数個の場合もある。また、０個の場合も往々にしてある。これをもとのサンプルのデータの個数分繰り返す。この場合、もとのサンプルに重複があれば、Bootstrap のときの index の番号にもとづいて Jackknife の切り落としをする。このｎ個の Bootstrap SE(j)から Jackknife SE を求める。これが、Jackknife After Bootstrap です。

プログラム

```
new; cls;
rndseed 911;
data={78,68,89,92,60,42,74,80,78};
call jab(data,1000);

proc(4)=jab(data,times);
    local n,thetahat,setheta,thetahat_i,i,j,index,jindex,temp;
    local bsthetahat,seb,jthetahat_ij,jthetahat_j,sejseb,jtemp;
```

```
    n=rows(data);
    /* Theta here is mean. */
    thetahat=meanc(data);
    setheta=stdc(data)/sqrt(n);
    thetahat_i=zeros(times,1);
    jthetahat_ij=zeros(times,n);
    i=1;
    do while i<=times;
        index=ceil(n*rndu(n,1));
        temp=data[index];
        thetahat_i[i]=meanc(temp);
        j=1;
        do while j<=n;
            jindex=zeros(n,1);
            jindex=(index.==j);
            jtemp=delif(temp,jindex);
            jthetahat_ij[i,j]=meanc(jtemp);
            j=j+1;
        endo;
        i=i+1;
    endo;
    bsthetahat=meanc(thetahat_i);
    seb=stdc(thetahat_i);
    jthetahat_j=stdc(jthetahat_ij);
    sejseb=sqrt( ((n-1)/n)*sumc((jthetahat_j-meanc(jthetahat_j))^2) );
    print/lz "     sample mean=" thetahat;
    print/lz "        sample SE=" setheta;
    print/lz "    bootstrap SE=" seb;
    print/lz "SEjack(SEboot)=" sejseb;
    retp(thetahat,setheta,seb,sejseb);
endp;
```

画面表示

```
    sample mean= 73.444444
      sample SE= 5.0965978
  bootstrap SE= 4.6194834
SEjack(SEboot)= 1.7115705
```

上の結果は、推定量を平均にとったときの Bootstrap SE をさらに Jackknife したもので、もし、平均の推定量のまたその Bootstrap Mean を計算して、その Jackknife SE を求めれば、当然結果は違ってきます。

Tibshirani の Bootstrap 修正 Formula と Weighted Jackknife after Bootstrap

Bootstrap の結果に対して、Tibshirani は次のような修正を加えました。すなわち、

$$\hat{SE}_{tib}(\hat{SE}_B) = \left[(1+\frac{1}{B})(\hat{SE}_B)^2 \right]^{1/2}$$

というように Bootstrap SE をルートの(Ｂ＋１)/B だけ大きする修正です。また、上で行なった Jackknife after Bootstrap においては、平均して１/ｎずつ Jackknife していくと考えましたが、実際には、それぞれの Bootstrap サンプルにはもとのサンプルのｊ番目の値が含まれていないことも、また、複数個あることも考えられる確率過程にしたがっているので、実際１/ｎよりも小さいまたは大きい割合でカットするケースもあるわけです。それを実際のｊ番目の値を取り除いた Bootstrap サンプルのそれぞれのデータの数の合計をそれぞれのｊ番目の Jackknife に対して出して、その実際の数によってウエイトしようというものです。このウエイトは計ｎ個あって、それぞれが１前後の数になります。ウエイトが１であれば、等しく 1/ｎだけの割合で Jackknife されていることになります。ここで、ウエイトをタイプ１として、Bootstrap の数だけに依存するものとして

$$w_j = \frac{B_j}{\sum_{j=1}^{n} B_j / n}, \qquad j = 1,2,...,n$$

$$\tilde{SE}_{B(j)} = w_j \hat{SE}_{B(j)}$$

として、Jackknife SE を求めるものと、

$$w_j = \frac{B_j}{\sum_{j=1}^{n} B_j / n + n}, \qquad j = 1,2,...,n$$

$$\tilde{SE}_{B(j)} = w_j \hat{SE}_{B(j)}$$

として、Bootstrap の数およびサンプルサイズｎの数に依存するタイプ２のバージョンとの２つのウエイトを考えて、それぞれのウエイトとそれに対する SE_{wjack} SE_{boot} を求めます。どちらも上の計算をもとにして、Jackkife の SE または Variance を求める公式を使うものとします。

プログラム

```
new; cls;
rndseed 911;
data={78,68,89,92,60,42,74,80,78};
call jab(data,1000);
```

```
proc(7)=jab(data,times);
    local n,thetahat,setheta,thetahat_i,i,j,index,jindex,temp;
    local bsthetahat,seb,jthetahat_ij,jthetahat_j,sejseb,jtemp,setibseb;
    local j_ij,Bj,wj,wj2,wjthetahat_j,sewjseb,wj2thetahat_j,sewj2seb;
    n=rows(data);
    /* Theta here is mean. */
    thetahat=meanc(data);
    setheta=stdc(data)/sqrt(n);
    thetahat_i=zeros(times,1);
    jthetahat_ij=zeros(times,n);
    j_ij=zeros(times,n);
    i=1;
    do while i<=times;
        index=ceil(n*rndu(n,1));
        temp=data[index];
        thetahat_i[i]=meanc(temp);
        j=1;
        do while j<=n;
            jindex=zeros(n,1);
            jindex=(index.==j);
            jtemp=delif(temp,jindex);
            jthetahat_ij[i,j]=meanc(jtemp);
            j_ij[i,j]=sumc(jindex);
            j=j+1;
        endo;
        i=i+1;
    endo;
    Bj=n*times-sumc(j_ij);
    wj=Bj/(sumc(Bj)/n);
    wj2=Bj/(sumc(Bj)/n+n);
    bsthetahat=meanc(thetahat_i);
    seb=stdc(thetahat_i);
    jthetahat_j=stdc(jthetahat_ij);
    sejseb=sqrt( ((n-1)/n)*sumc((jthetahat_j-meanc(jthetahat_j))^2) );
    wjthetahat_j=wj.*stdc(jthetahat_ij);
```

```
    setibseb=sqrt((1+1/times)*seb^2);
    sewjseb=sqrt( ((n-1)/n)*sumc((wjthetahat_j-meanc(wjthetahat_j))^2) );
    wj2thetahat_j=wj2.*stdc(jthetahat_ij);
    sewj2seb=sqrt( ((n-1)/n)*sumc((wj2thetahat_j-meanc(wj2thetahat_j))^2) );
    print/lz "     sample mean=" thetahat;
    print/lz "        sample SE=" setheta;
    print/lz "    bootstrap SE=" seb;
    print/lz " SEtib(SEboot)=" setibseb;
    print/lz "SEjack(SEboot)=" sejseb;
    print "weight";
    print "          wj(Type 1)          wj(Type 2)";;
    print wj~wj2;
    print/lz "SEwjack(SEboot)="; print sewjseb~sewj2seb;
    retp(thetahat,setheta,seb,sejseb,setibseb,sewjseb,sewj2seb);
endp;
```

画面表示

```
    sample mean= 73.444444
      sample SE= 5.0965978
  bootstrap SE= 4.6194834
 SEtib(SEboot)= 4.6217925
SEjack(SEboot)= 1.7115705
weight
       wj(Type 1)         wj(Type 2)
        1.0017500          1.0006243
       0.99875000         0.99762767
        1.0011250          1.0000000
        1.0085000          1.0073667
       0.99562500         0.99450618
       0.99987500         0.99875140
        1.0023750          1.0012486
       0.99200000         0.99088525
        1.0000000         0.99887626
SEwjack(SEboot)=
        1.6998996          1.6979894
```

Bootstrap after Boostrap

プログラム

```
new; cls;
rndseed 911;
data={78,68,89,92,60,42,74,80,78};
call doubleboot(data,100,100);

proc(5)=doubleboot(data,times1,times2);
    local n,thetahat,setheta,thetahat_i,i,j,index1,index2,temp,sebseb;
    local bsthetahat,seb,bthetahat_ij,bbthetahat_i,sebb,temptemp;
    n=rows(data);
    /* Theta here is mean. */
    thetahat=meanc(data);
    setheta=stdc(data)/sqrt(n);
    thetahat_i=zeros(times1,1);
    bthetahat_ij=zeros(times1,times2);
    i=1;
    do while i<=times1;
        index1=ceil(n*rndu(n,1));
        temp=data[index1];
        thetahat_i[i]=meanc(temp);
        j=1;
        do while j<=times2;
            index2=ceil(n*rndu(n,1));
            temptemp=temp[index2];
            bthetahat_ij[i,j]=meanc(temptemp);
            j=j+1;
        endo;
        i=i+1;
    endo;
    bsthetahat=meanc(thetahat_i);
    seb=stdc(thetahat_i);
    bbthetahat_i=meanc(bthetahat_ij');
    sebb=stdc(bbthetahat_i);
    sebseb=stdc(stdc(bthetahat_ij'));
    print/lz "     sample mean=" thetahat;
```

```
    print/lz "        sample SE=" setheta;
    print/lz "    bootstrap SE=" seb;
    print/lz "double boot SE=" sebb;
    print/lz "SEboot(SEboot)=" sebseb;
    retp(thetahat,setheta,seb,sebb,sebseb);
endp;
```

画面表示

```
    sample mean= 73.444444
      sample SE= 5.0965978
  bootstrap SE= 4.880302
double boot SE= 4.8983644
SEboot(SEboot)= 1.1733069
```

画面表示(call doubleboot(data,1000,1000);の場合　要 Full バージョン)

```
    sample mean= 73.444444
      sample SE= 5.0965978
  bootstrap SE= 4.7198154
double boot SE= 4.7252967
SEboot(SEboot)= 1.2207583
```

一番下の SEboot(SEboot)は Jackknife after Bootstrap と同じように、SE の SE を計算したものです。ただし、サンプルの全データを使うので Jackknife のようにかけ合わせてバランスする部分はありません。通常の標準偏差を計算しています。(一部の教科書では、分散および標準偏差の計算で、Ｂすなわち bootstrap サンプルの数で割る計算をするものもありますが、Ｂ－１で割っても考え方の違いなので無視することにします。）一方、その上の double boot SE としたところは、内側の bootstrap ループのそれぞれを平均をさらに平均して推定量を求め（この場合、平均)、外側の bootstrap ループでできた数だけの平均からその平均の SE を計算しています。途中までは同じですが、こちらは２重の bootstrap の計算をして、SE を求めています。内側と外側の推定量をどう扱うかによって、様々なパターンがあります。上の double boot は通常 Double Bootstrap と呼ばれているものではなく、Bootstrap after Bootstrap に相当します。次にあげるようなＰ値に関するもの、あるいは信頼区間に関する特殊なものが、Double Bootstrap Test と呼ばれます。

Double Bootstrap Test

この Double Bootstrap は、内側の bootstrap サンプルの１つ１つに対して、それぞれ外側で bootstrap することにはかわりないのですが、ここでは、あるテスト統計量（例えば、ｔ値など）を内側の bootstrap で計算して、それをもとのサンプルの統計量との大小関係を比較して、それ以上になるケースの割合を勘定してＰ値とします。この過程のそれぞれに

対して、外側の bootstrap で今度はＰ値を内側の手続きにしたがって計算したものを内側のループで既に計算されたＰ値に対して比べて、それ以下になるケースの割合を勘定して、これを Double Bootstrap によって修正されたＰ値とします。すなわち、

$$\hat{p}^* = \frac{1}{B_1}\sum_{i=1}^{B1} I(t_i^* \geq t)$$

$$\hat{p}^*_{corrected} = \frac{1}{B_2}\sum_{i=1}^{B2} I(\hat{p}_i^{**} \leq \hat{p}^*)$$

を計算させます。下のプログラムでは、一番下に t 統計量を求める procedure を置いて、これを真のμが０のときの値を計算させるようにしておきます。そして、下から２番目の procedure によって、times 回の通常の Bootstrap Ｐ値を求められるようにしておきます。これらを利用して最上階の doublepval という procedure によって、通常の Bootstrap Ｐ値を求める bootpstar に times1 回分計算させてたものを、doublepval のループで times2 回分計算させます。呼び方にもよりますが、こちらのループが外側のループに相当します。最終的に、data から求めた通常の Bootstrap Ｐ値以下の値になる Double Bootstrap Ｐ値を勘定して全体に占める割合を計算して、それを修正されたＰ値として出しています。ここで、最上階の doublepval でそれぞれ Bootstrap されたサンプルを temp として、さらにこれをその内部で bootpstar を呼び出してそのインプットとしてそれぞれのループの回のデータ temp を与えているので、結果としてそれぞれの Bootstrap 値に対する Bootstrap が計算され、Double Bootstrap が成立しています。

プログラム

```
new; cls;
rndseed 911;
n=100;
data=rndn(n,1);
print doublepval(data,199,999);

proc doublepval(data,times1,times2);
    local n,pstarstar,i,index,temp,pstar,pval;
    n=rows(data);
    pstarstar=zeros(times2,1);
    i=1;
    do while i<=times2;
        index=ceil(n*rndu(n,1));
        temp=data[index];
        pstarstar[i]=bootpstar(temp,times1);
        i=i+1;
```

```
    endo;
    pstar=bootpstar(data,times1);
    pval=(sumc(pstarstar.<=pstar)+1)/(times2+1);
    retp(pval);
endp;

proc bootpstar(data,times);
    local n,tstar,i,index,temp,t,pval;
    n=rows(data);
    tstar=zeros(times,1);
    i=1;
    do while i<=times;
        index=ceil(n*rndu(n,1));
        temp=data[index];
        tstar[i]=stat(temp);
        i=i+1;
    endo;
    t=stat(data);
    pval=(sumc(tstar.>=t)+1)/(times+1);
    retp(pval);
endp;

proc stat(data);
    local n,t;
    n=rows(data);
    t=(meanc(data)-0)/(stdc(data)/sqrt(n)); /* Here, true myu=0 . */
    retp(t);
endp;
```

画面表示

```
        0.92000000
```

μ＝０に設定してｔ値を計算させているので、Ｐ値は当然のことながら１に近い値になります。ｔ値の計算の proc stat ところを何かに変更しても、１変数の統計量であるかぎり、上の一連の計算 procedure は何にでも使えます。

各種 Bootstrap 信頼区間

ここでは、もう一度単純な Bootstrap に戻って、その信頼区間の求め方のいろいろな方法についてプログラムしておきます。簡単に整理しておくと、

Percentile	$Fb^{-1}(\alpha/2)$
Basic	$2\hat{\theta} - Fb^{-1}(1-\alpha/2)$
Normal	$\hat{\theta} - se_{boot} t_{df}$
Studentized	$\hat{\theta} - se_{boot} Ft^{-1}(1-\alpha/2)$

上の２つは BootstrapCDF の inverse、すなわち順位点によって求めます。なお、Baisc の推定量はもともとの Bias 推定量を用いてそれを２倍したものから差し引きします。反対側のテイルは足すものと考えてください。他方、下の２つは Parametric に分布から求めるものです。どちらも Bootstrap した推定量の SE をもとにしています。Normal の方はそれにかけて差し引きする t を求めるのに、もとのデータから下の場合には自由度 9‐1　CDF t のその Percentile での inverse を tval=cdftci((1-pinterval)/2,rows(data)-1);でもってあらかじめ計算したものを単純に用いています。一方、Studentized の方は Bootstrap した推定量から t を求めることになります。

プログラム

```
new; cls;
rndseed 911;
data={78,68,89,92,60,42,74,80,78};
times=2000; pinterval=0.95;
{thetahat,thetaboot,sdboot}=bootmean(data,times);
print "Theta estimate here is mean:";
print "Bias mean=" meanc(data);
print/rz pinterval*100 "% CI";
print "Percentile";
print percentile_ci(thetahat,thetaboot,pinterval);
print "Basic";
print basic_ci(thetahat,thetaboot,pinterval);
print "Normal"; tval=cdftci((1-pinterval)/2,rows(data)-1);
print normal_ci(thetahat,thetaboot,tval,pinterval);
print "Studentized(Single)";
print studentized_ci(thetahat,thetaboot,sdboot,pinterval);

proc percentile_ci(thetahat,thetaboot,pinterval);
    local alpha,times,low,up;
```

```
    alpha=1-pinterval;
    times=rows(thetaboot);
    thetaboot=sortc(thetaboot,1);
    low=thetaboot[round(times*alpha/2)];
    up=thetaboot[round(times*(1-alpha/2))];
    retp(low~up);
endp;

proc basic_ci(thetahat,thetaboot,pinterval);
    local alpha,times,low,up;
    alpha=1-pinterval;
    times=rows(thetaboot);
    thetaboot=sortc(thetaboot,1);
    low=2*thetahat-thetaboot[round(times*(1-alpha/2))];
    up=2*thetahat-thetaboot[round(times*alpha/2)];
    retp(low~up);
endp;

proc normal_ci(thetahat,thetaboot,tval,pinterval);
    local alpha,times,low,up;
    alpha=1-pinterval;
    times=rows(thetaboot);
    thetaboot=sortc(thetaboot,1);
    low=thetahat-tval*stdc(thetaboot);
    up=thetahat+tval*stdc(thetaboot);
    retp(low~up);
endp;

proc studentized_ci(thetahat,thetaboot,sdboot,pinterval);
    local alpha,times,tval,low,up;
    alpha=1-pinterval;
    times=rows(thetaboot);
    thetaboot=sortc(thetaboot,1);
    tval=(thetaboot-thetahat)./sdboot;
    tval=sortc(tval,1);
    low=thetahat-tval[round(times*(1-alpha/2))]*stdc(thetaboot);
```

```
    up=thetahat-tval[round(times*alpha/2)]*stdc(thetaboot);
    retp(low~up);
endp;

proc(3)=bootmean(data,times);
    local n,thetahat,thetaboot,sdboot,i,index,temp;
    n=rows(data);
    thetahat=meanc(data);
    thetaboot=zeros(times,1); sdboot=zeros(times,1);
    i=1;
    do while i<=times;
        index=ceil(n*rndu(n,1));
        temp=data[index];
        thetaboot[i]=meanc(temp);
        sdboot[i]=stdc(temp);
        i=i+1;
    endo;
    retp(thetahat,thetaboot,sdboot);
endp;
```

画面表示

```
Theta estimate here is mean:
Bias mean=          73.444444
                95 % CI
Percentile
        63.111111           81.666667
Basic
        65.222222           83.777778
Normal
        62.525809           84.363080
Studentized(Single)
        69.922152           77.690711
```

上の Studentized の t 値は tval=(thetaboot-thetahat)./sdboot;によって、簡易に求められているにすぎず、これは通常の Studentized ではありません。なぜなら、sdboot のところがそれぞれの Bootstrap 推定量に対するものではなくて、Bootstrap 推定量のその散らばりをその点で示しているにすぎないからです。ここで、さらに、上の Studentized 信頼区間の

計算でｔ値計算で用いる標準偏差を２重の Bootstrap で求めると次のようになります。一番下層の proc bootmean だけ２重の Bootstrap に変更して新たに sdboot2 を得てから、上と同じ proc studentized のインプット sdboot にこの計算で得られた sdboot2 を置き換えます。結果をまず示します。

画面表示

```
Studentized
          61.699578           85.544601
```

プログラム（所要数分）

```
new; cls;
rndseed 911;
data={78,68,89,92,60,42,74,80,78};
times1=2000; times2=2000; pinterval=0.95;
{thetahat,thetaboot,sdboot2}=bootmean2(data,times1,times2);
print "Studentized";
print studentized_ci(thetahat,thetaboot,sdboot2,pinterval);

proc studentized_ci(thetahat,thetaboot,sdboot,pinterval);
    local alpha,times,tval,low,up;
    alpha=1-pinterval;
    times=rows(thetaboot);
    thetaboot=sortc(thetaboot,1);
    tval=(thetaboot-thetahat)./sdboot;
    tval=sortc(tval,1);
    low=thetahat-tval[round(times*(1-alpha/2))]*stdc(thetaboot);
    up=thetahat-tval[round(times*alpha/2)]*stdc(thetaboot);
    retp(low~up);
endp;

proc(3)=bootmean2(data,times1,times2);
    local n,thetahat,thetaboot,sdboot,i,index,temp;
    local j,index2,temp2,sdboot2;
    n=rows(data);
    thetahat=meanc(data);
    thetaboot=zeros(times1,1);
    sdboot=zeros(times2,1); sdboot2=zeros(times1,1);
    i=1;
```

```
    do while i<=times1;
        index=ceil(n*rndu(n,1));
        temp=data[index];
        thetaboot[i]=meanc(temp);
        j=1;
        do while j<=times2;
            index2=ceil(n*rndu(n,1));
            temp2=temp[index2];
            sdboot[j]=meanc(temp2);
            j=j+1;
        endo;
        sdboot2[i]=stdc(sdboot);
        i=i+1;
    endo;
    retp(thetahat,thetaboot,sdboot2);
endp;
```

こちらが、正式な Studentized です。２重の Bootstrap をやっているのように見えますが実はこれはそれぞれの Bootstrap 推定量が求まる Bootstrap Sample をさらにもう一度 Bootstrap させてそれぞれに対する sdboot2 を求めてｔ値を求めているにすぎず、通常の Bootstrap と同じです。

BCa 信頼区間

ここでは、若干求め方が複雑な Bias-Corrected Accelerated 信頼区間を求めます。基本的に、BCa は Percentile の仲間で、Bootstrap データにもとづいて、その順位点をに若干の Adjust をほどこすものです。すなわち、Fb を Bootstrap CDF とすれば、

Percentile　　$Fb^{-1}(\alpha/2)$

Bca　　$Fb^{-1}\left[\Phi(z0+\frac{z0+za}{1-a(z0+za)})\right]$

というふうに、Percentile で計算したような Bootstrap の順位点を Bias と skew を修正した関数Φでもって計算します。関数Φ自体は Percentile のパラメータと同じ順位点です。関数Φの中味は少々複雑ですが、z0 は Bias-Corrected パラメータで通常は０前後の小さな値をとります。これは Bias パラメータに対して

$$\hat{\theta}_{bi} \le \hat{\theta}$$

となるBootstrap推定量を勘定して、その全体に占める割合ratioをまず求めておきます。これを標準正規分布にもとづくＺの値を可能な限り小さな-3からstep間隔でずらしていくこ

とにより、z=cdfn(z0)がこの割合ratio以下であるいっぱいいっぱいまで続けてBias修正項を求めます。

```
step=0.001;
    z=0; z0=-3;                    @ initial points @
    do while z<=ratio;
        z=cdfn(z0);
        z0=z0+step;
    endo;
```

一方、パラメータ a は Jackknife 過程から得られる skew の度合いを表す次のような

```
    a=sumc((thetahat_i-mean)^3)/(6*(sumc((thetahat_i-mean)^2))^1.5);
```

というパラメータです。また、za は標準正規分布の percentile を表します。この場合、両側 0.05 に対して（あるいは 0.975 にあたる）1.96 がきます。以下のプログラムでは、最後の procedure で今までの第 3 項を ratio 計算を返すものに変更して、あらかじめ ratio をそこで計算しておき、その上の BCa の procedure に受け渡して計算をさせています。なお、比較のため Percentile と BCa を列挙させると同じに、skew に相当する acceleration constant である項が a=0 のケースの BC も表示させています。内部で強制的に a=0 にすればこれは簡単に計算できます。３つすべてが、Bootstrap 推定量を一同に並べたインバース CDF の順位点をなにがしかの方法で求めていることになります。

プログラム

```
new; cls;
rndseed 911;
data={78,68,89,92,60,42,74,80,78};
times=2000; pinterval=0.95;
{thetahat,thetaboot,ratio}=bootmeanratio(data,times);
print "Theta estimate here is mean:";
print "Bias mean=" meanc(data);
print/rz pinterval*100 "% CI";
print "Percentile";
print percentile_ci(thetahat,thetaboot,pinterval);
print "BCa";
print bca_ci(data,thetahat,thetaboot,ratio,pinterval);

proc bca_ci(data,thetahat,thetaboot,ratio,pinterval);
    local alpha,n,times,thetahat_i,j,index,temp,mean,a;
    local za,z,z0,step,phi,phi2,low,up,sorttheta;
    alpha=1-pinterval;
```

```
    n=rows(data); times=rows(thetaboot);
/* acceleration constant by jackknife */
    thetahat_i=zeros(n,1);
    j=1;
    do while j<=n;
        index=zeros(n,1);
        index[j]=1;
        temp=delif(data,index);
        thetahat_i[j]=meanc(temp);
        j=j+1;
    endo;
    mean=meanc(thetahat_i);
    a=sumc((thetahat_i-mean)^3)/(6*(sumc((thetahat_i-mean)^2))^1.5);
/* za and z0 */
    za=cdfni(1-alpha/2);            @ Here, about za=1.96 @
    step=0.001;
    z=0; z0=-3;                      @ initial points @
    do while z<=ratio;
        z=cdfn(z0);
        z0=z0+step;
    endo;
/* adjusted percentiles and confidence interval */
    @ Activate a=0 here when BC. @
    phi=cdfn(z0+((z0+za)/(1-a*(z0+za))));
    phi2=cdfn(z0+((z0-za)/(1-a*(z0-za))));
    @ Here, 0.975=1-alpha/2=cdfn(za) when a=0 & z0=0. @
    sorttheta=sortc(thetaboot,1);
    low=sorttheta[round(phi2*times)];
    up=sorttheta[round(phi*times)];
    retp(low~up);
endp;

proc percentile_ci(thetahat,thetaboot,pinterval);
    local alpha,times,low,up;
    alpha=1-pinterval;
    times=rows(thetaboot);
```

```
    thetaboot=sortc(thetaboot,1);
    low=thetaboot[round(times*alpha/2)];
    up=thetaboot[round(times*(1-alpha/2))];
    retp(low~up);
endp;

proc(3)=bootmeanratio(data,times);
    local n,thetahat,thetaboot,i,index,temp,ratio;
    n=rows(data);
    thetahat=meanc(data);
    thetaboot=zeros(times,1);
    i=1;
    do while i<=times;
        index=ceil(n*rndu(n,1));
        temp=data[index];
        thetaboot[i]=meanc(temp);
        i=i+1;
    endo;
    ratio=sumc(thetaboot.<=thetahat)/times;
    retp(thetahat,thetaboot,ratio);
endp;
```

画面表示

```
Theta estimate here is mean:
Bias mean=          73.444444
                  95 % CI
Percentile
         63.111111          81.666667
BCa
         64.333333          82.222222
BC       (BCaのa=0を有効にしたケース)
         63.111111          81.666667
```

上の a=0 のケースの BC と Percentile は結果的に同一になっていますが、これは z0 の値が確かに 0 ではないのだが、四捨五入して順位点を求める際に順位が変更できるほどその値が大きくないためです。なお、これを確認するためには、print phi cdfn(za) z0;の 1 行を proc の内部の適当な場所に置けば、若干の違いが認識できるでしょう。なお、a と z0 を同

時に０にすれば za だけが残って、これは Percentile そのものになります。

Scale が上下何割かで変動する Bootstrap

ここでは、もともとのサンプルのデータの数ｎと Bootstrap サンプルの計算の際のデータの数はｎで等しいという厳格な仮定がありましたが、これを緩めます。今度は、このデータの数自体がｎの前後何割かの割合で変動する Multi-scale のケースを考えます。ただし簡単化のためにそれぞれのデータの数が出てくる割合は一様で同じ割合で出てくるものとします。これは、例えば今までの９個のデータからなるもともとのサンプルが１つの小さなクラスの点数であると考えれば、いままでの Bootstrap はその Bootstrap Sample も常に９個のデータが繰り返される、つまり、いつどこでそのテストをやってもクラスの人数は常に９人に固定していたのでした。これは非現実的です。そこで、もともとの考えるデータの人数は９人で変りませんが、Bootstrap する際に、必ずしも９人ではなくて、現実の世界と同じように、データの人数が前後何割かで変動するものと考えます。

プログラム

```
new; cls;
rndseed 911;
data={78,68,89,92,60,42,74,80,78};
times=2000; pp=0.05; p=0.3;            @ varying +-30% of n here @
call scaleboot(data,times,pp,p);

proc(3)=scaleboot(data,times,pp,p);
    local n,thetahat,vartheta,thetahat_i,i,index,lowerb,upperb,count,seq;
    local temp,bsthetahat,bias,corrected,bsvar,sorttheta,nn,plusminus,j;
    n=rows(data);
    /* Theta here is mean. */
    thetahat=meanc(data);
    vartheta=stdc(data)^2/n;
/* n could change here. */
    plusminus=round(p*n);
    nn=(n-plusminus)+floor((2*plusminus+1)*rndu(times,1));
    thetahat_i=zeros(times,1)~nn;
    i=1;
    do while i<=times;
```

```
        temp=zeros(nn[i],1);
        j=1;
        do while j<=nn[i];
            index=ceil(n*rndu(1,1));
            temp[j]=data[index];
            j=j+1;
        endo;
        thetahat_i[i,1]=meanc(temp);
        i=i+1;
    endo;
/* Ratio Calculation */
    print/lz "+-" p*100 "% of original n";
    print " #                        ratio";
    seq=seqa(n-plusminus,1,2*plusminus+1);
    i=1;
    do while i<=2*plusminus+1;
        count=sumc(thetahat_i[.,2].==seq[i])/times;
        print/lz seq[i]~count;
        i=i+1;
    endo;
/* Formal Calculation as in normal bootstrap */
    thetahat_i=thetahat_i[.,1];
    bsthetahat=meanc(thetahat_i);
    bias=bsthetahat-thetahat;
    corrected=thetahat-bias;
    bsvar=stdc(thetahat_i)^2;
    sorttheta=sortc(thetahat_i,1);
    lowerb=sorttheta[round(times*pp/2)];
    upperb=sorttheta[round(times*(1-pp/2))];
    print/lz "# of original cases in sample=" n;
    print/lz "       sample mean=" thetahat;
    print/lz "sample variance=" vartheta;
    print/lz " bootstrap mean=" bsthetahat;
    print/lz "                bias=" bias;
    print/lz " bias-corrected=" corrected "[ thetahat-bias ]";
    print/lz "    bootstrap var=" bsvar;
```

```
    print/rz (1-pp)*100 " % Confidence Interval";
    print/lz "[       " lowerb ",       " upperb "]";
    retp(bsthetahat,bias,bsvar);
endp;
```

プログラム自体は前章で扱った推定量を平均にとった場合をフォーマルに計算したものと基本的に同じになっています。ただし、最終インプットにスケール変動割合ｐを設定していて、これによって今までｎ個の要素の Bootstrap サンプルばかりが作成されて、それにもとづいて平均が求められていたのが、今度は、９個の前後 30%、この場合四捨五入して６から 12 までの個数に、一様に、その都度抽出されるデータの個数がなるように設定してあります。その部分が、

```
    plusminus=round(p*n);
    nn=(n-plusminus)+floor((2*plusminus+1)*rndu(times,1));
```

になります。すなわち、四捨五入値で、もともとのｎ個のｐだけがプラスマイナスの値であるとして、まず n-plusminus で下限の整数値を決めて、そこから(２×plusminus＋１)分の一様乱数を作成します。これはサイコロやランダムウオークの設定と同じです。それを Bootstrap の回数である times 回分とるために times×１の一様乱数をかけています。そして、あらかじめ thetahat_i=zeros(times,1)~nn;として times×２の領域を確保して、先に２列目に一様乱数で得られたおのおののスケール数を格納しておきます。その後のｉとｊの２重ループでは、前章の Bootstrap で最初に取り組んだように内側のループも１つ１つ回します。なぜならば、その都度スケールが変化するからです。スケールの数は列ベクトル nn のｉ番目から取り出すことにします。この場合の index は列ではなくてスケラーであって、１つ１つピックして nn[i]番目まで毎回繰り返します。こうして出来あがった temp の推定量、ここでは平均を求めて、領域確保された変数の１列目のｉ行目に格納していきます。なお、temp の列の長さはその時々のスケールの数によって変化するので、少し工夫をして、外側のループの内部の冒頭でそのたびに nn[i]行１列のゼロベクトルで初期化しています。その後の一番最後の部分の諸計算は通常の Bootstrap と同じです。その間にある Ratio の計算は直接は本題に関わりありませんが、とりあえず目に見える形で各スケールがどれだけの割合ずつ「一様に」抽出されているかを確かめています。すなわち、あらかじめ seq という最小のスケールから最大のスケールまでの数列を作っておき、そのｉ番目とスケールの番号が入っている作成した行列の２列目を比べて等しければ論理で１を等しくなければ０を返す times×１の行列を次の括弧内で作り、

```
    count=sumc(thetahat_i[.,2].==seq[i])/times;
```

その列ベクトルを列方向に合計することで等しくなっている数を勘定して最終的に times の数で割ることによって、割合を出しています。これをｉ番目からまわしてその都度スケール数と割合を表示させています。

画面表示

```
+- 30                    % of original n
 #                      ratio
 6                      0.1395
 7                      0.139
 8                      0.136
 9                      0.1475
 10                     0.1495
 11                     0.14
 12                     0.1485
# of original cases in sample= 9
      sample mean= 73.444444
sample variance= 25.975309
 bootstrap mean= 73.254432
                bias=-0.19001245
 bias-corrected= 73.634457          [ thetahat-bias ]
   bootstrap var= 24.202779
                    95   % Confidence Interval
[       62.727273          ,       82                  ]
```

なお、上のプログラムで変動の割合をｐ＝0 に設定すれば、すべての Bootstrap サンプルがｎ＝９の通常の Bootstrap と結果は等しくなります。また、上のケースは各スケールが一様に出てくる特殊なケースですが、冒頭の乱数部を変更すれば離散的な正規分布にしたがうようにすることもできます。なお、簡単化のため、上の計算はすべてのスケールのおのおのの Bootstrap サンプルを等しく扱うプロトタイプのプログラムです。

本章で取り扱った諸方法は、統計的な推定だけでなく、前章で扱ったような３つの代表的な回帰方法などにそのままあてはめることができます。上の場合はすべて１変数を扱っていますから、可能な場合には行列で一括計算するか、または、１パラメータごとに諸方法を大きなループで回せば（論理上はこちらがメモリ消費が少なくなります）簡単に計算できます。