Maximum Likelihood Applied to Binary Time Series
By Bill Menke with input from Roger Creel, December 23, 2023

Consider the binary time series \( x \) where each element \( x_i \) takes only the values 0 and 1. We consider the case where the statistics of the time series are defined by the conditional pmf \( P(x_{i+1}|x_i) \). We assert that the time series has no directional asymmetry, so that \( P(x_i|x_{i+1}) = P(x_{i+1}|x_i) \), and limited memory, in the sense that \( P(x_{i+1}|x_i, x_{i-1}, x_{i-2}, \ldots) = P(x_{i+1}|x_i) \). A simple parameterization is:

\[
\begin{array}{ccc}
  x_i \setminus x_{i+1} & 0 & 1 \\
P(x_{i+1}|x_i): & a_1 & (1 - a_1) \\
               & 1 - a_2 & a_2
\end{array}
\]

Bayes theorem can be used to calculate the joint pmf, \( P(x_i, x_{i+1}) \):

\[
P(x_i, x_{i+1}) = P(x_i|x_{i+1})P(x_{i+1}) = P(x_{i+1}|x_i)P(x_i)
\]

We rearrange to

\[
\frac{P(x_{i+1}|x_i)}{P(x_i|x_{i+1})} = \frac{P(x_{i+1})}{P(x_i)}
\]

and the sum over values of \( x_{i+1} \)

\[
D(x_i) \equiv \sum_{x_{i+1}} \frac{P(x_{i+1}|x_i)}{P(x_i|x_{i+1})} = \sum_{x_{i+1}} \frac{P(x_{i+1})}{P(x_i|x_{i+1})} = \frac{1}{P(x_i)}
\]

Substituting into (2) yields

\[
P(x_i, x_{i+1}) = P(x_{i+1}|x_i)P(x_i) = \frac{P(x_{i+1}|x_i)}{D(x_i)}
\]

In our case, \( x_i \) can take only the values (0,1), so the sum in (4) becomes

\[
D(x_i = 0) = \frac{P(x_{i+1} = 0|x_i = 0)}{P(x_i = 0|x_{i+1} = 0)} + \frac{P(x_{i+1} = 1|x_i = 0)}{P(x_i = 0|x_{i+1} = 1)} = \left( \frac{a_1}{a_1} + \frac{(1 - a_1)}{(1 - a_2)} \right)
\]

\[
D(x_i = 1) = \frac{P(x_{i+1} = 0|x_i = 1)}{P(x_i = 1|x_{i+1} = 0)} + \frac{P(x_{i+1} = 1|x_i = 1)}{P(x_i = 1|x_{i+1} = 1)} = \left( \frac{(1 - a_2)}{(1 - a_1)} + \frac{a_2}{a_2} \right)
\]
Inserting values for the conditionals yields:

\[
P(x_i, x_{i+1}) = \frac{P(x_{i+1}|x_i)}{D(x_i)}:
\]

\[
\begin{array}{ccc|ccc}
  & x_i & x_{i+1} & & & \\
 0 & a_1 & 0 & 1 & & \\
 1 & (1 - a_2) & & & \frac{a_2}{(1 - a_1) + 1} & \\
\end{array}
\]

which can be simplified to:

\[
P(x_i, x_{i+1}) = \frac{a_1(1 - a_2)}{(1 - a_1) + (1 - a_2)} \\
\frac{(1 - a_1)(1 - a_2)}{(1 - a_1) + (1 - a_2)} \\
\frac{a_2(1 - a_1)}{(1 - a_1) + (1 - a_2)}
\]

(6)

Because \(P(x_i, x_{i+1})\) is symmetrical, row sums equal columns sums and the univariate pmf’s are \(P(x_{i+1}) = P(x_i)\). Their values are

\[
P(x_i) = \sum_{x_{i+1}} P(x_i, x_{i+1}):
\]

\[
\begin{array}{ccc|ccc}
  & x_i & & & & \\
 0 & & & (1 - a_2) & & \\
 1 & & & (1 - a_1) & & \\
\end{array}
\]

(7)

This result is abbreviated as

\[
P(x_i) = \begin{cases} 
m_0 & x_i = 0 \\
m_1 & x_i = 1 
\end{cases}
\]

(8)

Note the row sum is unity, as is required for a well-posed pmf. Note also when \(a_1 = a_2\), \(m_0 = m_1 = \frac{1}{2}\) (as is required by symmetry under exchange of \(a_1\) and \(a_2\)).

The probability of a particular time series \(\mathbf{x}\) of length \(N\) can be computed by starting with the chain rule \(P(\mathbf{x}) = P(x_1)P(x_2|x_1)P(x_3|x_2, x_1)P(x_4|x_3, x_2, x_1) \cdots P(x_N|x_{N-1}, \cdots x_1)\) and applying the limited memory principle. The result can be written either left to right or right to left:
\[ P(x) = \begin{cases} P(x_1)P(x_2|x_1)P(x_3|x_2)\cdots P(x_N|x_{N-1}) \\ P(x_1|x_2)P(x_2|x_3)\cdots P(x_{N-1}|x_N)P(x_N) \end{cases} \]

(10a,b)

These two forms are equal as long as the univariate pmf \( P(x_i) \) is independent of \( i \) and the conditional pmf \( P(x_{i+1}|x_i) \) is invariant under exchange \( x_{i+1} \) and \( x_i \). Thus, (10a) can be used to compute a realization of \( x \) by first drawing \( x_1 \) from \( P(x_1) \), then drawing \( x_2 \) from \( P(x_2|x_1) \), then drawing \( x_3 \) from \( P(x_3|x_2) \), and so forth. This process is reminiscent of an AR1 process.

As the time series \((x_i, x_{i+1}, x_{i+2}, \ldots, x_{i+N-1})\) can be represented by an \( N \)-bit integer, the joint pmf can be written \( P_I(I) \), where \( I \) is the integer.

\[ P_I(I) = P(x) \]

(11)

The joint \( P_I \) for a given \( I \) can be calculated by writing \( I \) as a sequence \((b_1, \ldots, b_N)\) of 0s and 1s, initializing \( P_I = P(b_1) \) and then performing a sequence of multiplies, in which \( P_0 \) is replaced by \( P_0P(b_2|b_1) \), and that result by \( P_0P(b_3|b_2) \) and so forth.

The conditional \( A(j) = P(x_{i+j} = 1|x_i = 1) \), with \( j \geq 0 \) is reminiscent of an autocorrelation function. We note that \( A(0) = 1 \), and \( A(1) = P(x_{i+1} = 1|x_i = 1) \). The values for \( j > 1 \) can be calculated by considering every possible bit string of the form \((1, b_2, \ldots, b_{N-1}, 1)\), calculating its probability (as above) and summing the probabilities. As the bit strings \((b_2, \ldots, b_{N-1})\) are \((N-2)\)-bit integers, they can be derived from the sequence of integers \((0, \ldots, 2^{N-2} - 1)\).

Analogous algorithms can be used for \( P(x_{i+j} = 0|x_i = 0) \) and etc.

![Graph](image)

Fig 1. \( A(j) \) as a function of \( j \) for \( (a_1, a_2) = (0.6, 0.9) \), with theoretical (green); forward-stepping numerical calculation (red) and backward-stepping numerical (black) (for \( N = 20,000 \)).

For \( a_1 > \frac{1}{2} \) and \( a_2 > \frac{1}{2} \), numerical experiments show that the conditional \( P(x_{i+j} = 1|x_i = 1) \) decreases monotonically with \( j \), from unity for \( j = 0 \), and approaches \( m_1 \) as \( j \) is increased. The behavior is reminiscent of an autocorrelation function.
Assuming that the prior pdf $P_A(a_1, a_2)$ is uniform, the posterior pdf is $P(a_1, a_2|x_0) = P(x_0|a_1, a_2)$. Then the maximum likelihood estimate of $(a_1, a_2)$ is:

$$(a_1^{est}, a_2^{est}) = \underset{(a_1, a_2)}{\text{argmax}} \log P(a_1, a_2|x_0)$$

(12)

<table>
<thead>
<tr>
<th>$N$</th>
<th>(A) time series $x$</th>
<th>(B) Likelihood surface</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td><img src="image" alt="32 time series" /></td>
<td><img src="image" alt="32 likelihood surface" /></td>
</tr>
<tr>
<td>128</td>
<td><img src="image" alt="128 time series" /></td>
<td><img src="image" alt="128 likelihood surface" /></td>
</tr>
<tr>
<td>1024</td>
<td><img src="image" alt="1024 time series" /></td>
<td><img src="image" alt="1024 likelihood surface" /></td>
</tr>
</tbody>
</table>

Maximum likelihood estimation of $(a_1, a_2)$. True values are (0.50,0.75) (white), estimated values (black), $P(a_1^{est}, a_2^{est}|x_0)$ (colors).

A single $N$-element binary time series contains only $N$ bits of information. Consequently, one need fairly large values of $N$ (say $N > 100$) to achieve accurate estimates of $(a_1, a_2)$.

Suppose that we denote the number of time that $P(x_i = i)$ appears in the Eq. 10a as $N_i$ and the number of time that $P(x_i = i|x_{i+1} = j)$ appears as $N_{ij}$. Then, the likelihood of a particular $x$ is

$$P(a_1, a_2|x) = [P(0)]^{N_0} [P(1)]^{N_1} \prod_{i=1}^{N_{00}} P(0|0) \prod_{i=1}^{N_{01}} P(0|1) \prod_{i=1}^{N_{10}} P(1|0) \prod_{i=1}^{N_{11}} P(1|1)$$
The log likelihood is then
\[ L \equiv \log P(a_1, a_2|x) = N_0 \log P(0) + N_2 \log P(1) + N_{00} \log P(0|0) + N_{01} \log P(0|1) + N_{10} \log P(1|0) + N_{11} \log P(1|1) \] (13)

The partial derivative with respect to \( a_i \) is
\[ \frac{\partial L}{\partial a_i} = \frac{N_0}{P(0)} \frac{\partial P(0)}{\partial a_i} + \frac{N_2}{P(1)} \frac{\partial P(1)}{\partial a_i} + \frac{N_{00}}{P(0|0)} \frac{\partial P(0|0)}{\partial a_i} + \frac{N_{01}}{P(0|1)} \frac{\partial P(0|1)}{\partial a_i} + \frac{N_{10}}{P(1|0)} \frac{\partial P(1|0)}{\partial a_i} + \frac{N_{11}}{P(1|1)} \frac{\partial P(1|1)}{\partial a_i} \] (14)

Let \( d \equiv [(1 - a_1) + (1 - a_2)]^{-1} \). Then,
\[ d' \equiv \frac{\partial d}{\partial a_1} = \frac{\partial d}{\partial a_2} = [(1 - a_1) + (1 - a_2)]^{-2} = d^2 \] (15)

Differentiating Eq. (8) we obtain
\[ \frac{\partial P(x)}{\partial a_1}; \begin{bmatrix} 0 & 1 \\ 1 & -d + (1 - a_1)d' \end{bmatrix} \] and \[ \frac{\partial P(x)}{\partial a_2}; \begin{bmatrix} 0 & 1 \\ 1 & (1 - a_1)d' \end{bmatrix} \] (16)

And differentiating Eq. (7) we obtain
\[ \frac{\partial P(x_i|x_{i+1})}{\partial a_1}; \begin{bmatrix} x_i \backslash x_{i+1} \\ 0 & 1 & -1 \end{bmatrix} \] and \[ \frac{\partial P(x_i|x_{i+1})}{\partial a_2}; \begin{bmatrix} x_i \backslash x_{i+1} \\ 0 & 0 & 0 \end{bmatrix} \] (17)

These derivatives can be used in conjunction with the gradient descent method to determine \( (a_1^{\text{est}}, a_2^{\text{est}}) \), and their confidence intervals by estimating by sampling \( P(a_1, a_2|x_0) \) on a grid and estimating its width in the coordinate directions.

Reasonably well-tested Python functions:
def doPc(a1, a2):
# in my notation P(j|i) is Pc[i,j]
    b1 = 1.0 - a1;  # P(1|0)
    b2 = 1.0 - a2;  # P(1|0)
    Pc = np.array([[a1, b1], [b2, a2]])
    return Pc;

def doPind(Pc):
    a1 = Pc[0,0];
    b1 = 1.0 - a1;
    a2 = Pc[1,1];
    b2 = 1.0 - a2;
    Pind = np.zeros((2,1));
    d = b1 + b2;
    Pind[0,0] = b2/d;
    Pind[1,0] = b1/d;
    return Pind;

def doPjoint(Pc):
    a1 = Pc[0,0];
    b1 = 1.0 - a1;
    a2 = Pc[1,1];
    b2 = 1.0 - a2;
    Pjoint = np.zeros((2,2));
    d = b1 + b2;
    Pjoint[0,0] = a1*b2/d;
    Pjoint[0,1] = b1*b2/d;
    Pjoint[1,0] = Pjoint[0,1];
    Pjoint[1,1] = a2*b1/d;
    return Pjoint;

def doPI(Pc, Pind, I0, N):
    fmt = "{0:0%db}" % (N);
    s = fmt.format(I0);
    if( s[0] == "0" ):
        PI = Pind[0];
    else:
        PI = Pind[1];
    for j in range(1,N):
        s1 = s[j-1];
        s2 = s[j];
        if( (s1=="1") and (s2=="1") ):  # P(1|1)
            PI = PI*Pc[1,1];
        elif( (s1=="1") and (s2=="0") ):  # P(0|1)
            PI = PI*Pc[1,0];
        elif( (s1=="0") and (s2=="1") ):  # P(1|0)
            PI = PI*Pc[0,1];
        else:
            PI = PI*Pc[0,0];
        return PI;
def dologPx(Pc, Pind, x0, N):
    logPc = np.log(Pc);
    logPind = np.log(Pind);
    if( x0[0,0] == 0 ):
        logPx = logPind[0];
    else:
        logPx = logPind[1];
    for j in range(1, N):
        x1 = x0[j-1,0];
        x2 = x0[j,0];
        if( (x1==1) and (x2==1) ):    # P(1|1)
            logPx = logPx+logPc[1,1];
        elif( (x1==1) and (x2==0) ):  # P(0|1);
            logPx = logPx+logPc[1,0];
        elif( (x1==0) and (x2==1) ):  # P(1|0)
            logPx = logPx+logPc[0,1];
        else:                             # P(0|0)
            logPx = logPx+logPc[0,0];
    return logPx;

def doA(Pc, L):
    A = np.zeros((L,1));
    A[0,0] = 1.0;
    A[1,0] = Pc[1,1];
    for K in range(3, L+1):
        Km2 = K-2;
        M = int(2**Km2);
        z = 0.0
        for i in range(M):
            fmt = "{0:0%db}" % (Km2);
            s = "1" + fmt.format(i) + "1";
            zz = 1.0
            for j in range(K-1):
                s1 = s[j];
                s2 = s[j+1]
                if( (s1=="1") and (s2=="1") ):    # P(1|1)
                    zz = zz*Pc[1,1];
                elif( (s1=="1") and (s2=="0") ):  # P(0|1);
                    zz = zz*Pc[1,0];
                elif( (s1=="0") and (s2=="1") ):  # P(1|0)
                    zz = zz*Pc[0,1];
                else:                             # P(0|0)
                    zz = zz*Pc[0,0];
                z = z + zz;
        A[K-1,0]=z;
    return A;

def dorealize(Pc, Pind, N):
    x = np.zeros((N,1), dtype=int);
    r = np.random.uniform(low=0.0, high=1.0);
    myp = Pind[1,0];
    if( r <= myp ):
        x[0,0]=1;
else:
x[0,0]=0;
for i in range(1,N):
p = x[i-1,0];
myp = Pc[p,1];  # P(1|p)
r = np.random.uniform(low=0.0, high=1.0);
if( r <= myp ):
x[i,0]=1;
else:
x[i,0]=0;
return x;

def doItos(I,N):
    fmt = "{0:0%d}b" % (N);
s = fmt.format(I);
    return s;

def dostoi(s,N):
    I=0;
    for i in range(N):
        if( s[i] == "1" ):
            I = I + int(2**(N-i-1));
    return I;

def dostox(s,N):
    x = np.zeros((N,1));
    for i in range(N):
        if( s[i] == "1" ):
            x[i,0]=1;
    return x;

def doxtos(x,N):
    s = "";
    for i in range(N):
        if( x[i,0] == 0 ):
            s = s + "0";
        else:
            s = s + "1";
    return s

def Pderivs(Pc):
Pind = doPind(Pc);
a1 = Pc[0,0];
a2 = Pc[1,1];
d = 1.0/((1.0-a1) + (1.0-a2));
dp = d**2;
    # univariate
dP0dal = (1.0-a2)*dp;
dPldal = -d + (1.0-a1)*dp;
dP0da2 = -d + (1.0-a2)*dp;
dPlda2 = (1.0-a1)*dp;
dPinddal = gda_cvec( dP0dal, dPldal );
dPindda2 = gda_cvec( dP0da2, dPlda2 );
# conditional

dPcda1 = np.array([ [1.0, -1.0], [0.0, 0.0] ]);  
dPcda2 = np.array([ [0.0, 0.0], [-1.0, 1.0] ]);  
return Pind, dPindda1, dPindda2, dPcda1, dPcda2;

def doLderivs(Pc,x0,N):
    Pind, dPindda1, dPindda2, dPcda1, dPcda2 = Pderivs(Pc);  
    logPc = np.log(Pc);  
    logPind = np.log(Pind);  
    N0 = 0;  
    N1 = 0;  
    if( x0[0,0] == 0 ):  
        N0 = N0 + 1; 
    else:  
        N1 = N1 + 1; 
    N00 = 0;  
    N01 = 0;  
    N10 = 0;  
    N11 = 0;  
    for j in range(1,N):  
        x1 = x0[j-1,0];  
        x2 = x0[j,0];  
        if( (x1==1) and (x2==1) ): # P(1|1)  
            N11 = N11 + 1;  
        elif( (x1==1) and (x2==0) ): # P(0|1);  
            N01 = N01 + 1;  
        elif( (x1==0) and (x2==1) ): # P(1|0)  
            N10 = N10 + 1;  
        else: # P(0|0)  
            N00 = N00 + 1; 
    # L is log likelihood  
    L = N0*logPind[0,0] + N1*logPind[1,0];  
    L = L+N00*logPc[0,0]+N01*logPc[0,1]+N10*logPc[1,0]+N11*logPc[1,1];  
    dLda1 = (N0/Pind[0,0])*dPindda1[0,0] + (N1/Pind[1,0])*dPindda1[1,0];  
    dLda1 = dLda1 + (N00/Pc[0,0])*dPcda1[0,0] + (N01/Pc[0,1])*dPcda1[0,1];  
    dLda1 = dLda1 + (N10/Pc[1,0])*dPcda1[1,0] + (N11/Pc[1,1])*dPcda1[1,1];  
    dLda2 = (N0/Pind[0,0])*dPindda2[0,0] + (N1/Pind[1,0])*dPindda2[1,0];  
    dLda2 = dLda2 + (N00/Pc[0,0])*dPcda2[0,0] + (N01/Pc[0,1])*dPcda2[0,1];  
    dLda2 = dLda2 + (N10/Pc[1,0])*dPcda2[1,0] + (N11/Pc[1,1])*dPcda2[1,1];  
return L, dLda1, dLda2;