Linear Prediction and Levinson-Durbin Algorithm

Cedrick Collomb
http://ccollomb.free.fr/
Copyright © 2009. All Rights Reserved.

Created: February 3, 2009
Last Modified: November 12, 2009

Contents

1. Description of Linear Prediction

Given a discrete set of original values $(y_n)_{n \in [\![0,M]\!]}$ which we extend to $(y_n)_{n \in \mathbb{Z}}$ with an infinite number of zeroes, we would like to find the best k coefficients $(a_n)_{n \in [\![1,k]\!]}$ that will approximate y_n by $-\sum_{i=1}^{k} a_i y_{n-i}$. A common way to define best is to use the least-squares sense. Which means finding $(a_n)_{n \in [\![1,k]\!]}$ so that to minimize the sum of the squares of the error between the original and approximated values.

$$E = \sum_{n=-\infty}^{\infty} \left(y_n - \left(-\sum_{i=1}^{k} a_i y_{n-i} \right) \right)^2 = \sum_{n=-\infty}^{\infty} \left(y_n + \sum_{i=1}^{k} a_i y_{n-i} \right)^2$$

Defining $a_0 = 1$ gives the simpler $E = \sum_{n=-\infty}^{\infty} \left(\sum_{i=0}^{k} a_i y_{n-i} \right)^2$ which is the value we would like to minimize.

2. Minimizing the error

a. Relations between coefficients a_n

At E's minimum for $j \in [\![1,k]\!]$ we have $\frac{\partial E}{\partial a_j} = 0$. Calculating the partial derivatives of E gives $\frac{\partial \sum_{n=-\infty}^{\infty} \left(\sum_{i=0}^{k} a_i y_{n-i} \right)^2}{\partial a_j} = \sum_{n=-\infty}^{\infty} \frac{\partial \left(\sum_{i=0}^{k} a_i y_{n-i} \right)^2}{\partial a_j} = \sum_{n=-\infty}^{\infty} 2 y_{n-j} \left(\sum_{i=0}^{k} a_i y_{n-i} \right) = 0$.

Although the sum is written as infinite, it is finite since all terms vanish to zero at some point, therefore we can swap the two sum signs and get $2\sum_{i=0}^{k} a_i \sum_{n=-\infty}^{\infty} y_{n-j} y_{n-i} = 0$.

Which can be rewritten $\sum_{i=0}^{k} a_i \sum_{n=-\infty}^{\infty} y_n y_{n+j-i} = 0$.

Defining
$$R_l = \sum_{n=-\infty}^{\infty} y_n y_{n+l} \qquad (1)$$

I

t takes the final following form $\forall j \in [\![1,k]\!], \sum_{i=0}^{k} a_i R_{|j-i|} = 0$.

Which can we presented in the matrix form $MA_k = 0$ with

$$M = \begin{bmatrix} R_1 & R_0 & R_1 & \cdots & R_{k-1} \\ R_2 & R_1 & R_0 & \cdots & R_{k-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_{k-1} & R_{k-2} & \cdots & R_2 & R_1 \\ R_k & R_{k-1} & \cdots & R_1 & R_0 \end{bmatrix} \text{ and } A_k = \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix}$$

b. Solving for the coefficients a_n

The matrix M has k+1 columns and k lines. The system is not under determined, however in order to solve it, it is more convenient to make the system under a square Matrix form.

We could rewrite $MA_k = 0$ into a square system easily as below, however there is an easier and better although less direct way to solve this system.

$$\begin{bmatrix} R_0 & R_1 & \cdots & R_{k-1} \\ R_1 & R_0 & \cdots & R_{k-2} \\ \vdots & \vdots & \ddots & \vdots \\ R_{k-1} & R_{k-2} & \cdots & R_0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix} = - \begin{bmatrix} R_0 \\ R_1 \\ \vdots \\ R_{k-1} \end{bmatrix}$$

Looking at M, we can notice that M is very close to be a Toeplitz symmetric Matrix, with only the top row missing. We also notice that expending the top row would complete it into a square Matrix and system.

$$N_k A_k = \begin{bmatrix} E_k \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \text{ with } N_k = \begin{bmatrix} R_0 & R_1 & \cdots & R_k \\ R_1 & R_0 & \cdots & R_{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_k & R_{k-1} & \cdots & R_0 \end{bmatrix} \text{ and } A_k = \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix}$$

We do not know the value of E_k at that point since it is a function of A_k and the coefficients $\left(R_j\right)_{j \in [\![0;k]\!]}$.

This is a regular square linear system that we can not solve with the usual linear system solver. However this system being a Toeplitz matrix, can actually be solved better and quicker with a very simple recursive method called the Levinson-Durbin recursion.

3. Levinson-Durbin recursion

The basic simple ideas behind the recursion are first that it is easy to solve the system for $k=1$, and second that it is also very simple to solve for a $k+1$ coefficients sized problem when we have solved a for a k coefficients sized problem. In general none of the coefficients of the different sized problem match, so it is not a way to calculate a_{k+1} but a way to calculate the whole vector A_{k+1} as a function of N_{k+1}, E_k and A_k. Thinking about it Levinson-Durbin induction would be a better name.

a. Solving the size one problem

We are looking for $A_1 = \begin{bmatrix} 1 \\ a_1 \end{bmatrix}$ so that $N_1 A_1 = \begin{bmatrix} E_1 \\ 0 \end{bmatrix}$ with $N_1 = \begin{bmatrix} R_0 & R_1 \\ R_1 & R_0 \end{bmatrix}$ and E_1 is not necessary at this stage. The dot product of the second line of N_1 and A_1 gives $R_1 + R_0 a_1 = 0$, with $R_0 = \sum_{n=-\infty}^{\infty} y_n^{\ 2} > 0$.

Therefore

$$a_1 = -\frac{R_1}{R_0} \tag{2}$$

Therefore, we have found $A_1 = \begin{bmatrix} 1 \\ a_1 \end{bmatrix}$ and also

$$E_1 = R_0 + R_1 a_1 \tag{3}$$

b. Solving the size k+1 problem

Suppose that we have solved the size k problem and have found A_k, N_k and E_k. Then we have

$$\begin{bmatrix} R_0 & R_1 & \cdots & R_k \\ R_1 & R_0 & \cdots & R_{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ R_k & R_{k-1} & \cdots & R_0 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} E_k \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

N_{k+1} has one more row and column than N_k so we can not apply it directly to A_k, however if we expend A_k with a zero and call this vector U_{k+1} we can apply N_{k+1} to it and we get the following interesting result

$$\begin{bmatrix} R_0 & R_1 & \cdots & R_{k+1} \\ R_1 & R_0 & \cdots & R_k \\ \vdots & \vdots & \ddots & \vdots \\ R_{k+1} & R_k & \cdots & R_0 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_k \\ 0 \end{bmatrix} = \begin{bmatrix} E_k \\ 0 \\ 0 \\ \vdots \\ 0 \\ \sum_{j=0}^{k} a_j R_{k+1-j} \end{bmatrix}$$

Since the matrix is symmetric, we also have something remarkable when reversing the order of coefficients of U_{k+1} and calling this vector V_{k+1}.

$$\begin{bmatrix} R_0 & R_1 & \cdots & R_{k+1} \\ R_1 & R_0 & \cdots & R_k \\ \vdots & \vdots & \ddots & \vdots \\ R_{k+1} & R_k & \cdots & R_0 \end{bmatrix} \begin{bmatrix} 0 \\ a_k \\ \vdots \\ a_2 \\ a_1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sum_{j=0}^{k} a_j R_{k+1-j} \\ 0 \\ \vdots \\ 0 \\ 0 \\ E_k \end{bmatrix}$$

We can notice that a linear combination $U_{k+1} + \lambda V_{k+1}$ is of the form wanted for A_{k+1} since the first element is a 1 for all values of λ. Now if there was a value of λ for which $N_{k+1}(U_{k+1} + \lambda)$ would look like $\begin{bmatrix} E_{k+1} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$, E_{k+1} not being known at this stage, that would mean that we have found A_{k+1}.

Calculating $N_{k+1}(U_{k+1} + \lambda)$ gives

$$\begin{bmatrix} R_0 & R_1 & \cdots & R_{k+1} \\ R_1 & R_0 & \cdots & R_k \\ \vdots & \vdots & \ddots & \vdots \\ R_{k+1} & R_k & \cdots & R_0 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 + \lambda a_k \\ a_2 + \lambda a_{k-1} \\ \vdots \\ a_k + \lambda a_1 \\ \lambda \end{bmatrix} = \begin{bmatrix} E_k + \lambda \sum_{j=0}^{k} a_j R_{k+1-j} \\ 0 \\ 0 \\ \vdots \\ 0 \\ \sum_{j=0}^{k} a_j R_{k+1-j} + \lambda E_k \end{bmatrix}$$

So we just need to find λ satisfying $\sum_{j=0}^{k} a_j R_{k+1-j} + \lambda E_k = 0$ which is trivial.

Therefore $$\lambda = \frac{-\sum_{j=0}^{k} a_j R_{k+1-j}}{E_k} \tag{4}$$

And also $$A_{k+1} = U_{k+1} + \lambda V_{k+1} \tag{5}$$

Finally $$E_{k+1} = E_k + \lambda \sum_{j=0}^{k} a_j R_{k+1-j} = \left(1 - \lambda^2\right) E_k \tag{6}$$

.

c. Summary of the algorithm

- Choose m the number of coefficients wanted
- Compute all the $\left(R_j\right)_{j\in[\![0;m]\!]}$ using (1)
- Compute A_1 using (2)
- Compute E_1 using (3)
- For k from 1 to m
  - Calculate λ using (4)
  - Calculate U_{k+1}, V_{k+1}, A_{k+1} using (5)
  - Update E_{k+1} using (6)

4. Appendix. Non optimized C++ code

```
#include <math.h>
#include <vector>

using namespace std;

// Returns in vector linear prediction coefficients calculated using Levinson Durbin

void ForwardLinearPrediction( vector<double> &coeffs, const vector<double> &x )
{
    // GET SIZE FROM INPUT VECTORS
    size_t N = x.size() - 1;
    size_t m = coeffs.size();

    // INITIALIZE R WITH AUTOCORRELATION COEFFICIENTS
    vector<double> R( m + 1, 0.0 );
    for ( size_t i = 0; i <= m; i++ )
    {
        for ( size_t j = 0; j <= N - i; j++ )
        {
            R[ i ] += x[ j ] * x[ j + i ];
        }
    }

    // INITIALIZE Ak
    vector<double> Ak( m + 1, 0.0 );
    Ak[ 0 ] = 1.0;

    // INITIALIZE Ek
    double Ek = R[ 0 ];

    // LEVINSON-DURBIN RECURSION
    for ( size_t k = 0; k < m; k++ )
    {
        // COMPUTE LAMBDA
        double lambda = 0.0;
        for ( size_t j = 0; j <= k; j++ )
        {
            lambda -= Ak[ j ] * R[ k + 1 - j ];
```

```
        }
        lambda /= Ek;

        // UPDATE Ak
        for ( size_t n = 0; n <= ( k + 1 ) / 2; n++ )
        {
            double temp = Ak[ k + 1 - n ] + lambda * Ak[ n ];
            Ak[ n ] = Ak[ n ] + lambda * Ak[ k + 1 - n ];
            Ak[ k + 1 - n ] = temp;
        }

        // UPDATE Ek
      Ek *= 1.0 - lambda * lambda;
    }

    // ASSIGN COEFFICIENTS
    coeffs.assign( ++Ak.begin(), Ak.end() );
}
```

// Example program using Forward Linear Prediction

```
int main( int argc, char *argv[] )
{
     // CREATE DATA TO APPROXIMATE
     vector<double> original( 128, 0.0 );
     for ( size_t i = 0; i < original.size(); i++ )
     {
          original[ i ] = sin( i * 0.01 ) + 0.75 * sin( i * 0.03 )
                              + 0.5 * sin( i * 0.05 ) + 0.25 * sin( i * 0.11 );
     }

     // GET FORWARD LINEAR PREDICTION COEFFICIENTS
     vector<double> coeffs( 4, 0.0 );
     ForwardLinearPrediction( coeffs, original );

     // PREDICT DATA LINEARLY
     vector<double> predicted( original );
     size_t m = coeffs.size();
     for ( size_t i = m; i < predicted.size(); i++ )
     {
          predicted[ i ] = 0.0;
          for ( size_t j = 0; j < m; j++ )
          {
               predicted[ i ] -= coeffs[ j ] * original[ i - 1 - j ];
          }
     }

     // CALCULATE AND DISPLAY ERROR
     double error = 0.0;
     for ( size_t i = m; i < predicted.size(); i++ )
     {
          printf( "Index: %.2d / Original: %.6f / Predicted: %.6f\n", i, original[ i ], predicted[ i ] );
          double delta = predicted[ i ] - original[ i ];
          error += delta * delta;
     }
     printf( "Forward Linear Prediction Approximation Error: %f\n", error );

     return 0;
}
```