Carolyn Cotterman
Stat 244
April 15, 2011

Assignment 3

1 Introduction

This assignment focuses on stationary time series data. Such data consists of a sequence of observations, X_1,...X_n, measured at even time intervals, $t_1, ...t_n$. *Stationary* refers to the assumption that the joint distribution of the observations at any time $t_1, t_2, ...t_k$ is identical to the joint distribution at $t_1 + c, t_2 + c, ...t_k + c$. If we further assume that the process is Gaussian, then $E[X_t]$ and $Var(X_t)$ do not depend on t, and $Cov(X_t, X_s)$ only depends on the distance between time t and s.

In this assignment, I used the arima.sim command in R to generate sequences of observations from five different time series models, all of which are autoregressive moving average (ARMA) models. My underlying models differed not only in terms of the *values* of their parameters, but also in terms of the *number* of parameters that they contained. Using the resulting data, I then estimated the parameters (i.e., the AR component(s) and the MA component(s)) of the ARMA processes using 2 different minimization methods in C, and also using the arima command in R. Finally, I calculated the autocovariances for the first twenty observations of my time series sequences again by employing two different methods in C as well as one in R.

For this assignment, I developed original code and also altered existing code obtained from the book, "Numerical Recipes in C" (via the course website), and the Netlib library. My efforts resulted in a large number of source files, which are all referenced in "makefile." Here is a brief overview of the files necessary for reproducing my results:

- Stat244_hw3_R_code.R: All of my R code for this assignment.
- Stat244_hw3_main_v2.c: Main function for implementing this assignment's C code.
- Stat244_hw3_header1.h: My personal repertoire of functions I commonly use.
- armalk.c and as154.f: Functions for the Kalman Filter likelihood estimation.
- arma_amoeba.c: Functions for the Downhill Simplex minimization method.
- arma_powell.c, arma_linmin.c, arma_brent.c, arma_mnbrak.c, nrutil.c, nrutil.h, and arma_headers.h: Code necessary for Powell's minimization method.
- drfftb.c, drffti.c, drfftf.c, drftb1.c, drfti1.c, drftf1.c, dradb2.c, dradb3.c, dradb4.c, dradb5.c, dradf2.c, dradf3.c, dradf4.c, dradf5.c, dradfg.c, and dradbg.c: Functions for running the fast Fourier transform (and its inverse function).

The most pertinent output produced by the above code is contained in the following three files (and is also summarized in the discussion, below):

- hw3_C_out_Results.txt: Contains results of algorithms run in C.
- hw3_C_out_RunTimes.txt: Contains execution times of algorithms in C.
- hw3_R_out.txt: Contains results of R code.

2 ARMA Parameter Estimation

The autoregressive moving average (ARMA) model takes the following form:

$$X_t = \sum_{j=1}^{p} \alpha_j X_{t-j} + \sum_{j=1}^{q} \beta_j \epsilon_{t-j} + \epsilon_t \qquad \epsilon \sim N(0, \sigma^2) \text{ i.i.d.}$$

where $\epsilon_{t-j}, \epsilon_t$ are white noise error terms, α_j and β_j are the parameters of the autoregressive (AR) and moving average (MA) terms that we wish to estimate, and p and q are pre-specified parameters that refer to the orders of the AR and MA components, respectively.

2.1 Parameter Estimation Methods

Estimating the parameters of the ARMA process consists of finding the parameter values (α_js and β_js) that maximize the data's likelihood, given values for p and q. To do this, I used the Kalman filter method for calculating the following quantity, and two different minimization algorithms for minimizing it:

$$L^*(\alpha, \beta) = nlog(\sum v_t^2) + \sum_{t=1}^{n} log f_t$$

where v_t is the standardized residual for observation t, and f_t is proportional to the one-step prediction mean square error. Minimizing this quantity with respect to (α, β) is equivalent to maximizing the likelihood function.

The minimization algorithms I used were the downhill simplex method (by Nelder and Mead) and Powell's method. These methods are well-suited for the ARMA problem because they are able to minimize functions across multiple independent variables.

2.2 Parameter Estimation Results

I found that while my parameter estimation procedures were accurate when dealing with ARMA models of AR order 1 and MA order 1, none were able to recover my original parameters with reasonable accuracy for higher order models. This is a symptom of a classic problem with estimating the coefficients of correlated covariates – very different sets of parameter values can result in very similar likelihoods. Indeed, in comparing the estimated parameters resulting from the downhill simplex method to those resulting from Powell's method, I find that even though they are substantially different for higher order ARMA models, the corresponding L* values are very similar.

Unsurprsingly, both minimization algorithms take longer to complete when provided with poor initial guesses. However, I did not witness either method suffering too badly with poor initial geusses (provided these guesses were within the allowed parameter range). For example, the actual parameters for data set 5 were $(\alpha_1, \beta_1) = (.3, .3)$. Initially guessing $(.1, .1)$ for these parameters under the Powell method increased the time to complete 1,000 iterations by about 30% (from 6.2 seconds to 9.7 seconds), as compared to initially guessing $(.4, .4)$. The downhill simplex experienced similar run-time increases when poor starting vertices where chosen. Overall, run-times for both routines were in the same order of magnitude. Notably, though, the downhill simplex method generally evaluated L* many more times than did the Powell method.

3 Autocovariance using the Fast Fourier Transform

For time series data, the autocovariance is the covariance between the signal at time t and the signal at time s. If the mean signal at time t is $E[X_t] = \bar{X}_t$ (and the mean signal at time s is $E[X_s] = \bar{X}_s$), then the autocovariance is given by

$$C(t, s) = E[(X_t - \bar{X}_t)(X_s - \bar{X}_s)].$$

3.1 Autocovariance Methods

As discussed previously, our ARMA model assumes a stationary time series. Therefore, $E[X_t]$ does not depend on t, and $Cov(X_t, X_s)$ only depends on the distance between t and s (call this distance k). Thus, I can estimate the autocovariances in my simulated data using the following formula

$$C(k) = \frac{1}{N} \sum_{t=1}^{N-k} (X_t - \bar{X})(X_{t+k} - \bar{X}).$$

While I can do this for any value k=1,...,(N-1), the autocovariance estimates for large values of k will be highly variable since we have fewer and fewer measurements as k increases. Instead, I calculated the autocovariances for only k=1,...,20.

I calculated the covariances using the above formula for C(k) directly, and also using the fast Fourier transform (FFT) – an efficient algorithm used to compute the discrete Fourier transform and its inverse. This latter technique is implemented as follows:

1. Substract the mean, $\bar{X}$, from each observation. Call the resulting vector X^*.
2. Add K zeros to the end of the X^* (where K is the maximum lag desired.)
3. Calculate the fast Fourier transform of the vector created in step 2. Call this the DFT.
4. Calculate the squared modulus of DFT. (This means summing the real part squared with the complex part squared.)

5. Perform an inverse tast Fourier transform on the sequence generated in step 4, and divide each element by (N+k)*N.

Amazingly enough, the kth element of the resulting vector is the kth covariance value (i.e., the covariance between element t and element t+k of the original data vector).

3.2 Autocovariance Results

The results from my covariance calculations are contained in the output files.[1] I found the execution time to be much lower for my implementation of the basic formula than for my FFT routine in C....when k is reasonably small. In fact, it took only about 1.44 seconds to find the first 20 covariances 10,000 times when using the the basic formula, whereas it took about 67 seconds for the FFT covariance routine to complete the same task. When I doubled the desired number of covariances – to 40 – it took about twice as long (2.84 seconds) for the basic routine to finish running, as compared to when I cared only about estimating the first 20 covariances. This came as no surprise, as I surmise that the run-time for the basic formula routine is O(NK). In other words, doubling the number of observations and doubling the number of calculated covariances would both result in a doubling of the run-time for the basic routine. On average, this routine takes .000144 seconds to execute for K=20, and .000348 seconds to execute for K=50 (a little more than double).

In theroy, the FFT is an O(NlogN) operation. In other words, the FFT does *not* depend on the value of K. I witnessed this in practice: my FFT covariance routine took 66.89 seconds to complete 10,000 runs when K=20, and approximately the same amount of time (63.73 seconds) to complete 10,000 runs when K=40. This is becuase the FFT routine essentially implements the same vector manipulations regardless of the number of covariances we wish to obtain. Essentially, the only difference is that we feed the routine a slightly longer vector (i.e., we include more padded zeros) when we wish to obtain more covariances. So, for large values of K, the FFT method may be preferable in terms of total run-time.

4 Conclusion

In completing this assignment, I became better acquainted with mathematical procedures and statistical concepts with which I had previously been estranged. This assignment also pushed me to practice some new programming techniques. For example, I worked with so-called "two-dimensional arrays" (i.e., vectors of pointers) for the first time. Additionally, I was introduced to Netlib – a treasure trove of routines written by generous nerds greater than myself. Even though much of the code for this assignment was supplied, I found piecing everything together to be a very challenging exercise: I found myself wishing I were

[1] Unfortunately, I did not get the results of my FFT method implemented in C to exactly align with my results from using the basic formula, or from implementing the FFT method in R (though the latter two methods did produce compatible results). I believe that the problem lies in how the drfftb routine treats complex numbers.

a naturally more detail-oriented person, as parameters kept slipping through my fingers and one mistake would often lead to disaster (i.e., segmentation faults). But, through the struggle, I developed some new tools for staying organized: header files and make files were very useful....as was maintaining a large mug of coffee by my side.