Bayesian Dual GBTM

In this vignette we will demonstrate how to use BayesTraj to estimate a Bayesian Dual GBTM with normal likelihoods. We will use simulated data in order to verify that the estimation routine can recover the true parameters, and to demonstrate how the data should be formatted before calling the estimation routines.

Begin by loading the BayesTraj library:

```
library(BayesTraj)
```

Simulating Data

First, we will simulate data. This will not be necessary in your own projects, but it is useful both for testing the package and for using as a template for formatting your own datasets.

```
N=1000 #number of units
T1=9 #Time periods for Group 1
T2=9 #Time periods for Group 2
pi1=c(0.5,0.2,0.3) #Group 1 membership probabilities
#Transition Matrix
pi1_2=matrix(c(0.3,0.3,0.4,
               0.49,0.50,0.01,
               0.7,0.2,0.1),
             nrow=3,ncol=3,byrow=TRUE)
K1 = length(pi1) #Number of groups in series 1
K2 = dim(pi1_2)[2] #Number of groups in series 2
#Coefficients for Series 1
beta1=matrix(c(110,5,-0.5,
                111,-2,0.1,
                118,3,0.1),nrow=3,ncol=3,byrow=TRUE)
#Coefficients for Series 2
beta2=matrix(c(110,6,-0.6,
                111,-3,0.1,
                112,2,0.7),nrow=3,ncol=3,byrow=TRUE)
sigma1=2 #standard deviation of Series 1 outcomes
sigma2=4 #standard deviation of Series 2outcomes

set.seed(1)
data = gen_data_dual(N=N,
                 T1=T1,
                 T2=T2,
                 pi1=pi1,
                 pi2=pi1_2,
                 beta1=beta1,
                 beta2=beta2,
                 sigma1=sigma1,
                 sigma2=sigma2,
                 poly = 2) #degree of polynomial
```

In this example we have simulated data for 1000 paired-units with 9 time periods each, for a total of 18000

observations. While we have set each unit to have observations in 9 time periods, the estmation function below allows for the number of observations to vary. We have chosen the Group 1 membership probabilities to be 50%, 20%, and 30%. `pi1_2` is the transition matrix. `pi1_2[i,j]` represents the probability that the second pair member is in Group j conditional on the first pair member being in Group i. From this, the `gen_data_dual` function can infer that there should be three groups for both series.

Each row of the `beta` matrices defines the trajectory coefficients for the respective group. For example, the expected value at time t in Series 1 Group 1 is $110 + 5t - 0.5t^2$. Sigma1 and sigma2 define the standard deviation of the outcomes.

When calling the `gen_data_dual` function, we also specify `poly=2` in order to tell model to use a second-degree polynomial for time. If there are more non-intercept columns of `beta` than `poly`, `gen_data_dual` will generate random covariates corresponding to the remaining columns. In general, the last `poly` columns of the `beta` matrices correspond to the polynomial coefficients.

Now let's take a look at the generated data. We can unpack the individual attributes from the data object.

```
X1=data$X1
X2=data$X2
y1=data$Y1
y2=data$Y2
```

While we will restrict our discussion to the first series, everything applies to the second series as well. The first 18 rows of `X1` are:

```
print(head(X1,18))
#>       [,1] [,2] [,3]
#>  [1,]    1    1    1
#>  [2,]    1    2    4
#>  [3,]    1    3    9
#>  [4,]    1    4   16
#>  [5,]    1    5   25
#>  [6,]    1    6   36
#>  [7,]    1    7   49
#>  [8,]    1    8   64
#>  [9,]    1    9   81
#> [10,]    2    1    1
#> [11,]    2    2    4
#> [12,]    2    3    9
#> [13,]    2    4   16
#> [14,]    2    5   25
#> [15,]    2    6   36
#> [16,]    2    7   49
#> [17,]    2    8   64
#> [18,]    2    9   81
```

The first column identifies the unit. For example, the first 9 rows correspond to unit 1, the second 9 rows correspond to unit 2, and so forth. The second column is the time variable. Rows 1 and 10 correspond to time 1, rows 2 and 11 correspond to time 2, and so forth. Similarly, the third column is the square of the time column.

Now we take a look at `y1`. These are the outcomes. `y1[1]` corresponds to the outcome for unit 1 at time 1. `y1[2]` corresponds to the outcome for unit 1 at time 2, and so forth. The values of `y1` must correspond with the rows of `X1`. Therefore `X1` and `y1` should have the same length.

```
print(head(y1,18))
#>  [1] 116.1051 119.5725 119.2685 122.2980 122.5981 119.6487 121.6467
#>  [8] 115.2955 112.7368 115.9116 117.2351 120.1940 119.7062 120.4480
```

```
#> [15] 122.4963 120.2532 117.1636 112.6134
```

Estimating the model

We now turn our attention toward estimating the model. We can do this by calling the `dualtraj` function. This function uses the following weakly-informative hyperparameters: a uniform prior on group membership probabilities, an Inverse-Gamma(0.0005,0.0005) priors on the variances, and a Normal(0,λ*I) priors on the regression coeffcients, with $\lambda = 100$ as the default. A lambda parameter can be passed into the function if 100 is not sufficiently uninformative for the scale of your data.

```
iter = 5000
thin = 1
z1 = matrix(1,nrow=K1,ncol=dim(X1)[2])
z2 = matrix(1,nrow=K2,ncol=dim(X2)[2])
model = dualtraj(X1=X1, #data matrix Series 1
                 X2=X2, #data matrix Series 2
                 y1=y1, #outcomes Series 1
                 y2=y2, #outcomes Series 2
                 K1=K1, #number of groups Series 1
                 K2=K2, #number of groups Series 2
                 z1=z1, #functional form matrix Series 1
                 z2=z2, #functional form matrix Series 2
                 iterations=iter, #number of iterations
                 thin=thin, #thinning
                 dispIter=1000) #Print a message every 1000 iterations
#> [1] 1000
#> [1] 2000
#> [1] 3000
#> [1] 4000
#> [1] 5000
```

Here we run the model for 5000 MCMC iterations. In practice, more iterations may be desirable to ensure the posterior results are valid. Setting the `thin` parameter to 1 tells us to keep every sample. We can set `thin=10`, for example, to only keep 1 out of every 10 samples. Thinning is not necessary unless your computer has memory limitations. We also set `dispIter=1000` to tell the program to send us a message every 1000 MCMC iterations. This will help us monitor the progress of the program.

The only arguments we have not touched on yet is `z1` and `z2`. `z1` allows us to specify which columns of `X1` should be included in the model for each group. If `X1` has d columns, then `z1` should be a $K1 \times d$ matrix. In practice, this allows us to specify different polynomial degrees for different groups. `Z1[i,j]` indicates whether to include the jth column of `X1` in Series 1 Group i's model. The first column of `z1` corresponds to the intercept and should always be set to 1. In this example, have set `z1[i,j]=1` for each (i,j), indicating we'd like to use a second-degree polynomial for all three groups. The same applies to `z2`.

Some researchers prefer to specify different polynomial in each group. For example, a researcher who wants 2nd-degree polynomials in groups 1 and 2, but a 1-degree polynomial in group 3, could specify `z1` as follows:

```
z1_=matrix(c(1,1,1,
             1,1,1,
             1,1,0),nrow=3,ncol=3,byrow=TRUE)
```

Analyzing the Model

The model object contains the MCMC samples for each of the model's parameters. We can access the MCMC samples as follows, where each row represent an iteration of the MCMC:

```r
head(model$beta1[[1]]) #Series 1 group 1's coefficients
#>          [,1]     [,2]       [,3]
#> [1,] 112.3478 3.108181 -0.2170120
#> [2,] 111.8510 2.912952 -0.1798173
#> [3,] 111.7453 3.188596 -0.2175543
#> [4,] 111.9203 3.387903 -0.2301950
#> [5,] 111.1213 4.932923 -0.3530412
#> [6,] 110.5055 4.934091 -0.4974040
head(model$beta1[[2]]) #Series 1 group 2's coefficients
#>          [,1]     [,2]        [,3]
#> [1,] 112.4022 3.045103 -0.21070496
#> [2,] 112.2860 2.973891 -0.19491278
#> [3,] 113.0695 2.787127 -0.17839579
#> [4,] 112.8467 2.971369 -0.16868172
#> [5,] 115.5471 3.411327 -0.14463919
#> [6,] 115.6801 3.737982  0.01219868
head(model$beta1[[3]]) #Series 1 group 3's coefficients
#>          [,1]       [,2]       [,3]
#> [1,] 112.5249 3.02418836 -0.2097689
#> [2,] 111.3067 3.38181724 -0.2437130
#> [3,] 110.3313 4.26716536 -0.3524783
#> [4,] 110.6271 3.18435805 -0.3169066
#> [5,] 108.1256 2.13750993 -0.2565189
#> [6,] 111.2168 0.09689134 -0.0643746
head(model$sigma1) #Series 1 variance - NOT THE STANDARD DEVIATION
#> [1] 155.45766 152.82791 153.15341 147.74104  86.55189  30.18823
model$c1[1:6,1:10] #Series 1 unit-level group memberships
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    2    2    3    2    2    3    1    2    2     2
#> [2,]    2    2    3    3    1    1    2    2    2     1
#> [3,]    2    2    1    1    1    2    1    1    2     2
#> [4,]    2    2    1    1    1    1    3    1    2     1
#> [5,]    1    2    2    3    1    3    3    1    2     1
#> [6,]    3    3    2    3    1    3    3    2    2     1
head(model$pi1) #Series 1  group-membership probabilities
#>           [,1]      [,2]      [,3]
#> [1,] 0.3461449 0.4855474 0.1683076
#> [2,] 0.3802224 0.4493701 0.1704076
#> [3,] 0.3593258 0.4535635 0.1871108
#> [4,] 0.3853767 0.4203140 0.1943093
#> [5,] 0.2654728 0.3755130 0.3590142
#> [6,] 0.4058449 0.2853656 0.3087895
head(model$pi2) #Series 2  group-membership probabilities
#>           [,1]      [,2]      [,3]
#> [1,] 0.4924695 0.1499625 0.3575681
#> [2,] 0.4639298 0.1677923 0.3682780
#> [3,] 0.5677825 0.2472845 0.1849331
#> [4,] 0.2971823 0.2685213 0.4342964
#> [5,] 0.3059915 0.2310231 0.4629854
#> [6,] 0.3069569 0.2646378 0.4284053
model$pi1_2[1,,] #Transition probabilities from Series 1 Group 1.
#>           [,1]        [,2]       [,3]
#> [1,] 0.3300167 0.008194046 0.66178925
```

```
#> [2,] 0.7290138 0.181563670 0.08942251
#> [3,] 0.1441710 0.350360607 0.50546842
model$pi12[1,,] #Joint probability of both Series group memberships
#>             [,1]        [,2]       [,3]
#> [1,] 0.11423360 0.002836327 0.22907498
#> [2,] 0.35397080 0.088157776 0.04341887
#> [3,] 0.02426508 0.058968368 0.08507420
```

A conveniant way to summarize the posterior is with the `summary_dual` function:

```
burn = 0.9
summary = summary_dual(model,X1,X2,y1,y2,z1,z2,burn)
```

The `burn` parameter specifies the fraction of draws to keep. In this example, we keep the last 90% of MCMC samples. The first 10% are discarded as the burn-in period.

We can now print out a posterior summary to obtain the posterior mean, standard deviation, and 95% credible interval, as follows:

```
print(summary$estimates)
#>                Estimate Standard Deviation         2.5%          50%
#> beta1_1[1] 109.98522560        0.079614077 109.82931852 109.98435304
#> beta1_1[2]   4.99856998        0.036304658   4.92921045   4.99906506
#> beta1_1[3]  -0.49925197        0.003544099  -0.50639580  -0.49927573
#> beta1_2[1] 117.92717660        0.109382510 117.71664263 117.92764134
#> beta1_2[2]   3.02896668        0.050306737   2.93271228   3.02843830
#> beta1_2[3]   0.09725250        0.004879531   0.08765819   0.09728037
#> beta1_3[1] 110.87374935        0.128089519 110.61801797 110.87317267
#> beta1_3[2]  -2.00055392        0.058938508  -2.11484784  -2.00094427
#> beta1_3[3]   0.10306247        0.005791597   0.09160766   0.10303592
#> sigma1       1.42815862        0.010794454   1.40710738   1.42795255
#> pi1[1]      51.95678631        1.566742218  48.98498794  51.93541214
#> pi1[2]      28.20776019        1.406567275  25.46253030  28.20420376
#> pi1[3]      19.83545350        1.258344114  17.38386884  19.84036062
#> beta2_1[1] 110.71949117        0.143776824 110.44179338 110.71769598
#> beta2_1[2]  -2.84720039        0.066115606  -2.97928828  -2.84656551
#> beta2_1[3]   0.08474404        0.006439290   0.07245484   0.08468330
#> beta2_2[1] 111.73327287        0.157784587 111.42895646 111.73525138
#> beta2_2[2]   2.16403824        0.072657890   2.02245105   2.16338690
#> beta2_2[3]   0.68345407        0.007101689   0.66954780   0.68354287
#> beta2_3[1] 109.97534291        0.120699111 109.74614709 109.97484676
#> beta2_3[2]   5.99023455        0.055237910   5.88174352   5.99105794
#> beta2_3[3]  -0.59851786        0.005401227  -0.60896733  -0.59859001
#> sigma2       1.97964037        0.014642597   1.95094365   1.97961708
#> pi2[1]      30.91646465        1.449393983  28.06172584  30.91849346
#> pi2[2]      25.89030151        1.377086800  23.21905474  25.88194747
#> pi2[3]      43.19323384        1.567204143  40.09526602  43.19159136
#> 1->2,1->1   27.17989416        1.964947475  23.29759336  27.16573397
#> 1->2,1->2   42.40506914        2.162207533  38.19569205  42.38946143
#> 1->2,1->3   30.41503670        2.028746051  26.56702716  30.38453879
#> 1->2,2->1   19.99536296        2.371831948  15.58943694  19.92740302
#> 1->2,2->2   12.62884752        1.975492369   9.02491164  12.53601221
#> 1->2,2->3   67.37578953        2.780037416  61.75274096  67.39674998
#> 1->2,3->1   56.22353473        3.473197420  49.28180533  56.24798754
#> 1->2,3->2    1.49295899        0.843898159   0.32756274   1.34153965
```

```
#> 1->2,3->3   42.28350627        3.465220234  35.68722536  42.23387897
#> 2->1,1->1   45.67893521        2.862544965  40.20088905  45.59218221
#> 2->1,1->2   18.24681629        2.210343381  14.16169120  18.20074466
#> 2->1,1->3   36.07424850        2.738360496  30.68486634  36.10211451
#> 2->1,2->1   85.09898734        2.214687186  80.50934128  85.15657057
#> 2->1,2->2   13.75775167        2.123712659   9.82409104  13.68144202
#> 2->1,2->3    1.14326099        0.645464693   0.25761191   1.02829573
#> 2->1,3->1   36.58452494        2.308175747  32.28103424  36.54768462
#> 2->1,3->2   44.00042970        2.374328217  39.34415202  44.02084088
#> 2->1,3->3   19.41504536        1.867771882  15.85287842  19.37275383
#> 1,1         14.12233621        1.110906994  11.97934299  14.12036678
#> 1,2         22.03270194        1.306719168  19.50232766  22.01526485
#> 1,3         15.80241702        1.151507978  13.68045900  15.78503280
#> 2,1          5.64101032        0.733101696   4.29453852   5.61811422
#> 2,2          3.56174360        0.580607785   2.49596786   3.53771436
#> 2,3         19.00495732        1.226066610  16.65179705  19.00844634
#> 3,1         11.15213468        0.991876933   9.26608798  11.13087433
#> 3,2          0.29603845        0.168359456   0.06570769   0.26397172
#> 3,3          8.38666046        0.868138959   6.76503634   8.36440064
#>                   97.5%
#> beta1_1[1] 110.13886202
#> beta1_1[2]   5.06969057
#> beta1_1[3]  -0.49248620
#> beta1_2[1] 118.13832617
#> beta1_2[2]   3.12675979
#> beta1_2[3]   0.10672823
#> beta1_3[1] 111.12661988
#> beta1_3[2]  -1.88553690
#> beta1_3[3]   0.11427132
#> sigma1       1.44964518
#> pi1[1]      55.08969641
#> pi1[2]      31.00199143
#> pi1[3]      22.30989903
#> beta2_1[1] 111.01019848
#> beta2_1[2]  -2.71958268
#> beta2_1[3]   0.09752007
#> beta2_2[1] 112.03514215
#> beta2_2[2]   2.30466980
#> beta2_2[3]   0.69723003
#> beta2_3[1] 110.21038472
#> beta2_3[2]   6.09725523
#> beta2_3[3]  -0.58810254
#> sigma2       2.00800762
#> pi2[1]      33.84300720
#> pi2[2]      28.71009928
#> pi2[3]      46.27647787
#> 1->2,1->1   31.11345694
#> 1->2,1->2   46.71075996
#> 1->2,1->3   34.57425088
#> 1->2,2->1   24.89281201
#> 1->2,2->2   16.75306725
#> 1->2,2->3   72.70867359
#> 1->2,3->1   62.85477299
```

```
#> 1->2,3->2    3.56169495
#> 1->2,3->3   49.35877404
#> 2->1,1->1   51.32799654
#> 2->1,1->2   22.80208797
#> 2->1,1->3   41.54217428
#> 2->1,2->1   89.15658782
#> 2->1,2->2   18.09914548
#> 2->1,2->3    2.71734769
#> 2->1,3->1   41.10547353
#> 2->1,3->2   48.56624310
#> 2->1,3->3   23.17509563
#> 1,1         16.39284989
#> 1,2         24.63702846
#> 1,3         18.14511900
#> 2,1          7.17531405
#> 2,2          4.77481176
#> 2,3         21.44685797
#> 3,1         13.18678714
#> 3,2          0.71387623
#> 3,3         10.14271377
```

The notation for the transitions and joint probabilities requires some understanding. 1->2,i->j is $\Pr(c_2 = j|c_1 = i)$ and 2->1,i->j is $\Pr(c_1 = j|c_2 = i)$. The final columns i, j denote the joint probabilities $\Pr(c_1 = i \cap c_2 = j)$.

Checking for Label Switching and Local Modes

One issue with GBTMs is the tendency for estimation routines to find a local mode which is not globally optimal. This is a problem for GBTMs estimated using maximum likelihood as well. To increase the probability that we are in a global optimum rather than a local optimum, we often run the Gibbs sampler using several seeds and print out the likelihood at the posterior mean:

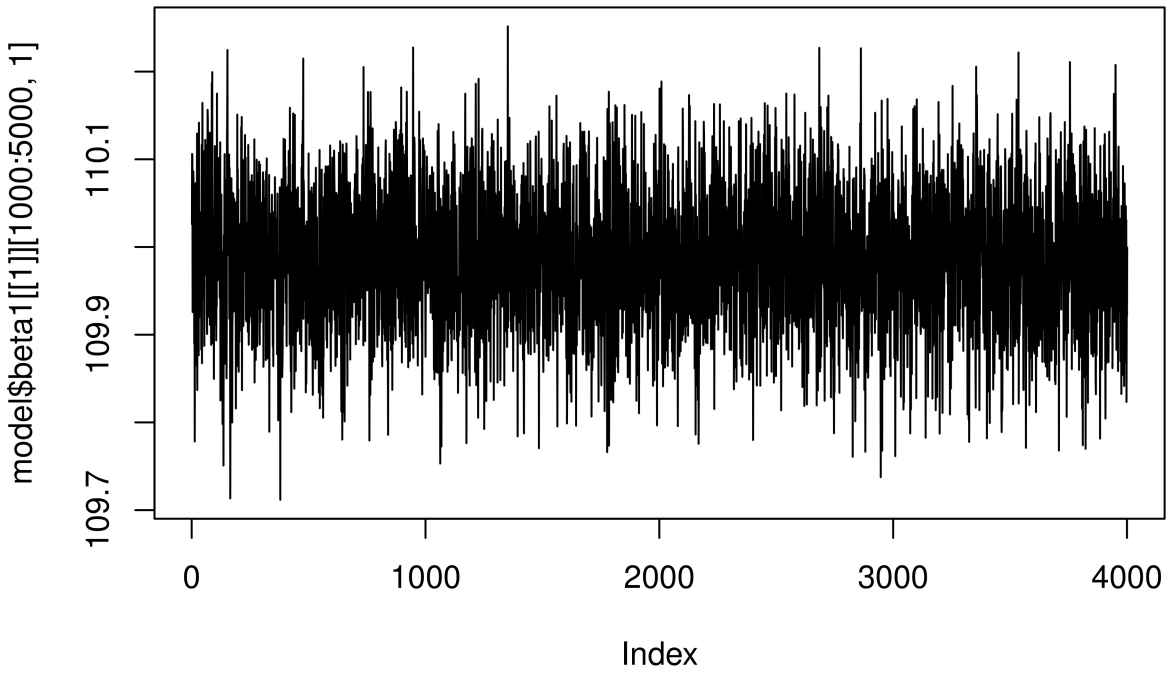
```
print(summary$log.likelihood)
#> [1] -36847.21
```

We then use the seed which maximize the likelihood. This solution has no optimality guarantees, but we have found that we can often reach better optimas this way than other existing packages using maximum likelihood.

The main drawback of the Bayesian approach is the tendency for label-switching and mode-switching. In the label-switching problem, the group labels switch in the middle of the algorithm. As a consequence, the group labeled "1" for the first 1000 draws may be labeled "2" in the second 1000 draws and vice versa. This would render and posterior summary of these coefficients meaningless. In our experience, label switching has not been a problem. However, switching between local-modes during the sampling process has occasionally been an issue.

There is no consensus for the best way to deal with label and mode switching. Either problem can be easily observed by plotting the draws sequentially and checking for sudden and sustained breaks in the trend. For example, the plot below looks consistent throughout the post-burn-in samples:

```r
plot(model$beta1[[1]][1000:5000,1],type='l')
```



This indicates that neither label-switching nor mode-siwtching occured.

If we do observe a sudden break, there are multiple possible solutions. From our experience, we usually find that re-estimating the model using a different seed will solve the problem with least amount of effort.