Bayesian GBTM

In this vignette we will demonstrate how to use BayesTraj to estimate a Bayesian GBTM with a normal likelihood. 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
T=9 #time periods
pi=c(0.5,0.2,0.3) #group membership probabilities
K = length(pi) #number of groups
#coefficients
beta=matrix(c(110,5,-0.5,
              111,-2,0.1,
              118,3,0.1),nrow=3,ncol=3,byrow=TRUE)
sigma=2 #standard deviation of outcomes

set.seed(1)
data = gen_data(N=N,
                T=T,
                pi=pi,
                beta=beta,
                sigma=sigma,
                poly = 2 #degree of polynomial
                )
```

In this example we have simulated data for 1000 units with 9 time periods each, for a total of 9000 observations. We have chosen the group-membership probabilities to be 50%, 20%, and 30%. From this, the `gen_data` function can infer that there should be three groups.

Each row of the `beta` matrix defines the trajectory coefficients. For example, the expected value at time t in Group 1 is $110 + 5t - 0.5t^2$. Sigma, defines the standard deviation of the outcomes.

When calling the `gen_data` 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` will generate random covariates corresponding to the remaining columns. In general, the last `poly` columns of the `beta` matrix 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.

```
X=data$X
y=data$Y
```

The first 18 rows of `X` are:

```
print(head(X,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 `y`. These are the outcomes. `y[1]` corresponds to the outcome for unit 1 at time 1. `y[2]` corresponds to the outcome for unit 1 at time 2, and so forth. The values of `y` must correspond with the rows of `X`. Therefore `X` and `y` should have the same length.

```
print(head(y,18))
#>  [1] 114.6093 117.5802 118.8266 122.0160 123.9023 124.2542 118.5587
#>  [8] 117.6470 116.1397 112.9243 114.4242 119.1764 120.6321 122.5672
#> [15] 121.4290 120.8274 117.4027 115.0291
```

Estimating the model

We now turn our attention toward estimating the model. We can do this by calling the `traj` function. This function uses the following weakly-informative hyperparameters: a uniform prior on group membership probabilities, an Inverse-Gamma(0.0005,0.0005) prior on the variance, and a Normal(0,λ*I) prior 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
z = matrix(1,nrow=K,ncol=dim(X)[2])
model = traj(X=X, #data matrix
             y=y, #outcomes
             K=K, #number of groups
             z=z, #functional form matrix
             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 argument we have not touched on yet is `z`. `z` allows us to specify which columns of `X` should be included in the model for each group. If `X` has d columns, then `z` should be a $K \times d$ matrix. In practice, this allows us to specify different polynomial degrees for different groups. `Z[i,j]` indicates whether to include the jth column of `X` in group i's model. The first column of `z` corresponds to the intercept and should always be set to 1. In this example, have set `z[i,j]=1` for each (i,j), indicating we'd like to use a second-degree polynomial for all three groups.

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 z as follows:

```
z_=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:

```
head(model$beta[[1]]) #group 1's coefficients
#>          [,1]     [,2]        [,3]
#> [1,] 111.9196 3.505139 -0.24404161
#> [2,] 112.6969 3.563656 -0.13229219
#> [3,] 117.5973 3.179666  0.06592185
#> [4,] 117.5597 3.207062  0.08240667
#> [5,] 117.3452 3.303807  0.07058685
#> [6,] 117.1609 3.487615  0.05161241
head(model$beta[[2]]) #group 2's coefficients
#>          [,1]     [,2]       [,3]
#> [1,] 112.4905 3.159789 -0.2205663
#> [2,] 112.4614 3.333562 -0.1736958
#> [3,] 113.6614 3.934240 -0.2647064
#> [4,] 106.0704 7.382808 -0.7362604
#> [5,] 109.9524 5.320701 -0.5354991
#> [6,] 110.2450 4.920892 -0.4977111
head(model$beta[[3]]) #group 3's coefficients
#>          [,1]      [,2]       [,3]
#> [1,] 112.3980 3.0077717 -0.2117297
#> [2,] 111.0502 3.0760145 -0.2438018
#> [3,] 110.2607 2.8519129 -0.3182364
#> [4,] 109.8328 3.3208052 -0.3651014
#> [5,] 110.0424 2.8698523 -0.3181357
#> [6,] 110.1171 0.4191075 -0.1046469
head(model$sigma) #variance - NOT THE STANDARD DEVIATION
#> [1] 151.84572 144.46687  49.30993  36.16892  34.89276  17.69164
model$c[1:6,1:10] #unit-level group memberships
```

```
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    3    3    3    3    2    3    3    3    1     3
#> [2,]    1    3    2    3    3    3    3    3    3     3
#> [3,]    3    1    1    3    3    3    3    1    1     3
#> [4,]    3    3    1    3    3    3    3    1    1     3
#> [5,]    3    3    1    3    2    3    3    1    1     3
#> [6,]    2    3    1    3    2    3    3    1    1     2
head(model$pi) #group-membership probabilities
#>           [,1]       [,2]      [,3]
#> [1,] 0.1025598 0.23972161 0.6577186
#> [2,] 0.1373852 0.20734876 0.6552660
#> [3,] 0.2847773 0.08506965 0.6301531
#> [4,] 0.2654934 0.01765258 0.7168540
#> [5,] 0.2799095 0.11079554 0.6092950
#> [6,] 0.2583544 0.43391876 0.3077269
```

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

```
burn = 0.9
summary = summary_single(model,X,y,z,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%
#> beta_1[1] 117.80613054        0.109519011 117.58752449 117.80419668
#> beta_1[2]   3.08281049        0.050427305   2.98408181   3.08380902
#> beta_1[3]   0.09257394        0.004940248   0.08291855   0.09249339
#> beta_2[1] 110.08375701        0.080716409 109.92376758 110.08534727
#> beta_2[2]   4.98817924        0.037011004   4.91606859   4.98783134
#> beta_2[3]  -0.50092401        0.003597028  -0.50818874  -0.50092371
#> beta_3[1] 110.85174235        0.130591050 110.59647034 110.85333483
#> beta_3[2]  -1.98938255        0.059184151  -2.10489837  -1.98975345
#> beta_3[3]   0.10208004        0.005771702   0.09069248   0.10209157
#> sigma       1.43412214        0.010845754   1.41304332   1.43406691
#> pi[1]      28.22730709        1.454500217  25.45228838  28.21183445
#> pi[2]      51.92959012        1.594010463  48.74290919  51.93541618
#> pi[3]      19.84310279        1.249415335  17.42586160  19.82166590
#>                 97.5%
#> beta_1[1] 118.0226768
#> beta_1[2]   3.1824795
#> beta_1[3]   0.1023227
#> beta_2[1] 110.2411201
#> beta_2[2]   5.0613396
#> beta_2[3]  -0.4939601
#> beta_3[1] 111.1085527
#> beta_3[2]  -1.8739339
#> beta_3[3]   0.1134287
#> sigma       1.4555217
#> pi[1]      31.1037223
#> pi[2]      55.0190359
```

```
#> pi[3]       22.3461472
```

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:

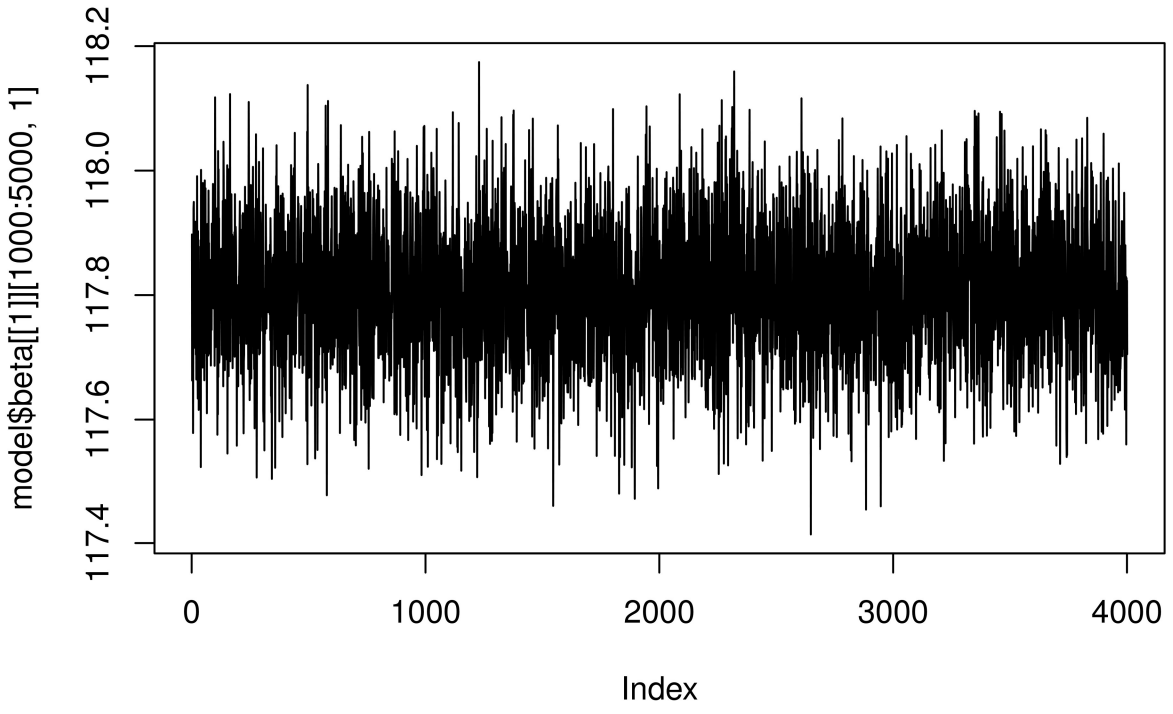
```
print(summary$log.likelihood)
#> [1] -14866.08
```

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:

```
plot(model$beta[[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.