Optimizing batting order for Little League Baseball using Bayesian statistics and cellular automation

Perry Williams

April 19, 2022

Contents

1 Methods

1.1 Statistical Model

I divided potential batting outcomes of a plate appearance into five categories. They were

$$\boldsymbol{y}_i = \begin{bmatrix} \text{strikeout}_i \\ \text{force out}_i \\ \text{single}_i \\ \text{double}_i \\ \text{triple}_i \end{bmatrix}, \tag{1}$$

where each category in the vector represents the totals of each outcome for player $i = 1, \ldots, n$ for the season. These are the 5 most common outcomes that occur and each plate appearance can be classified into one of these categories. There are usually no home runs in the league and team to which I am applying these methods. However, the model could easily be extended to account for additional batting outcomes if sufficient data exist to estimate their probabilities. To estimate the probability of each outcome for each batter, I modeled the data as

$$\begin{aligned} \boldsymbol{y}_i &\sim \text{Multinomial}(\boldsymbol{p}_i, N_i) \\ \boldsymbol{p}_i &\sim \text{Dirichlet}(\boldsymbol{\alpha_i}, K) \end{aligned} \tag{2}$$

I estimated the unknown parameter vector $\boldsymbol{p}_i$ (the probability of each batting outcome) for each player in a Bayesian framework with a custom MCMC algorithm. The vector $\boldsymbol{\alpha}_i$ is the hyper-parameter for the prior probabilities of each player. I set these hyper-parameters using previous year batting data when available, or observations during practice when the previous year data were not available. The model shown in equation 2 results in a conjugate Dirichlet posterior distribution which is easy to sample from. An example of R script to fit the model is shown below

```
library(data.table)
library(DirichletReg)

## Loading required package:  Formula
```

```r
# library(gtools)
library(readxl)

###
### Set Working Directory
###

setwd("~/Dropbox/Baseball/2022_JACK_FARM/BayesianAnalysis")

###
### Load Batting Data
###

batting=read_xlsx("BattingData.xlsx",sheet="BattingData")
game.results=read_xlsx("BattingData.xlsx",sheet="GameResults")
dt.tmp=data.table(batting)

###
### Remove players who missed games
###

ind=!is.na(dt.tmp$AB)
dt=dt.tmp[ind,]

## ID=dt[,sum(AB,na.rm=TRUE),by=playerID]£playerID
## hide identities of batters
ID=1:11

###
### At bats (AB), putouts (PO), strikeouts (K), singles (S)
### doubles (D), triples (Tr)
###
AB=dt[,sum(AB,na.rm=TRUE),by=playerID]$V1
K=dt[,sum(SO,na.rm=TRUE),by=playerID]$V1
S=dt[,sum(X1B,na.rm=TRUE),by=playerID]$V1
D=dt[,sum(X2B,na.rm=TRUE),by=playerID]$V1
Tr=dt[,sum(X3B,na.rm=TRUE),by=playerID]$V1
PO=AB-K-S-D-Tr

###
### Create respons variable y
###

y=cbind(K,PO,S,D,Tr)

###
### Required data format
###

y
```

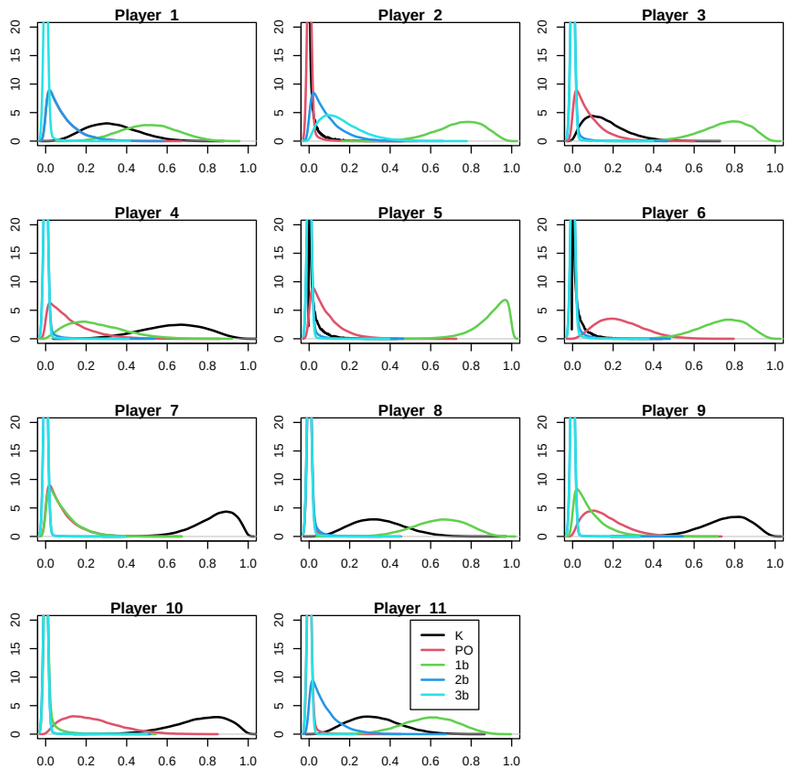
```
##        K PO  S D Tr
##  [1,]  4  1  6 1  0
##  [2,]  0  0  9 1  2
##  [3,]  2  1  9 0  0
##  [4,]  5  1  2 0  0
##  [5,]  0  1 11 0  0
##  [6,]  0  3  9 0  0
##  [7,] 10  1  1 0  0
##  [8,]  4  0  8 0  0
```

```
##  [9,]  9  2  1 0  0
## [10,]  6  2  0 0  0
## [11,]  4  0  7 1  0

###
### Estimate probabilities of each outcome for each player
###

###
### MCMC settings
###

n.iter=50000

###
### Prior information
###

K=ncol(y)
alpha=matrix(
  c(0.25,0.05,0.62,0.05,.03, # player 1
  0.15,0.05,0.65,0.1,.05,    # player 2
  0.1,0.05,0.79,0.05,.01,    # player 3
  0.64,0.06,0.25,0.04,.01,   # player 4
  0.2,0.05,0.69,0.05,.01,    # player 5
  0.22,0.1,0.57,0.08,.03,    # player 6
  0.83,0.05,0.1,0.01,.01,    # player 7
  0.45,0.05,0.4,0.08,.02,    # player 8
  0.83,0.05,0.1,0.01,.01,    # player 9
  0.83,0.05,0.1,0.01,.01,    # player 10
  0.18230, 0.04742, 0.76430, 0.00294, 0.00304)
      # player 11: strong priors  from 2021 batting data
 ,length(ID),byrow=TRUE)

###
### Book-keeping
###

p.save=array(data=NA,dim=c(n.iter,nrow(alpha),K))

###
### MCMC
###

for(k in 1:n.iter){
  p.save[k,,]=DirichletReg::rdirichlet(length(ID),alpha+y)
}
```

We can plot the probabilities of each outcome for each player

```
###
###
###

par(mfrow=c(4,3),mar=c(4,2,1,1))
for(j in 1:11){
  plot(density(p.save[,j,1]),main=paste("Player ", j),
       xlim=c(0,1),ylim=c(0,20),lwd=2,xlab="")
  for(i in 2:5){
    lines(density(p.save[,j,i],bw=.01),,col=i,lwd=2)
```

```
  }
}
legend(.5,20,c("K","PO","1b","2b","3b"),lty=1 ,col=1:5,lwd=2)
```



1.2 Cellular Automatation - simulating a little league game

Cellular automation consists of a regular grid of cells, each with a finite number of states. Each cell has a neighborhood defined relative to each cell. Each cell has an initial state at time $t = 0$. A new generation is crated according to fixed rules that determine the new state of each cell with respect to its state and neighboring states.

Baseball offense can be approximated using a simple cellular automation. There are four grid cells representing first base, second base, third base, and home plate. Home plate and second base are neighbors of third base, third base and first base are neighbors of second base, and so on. The state of of each grid cell is determined by a set of fixed rules. Many little leagues have a set of rules that vary among organizations. For example, the league described in the example I provide has 6 innings, 11 players per team who all bat each game, and a maximum of 5 runs per inning.

The rules the cellular automation can be described with variations of the following pseudo-code:

1. set inning = 1 and batter = 0
2. While inning < 6 do:
   (a) set first base = 0, second base = 0, third base = 0, and outs = 0.
   (b) While outs < 3 and runs < 5 do:
      i. batter = batter + 1, if batter = 12, batter = 1,

ii. draw a random $\boldsymbol{y}_i$ from equation 1 from the posterior predictive distribution of the model.

iii. if $y_{1,i} = 1$ do:
- batter strikes out and outs = outs + 1, go back to ii.

else go to iv.

iv. if $y_{2,i} = 1$ do:
- batter is forced out at first, outs = outs + 1,
- if third = 1, runs = runs + 1
- if second = 1, third = 1
- if first = 1, second = 1
- first = 0
- go back to ii.

else, go to v.

v. if $y_{3,i} = 1$
- batter hits single
- if third = 1, runs = runs + 1
- if second = 1, third = 1
- if first = 1, second = 1
- first = 1
- go back to ii

else, go to vi.

vi. if $y_{4,i} = 1$
- batter hits double
- if third = 1, runs = runs + 1
- if second = 1, runs = runs + 1
- if first = 1, third = 1
- first = 0
- second = 1
- go back to ii,

else, go to vii.

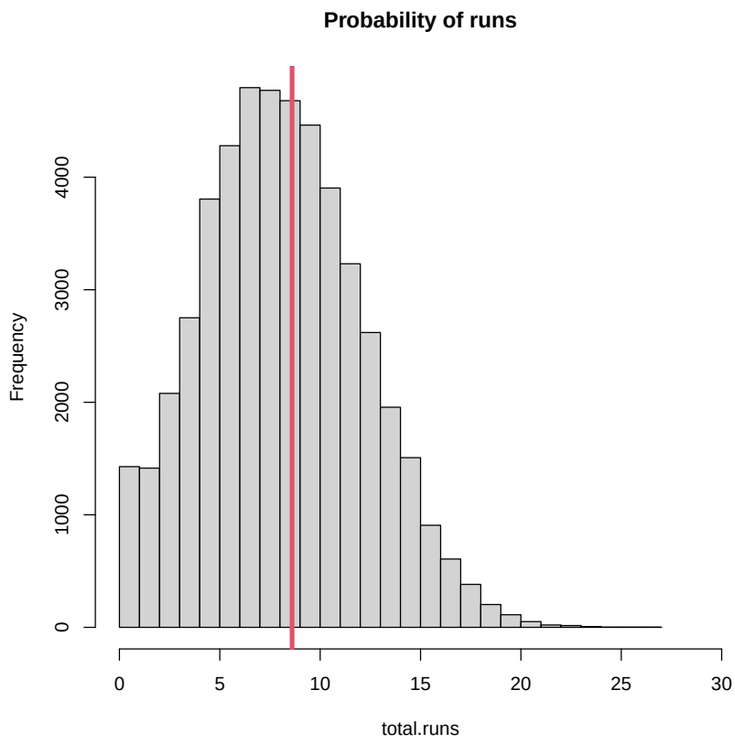
vii. if $y_{5,i} = 1$
- batter hits triple
- if third = 1, runs = runs + 1
- if second = 1, runs = runs + 1
- if first = 1, runs = runs + 1
- first = 0
- second = 0
- third = 1
- go back to ii

While there are many more than 5 outcomes that could occur during each plate appearance, the five outcomes described in these rules are the most common. For example, a force out at first base is much more common than a force out at second or third and restricting force outs to occur at first greatly simplifies model complexity, reducing the number of parameters we need to estimate.

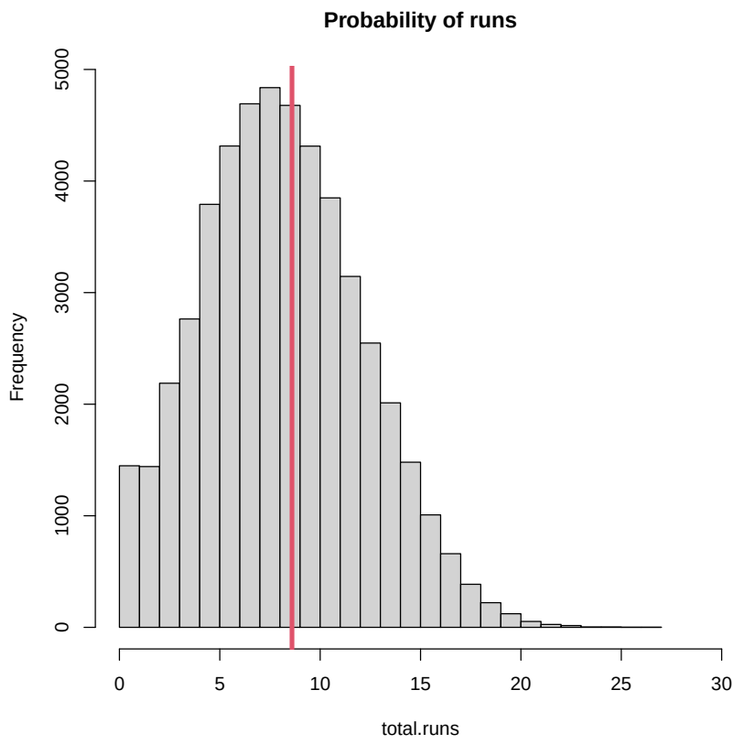
After running the cellular automation for one value sampled from the posterior predictive distribution $[\boldsymbol{y}_{\text{new}}|\boldsymbol{y}]$ we can record how many runs were scored as a derived parameter. We can then repeat the cellular automation for each $k = 1, \ldots, K$ MCMC sample from $[\boldsymbol{y}_{\text{new}}|\boldsymbol{y}]$ to estimate the probability of scoring any number of runs. Example R code of how to run the cellular automation is shown below

```
###
### Function to simulate game using the rules describe above
###

simulate.game=function(l,n.iter,perms,p.save){
  tot.innings=6
  tok=Sys.time()
  outs=0;first=0;second=0;third=0
  runs=0;i=1;inning=1;batter=1;j=1;k=1
  ab.save=array(data=0,dim=c(5,ncol(perms),tot.innings))
  ab.l=list();q.save=0
  runs.save=matrix(NA,tot.innings,n.iter)
  for(k in 1:n.iter){
    ab.save=array(data=0,dim=c(5,ncol(perms),tot.innings))
    for(inning in 1:tot.innings){
      while(outs<3&runs<5){

        ###
        ###

        ab=rmultinom(1,size=1,prob=p.save[k,perms[l,batter],])

        ##
        ##

        ab.save[,batter,inning]=ab
        if(ab[1]==1) outs=outs+1
        if(ab[2]==1){
          outs=outs+1
          if(third==1&outs<3) runs=runs+1
          if(second==1) third=1
          if(first==1) second=1
          first=0
        }
        if(ab[3]==1){
          if(third==1) runs=runs+1
          if(second==1) third=1
          if(first==1) second=1
          first=1
        }
        if(ab[4]==1){
          if(third==1) runs=runs+1
          if(second==1) runs=runs+1
          if(first==1) third=1
          second=1
          first=0
        }
        if(ab[5]==1){
          if(third==1) runs=runs+1
          if(second==1) runs=runs+1
          if(first==1) runs=runs+1
          third=1
          second=0
          first=0
        }
        i=i+1
        #print(batter)
        batter=batter+1
        if(batter==(ncol(perms)+1)) batter=1
```

```
      }
      i=1
      runs.save[inning,k]=runs
      runs=0
      inning=inning+1
      outs=0
      first=0
      second=0
      third=0
    }
    ab.l[[k]]=ab.save

  }

  tot.runs=apply(runs.save,2,sum)
  ## hist(tot.runs)
  q.save=sum(tot.runs)/n.iter
  #}
  tik=Sys.time()
  output=matrix(NA,,n.iter)
  # output[[1]]=tik-tok
  # output[[2]]=q.save
  output[1,]=tot.runs
  # output[[4]]=runs.save
  # output[[5]]=ab.l
  return(output)

}


###
### Execute the function for one lineup
### with order: player 1, player 2,...,player 11
###

perms=matrix(1:11,1,11)

output=simulate.game(1,n.iter,perms,p.save)
total.runs=output
hist(total.runs,main="Probability of runs",xlim=c(0,30),
     breaks=30)
abline(v=mean(total.runs),col=2,lwd=4)
```




```
(mean1=mean(total.runs))

## [1] 8.6

###
### How does reversing the lineup affect the probability
### of runs?
### with order: player 11, player 10,...,player 1
###

perms=matrix(11:1,1,11)

output=simulate.game(1,n.iter,perms,p.save)
total.runs=output
hist(total.runs,main="Probability of runs",xlim=c(0,30),
     breaks=30)
abline(v=mean(total.runs),col=2,lwd=4)
```




```
(mean2=mean(total.runs))

## [1] 8.6
```

We see that the mean of the posterior distribution for the number of runs for the batting order player 1, player 2, ..., player 3 is 8.6 and the mean of the posterior distribution for the total number of runs or the batting order player 11, player 10,...,player 1 is 8.6. Changing the order can give us different probabilities associated with the number of runs we expect (even if they are the same for these two examples). **Can we find a batting order that maximizes the expected number of total runs?**

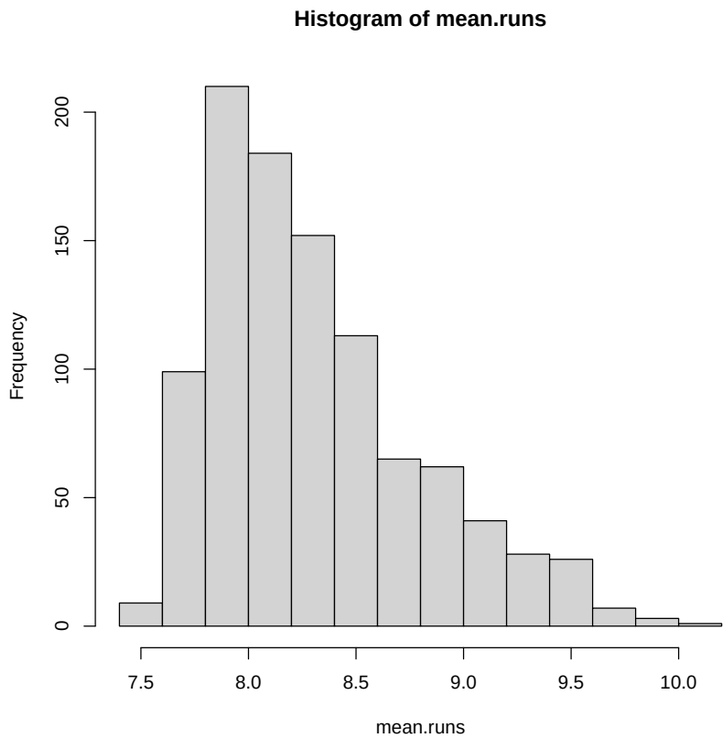
1.3 Optimizing Batting Order

To optimize the batting order we will complete the above calculations for a large number of permutations of the batting order. This is computationally expensive and I will leverage parallel processing. Here is some example R code to do this task

```
###
### Load parallel processing package
###

library(doSNOW)

###
### Permutate all possible batting orders
### (slow so I saved the results after doing it once)
###

# perms=permutations(n=length(ID),
#                    r=length(ID),
```

```
#                     v=1:length(ID),
#                     repeats.allowed=F)
# save(perms,file=paste0("~/Dropbox/Baseball/",
#                        "2022_JACK_FARM/BayesianAnalysis/",
#                        "perms.RData"))

load(paste0("~/Dropbox/Baseball/2022_JACK_FARM/",
            "BayesianAnalysis/perms.RData"))
nrow(perms) # 39.9 million ways to make the lineup

## [1] 39916800

###
### How many lineups to evaluate (can't do all 39.9 million)
###

No.lineups=1000  # about 4.6 minutes to complete

###
### setup parallel backend to use many processors
###

cl <- makeCluster(18)  # 7 cores for parallel
registerDoSNOW(cl)

###
### Optional  progress bar so we know where we're at
### in the computation
###

# pb=txtProgressBar(max=No.lineups,style=3)
# progress=function(n)setTxtProgressBar(pb,n)
# opts=list(progress=progress)

###
### randomly sample lineups from 1,...,39.9 million
###

samp=sort(sample(1:dim(perms)[1],No.lineups))

###
### Start calculations, keeping track of time
###

tok=Sys.time()
# parallel.out <- foreach(l=samp,
#                         .combine=rbind,
#                         .options.snow=opts) %dopar% {
#                           tempMatrix =
#                             simulate.game(l,n.iter,perms
#                                           ,p.save)
#   return(tempMatrix)
# }

parallel.out <- foreach(l=samp,
                        .combine=rbind) %dopar% {
                          tempMatrix =
                            simulate.game(l,n.iter,perms,
                                          p.save)
  return(tempMatrix)
```

```
}



tik=Sys.time()
tik-tok

## Time difference of 18.1 mins

###
### close backend processors
###

## close(pb)
stopCluster(cl)

###
### Calculate the mean of the posterior distribution
### of number of runs for each batting order. Some
### batting orders are better than others
###

mean.runs=apply(parallel.out,1,mean)
hist(mean.runs)
```




```
###
### What is the best lineup?
###
best.designs=which(mean.runs==max(mean.runs))
(lineup=ID[perms[best.designs,]])

##  [1]  1  2  3  4  5  6  8  7  9 11 10
```

```
runs.ppd=parallel.out[best.designs,]

###
### what are the mean number of runs to expect
###

mean(runs.ppd)

## [1] 10

###
### Credible intervals
###

quantile(runs.ppd,c(0.025,0.975))

##  2.5% 97.5%
##     3    17
```

The (locally) optimal batting order is:
player 1
player 2
player 3
player 4
player 5
player 6
player 8
player 7
player 9
player 11
player 10

Computation time for 100 lineups is approximately 5 minutes on my machine. Computation time for 5000 lineups is approximately 1.9 hours to complete. Parents, please let me know if your player won't be able to make it to the game 2 hours prior to game time ;).