Regression addendum

Carl Larson

6/28/2018

In completion of the Springboard Introduction to Data Science curriculum unit 7.4:

One big hurdle with this project is that one of the few variable user-inputs is time remaining, yet the back-end dataframe does not have a constant time remaining; instead, the index in the back-end dataframe is made baskets by either team.

The problem is that given the many random input variables in the basketball game simulator function, we don't yet have a reliable way of predicting how often a basket is scored.

Basketball games don't always have a game-action or basket at a specific time, say 5 minutes flat remaining, yet samples that have 4:59 remaining or 5:01 remaining are sufficiently functionally similar for a user at 5 minutes flat that it's legitimate to break up the game into an index so we can more easily put it into a matrix for a contour plot. It would be inefficient to throw out 99% of results because they didn't include a basket being scored at *precisely* 5.00 minutes remaining, especially because scores in basketball change so quickly, it doesn't make any sense to isolate for specific times, but rather, to group times into buckets - essentially the regression is asking how wide should those buckets be.

So in essense, rather than holding time constant for an index, this project uses made baskets as the matrix index, with a computed average-mean time over all the games for each column.

Using Linear Regression

Essentially, this is a function of many inputs via the Gameflowmaker4() function, and we are measuring all these inputs with one output which would be essentially the average rate at which baskets are scored, or, roughly, "Minutes-Left" per "GameActionIndex." This will allow us to create a relevant frame of y-values for the "score-difference" axis we will show the user in the final graph along with the program's recommendation.

This will also allow us to convert the user's "Minutes Left" into the most appropriate time-column.

During basketball games, score differences are never static for long. The index is under the control of no one single player or team.

Note On The Round Function

Ultimately, rounding up or down to an index from continuous time is a challenge. The loop index must be in integer format, but time comes from the user in continuous format. The issue is border cases where users are close to two different scenarios; whether to round those users down to a common index, or up.

The choice was made to round up, which possibly introduced bias, because some edge case users may be seeing higher probabilities than they should, because in the rounding up, the machine thinks there is more time in the game than there actually is - never by more than 30 seconds or so, as will be shown by the linear regression below.

Later in the app itself, the same value is then rounded down to the nearest integer using the floor() function, ideally to compensate slightly for the rounding bias above.

Fortunately for the sake of the app, this is distinctly linear bias. This type of bias can be corrected linearly by simply adjusting the "Confidence Interval" tab in the app, and raising it to require the maximum amount

of certainty. Later on in this project's model validation section in the main paper, we will take steps to empirically measure this linear bias.

Linear Regression From Minutes To Index

Basketball is a game that can not just go down to the second, but the tenth of a second. For this project, we just have users put in minutes and seconds, and then a global variable is formed called "Minutes Left" which decimalizes the and sums the minutes and seconds.

One issue with this project is that it has to quantize otherwise continuous time. Essentially this is like casting a net in open seas to be able to constrain a variable; the coefficients on this linear regression are the lines of the netting.

Essentially, this linear regression allows us to take "Minutes Remaining" formatted time data and then quickly and reliably convert it to "Index" formatted time data. This allows the loop to roughly understand where the user is in the map.

First we will build a few essential functions that comprise the heart of this proejct:

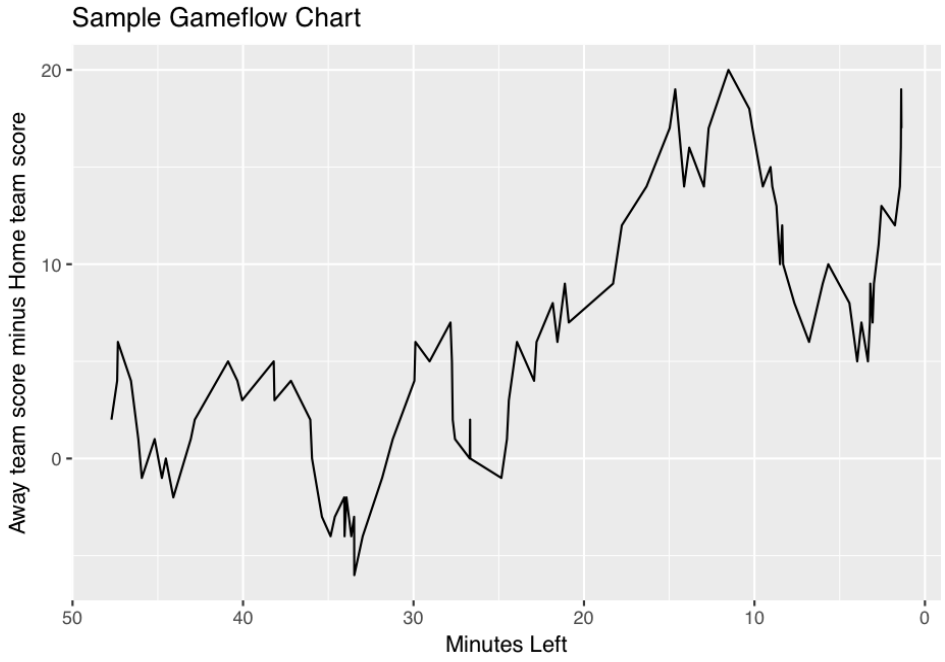
1. GameflowMaker4 - the heart of the simulator, this function has only optional arguments and outputs a simulated basketball gameflow chart, with 3 columns, time remaining, score of team 1, and score of team 2.
2. algo2app - This function loops GameflowMaker4. Taking the argument h which stands for the h-th to last basket made by either team in the game, and the argument p which stands for score difference, this function is capable of looping "m" number of games matching that criteria. Outputs of this function are a.) a repeat of the index b.) the percent of times the leading team actually won in this scenario c.) the average time-remaining for the h-th basket-made, and d.) the average number of baskets made in each game.
3. GFMapper - This function uses algo2app's outputs to create a graph where score difference is held constant, and the leading team's win percentage is plotted out for the time between h and 0, creating a chart similar to others seen in the literature of this field.

These 3 functions are essentially the heart, bones and muscle of the project behind the app Hooply, as we use regression to better understand and measure this particular basketball simulation, so as to quantitatively compare it to others.

```
# A crude but effective basketball simulator

GameflowMaker4 <- function(
  probA = c(abs(rnorm(1, 0.65, 0.005)),
            abs(rnorm(1, 0.09, 0.002)),
            abs(rnorm(1, 0.17, 0.005)),
            abs(rnorm(1, 0.09, 0.005))),
  probB = c(abs(rnorm(1, 0.64, 0.005)),
            abs(rnorm(1, 0.10, 0.003)),
            abs(rnorm(1, 0.17, 0.005)),
            abs(rnorm(1, 0.09, 0.005))),
  n = as.integer(abs(rnorm(1, 218, 22))),   #ad hoc game-length index variable
  dt = 0.005,   #min time between scores to allow for physics
  GL = 48){   #game length of 48 minutes per NBA rules
  sta = sample(0:3, n, replace=TRUE, prob=probA)
  stb = sample(0:3, n, replace=TRUE, prob=probB)
  w = which(sta > 0 & stb > 0)
  sta[w] = 0
  stb[w] = 0
```

```
  v = which(sta == 0 & stb == 0)
  sta <- sta[-c(v)]
  stb <- stb[-c(v)]
  m <- length(sta)
  listA <- as.vector(seq(from=0, to=((m * dt)-dt), by=dt))
  listB <- as.vector(sort(runif(m, 0, c(GL-listA[m]))))
  MinLeft <- listA + listB
  MinLeft <- rev(MinLeft)
  scoreA <- cumsum(sta)
  scoreB <- cumsum(stb)
  df = data.table(MinLeft, scoreA, scoreB)
  if(df$scoreA[nrow(df)] == df$scoreB[nrow(df)])
  df = df[-c(nrow(df))]
  return(df)
}
df <- GameflowMaker4()
#plot(df$MinLeft, (df$scoreA - df$scoreB), type = "b", main = "Sample Gameflow Chart",
     #xlab = "Minutes Left", ylab = "Away team score minus Home team score")
ggplot(df, aes(x=df$MinLeft, y=(df$scoreA-df$scoreB))) +
  geom_line() +
  labs(y="Away team score minus Home team score", x="Minutes Left") +
  ggtitle("Sample Gameflow Chart") +
  scale_x_reverse()
```



Looping Batches of Games

To explain the below function in more detail, "m" is essentially a "focus" meter; the eventual graph of probabilities might be out of focus with a lower "m" value like 200 (which requires less computing power) and would be in, essentially, greater focus with an "m" value more like 1000, meaning, 1000 game-flow charts were used per game action index (the game action index literally being only a made basket by either team, but also acting as an index counting down the final relevant "basketball-actions" allowing other meaningful variables to be in the same row, such as time, which eventually creates the x-axis, and percent chance, or the y-axis).

At this juncture, there was a need for the function "algo2app," which was tailored to output key data such as who was leading by what at about which time in the game, and then who won that game? This was required to know the leading team's win percentage out of a certain scenario. With m = 900 simulations per scenario you can start to see some very smooth contour lines, but the default data was turned down to 90 to lessen computing time, in case someone wants to casually run the below snippet without changing any figures, the code should run fairly quickly.

```
#"h" here (for "hoop" literally since made-baskets create this index) controls
#the index of the function, defaulting to 30 or the 30th-to-last
#made basket by either team, allowing for easy looping later.
algo2app <- function(h=30, p=scdiff){
  m=90
  mec <- matrix(data = 0, nrow = 1, ncol = 5)
  for(i in 1:m){
  df <- GameflowMaker4()
  q <- (nrow(df) - h)
  mec[1,4] <-  mec[1,4] + as.numeric(df$MinLeft[q])
  mec[1,5] <- mec[1,5] + as.numeric(nrow(df))
  if(((df$scoreA[q]-p) > df$scoreB[q]) && (df$scoreA[nrow(df)] > df$scoreB[nrow(df)])) {
  mec[1,1] <- mec[1,1] + 1
  next} #team A led at q and eventually won

  else if(((df$scoreB[q]-p) > df$scoreA[q]) && (df$scoreB[nrow(df)] > df$scoreA[nrow(df)])){
  mec[1,1] <- mec[1,1] + 1
  next} #team B led at q and eventually won

  else if(((df$scoreA[q]-p) > df$scoreB[q]) && (df$scoreB[nrow(df)] > (df$scoreA[nrow(df)]))){
  mec[1,2] <- mec[1,2] + 1
  next} #team A led at q but team B came back and won

  else if(((df$scoreB[q]-p) > df$scoreA[q]) && (df$scoreA[nrow(df)] > df$scoreB[nrow(df)])){
  mec[1,2] <- mec[1,2] + 1
  next} #team B led at q but team A came back and won

    else{mec[1,3] <- mec[1,3] + 1 #functional "q-ties" irrelevant
      next}
if(i==n)
  break}
  mec2 <- matrix(data=0, nrow = 1, ncol = 4)
  mec2[1,2] <- (mec[1,1]/(mec[1,1]+mec[1,2]))
  mec2[1,1] <- h
  mec2[1,3] <- (mec[1,4] / m)
  mec2[1,4] <- (mec[1,5] / m)
  colnames(mec2) <- c("Game_Action_index", "Leader_win_chance:", "Avg_sample_MinsLeft", "avg_Game_Actio
  return(mec2)
```

```
}
#Below is a couple of sample runs of this function as an example.
algo2app()
```

```
##      Game_Action_index Leader_win_chance: Avg_sample_MinsLeft
## [1,]                30          0.9814815            15.11304
##      avg_Game_Actions
## [1,]         100.5667
```

```
algo2app(h=10)
```

```
##      Game_Action_index Leader_win_chance: Avg_sample_MinsLeft
## [1,]                10          0.9661017            5.416969
##      avg_Game_Actions
## [1,]             96.1
```

The above function "algo2app" was one of the first functions used in this project to usefully reduce the dimensions of potentially thousands or millions of game-flow chart-style basketball game simulations.

```
#This runs for the argument "k" game-action index down to 1 creating a dataframe
#of row number k and column number 4. Basically expands the above function into a loop,
#nothing too fancy but very needed.
GFmapper <- function(k=35){
  matx <- matrix(data = 0, nrow = k, ncol = 4)
  h <- k
  for(i in 1:k){
    h <- h
    dfJ <- as.matrix(algo2app(h=h))
    matx[i,1] <- dfJ[1,1]
    matx[i,2] <- dfJ[1,2]
    matx[i,3] <- dfJ[1,3]
    matx[i,4] <- dfJ[1,4]
    h <- h - 1
    if(h == 0){
      break
      }else{
        i <- i+ 1
        next}}
  return(matx)
}
```

The Y-vector below is actually just the values 1 through k, in this case K is set to 35.

To increase the number of simulations run in this regression, refer up to algo2app and increase m.

```
#Goal here is to run linear regression and find the slope (or calculus derivative)
#of the two variables Avg_sample_MinsLeft and game_action_index.
#This will allow us to later graph relevant ranges for score differences

k <- 35
df <- as.data.frame(GFmapper(k=k))
colnames(df) <- c("Game_Action_index", "Leader_win_chance:", "Avg_sample_MinsLeft", "avg_Game_Actions")

X <- df[1:k, 3] #Minutes left
Y <- df[1:k, 1] #output game action guess

LinReg <- lm(X ~ Y, data=df)
```

```
print(LinReg)
```

```
##
## Call:
## lm(formula = X ~ Y, data = df)
##
## Coefficients:
## (Intercept)            Y
##      0.4766       0.4880
```

This shows us the line-of-best-fit that was drawn through the average minutes-remaining values at each of the 35 final baskets made.

This function runs the algo2app dimension reducing function on the GameflowMaker4 simulator. Recall that in algo2app, m was the number of simulations to be run per quantized unit of time. This means "m" is essentially a meter for "focus," allowing us to increase the focus of our data on the "true" values, at the cost of increased computing time.

Running this at a focus value of m = 200 yielded a slope of 0.4835, which is right on target for what would be expected; teams scored a basket on average every 0.4835 minutes (or 29.01 seconds), which is roughly 2 baskets a minute, or one basket for each team on average, or about 2 points per minute per team.

```
#to find the slope (or calculus derivative) of time vs index
k <- 35
df <- as.data.frame(GFmapper(k=k))
colnames(df) <- c("Game_Action_index", "Leader_win_chance:", "Avg_sample_MinsLeft", "avg_Game_Actions")

X <- df[1:k, 3] #Minutes left
Y <- df[1:k, 1] #output game action guess

LinReg <- lm(X ~ Y, data=df)

print(LinReg)
```

```
##
## Call:
## lm(formula = X ~ Y, data = df)
##
## Coefficients:
## (Intercept)            Y
##      0.4361       0.4887
```

We can also increase the focus and run an m-value of m = 1000: this yielded a slope of 0.4847, just a hair above the lower-focus value; also it had a y-intercept essentially representing the final basket of the game of roughly the same size as the slope. I will go with the higher-focus value of m=1000, meaning, one thousand game-flow simulations were ran for each game-action-index, of which there were 35 in total across the x-axis, and if you run this yourself you'll see it takes much longer to compute 35,000 game-flow charts than 7,000 (more than 5x as long).

Game_Action_Index = round(MinzLeft * 0.4847 + 0.4733)

With this equation, we can now convert the user's time remaining to the index of the matrix.

We'd be able to multiply the total number of game actions by this equation and get the computable game-flow action the user meant by the amount of minutes remaining. Again, by using "round" here there has to be some bias introduced, one way or the other, to get the user's continuous time to a specific pre-computed game index.

This shows us the line-of-best-fit that was drawn through the average minutes-remaining values at each of the 35 final baskets made.

This function runs the algo2app dimension reducing function on the GameflowMaker4 simulator. Recall that in algo2app, m was the number of simulations to be run per quantized unit of time. This means "m" is essentially a meter for "focus," allowing us to increase the focus of our data on the "true" values, at the cost of increased computing time.

Running this at a focus value of m = 200 yielded a slope of 0.4835, which is right on target for what would be expected; teams scored a basket on average every 0.4835 minutes (or 29.01 seconds), which is roughly 2 baskets a minute, or one basket for each team on average, or about 2 points per minute per team.

We can also increase the focus and run an m-value of m = 1000: this yielded a slope of 0.4847, just a hair above the lower-focus value; also it had a y-intercept essentially representing the final basket of the game of roughly the same size as the slope. I will go with the higher-focus value of m=1000, meaning, one thousand game-flow simulations were ran for each game-action-index, of which there were 35 in total across the x-axis, and if you run this yourself you'll see it takes much longer to compute 35,000 game-flow charts than 7,000 (more than 5x as long).

Game_Action_Index = round(MinzLeft * 0.4847 + 0.4733)

This allows us to convert it to the nearest game-flow action. We'd be able to multiply the total number of game actions by this equation and get the computable game-flow action the user meant by the amount of minutes remaining. Again, by using "round" here there has to be some bias introduced, one way or the other, to get the user's continuous time to a specific pre-computed game index.

Ultimately, this is a supervised linear regression analysis to get a back-end view of the average total output of the Gameflowmaker4 basketball game simulator. This analysis was used in furtherance of the development of the Hooply application in allowing Hooply to easily and reliably convert user-compatible time-information to machine-compatible time-information.