Additional guidance.

For Stat 847 Final Project

Data anomaly Guidance

If a game seems too strange to be true (200901 runs, 24 wickets, no league, two copies of the same game, more than two teams in a game, !(MS Dhoni showing anything less than excellent sportsmanship)[https://www.youtube.com/watch?v=uBIUzYFVnmE]), feel free to either ignore the NA values (e.g. for missing batter / bowler / fielder IDs), just remove and ignore that game. There are enough matches that it shouldn't change your results much in any meaningful way.

Clustering Guidance

The `ddply` function in the `plyr` package is used in the LinkedIn profile example like so:

(Week 06-1)

```
df_profiles = ddply(df_jobs, "m_urn", summarize,
    avg_n_pos_per_prev_tenure = mean(avg_n_pos_per_prev_tenure),
    avg_pos_len = mean(avg_pos_len),
    avg_prev_tenure_len = mean(avg_prev_tenure_len),
    n_pos = sum(n_pos),
    tenure_len = sum(tenure_len),
    age = age[1],
    ...
```

Here...

- `df_jobs` is the input dataframe in which one row represents one job, but multiple rows can fit a single profile.
- `df_profiles` is the output dataframe in which one row represents one profile.

So how we merge multiple job rows in one profile row in an intelligent way?

We need an index variable, in this case `m_urn`, where each profile has a unique value in the `m_urn` variable. (e.g., the first six rows of `df_jobs` all have the same `m_urn` because they are jobs for the same profile. No other rows have that value for `m_urn`.)

Then we `summarize` the data for those rows into new variables.

The summary `avg_pos_len = mean(avg_pos_len),` is interpreted as "for each of the jobs on this profile in `df_jobs`, take the `mean` of the variable `avg_pos_len` and save the output into a single row of a variable named `avg_pos_len` in `df_profiles`.

The summary `age = age[1],` is interpreted as "for each profile in `df_jobs`, take the age value in the first row (the first job) for that profile and save it as a value of `age` in `df_profiles`.

You can make similar summaries by `ddply` the data you have by match number.

Things you could do include:

- `total_runs = sum( pmax(numOutcome, 0))`

- `runs_1stinning = sum(pmax(numOutcome*1(inning == 1), 0)))`
- `total_wickets = length(which(numOutcome == 1))`
- `fielder_mentions = length(which(Fielder != ""))`
- `balls_until_1st_wicket = length(which(wickets == 0))`
- `average_wickets_in_during_match = mean(wickets, na.rm=TRUE)`
- `balls_2nd_inning` (you figure this one out by adapting the above code)

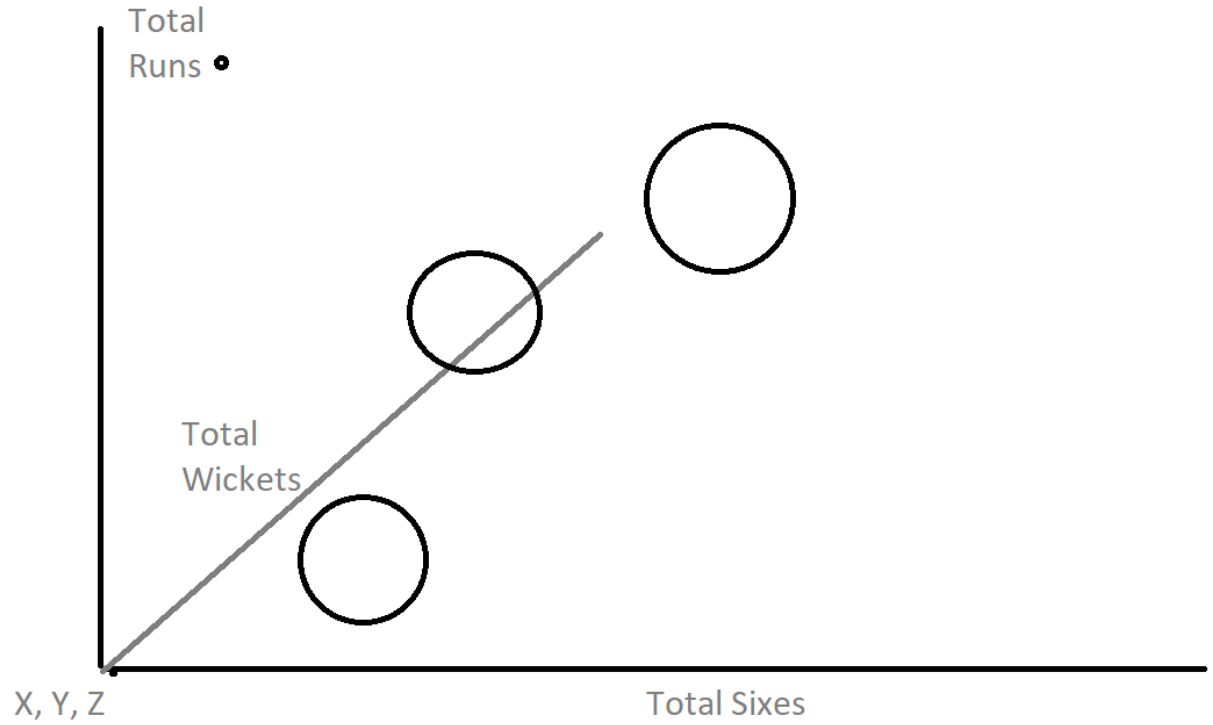
But these are only some of the workable possibilities. Since you'll be using these for clustering, it's mostly important to have numeric variables, especially ones you can standardized (x - mean)/sd.

Once you have variables like those above to describe the game, you can use kmeans as described in Lecture 6-1.

Make sure to justify your choice of the number of clusters. The actual choice doesn't matter so much as having a rational explanation for using the number of clusters you have. Graphing the eblow plot will help. Graphing the density isn't necessary, and gets quite hard when you use more than 2 variables to cluster anyways.

Don't worry if the clusters all fall on a diagonal line, as many variables are strongly correlated.

Also don't worry about characterizing the clusters with meaningful cricket explanations. Simply saying something like "this cluster has high total wickets and low total fielder events" is enough.



Duckworth Lewis Guidance

```
DLT = read.csv("DLS_T20.csv")[,-1]
DLT[16,2] = 0.3 # Was 0.03. This was a typo
#DLS # jth row, kth column is jth over, kth wicket
```

Resource numbers represent the proportion of runs that a team is expected to still score in a match, given the current overs and wickets lost. For example, teams at the beginning of their 7th over with 3 wickets lost have 0.68 resources.

```
DLT[7,4]
```

```
## [1] 0.68
```

That resource value implies that an average team has 68% of their runs left to score. In other words, we would expect them to have scored 32% of their total runs at this point in the match.

What part D of the final project is essentially asking is "Does this table reflect our data?", or "Does an average team in our dataset actually score 32% of their runs by this point in the match?"

This tricky to determine for a few reasons,

- First, not every match has a situation with a 7th over and 3 wickets lost. In many matches, the batting team has lost 0, 1, 2, 4, or 5 wickets at this point in the match. So how do we incorporate all these diverse situations together?
- Second, some simulations never appear in a match, at least in our data. There are (almost?) observations in cases like 16-20th overs, 0 wickets lost, and 1st-2nd overs, 9 wickets lost. In fact, anything in the gray region in the following screencap is effectively unprecedented and based on guesses. (Red is typo, yellow is 50%)

Wickets.Lo	X0	X1	X2	X3	X4	X5	X6	X7	X8	X9
20	1	0.968	0.926	0.867	0.788	0.682	0.544	0.375	0.21	0.083
19	0.961	0.933	0.892	0.839	0.767	0.666	0.535	0.373	0.21	0.083
18	0.922	0.896	0.859	0.811	0.742	0.65	0.527	0.369	0.21	0.083
17	0.882	0.857	0.825	0.779	0.717	0.633	0.516	0.366	0.21	0.083
16	0.841	0.818	0.79	0.747	0.691	0.613	0.504	0.362	0.208	0.083
15	0.799	0.779	0.753	0.716	0.664	0.592	0.491	0.357	0.208	0.083
14	0.754	0.737	0.714	0.68	0.634	0.569	0.477	0.352	0.208	0.083
13	0.71	0.694	0.673	0.645	0.604	0.544	0.461	0.345	0.207	0.083
12	0.664	0.65	0.633	0.606	0.571	0.519	0.443	0.336	0.205	0.083
11	0.617	0.604	0.59	0.567	0.537	0.491	0.424	0.327	0.203	0.083
10	0.567	0.558	0.544	0.527	**0.5**	0.461	0.403	0.316	0.201	0.083
9	0.518	0.511	0.498	0.484	0.461	0.428	0.378	0.302	0.198	0.083
8	0.466	0.459	0.451	0.438	0.42	0.394	0.352	0.286	0.193	0.083
7	0.413	0.408	0.401	0.392	0.378	0.355	0.322	0.269	0.186	0.083
6	0.359	0.355	0.35	0.343	0.332	0.314	0.29	0.246	0.178	0.081
5	0.304	0.03	0.297	0.292	0.284	0.272	0.253	0.221	0.166	0.081
4	0.246	0.244	0.242	0.239	0.233	0.224	0.212	0.189	0.148	0.08
3	0.187	0.186	0.184	0.182	0.18	0.175	0.168	0.154	0.127	0.074
2	0.127	0.125	0.125	0.124	0.124	0.12	0.117	0.11	0.097	0.065
1	0.064	0.064	0.064	0.064	0.064	0.062	0.062	0.06	0.057	0.044
0	0	0	0	0	0	0	0	0	0	0

We can solve both of these problems by fitting a smooth surface to our observations. That way an observation about the 7th over, 3 wickets lost will be informed not only by the situations at 7th over, 3 wickets lost, but also the 6th and 8th overs wickets lost, and 7th over with 2, 3, or 4 wickets lost.

This is the sort of task that `optim()` can handle very well.

To set up for this you need:

A ball-by-ball measurement of how many overs have been used

```
x$over2 = x$over + x$ball/6
```

A variable that measures the proportion of the total runs that have been scored after each ball.

ONLY USE THE FIRST INNING FOR THIS BECAUSE THEY WEREN'T AIMING FOR A SPECIFIC TARGET.

For example, if a team got to 250 runs in a first inning, and they currently have 75 runs, the their proportion should be 0.3.

I'll leave that for you to find out how proportion is calculated, but it helps to use the non-negative values of the `numOutcome` variable as a very close approximation of the number of runs (see footnote:)

```
x$Nruns = pmax(x$numOutcome, 0)
```

Let's call your observed proportion `prop` for now. The variable `prop_smooth` is going to be the results from your Duckworth-Lewis-like formula.

As a simple version you should mess with this to refine it, as other formulae will get you a better squared error. *Hint: Intercept, interaction.*

```
prop_smooth = A*over2 + B*wicket + C*over2^2
```

We don't know what A, B, or C should be, but optim can find the values that minimize, say, the sum-of-squared error between the fitted proportion values from your formula and the observed values from the data.

```
loss_function = function(x, prop)
{
  A = x[1]
  B = x[2]
  C = x[3]

  prop_smooth = A*over2 + B*wicket + C*over2^2
  error = sum( (prop - prop_smooth)^2)
  return(error)
}
```

The best values for A, B, and C, are found by calling the `optim()` function like so:

```
best_params = optim(par=c(1,1,1), loss_function, prop=prop)$par
A = best_params[1]
B = best_params[2]
C = best_params[3]
```

This will give you a formula you can use to fill in your own Duckworth Lewis-like table.

```
# Example parameter variables
A = 0.02
B = 0.07
C = 0.01

# Make a matrix. 20 rows for 20 overs, 10 columns for the 0-9 wickets taken
# NA for cell values to start so that
# we know if we missed something because it will still be NA
newDLT = matrix(NA, nrow=20, ncol=10)

# Compute the matrix row by row, where each row is an over
for(overcount in 1:20)
{
  # Apply the example formula. 1 - (formula) because resource = 1 - proportion.
  newDLT[overcount,] = 1 - A*overcount + B*(0:9) + C*overcount^2
```

```
}

# Compare
newDLT
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,] 0.99 1.06 1.13 1.20 1.27 1.34 1.41 1.48 1.55  1.62
##  [2,] 1.00 1.07 1.14 1.21 1.28 1.35 1.42 1.49 1.56  1.63
##  [3,] 1.03 1.10 1.17 1.24 1.31 1.38 1.45 1.52 1.59  1.66
##  [4,] 1.08 1.15 1.22 1.29 1.36 1.43 1.50 1.57 1.64  1.71
##  [5,] 1.15 1.22 1.29 1.36 1.43 1.50 1.57 1.64 1.71  1.78
##  [6,] 1.24 1.31 1.38 1.45 1.52 1.59 1.66 1.73 1.80  1.87
##  [7,] 1.35 1.42 1.49 1.56 1.63 1.70 1.77 1.84 1.91  1.98
##  [8,] 1.48 1.55 1.62 1.69 1.76 1.83 1.90 1.97 2.04  2.11
##  [9,] 1.63 1.70 1.77 1.84 1.91 1.98 2.05 2.12 2.19  2.26
## [10,] 1.80 1.87 1.94 2.01 2.08 2.15 2.22 2.29 2.36  2.43
## [11,] 1.99 2.06 2.13 2.20 2.27 2.34 2.41 2.48 2.55  2.62
## [12,] 2.20 2.27 2.34 2.41 2.48 2.55 2.62 2.69 2.76  2.83
## [13,] 2.43 2.50 2.57 2.64 2.71 2.78 2.85 2.92 2.99  3.06
## [14,] 2.68 2.75 2.82 2.89 2.96 3.03 3.10 3.17 3.24  3.31
## [15,] 2.95 3.02 3.09 3.16 3.23 3.30 3.37 3.44 3.51  3.58
## [16,] 3.24 3.31 3.38 3.45 3.52 3.59 3.66 3.73 3.80  3.87
## [17,] 3.55 3.62 3.69 3.76 3.83 3.90 3.97 4.04 4.11  4.18
## [18,] 3.88 3.95 4.02 4.09 4.16 4.23 4.30 4.37 4.44  4.51
## [19,] 4.23 4.30 4.37 4.44 4.51 4.58 4.65 4.72 4.79  4.86
## [20,] 4.60 4.67 4.74 4.81 4.88 4.95 5.02 5.09 5.16  5.23
```

```
DLT
```

```
##       X0    X1    X2    X3    X4    X5    X6    X7    X8    X9
## 1  1.000 0.968 0.926 0.867 0.788 0.682 0.544 0.375 0.210 0.083
## 2  0.961 0.933 0.892 0.839 0.767 0.666 0.535 0.373 0.210 0.083
## 3  0.922 0.896 0.859 0.811 0.742 0.650 0.527 0.369 0.210 0.083
## 4  0.882 0.857 0.825 0.779 0.717 0.633 0.516 0.366 0.210 0.083
## 5  0.841 0.818 0.790 0.747 0.691 0.613 0.504 0.362 0.208 0.083
## 6  0.799 0.779 0.753 0.716 0.664 0.592 0.491 0.357 0.208 0.083
## 7  0.754 0.737 0.714 0.680 0.634 0.569 0.477 0.352 0.208 0.083
## 8  0.710 0.694 0.673 0.645 0.604 0.544 0.461 0.345 0.207 0.083
## 9  0.664 0.650 0.633 0.606 0.571 0.519 0.443 0.336 0.205 0.083
## 10 0.617 0.604 0.590 0.567 0.537 0.491 0.424 0.327 0.203 0.083
## 11 0.567 0.558 0.544 0.527 0.500 0.461 0.403 0.316 0.201 0.083
## 12 0.518 0.511 0.498 0.484 0.461 0.428 0.378 0.302 0.198 0.083
## 13 0.466 0.459 0.451 0.438 0.420 0.394 0.352 0.286 0.193 0.083
## 14 0.413 0.408 0.401 0.392 0.378 0.355 0.322 0.269 0.186 0.083
## 15 0.359 0.355 0.350 0.343 0.332 0.314 0.290 0.246 0.178 0.081
## 16 0.304 0.300 0.297 0.292 0.284 0.272 0.253 0.221 0.166 0.081
## 17 0.246 0.244 0.242 0.239 0.233 0.224 0.212 0.189 0.148 0.080
## 18 0.187 0.186 0.184 0.182 0.180 0.175 0.168 0.154 0.127 0.074
## 19 0.127 0.125 0.125 0.124 0.124 0.120 0.117 0.110 0.097 0.065
## 20 0.064 0.064 0.064 0.064 0.064 0.062 0.062 0.060 0.057 0.044
## 21 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

The smoothing won't necessarily look good for the extreme values because the `optim` will prioritize the table values with lots of observations. This is a feature, not a bug.

You can also use your formula to get a Duckworth-Lewis-like estimate of resources for things like determining

how much a team fell ahead or behind their opponent when looking for highlights in part C

Footnote: `NumOutcome` isn't exactly the same as the number of runs because a four or six is marked as 4 or 6, respectively, as there are rare cases where a batter gets an extra run as a penalty to the bowling team AND a four/six is earned. These should not throw off your proportion calculations much. If you use these to determine a game's winner, you will run into issues. A better determinant of a game's winner is state that the 2nd inning ended. If it ends with a 10th wicket taken or a 20th over being used, then the 2nd inning batting team didn't reach the target to win (unless they got it on the very last ball). If the 2nd inning game before 20 overs or 10 wickets, then the 2nd inning reached their target and win (they scored more runs than the 1st inning team, so no more play is necessary).