SMT Data Challenge: Evaluating Outfielder Route Efficiency and Route Paths By Level

Team 60

Division: Undergrad

1. Abstract

Outfielders have one of the most important jobs in today's game. In a game where power and extra bases win baseball games, great outfielders must excel at limiting the damage done on hard-hit balls. There are a couple of attributes that an outfielder may possess to prevent big plays, among them is their route to the baseball. A few steps in the wrong direction could mean the difference between an out, hit, or extra bases. The aim of this project is to identify the optimal route path on any given flyball based on a number of factors including where the ball lands, where the fielder starts from, how long he has to get there, among other things. I also created a model that predicted the route taken to a ball at different levels of the farm system to see if there

were any major or minor differences between routes at different levels that impacted the fielder's catch probability, finding a wider range of fielding acumen among the lower levels of the farm system compared to the higher levels having more experienced players. I created a Shiny App that utilizes my models so you can try them for yourself.

2. Introduction

Outfielders have one of the more underrated jobs in baseball. Three players are tasked with covering the massive outfield, many times in a matter of only a few seconds. Milliseconds after the ball is hit, the fielder must accurately assess where the ball will land down to the foot and then determine the most efficient path to that spot. Even a few wrong steps can be the difference between a hit and an out. Throughout this project, the goal was to create a model that would create the most efficient path to the ball to maximize catch probability. This type of model could be displayed side-by-side with an actual route taken by the outfielder and can help him get an idea of how he can improve his route to the ball. I also wanted to evaluate how fielder route path and route efficiency varies throughout each level in the farm system. Assuming fielders get better as they make their way through the minors, this could help teams determine if a player is prepared defensively to move up to the next level.

3. Data

I started by loading in ball position, outfielder position, game info, and game events data supplied by SMT. I then joined all these tables together on the game_str, play_id, and timestamp variables. I edited the table further to only include plays where a flyball or line drive was hit and

only included the fielder position data of the outfielder who fielded/touched the ball first. To determine if a batted ball was a line drive or a flyball I calculated its launch angle (LA) and included the batted ball events that had a LA greater than 10 degrees, which Statcast considers the minimum LA for a line drive. I also removed any flyballs that didn't travel further than 120 feet (in the y-direction) and assumed these to be either data errors or popups. Using the data provided, I calculated the max fielder speed (during the route), exit velo, pitch velo, pitch break, pitch type, distance from final ball position (x-direction, y-direction, and total), and whether or not a flyball was caught (see Appendix 9.1). I noted whether a ball was caught or dropped in two steps. First, I checked which event code from the game events data came first (see Figure 1). Second, I removed any plays that claimed that a fielder caught the ball while being more than 8 feet away from the final ball position by the end of the play and assumed these plays to be data errors. While 8 feet seems like a big area, player tracking is often subject to very high levels of error and rarely tracks a player's coordinates precisely. Using a bigger area accounts for this level of error, while admittedly using 8 feet was an arbitrary number I used that is open for debate. I ended every play after either the ball was caught or the ball hit the ground.

Event Codes	Definition
1	Pitch
2	Ball Acquired
4	Ball In Play (i.e. Contact Made)
9	Ball Deflection
10	Ball Deflection (off wall)
16	Bounce

Figure 1: The event codes above correspond to what happened over the course of the play. If event code 2 is the first event code to occur after contact is made, the ball was caught. Otherwise, it was a hit.

4. Methodology

4.1. Catch Probability Model

For my catch probability model, I utilized a random forest probability model due to its high prediction ability. A random forest model consists of hundreds of decision trees and is particularly good with predicting on data with high dimensionality like this one contains. A decision tree starts with a subset of the data, going through each independent variable and splitting on them until it finds the variable that minimizes error. It continues to split until there are only a few observations left in the final node (terminal node), at which point the tree takes the average of the dependent variable from the remaining observations and makes its prediction. The forest then takes the average of the predictions from all the trees and makes a final prediction on the observation (see Figure 2). I split my data into a training and testing dataset to ensure that my model didn't overfit my data and was best suited to make accurate predictions on new data. I built my model only on observations after contact. For my model, I used max fielder speed, final ball position, distance (x-direction, y-direction, and total), and time remaining before catch/bounce (see Figure 3). I originally included exit velo and LA in the model and although it

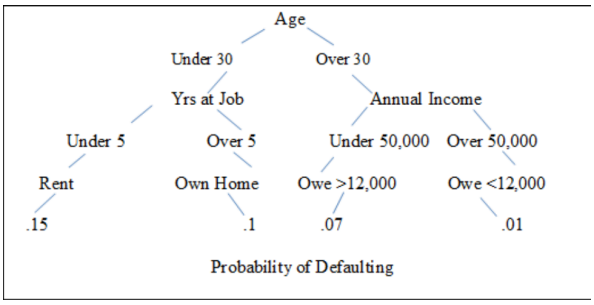

Figure 2: Here's an example of a decision tree where the user is trying to predict the probability of a client defaulting on his loan. Age, Yrs at Job, and Annual Income are all independent variables that the tree is splitting on and the probabilities at the bottom of the tree are the averages of the terminal node. (Reference: SAL 313 Powerpoint)

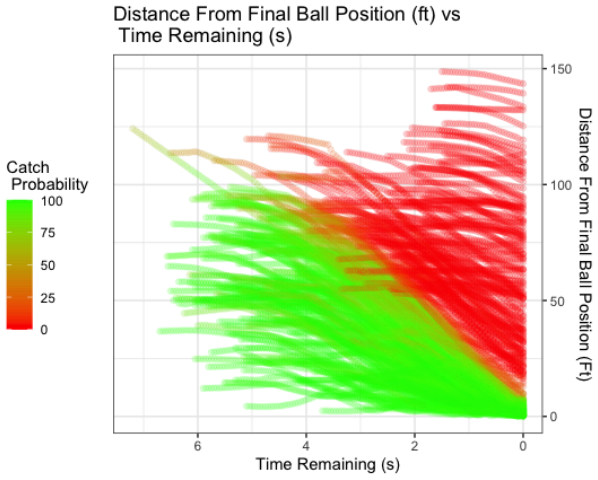

Figure 3: Plot showing how the relationship between time remaining and distance from final ball position impacts catch probability.

caused the model to perform better, it didn't work well with the Shiny App where the user has to include their own inputs since exit velo and LA are heavily related to final ball position and time remaining, so I left them out of the model. I also originally had pitch type in the model due to its potential effects on LA and final ball position, but it didn't have much effect on the model.

4.2. Route Optimization Model

For the optimization model, I first took the fielder's max speed, used it to determine the max distance the fielder could travel in 0.2 seconds and found every field_x and field_y combination within that max distance (to the nearest half inch). I then kept the 50 combinations that minimized the distance from the final ball position to get the model moving in the right direction and used my catch probability model to find the field_x field_y combination that maximized catch probability. I also created a pseudo acceleration algorithm for the first few seconds of the route using running splits data from Statcast (see Figure 4). As you can see, players seem to reach top acceleration at 10 feet and continue accelerating until they reach top speed at around 35 feet. According to the running splits data from Statcast, this occurs around 0.85 seconds (10 feet) and 2 seconds (35 feet). Thus, in my optimization model, I limited the top speed of the fielder by half in the first 0.85 seconds after contact and then by ⅘ of their top speed between 0.85 and 2 seconds after contact.

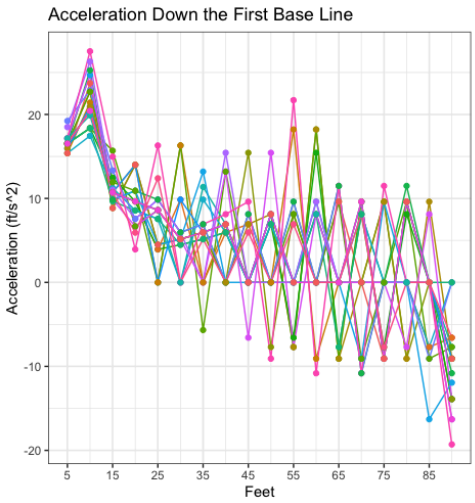

Figure 4: Acceleration (ft/s^2) plot of 30 random players. Seems to be a clear spike in acceleration at 10 feet, and a slower continuous acceleration until players reach top speed at around 35 feet on average.

4.3. Fielder Path Model

The goal of the fielder path model was to try and estimate what path the average fielder at each level in the farm system would take to the ball. I used the same random forest model approach that I used for the catch probability model, creating one model for the x coordinates and another for the y coordinates. At first, I created two models that used level as an independent variable. However, this process caused the models to take up too much memory space and took too long to predict on new data. Therefore, I created an x and y coordinate model at each level instead. Similarly to the catch probability model, I only included observations that occurred after contact when creating my model. The independent variables for these models included previous field position, previous distance, time remaining, time in play (how long the play has gone on for up until that point), final ball position, and previous distance x & y (depending on the model). On average, I was able to get the error within 6 feet of the actual position for each model.

5. Results

5.1. Catch Probability Model and Metrics

Using the catch probability model (see Appendix 9.2), I was able to calculate 2 metrics: the first was the Outs Above Average (OAA) statistic that was first created by Statcast. At the moment of contact, I subtracted the number of outs made by the catch probability at contact to get the OAA for that play. The second metric I calculated is a statistic I call Route Score, which as far as I know, is a statistic I created to evaluate route efficiency. You take the catch probability at the end of the play and subtract it from the

catch probability at contact to get the Route Score, which effectively calculates how much catch probability a fielder adds throughout their route to the baseball. I then found the total OAA and Route Score as well as the percentile for every unique fielderID, level, position, and season combination. The average for both metrics is 0. The leaderboard is available on my Shiny App (see Figure 5).

	Fielder ID	Position	Level	Plays	OAA Percentile	Route Score Percentile	Avg. OAA Per Play Percentile	Avg. Route Score Per Play Percentile
	All	All	All	8.00 ... 42.0	All	All	All	All
57	624	CF	4A	42	16.4	10.4	53.7	53.7
62	495	CF	1A	20	9	13.4	40.3	46.3
2	651	CF	2A	18	98.5	97	80.6	74.6
3	594	LF	1A	18	97	98.5	76.1	77.6
52	467	LF	4A	18	23.9	9	47.8	40.3
64	460	RF	3A	17	6	1.5	32.8	20.9
1	975	RF	2A	14	100	100	92.5	94
4	461	CF	3A	14	95.5	94	73.1	70.2
16	637	LF	2A	14	77.6	16.4	62.7	38.8

Figure 5: Example OAA and Route Score Leaderboard that you can find on my Shiny App. Here, We have we can see the leaderboard in descending order of plays. Keep in mind that the game info data that contained the fielderIDs for each play had a lot of missing data. As a result, there are a lot of plays that didn't have a player attached to them the I've left out of the table.

5.2. Route Optimization Model

The optimization model (see Appendix 9.3) proved to be largely successful at finding routes that had a better catch probability than the fielders took (see Figures 6 & 7).

Summary Statistics Of Optimized Catch Probability - Catch Probability

Min	1st	Median	Mean	3rd	Max
-0.51	0.02	0.19	0.27	0.47	0.99

Figure 6: Summary statistics of the difference in catch probability where Optimized Catch Probability is the catch probability of the optimized field_x & field_y combination and Catch Probability is the catch probability at the field_x & field_y combination of the fielder.

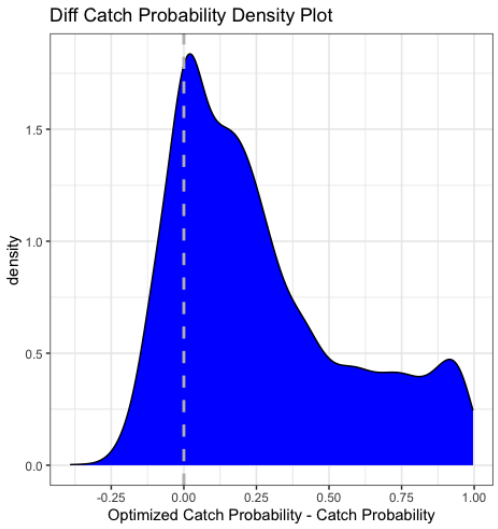

Figure 7: Density plot showing the distribution of the difference in catch probability between the optimized field_x field_y combination and the fielder field_x field_y combination.

However, although there were points in the middle of the routes that had lower catch probabilities than the route the fielder took, the catch probability of the optimized route was nearly always better or as good as the route the fielder took (see Figures 8 & 9). Granted, some of this could be because of the arbitrary acceleration system I have in place for this model which might have inflated the difference (See Figure 10).

Summary Statistics Of Optimized Catch Probability - Catch Probability (End of Play)

Min	1st	Median	Mean	3rd	Max
-0.03	0.01	0.13	0.36	0.85	0.99

Figure 8: Summary statistics of the difference in catch probability by the end of the play where Optimized Catch Probability is the catch probability of the optimized field_x & field_y combination and Catch Probability is the catch probability at the field_x & field_y combination of the fielder.

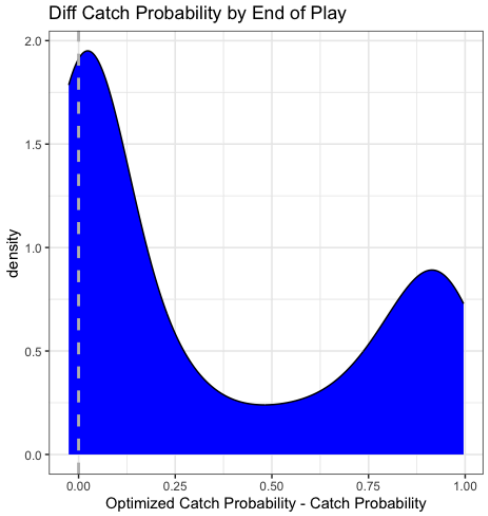

Figure 9: Density plot showing the distribution of the difference in catch probability between the optimized field_x field_y combination and the fielder field_x field_y combination by the end of the play.

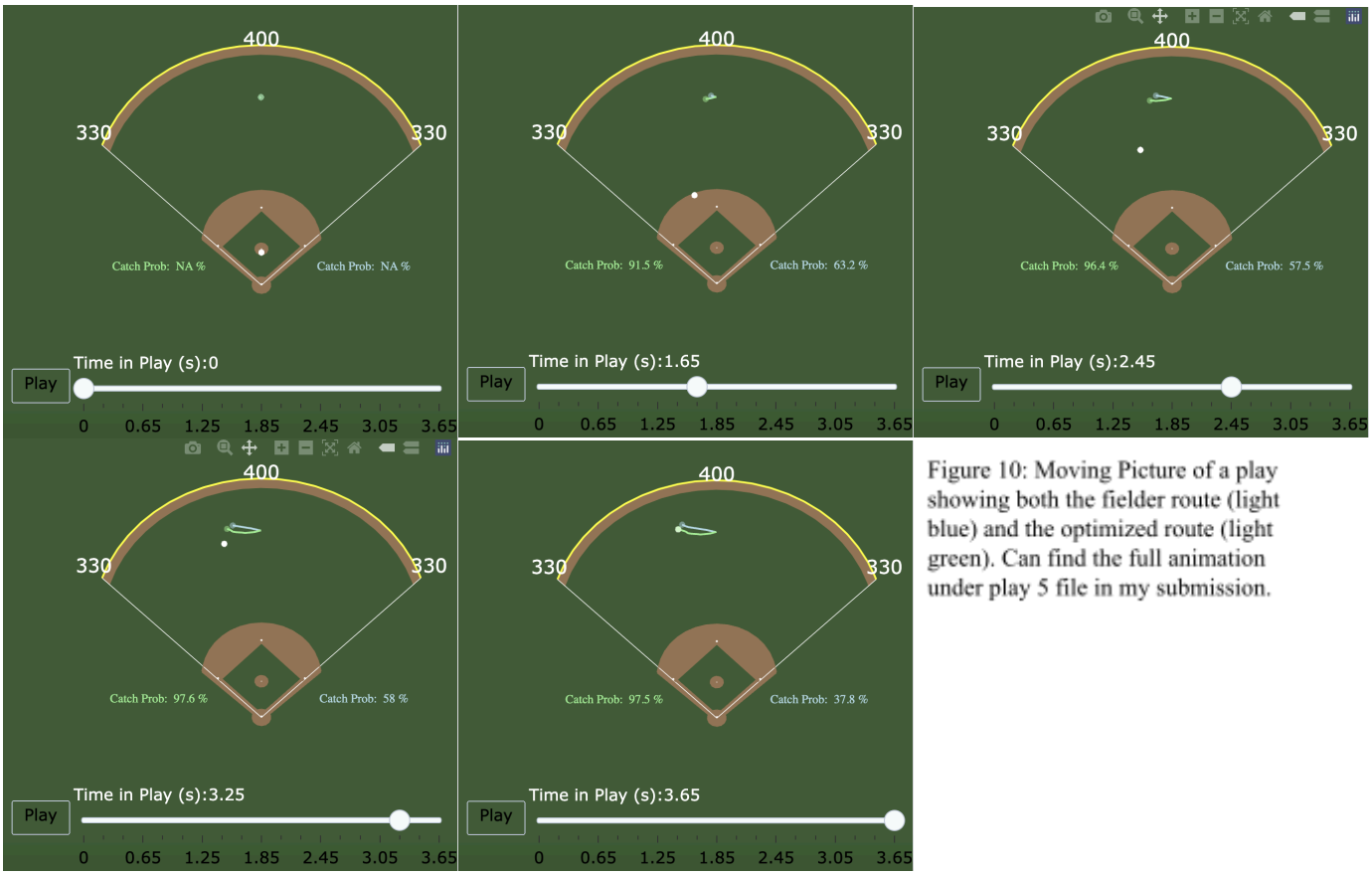

Figure 10: Moving Picture of a play showing both the fielder route (light blue) and the optimized route (light green). Can find the full animation under play 5 file in my submission.

5.3. Fielder Path Model

Although the fielder paths themselves didn't show huge differences between levels (see Figure 11), one thing I did notice was that the models at the lower levels often had a much higher level of error while tuning the models. This makes sense, as the players at the lower levels are often young and inexperienced, making variance in fielding ability from player to player more extreme at the lower levels than at the upper levels where players have more experience in professional baseball and the skill gap between players has decreased. Despite this, I was still able to get a fairly decent error term for all levels with precise tuning of the hyperparameters of the model (see Appendix 9.4).

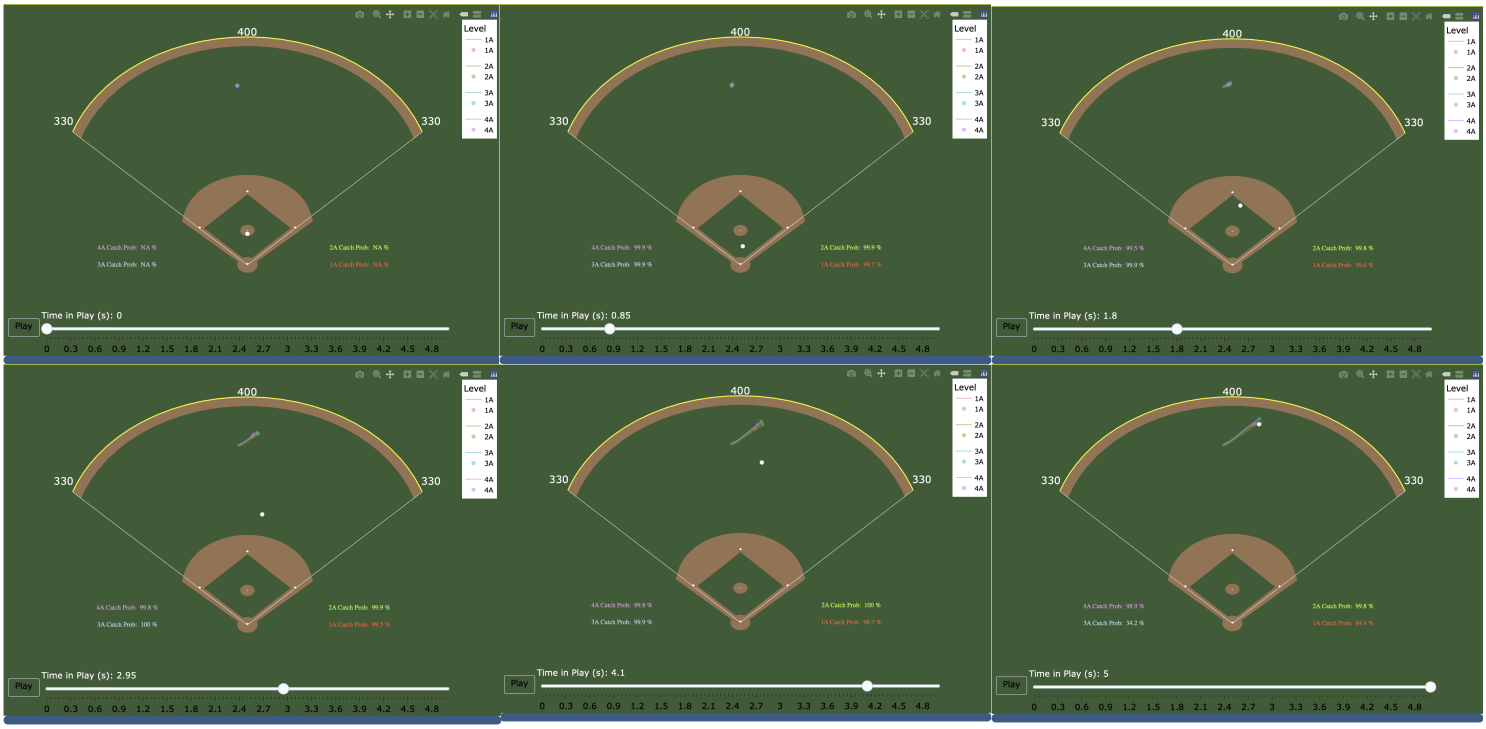

Figure 11: Example graphic utilizing the Fielder Route Model within the Shiny App

5.4. Shiny App

One of the biggest things that came as a result of this project was the Shiny App I created that allows users to alter the inputs for each model. In Figure 12, we can see that the inputs for the model are field position, final ball position, max fielder speed, and length of play (from pitch to catch/ball bounce). To input field position and final ball position, you simply need to click a point on the graph to fill in the coordinates for the model. The Shiny App gives users unique control to check any hypothetical situations they are curious about. The Shiny App also contains OAA and Route Score leaderboards (as mentioned above) as well as example optimization plots using real plays displaying the optimal path against the path taken by the outfielder. Please note that the app may need to be relaunched after the fielding paths model (under the Fielding Across Levels tab). This is due to the memory constraints imposed by Shiny and there isn't anything I can do about it (see Appendix 9.5 for more details).

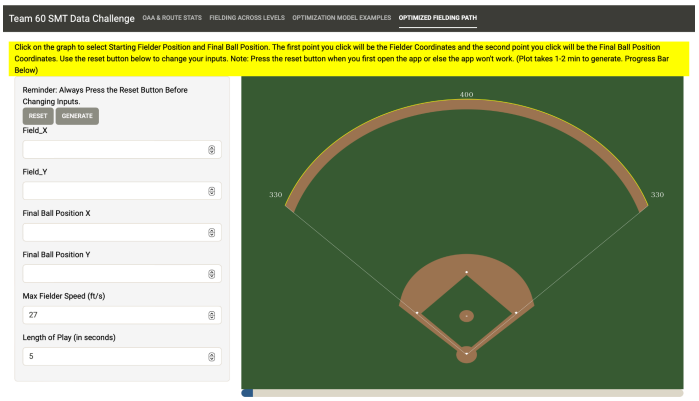

Figure 12: Here's an example of how the optimized fielding path tab within the Shiny App looks before inputs have been filled.

6. Application

I think the optimization and fielding path models have great potential to further player development and player evaluation. Allowing an outfielder to compare his route to an optimized route can allow him to make changes on the fly and positively impact his plan of attack when tracking down similar flyballs. Similarly, the level-by-level fielder paths are a great resource to

see how a particular fielder's route is compared to fielders at both his level and the next level. This can allow coaching staffs and front offices to better evaluate if an outfielder is defensively capable to move onto the next level of the farm system.

7. Improvements and Future Work

The first thing I would do in the future is create a second catch probability model with exit velo and LA included. This would allow users to use a more accurate catch probability and optimization model when exit velo and LA are known to them and would be more useful for players in the future. Also, the field models weren't necessarily the best they could be because I had to use a smaller test dataset than I wanted since tuning the models took an extremely long time. This caused my dataset to only be 10 plays big, and caused a high risk of not only overfitting on the training dataset, but also the test dataset as well. Thus, in the future, I would like to use more test data to ensure the best performance. I also had memory space issues with the Shiny App as a result of my models, which caused me to have to change certain aspects of my models at the last minute which slightly decreased its accuracy (There are a lot more technical improvements I would make to the model which I left in Appendix 9.4).

8. References

8.1 "Allow for Random Splits in Regression." *GitHub*, github.com/imbs-hl/ranger/issues/87. Accessed 6 Aug. 2024.

8.2 "Cumulative Animations in R." *Plotly*, plotly.com/r/cumulative-animations/. Accessed 5 Aug. 2024.

8.3 Drucker R (2024). _sportyR: Plot Scaled 'ggplot' Representations of Sports Playing Surfaces_. R package version 2.2.2, <https://CRAN.R-project.org/package=sportyR>.

8.4 Geurts, Pierre, et al. *Extremely Randomized Trees*, 2 Mar. 2006, orbi.uliege.be/bitstream/2268/9357/1/geurts-mlj-advance.pdf.

8.5 “How to Calculate Class Weights for Random Forests in R.” *GeeksforGeeks*, GeeksforGeeks, 26 June 2024, www.geeksforgeeks.org/how-to-calculate-class-weights-for-random-forests-in-r/.

8.6 Marvin N. Wright, Andreas Ziegler (2017). ranger: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. Journal of Statistical Software, 77(1), 1-17. doi:10.18637/jss.v077.i01

8.7 “Statcast Running Splits Leaderboard.” *Baseballsavant.Com*, baseballsavant.mlb.com/leaderboard/running_splits. Accessed 5 Aug. 2024.

8. 8 “What Exactly Is the Extratrees Option in Ranger?” *Stack Exchange*, stats.stackexchange.com/questions/406666/what-exactly-is-the-extratrees-option-in-ranger. Accessed 6 Aug. 2024.

9. Appendix

9.1. General/Data Manipulation

There was a lot of data manipulation I did before I even started thinking about the model. I figured I'd try to calculate everything that I **might need** before starting the model so I wouldn't have to continuously go back and add new variables. The first variable I calculated was time_in_play, which I calculated by simply subtracting the current timestamp from the timestamp at the beginning of the play (when the pitch was thrown). Timestamps in the data were a way to track time throughout the game and play. I found the time_remaining variable in a similar way, subtracting the timestamp at the end of the play (i.e. when the ball was caught/dropped) from the current timestamp.

9.1.1. Play Classification

To expand a bit further on section 3 where I explain how I figured out whether a flyball was caught or not, I first checked the event codes to see if the ball bounced or was deflected during the duration of the play (event code 9,10,16). If it hadn't, then the ball must've been caught and the play had ended. Otherwise, I need to figure out if the bounce occurred before or after the fielder acquired the ball. For instance, event code 16 could've occurred after the catch if the outfielder was making a throw back into the infield and the ball bounced before reaching the target. To figure out if the fielder caught the ball before it bounced, I

Event Codes	Definition
1	Pitch
2	Ball Acquired
4	Ball In Play (i.e. Contact Made)
9	Ball Deflection
10	Ball Deflection (off wall)
16	Bounce

Figure 1: The event codes above correspond to what happened over the course of the play. If event code 2 is the first event code to occur after contact is made, the ball was caught. Otherwise, it was a hit.

took the timestamp of the first ball bounce during any play and removed all observations that occurred after that timestamp. I then checked to see if an event code 2 popped up during the play before the first ball bounce. If it did, the ball was caught and I removed any observation that occurred after the ball was caught.

9.1.2. Fielder Speed

I calculated fielder speed (ft/s) in 0.25 second intervals and used the max fielder speed in the route for the max fielder speed variable within my model. I decided to use max speed in the route rather than the max speed that player exhibited throughout the dataset because the majority of the plays didn't have a player (fielderID) attached to them.

9.1.3. Batted Balls

I found the exit velocity of each batted ball attempt by finding the change in distance in all directions (x,y, and z) between the second and first observation after contact during the 50 ms interval and converting it to mph. I found the LA for each batted ball event by taking the tangent angle of the change in height (z) and the change in total distance in the xy direction one observation after contact. I found pitch velo in a similar manner. Figures 13 & 14 show the code for both:

```
#determining exit velo
exit_velo <- function(df){
  df %>%
    group_by(game_str, play_id, Day) %>%
    mutate(exit_velo = ifelse(event_code == 4,
                              round(sqrt((lead(ball_position_x, n = 2L) - lead(ball_position_x))^2 +
                                           (lead(ball_position_y, n = 2L) - lead(ball_position_y))^2 +
                                           (lead(ball_position_z, n = 2L) - lead(ball_position_z))^2)/
                                      50*681.8,2),NA)) %>%
    fill(exit_velo, .direction = c("downup")) %>%
    ungroup()
}
```

Figure 13: Exit Velo Code

```
#determining launch angle
LA <- function(df){
  df %>%
    group_by(game_str, play_id, Day) %>%
    mutate(LA = ifelse(event_code == 4,
                       round(atan((lead(ball_position_z) - ball_position_z)/
                                    sqrt((lead(ball_position_x) - ball_position_x)^2 +
                                           (lead(ball_position_y) - ball_position_y)^2))*(180/pi),2),NA)) %>%
    fill(LA, .direction = c("downup")) %>%
    ungroup()
}
```

Figure 14: LA Code

9.1.4. Pitch Identification

I then looked to figure out a way to calculate pitch break so I could identify pitch type. I wasn't exactly sure if this would have any big impact on the final results of the model. My thought process was perhaps some pitch types are associated with lower LA (like the sinker) which could impact the flight path and others with higher exit velos. However, going back to what I said at the beginning of the section, I would rather have it and not use it than need it and not have it. First, I had to identify which observations in each play were during the pitch flight. Then I found the change in x and z position at every observation in the pitch flight (approximately 50 ms between observations), saved as variables **delta_ball_x** and **delta_ball_z** in the dataset. I then found the actual horizontal and vertical movement by summing up those values. Now for the hard part: determining how much delta_ball_x and delta_ball_z movement was due to pitch break. For horizontal movement, I took the first delta_ball_x value and assumed that without magnus force, seam-shifted wake, or any other forces applied to the baseball throughout the flight, that delta_ball_x would remain constant throughout the pitch since no other forces are present in the horizontal field (except wind perhaps, but let's assume it isn't). Finding vertical movement was a bit trickier, since we now have to deal with gravity which will apply a

constant downwards force on the ball. However, we do know that the force of gravity is equal to 9.8 m/s^2, which is roughly equivalent to 32 ft/s^2. Also, we can find the distance an object moves through space with the following equation:

$\Delta d_z = (a/2)(\Delta t)^2 + v_{zi}(\Delta t)$ where $a/2$ = -16, Δt = pitch flight time, and v_{zi} = initial velocity after pitch release in the z direction. We can find the pitch flight time by using the time_in_play variable I created earlier and finding the value at the point of contact and we can find the initial velocity after pitch release in the z direction using the first delta_ball_z observation from the pitch. After finding the expected horizontal and vertical movement profiles of the pitch along with the actual movement profiles of the pitch, we can convert them to inches by multiplying by 12 and then taking the difference between the two to find pitch break. I also manipulated the horizontal break of each pitch such that negative horizontal break represents Arm-Side break while positive horizontal break represents Glove-Side break. I then used my baseball knowledge to identify pitch types based on their pitch velo and pitch break characteristics (See Figures 15 & 16).

```
# Identifying Pitches
final_dataset <- final_dataset %>%
  group_by(game_str, play_id, Day) %>%
  mutate(pitch_type = case_when(
    pitch_velo < 65 ~ "Eephus",
    pitch_velo > 88 & abs(Hor_Mov) < Ver_Mov & Hor_Mov < 0 ~ "4-Seam",
    pitch_velo > 88 & Ver_Mov < abs(Hor_Mov) & Hor_Mov < 0 ~ "Sinker",
     Hor_Mov < -10 & Ver_Mov < 12 | pitch_velo < 88 & Hor_Mov < -10  ~ "Changeup",
     Hor_Mov > -10 & Hor_Mov < 0 & Ver_Mov < 12 |
      pitch_velo < 88 & Hor_Mov > -10 & Hor_Mov < 0 ~ "Splitter",
     Hor_Mov >= 0 & Hor_Mov < 10 & Ver_Mov < 10 & Ver_Mov > -5.5 ~ "Slider",
      Hor_Mov >= 0 & Hor_Mov < 10 & Ver_Mov > 10 ~ "Cutter",
      Hor_Mov >= 10 & Ver_Mov >= -5  & abs(Hor_Mov) > abs(Ver_Mov) ~ "Sweeper",
     Hor_Mov >= 7 & Ver_Mov <= -5 ~ "Slurve",
     Hor_Mov >= 0 & Ver_Mov < -5 & abs(Ver_Mov) > abs(Hor_Mov)  ~ "Curveball"
    )) %>%
   mutate(pitch_type = factor(pitch_type, levels = c("4-Seam", "Sinker", "Cutter",
                                                     "Splitter", 'Changeup',
                                                     "Slider", 'Sweeper',
                                                     'Curveball', 'Slurve',
                                                     'Eephus')))%>%
  ungroup()
```

Figure 15: Pitch Identification Code. The vertical bar (|) is effectively an OR statement.

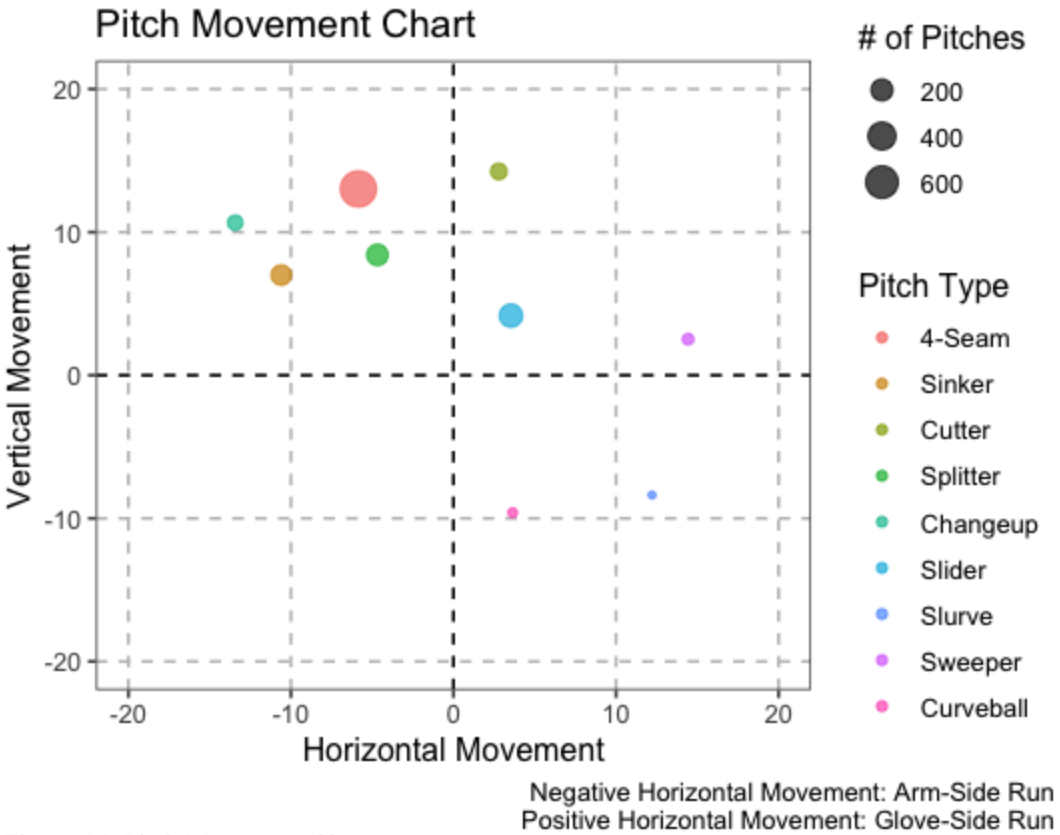

Figure 16: Pitch Movement Chart

9.2. Catch Probability Model

I utilized the ranger package for the catch probability model, which is a faster implementation of the random forest package. The ranger package also allowed for more customization within the hyperparameters of the model to get the most accurate estimates for both my catch probability model and especially my fielder path model (see Appendix 9.4). One specific change I was able to make in ranger was utilizing the Extremely Randomized Trees splitting technique first introduced by Geurts et al in 2006 (8.4). The way this method works is that for each feature X, it draws K random split points (referred to as num.random.splits in the ranger function) from a continuous uniform distribution

between A and B, where A and B are the minimum and maximum covariate values at that current node (8.1). It's very useful for extremely high dimensional data where finding the locally optimal cut point doesn't work as well for finding the overall best model fit for predicting on new data. I only just learned about this technique while doing this project so I'm fairly new to this method, but it seemed to have much more predictive power than the gini splitrule in my catch probability model as well as variance splitrule in my fielder path model (see Appendix 9.4). I also decided to use the class weights argument within ranger to help with the major class imbalance with regards to flyouts vs hits in my dataset, as nearly 68% of my data were flyouts. Thus, I used the inverse of class frequencies method to make sure hits weren't underrepresented when creating my forest. (8.5). When creating the training and test datasets, I made sure no play could be in both the training dataset and the test dataset. I found when plays were mixed between the training and test datasets, the model would overfit the data to the point where the catch probability would increase or decrease dramatically in the span of a tenth (0.1) of a second, which is extremely unnatural. By grouping each observation by play before splitting them into training and test datasets, I found that this issue was resolved. I created primary keys for each play by using their unique game_str and play_id combination which I used to create these test and training datasets with the help of the group_initial_split function in the tidymodels package. Overall, my model had a final Brier score of 0.046.

9.3. Route Optimization Model

For the optimization problem, I utilized a for loop that my model to go row by row in the data in order to both find the best field_x field_y combination for that particular row, but also utilize the prediction from the previous row to be used when finding the next

field_x field_y combination so the model will smooth over when in practice. To find all the possible field_x field_y combinations within the max distance a fielder could go, I utilized a while loop where the initial value of distance was the distance the fielder would travel at max speed. I then decreased distance by 0.05 after every loop until distance became negative, at which point the while loop ended and returned a large dataset of field_x field_y combinations. Within the while loop, I found the different x distance and y distance combinations using Pythagoras theorem, where:

$y_{dist} = \sqrt{(distance)^2 - (x_{dist})^2}$. I also had to add the negative values of y distance after the fact (since the equation above will only yield positive values) in order to get field_x field_y combinations in every direction. I then added these values to the previous field_x field_y values to get the possible values of the optimal field_x field_y combination before merging this data back to the row of the observation with all the original data. I then calculated the distance from the final ball position at each new field_x field_y combination and only kept the 50 observations that minimized distance from the final ball position. I did this for two reasons. First, it made the model run quicker since it had to predict on less data. Second, when slicing by catch probability first, especially when the catch probability was really high, the model took some really unusual routes to the baseball because it didn't sense a huge difference between one field_x field_y combination over the other. Slicing by distance first and then catch probability allowed the model to move in the right direction. In order to add an acceleration component to the model, I used the running splits data that's available on Statcast. This data contains the time it takes different batters to make their way down the first base line at every 5-foot interval. After re-arranging the data a bit using the pivot_longer function (see

Acceleration_Time.R script for details), I was able to create an acceleration variable that allowed me to visualize how 30 random players accelerated down the first baseline (see Figure 4). Using that visual, I was able to see that players typically reach top acceleration at 10 feet and top speed at around 35 feet. In terms of time, most players reach 10 feet around 0.85 seconds after contact, and most players reach 35 feet around 2 seconds after contact. Thus, I arbitrarily decided that within the model, fielders would be limited to 50% their top speed in the first 0.85 seconds after contact, and then they would be limited to 80% of their top speed between 0.85 and 2 seconds after contact before being able to truly travel at their top speed.

9.4. Fielder Path Model

The fielder path model required the most tuning time and overall gave me the most trouble. The main reason for this was because I needed the model to predict on newdata differently than how the model used predictions when building the forest. This is because when creating fielder paths, I needed the forest to make predictions using its previous field_x and field_y predictions, which is not a capability that ranger supports. This caused the earlier creations of the model to overfit on the prev_field_x and prev_field_y variables such that it would pretty much only use prev_field_x and prev_field_y to make predictions on new data. Which when prev_field_x and prev_field_y are moving on their own is fine, but when you want to make predictions where prev_field_x and prev_field_y are based off your previous predictions, it causes the predictions to pretty much never change after a second or two into the play. I tried using various time series models to see if perhaps they gave better predictions, but they actually gave worse predictions than the ranger model. I also considered using an XGBoost model due to its prediction capabilities, but I figured

the model would take a long time to run and XGBoost models typically take up a lot of memory, so I kept the ranger model while making some adjustments. In an attempt to solve this, I tried to implement a prediction system within my tuning functions that was similar to the prediction system that would be used in my Shiny App to limit the model from overfitting onto prev_field_x and prev_field_y. However, this led me to my second problem: tuning time. Predicting on my new prediction system takes a lot more time because it has to make predictions row by row rather than on all the data at once. This took a significant amount of time, and sometimes took me an entire day or two just to tune 1 of 8 models. I tried decreasing the number of trees to 100 at first, but the model didn't smooth over well without at least 200-300 trees. Knowing that I would probably be constantly going back to retune models, I decided it was in my best interests to significantly lower the testing dataset in order to save some time and tune models faster. Each testing dataset was decreased from roughly 40 plays (4000 rows, each play was on average 100 rows) to 10 plays (1000 rows), which while it made the tuning process much quicker, it also significantly increased the possibility of overfitting the testing data due the little amount of plays I was testing the model on (For level 2A, I used 750 rows because at the time wanted to train on as much data as possible and level 2A had significantly less data than the rest of the levels. If I had a chance to do it over, I would use 1000 rows in the test data of level 2A). In the future with more time, I would use more testing data so I could be more aggressive in the tuning process in order to minimize error more convincingly. I used the same extremely randomized trees method for these models as I did in the catch probability model due to better performance. I used 2 new hyperparameters for these models: split.select.weights & always.split.variables.

Split.select.weights allows the user to add weights (from 0 to 1) to each of their independent variables to either increase or decrease the probability that it gets chosen by mtry. The ranger function takes this vector of values and normalizes them afterwards, so the inputs don't need to sum up to 1. I used the optim function to find the optimal values for each variable. Even the smallest changes to the weights on these independent variables saw big increases in model accuracy, especially at the lower levels. Always.split.variables allows the user to choose which variables in the model should be available for splitting at every node in addition to the variables randomly chosen by mtry. For my model, I chose to make final ball position x available at every node for my field_x models and final ball position y available at every node for my field_y models. I did this so that even if the difference was big at the middle of the route, it would hopefully begin to converge on the place the fielder ended up by the end of the route (see Figure 17). I also held mtry at 5 for all models (except the field_x model for level 1A) to further reduce the chance of the models overfitting on prev_field_x and prev_field_y (I made the mtry 6 for the level 1A field_x model because it saw a significant improvement in accuracy that can't be said for the other models). Unfortunately, even then, I couldn't make each model as good as they could've been because of the memory constraints imposed by Shiny. I go more in-depth on this topic in the Shiny section of the appendix (9.5), but in order to solve this problem I needed to use a larger min.node.size value as

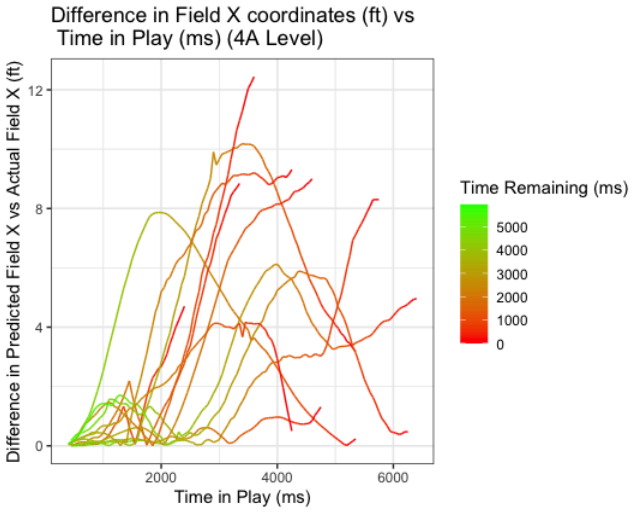

Figure 17: Margin of error of the Fielding Path Model throughout the route. Model has the toughest time identifying field x positions at the middle of the route. Keep in mind that the field_x models tend to have higher errors due to the larger range of values.

well as a smaller amount of trees, which hindered the predicting capabilities of the level 4A and level 1A models specifically. Despite this, I was able to get almost all the models within an RMSE of 6 feet with the exception of the field_x model for level 1A. I think with more time and possibly with more data, these models have the potential to be extremely useful when evaluating the defense of prospects compared to their peers. Among the improvements in section 7, another possible improvement in the model could be making predictions over longer time periods (ie made predictions every 200 ms rather than every 50ms). Perhaps if I made the time between predictions bigger, it could make the model more accurate since bad predictions wouldn't compound on themselves as often and the previous field_x and field_y values wouldn't be as close to the next field_x and field_y values.

9.5. Shiny App

The Shiny App contains 4 tabs that the user can experiment with: OAA and Route Leaderboards, the Fielder Path Model, Route Optimization Model Examples, and the Route Optimization Model. The OAA and Route Score leaderboards allow the user not only see each fielder's OAA and Route Score, but also their percentiles to easily compare how each player ranks compared to his peers (note that while the leaderboards are grouped

OAA ROUTE ALL

Show 10 entries Search:

	Fielder ID	Position	Level	Season	Plays	OAA	Avg. OAA Per Play	OAA Percentile	Avg. OAA Per Play Percentile
	All	["CF"]	["3A"]	All	All	All	All	All	All
4	461	CF	3A	Season_1883	14	1.07	0.08	95.5	73.1
15	475	CF	3A	Season_1883	6	0.28	0.05	79.1	67.2
36	475	CF	3A	Season_1884	2	-0.08	-0.04	47.8	37.3
55	337	CF	3A	Season_1883	1	-0.41	-0.41	19.4	1.5
59	651	CF	3A	Season_1884	11	-0.55	-0.05	13.4	34.3

Figure 18: OAA leaderboard filtered by position (CFs) and Level (3A)

by each unique fielderID, Position, Level, and Season combination, the percentiles are not grouped in this manner). The table also allows for unique filtering capabilities such that you can view the leaderboards by level, position, fielderID, minimum number of plays, etc. (see Figure 18).

The Fielder Path and Route Optimization models (2nd and 4th tab) allow users to create their own hypothetical situations. Looking at the route optimization model, you start by selecting the fielder's top speed and then the length of the desired play (from pitch release to catch/ball bounce). Then, you can select your fielder's start position and final

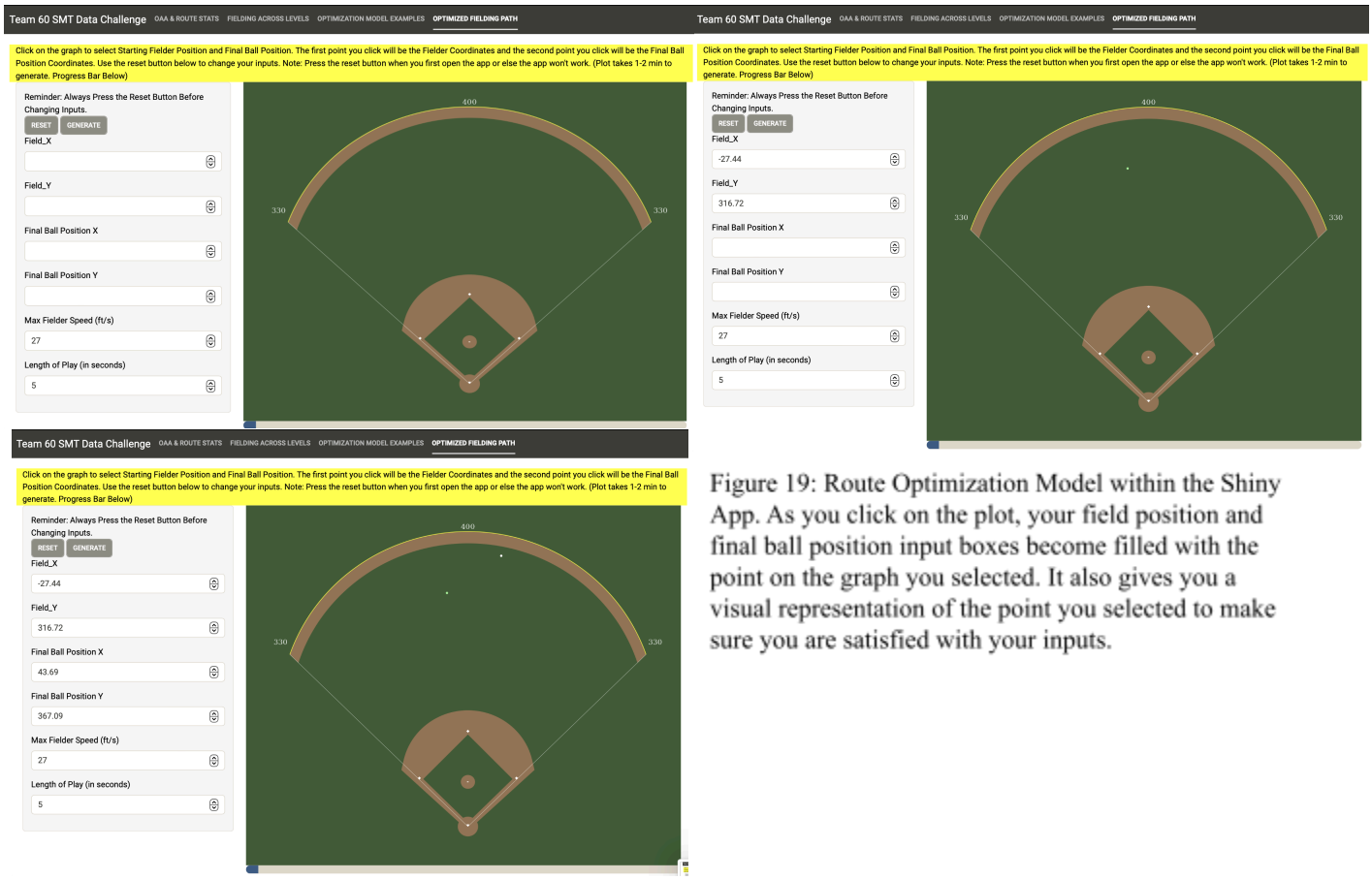

Figure 19: Route Optimization Model within the Shiny App. As you click on the plot, your field position and final ball position input boxes become filled with the point on the graph you selected. It also gives you a visual representation of the point you selected to make sure you are satisfied with your inputs.

ball position. You can do so by clicking on the plot. The first point you click will set your starting fielder position, and the second point selects the final ball position. The graph will show the position of each as you click the graph, with the light green dot representing the

starting fielder position and the white dot representing the final fielder position (see Figure 19). If you want to change your inputs, press the reset button to start over (note: you always have to press the reset button when inputting new values, even when you first open the app). Once you're satisfied with your inputs, you can press the generate button and wait for your play to be created. These animations take some time to make backend calculations and typically take anywhere from 1-3 minutes to generate. I added a progress bar at the bottom of each plot to help the user evaluate how much longer until their play is created. Once loaded in, you can play the animation and use the slider at the bottom to stop at certain timestamps of the animation. You can zoom in on the plot and hover over the points to gather their xy coordinates throughout different points in the play. These features are available in the top right of the plot (see Figure 20).

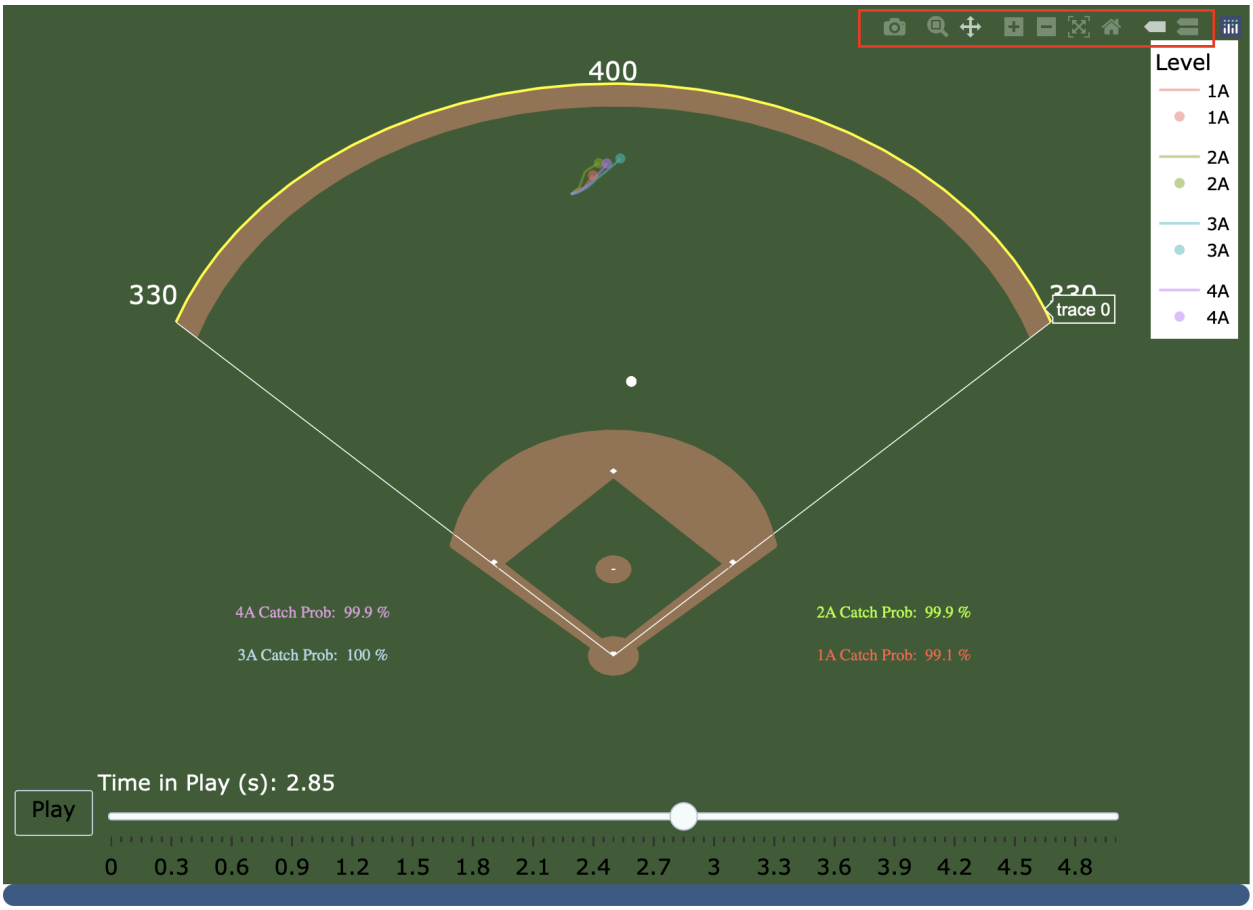

Figure 20: The red box shows the different features available with this graph. If you hover over them, they will describe but they do. The 2 boxes with + and - signs allow you to zoom in and out on the graph while the 4 hours allows you to move around while zoomed in. Note that most of these functions only work while the animation is not moving.

On the fielder path graph, you can also isolate a single line by either clicking the other 3 lines on the legend to remove them from the plot or double-clicking on the desired level you want to isolate (see Figure 21). You'll also see the blue line below the plot. This

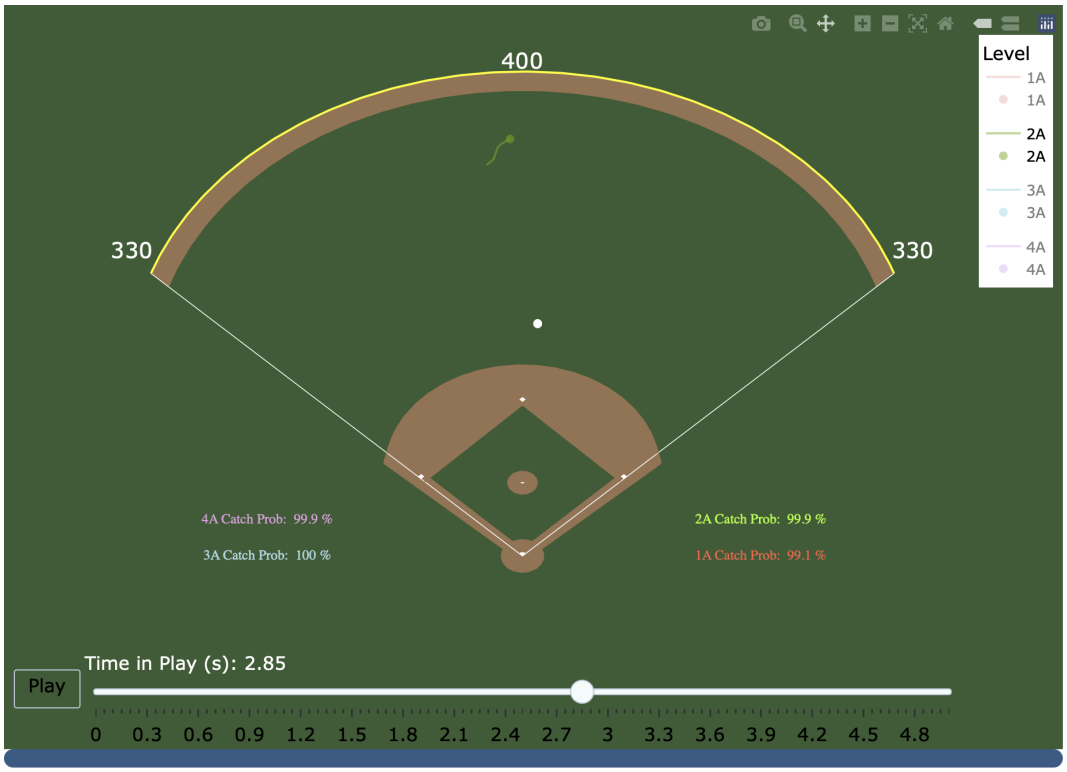

Figure 21: I isolated the level 2A fielder path by clicking on the other 3 levels in the legend in order to remove them from the plot. You can also do this by double clicking on the 2A level in the legend. You don't have to isolate one line either. If you wanted to look at 2A and 3A together, then you can simply click on level 1A and 4A in the legend to remove only those two levels.

is what the progress bar will look like when it's full and the plot is ready.

The optimized model examples (3rd tab) contain 6 different plays (2 at each outfield position) using real data that compares both the optimized route path (light green) and the fielder route path (light blue) (see Figure 22). This is a great look at how this could be used in player evaluation and/or film setting to help outfielders perfect their routes.

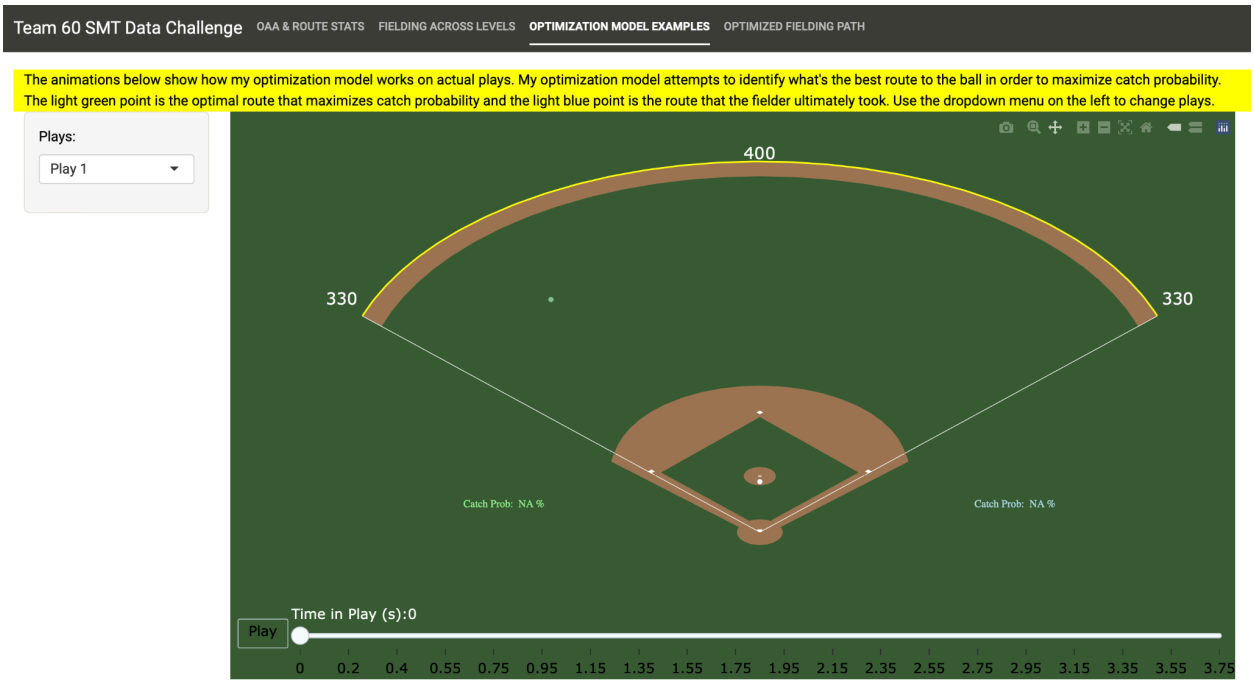

Figure 22: This is the display of the Route Optimization Model Examples. There are 6 plays total, where the plot compares the optimized route path (light green) to the fielder route path (light blue)

I mentioned before that you can only run the fielder path model once before you will have to exit the app and reopen it, or else the application will crash. This has to do with the shiny server itself and there isn't anything I can do with it. On the free plan, Shiny only allows for 1 GB of memory to be used for each Shiny App session upon opening it. Due to the size of my fielder path models, I have precisely enough memory available to run the Fielder Path model once before my application runs out of available memory. I did some digging and found that there is a paid subscription plan that allows for the use of greater memory (8GB, to be exact) in one session. Unfortunately, this plan costs $49/month or $550/year. I'm not paying that much for extra memory, so this will have to do.