Noah Hewes, Benjamin Schmid, Noah Swan
Dr. Zhang
Data Mining and Predictive Analysis
20 December 2022

Phase II Report: Statcast Model of Balls Hit Into Play

Introduction

In 2015, with analytics becoming more prominent in sports and the rise of Sabermetrics, Major League Baseball (MLB) implemented a new analytical system in all 30 of its stadiums: Statcast. This technology integrates doppler radar and high definition video to take high-speed, high-accuracy measurements of player movements and athletic abilities, creating the ability for all player movements and aspects of the game to be tracked. Each team in the MLB now has an analytics team that uses this data in hope of gaining a competitive advantage. Our project goal, using Statcast data, is to determine if we can create a model that will accurately predict if a ball that is hit into play will result in a hit or an out. Doing so will allow us to see which aspects of the pitch or the conditions under which it is thrown most influence the chances of it resulting in a hit. Players can use this information to focus their training and improve their skills.

Methodology

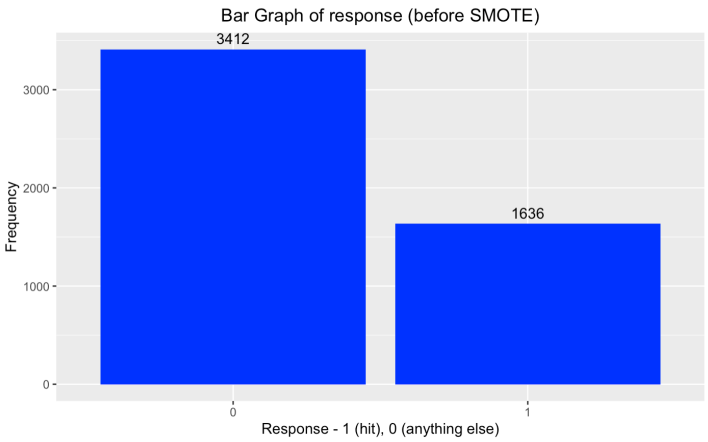
Data Introduction and Preparation

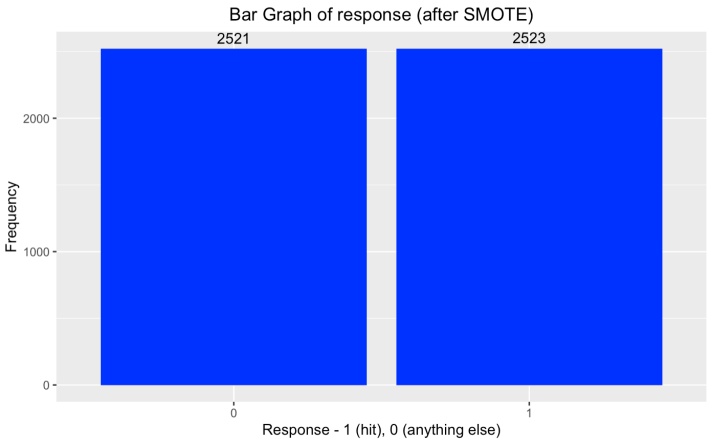
The full dataset that we will be using is the Statcast pitching dataset from 2021. This dataset consists of all pitches thrown in the 2021 MLB season. It is quite large, with 93 columns and 709,851 rows. Filtering it down to answer our question, we are left with 121,707 pitches hit into play. This was still too large of a dataset, so we decided to reduce it to a 16 day stretch of the season, days 48 to 64, which decreases our number of rows to 10,096. For Phase I exploration, we reduced the columns down to our created response variable and 18 potential predictors, some of which were created by us. Through continued work, we reduced our list down to 11 predictors, which will be included in all models.

Our dataset has very few missing values. The full dataset has missing values in about 1% of rows, but after filtering down to only the rows and columns of interest to us for Phase II, we have a dataset with only 48 of the 10,096 (.475%) rows having missing values. These values are in the variables launch_speed and launch_angle (21), and release_spin_rate (27). Since there are so few of these missing values, we decided that it would be best to simply impute the mean for these values. Other methods, including imputation by random forest, were considered but seemed excessive for so few missing values. We did imputation prior to addressing the class imbalance issue.

As was discussed in Phase I, our response variable has a somewhat concerning class imbalance. Below left is the distribution of the response variable before we worked with it. We see a roughly 68/32 split between the response levels. We considered multiple methods to combat this and ultimately decided to use Synthetic Minority Oversampling Technique (SMOTE). This is a technique that creates synthetic samples for the minority case using the K-Nearest Neighbors algorithm, which in our case is hits (resp=1), while also undersampling the majority case. We wrote the code so that the algorithm undersampled and oversampled by roughly the same amount to arrive at a balanced dataset. Below right is the distribution of the response variable in our new training dataset with SMOTE applied. This dataset now has 10,092 rows and the new training set has a class imbalance split of 50.3/49.7, which is much better.

Please note that both graphs below show the distribution of the response variable in the training dataset only.





Variables

As stated, through our work in Phase I and some continued exploration in Phase II, we have arrived at the following list of variables to include in our models. It is important to note that through our work and much discussion, we do not feel that there is any reason to transform any variables as the numerics were all approximately normal. Additionally, we do not feel it is necessary to include any interaction terms in our models. The numerics are not correlated with each other and interactions of categorical variables would add too many dummy variables for useful addition to the model. The table below lists all variables included in our model with their respective role, measurement level, and description.

Variable Name	Model Role	Measurement Level	Description
response	target	binary	0 = out, 1 = hit
launch_speed	input	nominal	Speed of ball upon contact (mph)
release_speed	input	nominal	Speed of ball upon release (mph)
pitch_type	input	nominal	Type of pitch. Levels: Fastball, OffSpeed, BreakingBall
outs_when_up	input	interval	Outs in the inning during at-bat. Levels: 0, 1, 2
inning	input	nominal	Inning of the game. Early = 1-3, Mid = 4-6, Late = 7-12
runners_on	input	binary	Runners on base during at-bat? Levels: no, yes

launch_angle	input	nominal	Angle of ball upon contact (degrees)
balls	input	interval	Number of balls in count during at-bat. Levels: 0, 1, 2, 3
strikes	input	interval	Number of strikes in count during at-bat. Levels: 0, 1, 2
release_spin_rate	input	nominal	Spin rate of ball upon release (rpm)
release_pos_z	input	nominal	Height of the ball upon release (ft)

Many of the above variables are in the same form in which they are in the original dataset. However, some variables had to be created. One of those is our response variable. To answer our question, we needed to narrow our dataset down from all pitches thrown to just pitches where contact was made. We did this by filtering the description variable, which gives a description of the resulting pitch, to "hit_into_play." We could then make our binary response variable be a 1 if the events variable, which states the event of the resulting plate appearance, is a single, double, triple, or home run. Otherwise, the response variable would be a 0.

Three of our input variables needed to be consolidated down as they were not as simple as we desired. The original dataset had three variables, runner_on_1b, runner_on_2b, and runner_on_3b, which seemed like overkill, so we combined these binary variables into one which simply stated if there was a runner on any of the bases. The pitch_type variable has 11 categories: SI (sinker), FF (four-seam fastball), FC (cutter), FS (splitter), FA (fastball), CU (curveball), SL (slider), KC (knuckle-curve), CS (slow curve), CH (change-up), and EP (eephus). To reduce the number of categories into a more manageable number, we did the following: SI, FF, FC, and FA were grouped into the Fastball category. CU, SL, KC, and CS were grouped into the BreakingBall category. Finally, CH, FS, and EP are grouped into the OffSpeed category. Finally, the inning variable takes the levels of 1 to 12. As this was too many we consolidated these down to three categories: early (innings 1-3), mid (innings 4-6), and late (innings 7-12). We considered using Weight of Evidence and Decision Trees to consolidate this variable, but it seemed most reasonable to use intuition instead.

<u>Model Architecture and Hyperparameters and Evaluation Metrics</u>

We considered 5 models for this project: Decision Tree, Logistic Regression, Artificial Neural Network, Random Forest, and XGBoost. For all of the models, we used the full Phase II list of input variables. Our evaluation metric for all of the models is F score due to our class imbalance issue.

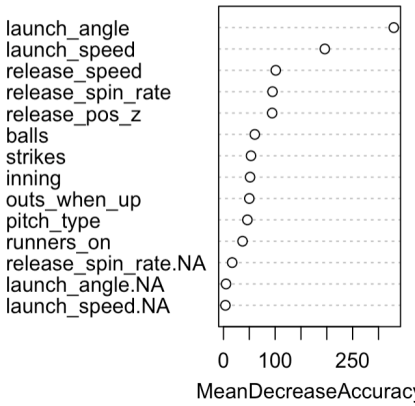
Decision Tree

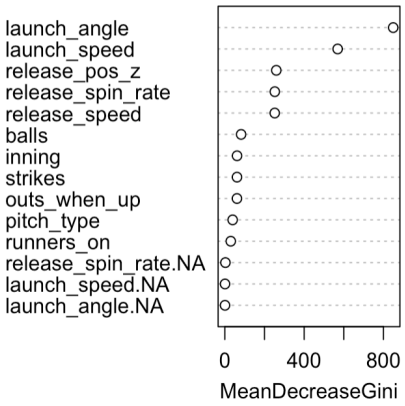
For the Decision Tree model, we used a cp value of .001 for our control. This resulted in a tree with 23 leaves. The variable launch_angle is the most important in the tree, being the split variable for many of the initial splits and providing 52% of the overall contribution in predicting hits. The variable launch_speed is the second most important variable (42%), with a huge drop in importance before any other variable. The third and fourth most important variables, respectively, are release_speed and release_pos_z. The full variable importance is shown below.

launch_angle	launch_speed	release_speed	release_pos_z	release_spin_rate	strikes	balls	inning	pitch_type
576.145362	461.955842	20.736753	10.714137	7.241111	6.674390	6.126488	5.537865	4.757714

Random Forest

In an attempt to obtain the strongest possible model, we left the ntrees parameter at 1000 trees. Our best m parameter by F score is 8. The resulting random forest had high agreement among the two variable importance methods. Both found launch_angle to be the most important, followed by launch_speed. Both have release_speed, release_spin_rate, and release_pos_z rounding out the top 5, though they are in different orders.





Logistic Regression

For this model, we chose to do stepwise regression. Forward and backwards regression resulted in the same model, so any would have worked, but our preference was to use stepwise. We also chose to use BIC so our model is tougher on the predictors. We had issues with logistic regression choosing too many predictors, so this was done in hopes of correcting that.

The stepwise regression chose a final model with the following predictors: launch_speed, launch_angle, balls, and outs_when_up. These variables were all found to be extremely significant with p-values of less than .0002. The AIC of this final model is 12737.

```
Call:
glm(formula = response ~ launch_speed + release_pos_z + outs_when_up +
    balls, family = binomial, data = data.mdl[split.new, ])

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.8841  -1.0970   0.5486   1.0164   2.7185

Coefficients:
             Estimate Std. Error z value             Pr(>|z|)
(Intercept)  -2.734094   0.384085  -7.118     0.00000000000109 ***
launch_speed  0.048689   0.002365  20.586 < 0.0000000000000002 ***
release_pos_z -0.278512   0.054627  -5.098     0.00000034241591 ***
outs_when_up1 -0.040805   0.070555  -0.578             0.563028
outs_when_up2 -0.347689   0.075226  -4.622     0.00000380240423 ***
balls1         0.267391   0.071605   3.734             0.000188 ***
balls2        -0.173258   0.084916  -2.040             0.041315 *
balls3        -0.089106   0.105485  -0.845             0.398262
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 6992.5  on 5043  degrees of freedom
Residual deviance: 6379.0  on 5036  degrees of freedom
AIC: 6395
```

Artificial Neural Network

Our ANN model has one hidden layer with 12 nodes. We picked 12 for our number of nodes as 12 is ⅔ of the dimension of the neural network. With dummy variables, we have 17 predictors, and ⅔ of this is roughly 12. We chose to use the hyperbolic tangent activation function for this hidden layer as we just had one hidden layer. When running the model, we chose to increase the number of epochs to 60 as it resulted in a higher F score. Our group feels that a longer model run time is a warranted trade off for a stronger model.

XGBoost

Through our research, we found that for baseball predictive modeling projects, people have had great success with a supervised learning model XGBoost, or Extreme Gradient Boosting. Thus, we decided to use it in our project. It is a gradient boosting decision tree algorithm that combines weak learners sequentially to correct the errors made by the previous models. Models continue to be added until no further improvements can be made. Many people

choose to use XGBoost as opposed to other models as it is praised for its high efficiency (it can be 10 times faster than its competitors) and its high accuracy. It is often considered an improvement on random forest.

To understand XGBoost, it is crucial to decipher what boosting is. In comparison to the bagging ensemble method learned in class, boosting is another ensemble method that reduces variance (stabilizes results). The key difference is that while decision trees are built in parallel (independently) for bagging, decision trees are built sequentially in boosting, adding new models that do well where the previous models fail. Like bagging, for the number trees specified, boosting draws a random sample with replacement of the training data for each tree. However, while each observation has an equal chance of being chosen for bagging, boosting assigns weights to the observations so that some show up in the random samples more than others. This is exactly how it builds sequentially. After each decision tree is built, the observations in the training data are reweighted such that the misclassified data receives higher weight, and therefore, the subsequent trees will focus on them during their training. As for classification, bagging does this through a simple average (majority vote) of the independent decision trees. On the other hand, boosting allocates weights to each of the decision trees, and obtains a classification result through a weighted average (where a decision tree with a good classification rate on the training dataset receives a higher weight than a poor one).

```r
#XGBoost
data.xgb <- data.new
tune_grid <- expand.grid(
  nrounds = seq(from = 200, to = 500, by = 100), # number of rounds for boosting
  eta = c(0.025, 0.05, 0.1, 0.3), # learning rate for each subsequent tree, higher means more conservative
  max_depth = seq(2,10,length.out = 5), # max depth of a tree, default is 6
  gamma = 0,
  colsample_bytree = 1,
  min_child_weight = 1,
  subsample = 1
)

xgb.fscores <- c()
for(i in 1:nrow(tune_grid)){

tune_control <- caret::trainControl(
  method = "none", # none
  number = 1, # only 1 model
  #index = createFolds(tr_treated$Id_clean), # fix the folds
  verboseIter = TRUE, # no training log
  allowParallel = TRUE # FALSE for reproducible results
)
set.seed(1234)

xgb.int <- data.xgb %>%
  filter(split.new) %>%
  # slice_sample(n = 50000) %>%
  caret::train(
  response ~ .,
  data = .,
  trControl = tune_control,
  tuneGrid = tune_grid[i,],
  method = "xgbTree",
  verbose = TRUE
)
```

The above code shows how we created a grid of hyperparameters to train and determine the best XGBoost model using a for loop. We varied three hyperparameters while leaving the remaining parameters at their defaults. Within the tune_grid, we give values to all of the seven hyperparameters. nRounds is the number of boosting rounds, which is equivalent to the number of trees. Learning_rate, or eta, is the size of the step of each iteration. Max_depth is the maximum depth of each tree, so we do not have overcomplexity. Gamma is the minimum loss reduction required to make a further partition of the tree. We fixed gamma at 0, its default value. Colsample_bytree is the proportion of columns to be randomly sampled for each tree. We fixed this at 1, its default value. Min_child_weight is the minimum sum of instance weight (Hessian)

needed in a child node. If the split results in a node with the sum of instance weight less than the min_child_weight value, then the model process will give up further splitting. We fixed this hyperparameter at 1, which is a conservative value and the default. Subsample is the proportion of observations to be randomly sampled for each tree. We chose 1, its default. We chose to vary nRounds to be one of 200, 300, 400, or 500, eta to be one of .025, .05, .1, or .3, and max_depth to be one of 2, 4, 6, 8, or 10. Thus, we built 80 unique models and used F Score to find the best one. The optimal model as determined by F Score had the following hyperparameter values for those that varied: nRounds of 200, eta of .1, and max_depth of 2.

The XGBoost package includes a function that shows the variable importance of the model. This metric is the scaled gain value so that the top variable has a value of 100. Gain for a specific variable relates to the increase in accuracy provided by the splits that use the variable. Thus, a higher value means the variable is more important. We see that launch_angle is by far the most important variable (scaled gain value of 100), followed by launch_speed (42.62), release_pos_z (3.58), and release_speed (3.22).

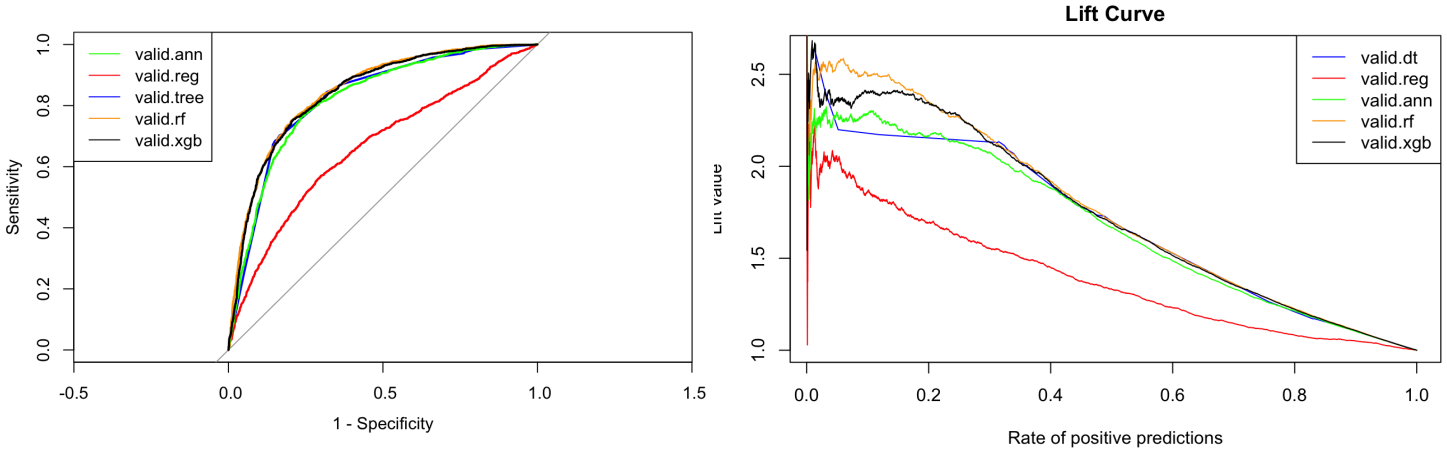
	Overall <dbl>
launch_angle	100.00000000
launch_speed	42.61529056
release_pos_z	3.58383280
release_speed	3.22336171
release_spin_rate	3.02719905
balls1	1.36131771
inningMid	0.94168373
outs_when_up2	0.69825387
runners_onyes	0.36983193
strikes2	0.32210845

Results

Model	**F Score (want highest)**
Decision Tree	.6816
Random Forest	.6833
Logistic Regression	.5196
Artificial Neural Network	.6657
XGBoost	.6879

From the table, based on F score, we see that XGBoost is the best model of the five. Decision Tree, Random Forest, and Artificial Neural Network are close behind, but XGBoost prevails. Our models have strong agreement in variable importance. All of them but logistic regression found launch_angle and launch_speed to be the two most important. This gives us good confidence since our models seem to strongly agree, while also explaining why the F Score for Logistic Regression is so much lower than those of the other models.

We further evaluated our models using an ROC chart (left), and lift graph (right), both of which are shown below.



These plots again indicate that the Logistic Regression model underperforms in comparison to the other 4 models, and that the 4 other models perform strongly. Looking specifically at the lift graph, for the top roughly 2% of observations (as ordered by predictive probability) the XGBoost model has the highest lift, before being overtaken by the Random Forest model. Overall, the Random Forest model seems to perform best on the top 20% of observations. At around the 40th percentile, all of the models besides the Logistic Regression converge. One thing to note is that all of the models have lift values above 1, indicating that they all are an improvement over randomly guessing whether a ball hit into play resulted in a hit or an out.

Conclusion

We have seen through all of our analysis that XGBoost is the best of the five models, although Decision Tree, Random Forest, and Artificial Neural Network are not far behind. We also see that these models agree that launch_angle is the best predictor. When starting this project, we knew that it would be tough to get a high accuracy model, but are very pleased with our results. With solid accuracy, we are able to predict when a ball hit into play will result in a hit.

Our project is not perfect. We are limited by time, ability, and data availability. As sports analytics is such a hot topic of work, new variables and metrics are becoming available. Future projects could include these more complex variables to make a stronger prediction. The dataset we started with was very large. With more time, we could have included more predictors to better our models. More detailed projects could go further in depth to our problem and provide deeper insights about baseball happenings. Finally, our models are tough to interpret. The inclusion of XGBoost was done to give us the best possible model, but it, along with the other models, can be very hard to interpret and visualize. We believe that a sacrifice of interpretability is justified for higher predictive power, but more time and research could allow us to do this better.