baseball-hit-probability

Katie Sharp

2020-06-21

Contents

Preface

This data science project is required for the ninth and final course in the HarvardX Professional Certificate in Data Science (HarvardX: PH125.9x). For this project we were instructed to select our own dataset and apply machine learning techniques to solve a problem of our choice.

Introduction

The goal of this project is to predict whether a batted ball is a hit or an out using data from MLB.com's Statcast database. This database contains every pitch thrown in MLB from 2008-present, and includes 90 different variables that measure characteristics of the pitch and the batted ball, and details about the game conditions. The documentation and description of the variables can be found here. For my analysis I used data from the 2019 season only. I also filtered the dataset to include only balls put into play. This subset contains 127,535 observations (i.e. batted balls).

Overview

Whether a batted ball put into play becomes a hit or an out depends a variety of factors, including the skill of the batter, quality and positioning of the defense, configuration of the ballpark, and trajectory of the ball.

The introduction of the Statcast tracking system in 2015 was a major breakthrough for baseball statistics, adding a massive amount of pitch- and play-level data that could be processed and analyzed in ways that were previously never possible. The technology has two tracking systems: a Doppler radar and high-definition cameras. The radar tracks everything related to the baseball, including pitch speed, pitch movement, batted ball velocity and launch angle, and batted ball distance. The camera tracks the movement of the players on the field, including player speed, distance, and direction for every play.

In 2017, MLB.com introduced its own version of "hit probability", a metric that calculates the probability of a batted ball becoming a hit based only on **exit velocity and launch angle** (the exact model specifications and algorithm are unknown, except for those two variables). Our goal is to create a more robust hit probability model and metric that incorporates additional information about the batted ball and accounts for other factors such as defense and ballpark.

We will use the Statcast dataset to train several machine learning algorithms and ultimately build a predictive model using the best-performing algorithm from our testing. This final model will be evaluated on a `validation` subset of our working dataset. The performance measures for model evaluation are: accuracy, specificity, sensitivity (recall), precision and F1 score.

The key steps in the project include: data import and preprocessing, data exploration and visualization, methods/analysis, results summary and model performance, and important conclusions.

Data import and preprocessing

Here are the required libraries to load for the project:

```
# libraries must be loaded in this order
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
if(!require(grid)) install.packages("grid", repos = "http://cran.us.r-project.org")
if(!require(gridExtra)) install.packages("gridExtra", repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
if(!require(e1071)) install.packages("e1071", repos = "http://cran.us.r-project.org")
if(!require(ROCit)) install.packages("ROCit", repos = "http://cran.us.r-project.org")
if(!require(cvms)) install.packages("cvms", repos = "http://cran.us.r-project.org")
if(!require(rpart)) install.packages("rpart", repos = "http://cran.us.r-project.org")
if(!require(randomForest)) install.packages("randomForest", repos = "http://cran.us.r-project.org")
if(!require(mlr)) install.packages("mlr", repos = "http://cran.us.r-project.org")
if(!require(parallel)) install.packages("parallel", repos = "http://cran.us.r-project.org")
if(!require(parallelMap)) install.packages("parallelMap", repos = "http://cran.us.r-project.org")
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-project.org")
if(!require(xgboost)) install.packages("xgboost", repos = "http://cran.us.r-project.org")
```

Data import

The data for the 2019 season was scraped from the Statcast website using the baseballr package. The code for the scraping process can be found in a separate **scrape_2019statcast_data.R** file on my github site. I then uploaded the .csv file (which was 346 MB) to Dropbox, where it can now be downloaded by anyone to a local computer using this code:

Data preprocessing

The first step is to filter the dataset for only balls in play. This reduces our data to 127,535 observations.

```
dat_ip <- bat_dat %>% filter(type == "X")
rm(bat_dat, dl)
```

Next, we know that some of the columns contain all NAs because they represent deprecated variables from prior versions of the database. We remove those columns with this code:

```
dat_ip <- dat_ip %>% select_if(~!all(is.na(.)))
```

We need to add some variables to help us build the model. First, we'll add our outcome variable – whether the batted ball becomes a hit or not.

```
dat_ip <- dat_ip %>% mutate(hit = ifelse(events %in% c("single", "double", "triple", "home_run"), 1, 0))
dat_ip$hit <- as.factor(dat_ip$hit)
```

We also want a variable for the horizontal spray angle of the batted ball. Statcast does not give a direct measurement of that variable, but it does include two pieces of related information, `hc_x` and `hc_y`, which are the hit coordinates (x,y) of the batted ball. We can use basic trigonometry to calculate the angle, and then make an adjustment for batter handedness. Thanks to Jim Albert for providing the math to calculate the horizontal spray angle.

```
dat_ip$spray_angle <- with(dat_ip, round(atan((hc_x-125.42)/(198.27-hc_y))*180/pi,1))
dat_ip$adj_spray_angle <- with(dat_ip, ifelse(stand == "L",
                                              -spray_angle, spray_angle))
```

Next, we add a few variables to account for the defensive team and positioning of the fielders. Here is the code for the fielding team:

```
dat_ip$fieldTeam <- with(dat_ip, ifelse(inning_topbot == "bot", away_team, home_team)) %>% as.factor()
```

Our measure of positioning is whether the infielders and/or outfielders were in a shifted alignment or a standard alignment.

```
dat_ip$inf_pos <- with(dat_ip, ifelse(if_fielding_alignment %in% c("Infield shift", "Strategic"), "inf_
dat_ip$of_pos <- with(dat_ip, ifelse(of_fielding_alignment %in% c("4th outfielder", "Extreme outfield sh
```

Now we are ready to subset our working dataset, which selects the relevant columns for our models and filters out any other NA values. Using the Statcast csv documentation as a reference guide, I choose the following variables from the original dataset: `home_team` (to account for park-specific factors), `stand` (whether the batter is on the left or right side of the plate), `hit_distance_sc` (projected distance of batted ball), `launch_speed` (exit velocity of batted ball), and `launch_angle` (vertical spray angle of batted ball). We also need to convert some of those desired variables from characters to factors.

```
dat_ip$home_team <- dat_ip$home_team %>% as.factor()
dat_ip$stand <- dat_ip$stand %>% as.factor()
dat_models <- dat_ip %>% select(home_team, fieldTeam, inf_pos, of_pos, stand, hit_distance_sc, launch_sp
  filter(!is.na(hit_distance_sc)) %>%
  filter(!is.na(launch_speed)) %>%
  filter(!is.na(launch_angle)) %>%
  filter(!is.na(adj_spray_angle))
```

Our clean working dataset is a dataframe with 10 variables and more than 100,000 observations.

```
str(dat_models)
```

```
## 'data.frame':    115577 obs. of  10 variables:
##  $ home_team      : Factor w/ 30 levels "ARI","ATL","BAL",..: 22 22 22 22 22 22 22 22 22 22 ...
##  $ fieldTeam      : Factor w/ 30 levels "ARI","ATL","BAL",..: 22 22 22 22 22 22 22 22 22 22 ...
##  $ inf_pos        : Factor w/ 2 levels "inf_shift","inf_standard": 2 2 2 1 2 2 2 2 2 2 ...
##  $ of_pos         : Factor w/ 2 levels "of_shift","of_standard": 2 2 2 2 2 2 2 2 2 2 ...
##  $ stand          : Factor w/ 2 levels "L","R": 2 1 2 2 2 1 2 2 2 2 ...
##  $ hit_distance_sc: int  14 4 29 322 6 256 32 6 307 160 ...
##  $ launch_speed   : num  92.1 73.4 96.9 89.5 59.4 97.8 92.2 68.1 86.2 78.9 ...
##  $ launch_angle   : num  -8.4 -33.1 -2.7 31.4 -22.3 52.4 -2.1 -18.4 21.1 62.5 ...
##  $ adj_spray_angle: num  25 -19.6 -1.1 13.9 0.8 -29.9 -37.7 -18.9 -21 44.4 ...
##  $ hit            : Factor w/ 2 levels "0","1": 1 1 2 1 1 1 1 1 1 1 ...
```

To get an idea of the distribution our outcome variable, here's the proportion of hits ("1") and outs ("0") in our entire working dataset.

```
table(dat_models$hit)
```

```
##
##     0     1
## 74493 41084
```

Our final preprocessing step is to create our `validation` set, which will be 20 percent of the working dataset, and will **only be used for a final test of our algorithm** at the end of the project. The remaining data will be stored in file called `dat_train` and will be used for training and testing our algorithm. We use an 80/20 split, which is general rule of thumb to follow for datasets of our size (~100,000 observations) and will allow us to cross-validate when training.

Let's check now to make sure that our train and validation sets are balanced, i.e. have a similar proportion of hits and outs.

```
prop.table(table(dat_train$hit))
```

```
##
##         0         1
## 0.6445312 0.3554688
```

```
prop.table(table(validation$hit))
```

```
##
##         0         1
## 0.6445319 0.3554681
```

Data Exploration and Visualization

Before training and testing our models, let's explore our train dataset. To get a basic overview, we'll first look at the first few rows to make sure its in a tidy format.

```
head(dat_train)
```

```
##   home_team fieldTeam      inf_pos      of_pos stand hit_distance_sc
## 1       PIT       PIT inf_standard of_standard     R              14
## 2       PIT       PIT inf_standard of_standard     L               4
## 4       PIT       PIT    inf_shift of_standard     R             322
## 5       PIT       PIT inf_standard of_standard     R               6
## 7       PIT       PIT inf_standard of_standard     R              32
## 9       PIT       PIT inf_standard of_standard     R             307
##   launch_speed launch_angle adj_spray_angle hit
## 1         92.1         -8.4            25.0   0
## 2         73.4        -33.1           -19.6   0
## 4         89.5         31.4            13.9   0
## 5         59.4        -22.3             0.8   0
## 7         92.2         -2.1           -37.7   0
## 9         86.2         21.1           -21.0   0
```

Looks good so far. Now we'll examine the structure of the data, to get an idea of the dimensions and attributes.

```
str(dat_train)
```

```
## 'data.frame':    92461 obs. of  10 variables:
##  $ home_team      : Factor w/ 30 levels "ARI","ATL","BAL",..: 22 22 22 22 22 22 22 22 22 6 ...
```

```
##  $ fieldTeam      : Factor w/ 30 levels "ARI","ATL","BAL",..: 22 22 22 22 22 22 22 22 22 6 ...
##  $ inf_pos        : Factor w/ 2 levels "inf_shift","inf_standard": 2 2 1 2 2 2 2 2 2 2 ...
##  $ of_pos         : Factor w/ 2 levels "of_shift","of_standard": 2 2 2 2 2 2 2 2 2 2 ...
##  $ stand          : Factor w/ 2 levels "L","R": 2 1 2 2 2 2 2 2 2 1 ...
##  $ hit_distance_sc: int  14 4 322 6 32 307 352 41 127 322 ...
##  $ launch_speed   : num  92.1 73.4 89.5 59.4 92.2 ...
##  $ launch_angle   : num  -8.4 -33.1 31.4 -22.3 -2.1 21.1 16.6 2 10.4 25.2 ...
##  $ adj_spray_angle: num  25 -19.6 13.9 0.8 -37.7 -21 14 -11.3 -12.8 33.9 ...
##  $ hit            : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 2 1 1 1 ...
```

Here we show a more detailed summary of each variable and its distribution:

```
summary(dat_train)
```

```
##    home_team       fieldTeam             inf_pos              of_pos
##  COL    : 3422   COL    : 3422   inf_shift   :31346   of_shift   : 8629
##  HOU    : 3372   HOU    : 3372   inf_standard:61115   of_standard:83832
##  WSH    : 3351   WSH    : 3351
##  BAL    : 3284   BAL    : 3284
##  DET    : 3213   DET    : 3213
##  PIT    : 3207   PIT    : 3207
##  (Other):72612   (Other):72612
##  stand     hit_distance_sc  launch_speed     launch_angle    adj_spray_angle
##  L:37866   Min.   :  0     Min.   :  7.60   Min.   :-87.20   Min.   :-89.800
##  R:54595   1st Qu.: 21     1st Qu.: 80.60   1st Qu.: -4.50   1st Qu.:-28.100
##            Median :174     Median : 91.60   Median : 12.40   Median : -9.200
##            Mean   :173     Mean   : 88.97   Mean   : 11.53   Mean   : -5.461
##            3rd Qu.:299     3rd Qu.: 99.50   3rd Qu.: 28.50   3rd Qu.: 16.800
##            Max.   :487     Max.   :118.90   Max.   : 89.50   Max.   : 89.900
##
##  hit
##  0:59594
##  1:32867
##
##
##
##
##
```

We can see that the "average" batted ball was hit a distance of 173 feet, with an exit velocity of 88.97 mph, at vertical angle of 11.53 degrees and a horizontal angle of -5.461 degrees. Remember, we calculated an adjusted horizontal spray angle, so that a negative number corresponds to a batted ball that is pulled and a positive number is a batted ball hit to the opposite field (and 0 is a batted ball hit up the middle).

Outcome variable

This is a binary classification problem, so there are two outcomes: hit or out. Here's the frequency and percentage of each outcome (0 is out, 1 is hit). Approximately 35% of batted balls in our data set are hits.

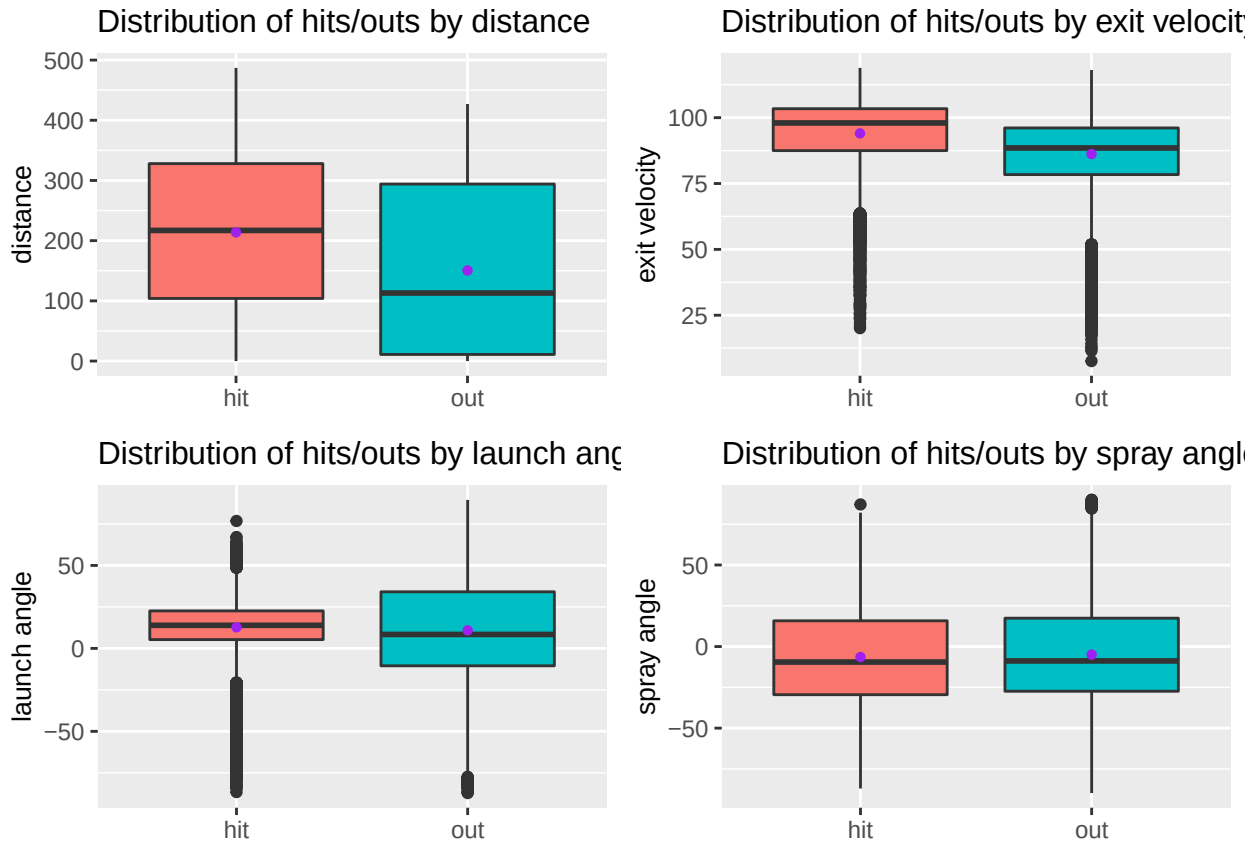
```
percent <- 100*prop.table(table(dat_train$hit))
cbind(freq=table(dat_train$hit), percent=percent)
```

```
##    freq  percent
## 0 59594 64.45312
## 1 32867 35.54688
```

Batted ball trajectory variables

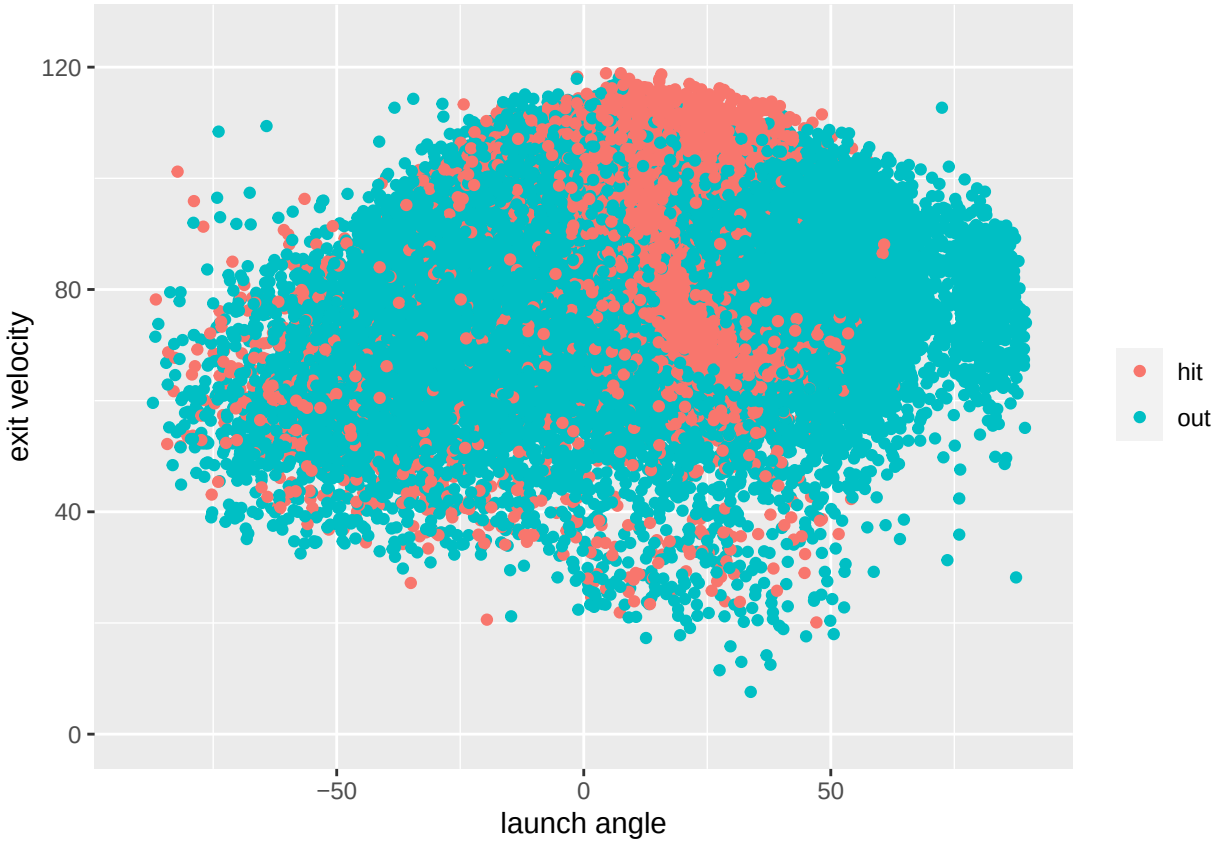
To further analyze the predictor variables and understand their significance, we'll split the analysis into two parts. First we'll examine the numeric variables related to the trajectory of the batted ball: distance, exit velocity, launch angle, spray angle. Below is a boxplot for each of these four variables: the vertical line shows numeric range of the variable, the boxes are the first and third quantiles, the horizontal line in each plot is the median, and the purple dot is the mean.

```
boxp_dist <- dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
  ggplot(aes(x=hit, y=hit_distance_sc, fill=hit)) +
  geom_boxplot() +
  stat_summary(fun = mean, geom = "point", shape = 16, size = 1.5, color = "purple") +
  theme(legend.position = "none") +
  labs(y="distance", title = "Distribution of hits/outs by distance") +
  theme(plot.title = element_text(size = 12),
        axis.title = element_text(size = 10),
        axis.title.x = element_blank())

boxp_ev <- dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
  ggplot(aes(x=hit, y=launch_speed, fill=hit)) +
  geom_boxplot() +
  stat_summary(fun = mean, geom = "point", shape = 16, size = 1.5, color = "purple") +
  theme(legend.position = "none") +
  labs(y="exit velocity", title = "Distribution of hits/outs by exit velocity") +
  theme(plot.title = element_text(size = 12),
        axis.title = element_text(size = 10),
        axis.title.x = element_blank())

boxp_la <- dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
  ggplot(aes(x=hit, y=launch_angle, fill=hit)) +
  geom_boxplot() +
  stat_summary(fun = mean, geom = "point", shape = 16, size = 1.5, color = "purple") +
  theme(legend.position = "none") +
  labs(y="launch angle", title = "Distribution of hits/outs by launch angle") +
  theme(plot.title = element_text(size = 12),
        axis.title = element_text(size = 10),
        axis.title.x = element_blank())

boxp_sa <- dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
  ggplot(aes(x=hit, y=adj_spray_angle, fill=hit)) +
  geom_boxplot() +
  stat_summary(fun = mean, geom = "point", shape = 16, size = 1.5, color = "purple") +
  theme(legend.position = "none") +
  labs(y="spray angle", title = "Distribution of hits/outs by spray angle") +
  theme(plot.title = element_text(size = 12),
        axis.title = element_text(size = 10),
        axis.title.x = element_blank())

grid.arrange(boxp_dist, boxp_ev, boxp_la, boxp_sa, ncol = 2)
```



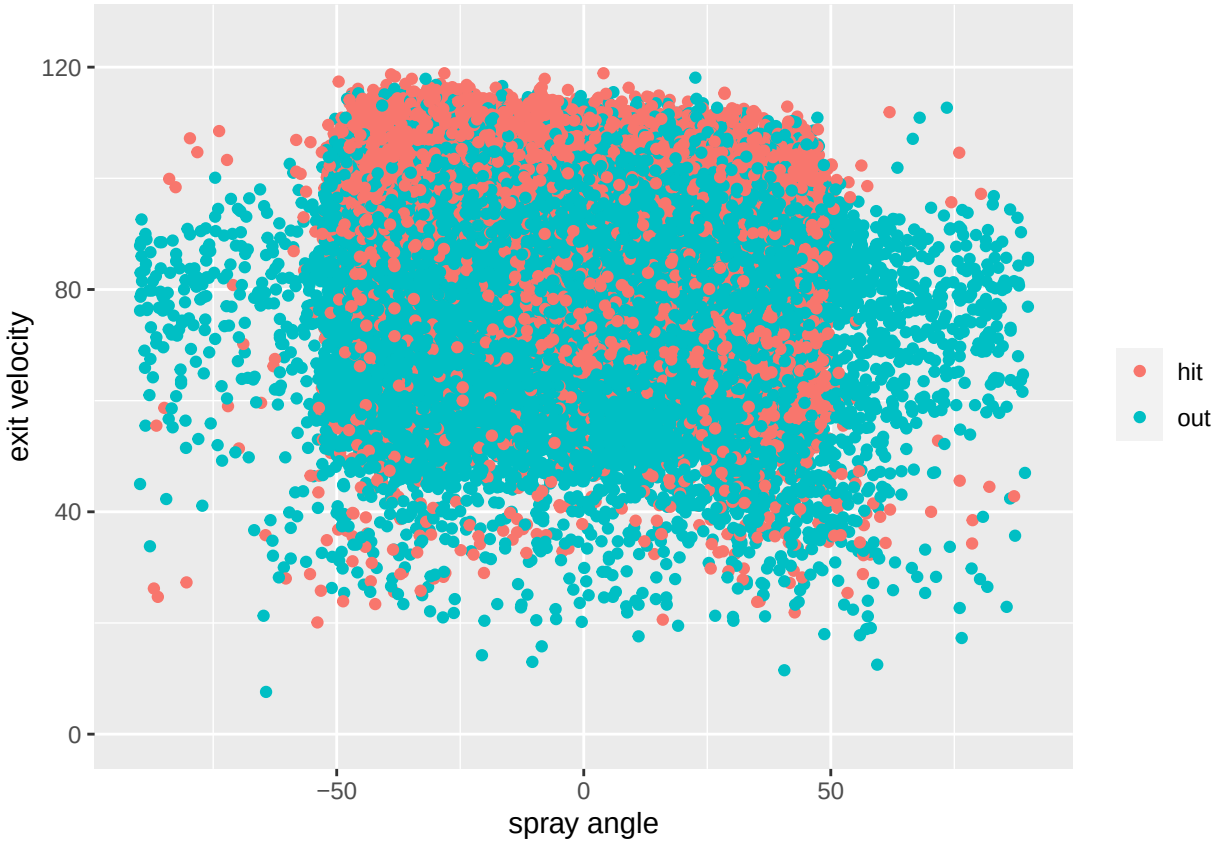
There are some interesting observations from this data. First, the distribution of the spray angle is similar for hits and outs. The plots for distance and exit velocity do have some overlap, but both do show some variation between hits and outs. To get a better understanding of how the variables are related to hits/outs, we'll use a scatter plot. First we'll plot the distribution of hits/outs as a function of launch (vertical) angle and exit velocity.

```
dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
ggplot(aes(x=launch_angle, y=launch_speed, color = hit)) +
  geom_point() +
  ylim(0, 125) + xlim(-90, 90) +
  theme(legend.title = element_blank()) +
  labs(y= "exit velocity", x="launch angle")
```



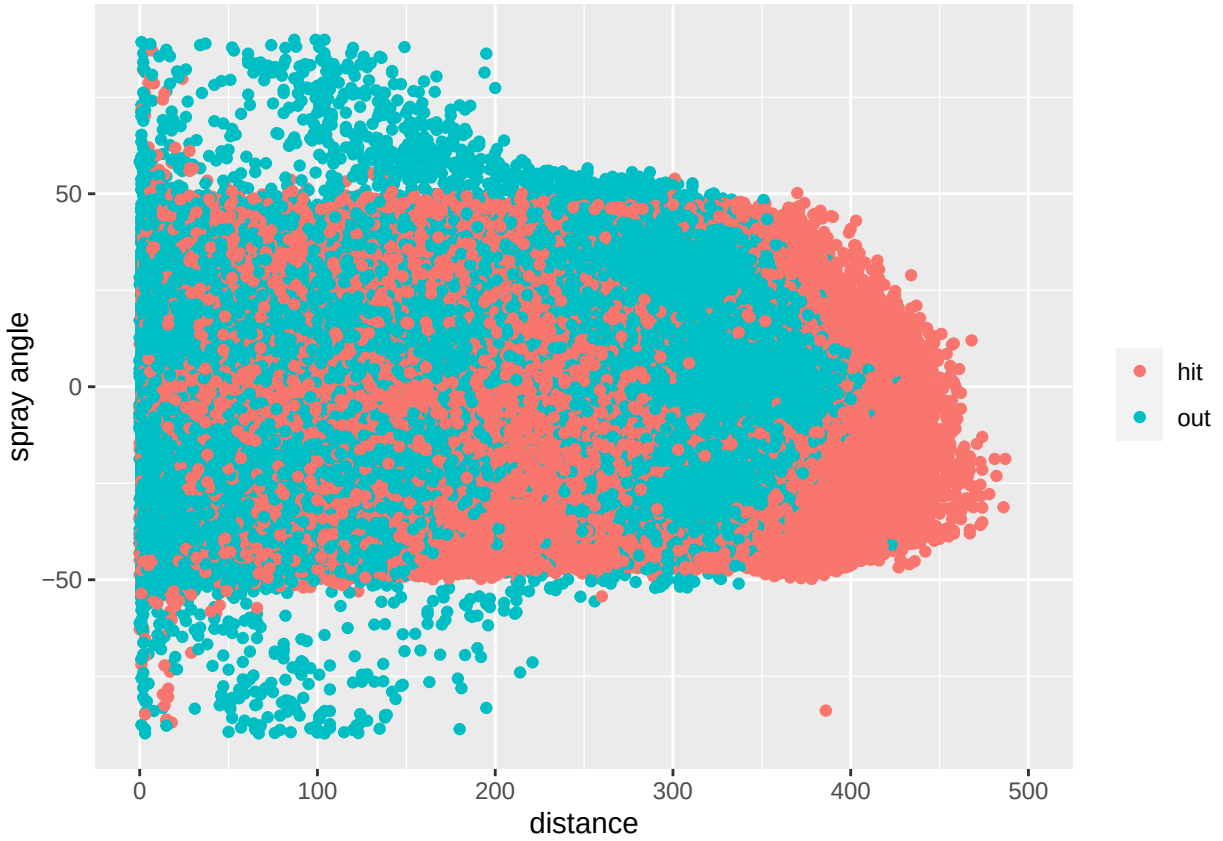
There appears to be a sweetspot for both variables: balls hit at higher velocities tend to be hits, while balls hit with a launch angle between 10 and 40 degrees also seem to turn into outs more often. But the relationship is not clear. This likely means we won't be able to use a linear model. Now let's look at exit velocity and spray (horizontal) angle.

```
dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
ggplot(aes(x=adj_spray_angle, y=launch_speed, color = hit)) +
  geom_point() +
  ylim(0, 125) + xlim(-90, 90) +
  theme(legend.title = element_blank()) +
  labs(y= "exit velocity", x="spray angle")
```



In contrast to launch angle, the sweetspot for spray angle is much larger as we see the hits distributed across the values of -50 and 50 degrees. Finally, we'll look at distance.

```
dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
ggplot(aes(x=hit_distance_sc, y=adj_spray_angle, color = hit)) +
  geom_point() +
  ylim(-90,90) + xlim(0,500) +
  theme(legend.title = element_blank()) +
  labs(y= "spray angle", x="distance")
```



This is very interesting as there seems to be two distinct sweetspots – from 150 to 250 feet and above 350 feet, with an obvious "hole" in between the pockets of hits. This makes sense given the configuration of an outfield, where the defensive players must cover a large amount of space. Those batted balls that traveled more than 350 feet are likely home runs or go over the head of the defensive outfielder, while the balls from 150 to 250 feet likely fall in front of him but are beyond the reach of the infielders (and the balls from 250 to 350 are within the catchable range of the outfielder).

Defense and ballpark variables

The other five predictor variables are not directly related to the physical characteristics of the batted ball, but rather relate to what happens after the ball is put into play. First, let's examine the positioning of the fielders: `inf_pos` and `of_pos`.

The advent of analytics in baseball has caused a dramatic change in defensive positioning over the last decade. Recognizing that the standard positioning of the fielders (i.e. two fielders on each side of second base) may not be optimal for every batter, teams now often use a "shifted" defensive alignment and position their defensive players based on where the batter is more likely to hit the ball. An example is to put three infielders on the right side of second base for a left-handed hitter that tends to pull the ball on the ground. More information Statcast's definition of shifts can be found here.

A quick calculation shows us that infield shifts occur on 34% of batted balls ...

```
mean(dat_train$inf_pos == "inf_shift")
```

```
## [1] 0.3390186
```

. . . while outfield shifts occur on just 9% of batted balls.

```
mean(dat_train$of_pos == "of_shift")
```

```
## [1] 0.09332583
```

So let's just focus on infield shifts. We also know that the purpose of infield shifts is to prevent hits on groundballs. Is there a relationship between the defensive alignment(shift/no shift) and the outcome of a groundball (hit/out)? Here is a table for the percent of hits for three categories: all groundballs, groundballs with a shifted defense, groundballs with a standard defense:

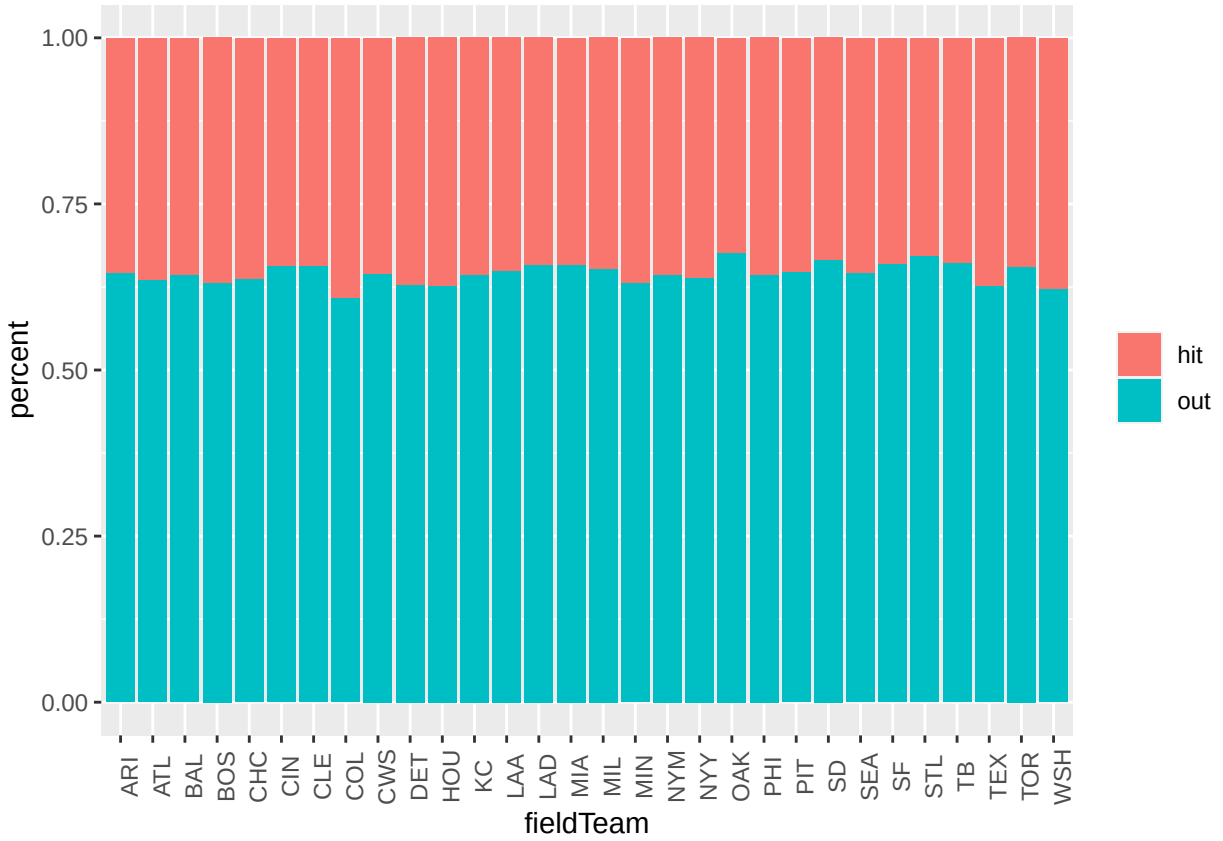
```
all_gb <- dat_ip %>%
  filter(bb_type == "ground_ball") %>%
  select(hit)
all_gb_tab <- mean(all_gb$hit == 1) * 100

shift <- dat_ip %>%
  filter(inf_pos == "inf_shift") %>%
  filter(bb_type == "ground_ball") %>%
  select(hit)
shift_tab <- mean(shift$hit == 1) * 100

no_shift <- dat_ip %>%
  filter(inf_pos == "inf_standard") %>%
  filter(bb_type == "ground_ball") %>%
  select(hit)
no_shift_tab <- mean(no_shift$hit == 1) * 100

pos_tab <- matrix(c(all_gb_tab, shift_tab, no_shift_tab), ncol = 1, byrow = TRUE)
colnames(pos_tab) <- "Percent_Hits"
rownames(pos_tab) <- c("all", "shift", "no_shift")
pos_tab <- as.table(pos_tab)
print(pos_tab)
```

```
##          Percent_Hits
## all          23.85275
## shift        21.94827
## no_shift     24.77418
```

Batters that hit a groundball into a shifted defense are less likely to get a hit compared to groundballs versus a standard defense, so it does appear that there is a relationship between defensive positioning and whether a groundball is a hit or an out.

Our variable for a team's overall defensive quality is `fieldTeam`. While the differences are not large, the ability of a team to turn batted balls into hits does vary across MLB.

```
dat_train %>% mutate(hit = ifelse(hit == 0, "out", "hit")) %>%
  ggplot(aes(fieldTeam, fill=hit)) +
  geom_bar(position = "fill") +
  theme(legend.title = element_blank()) +
  theme(axis.text.x = element_text(angle = 90)) +
  ylab("percent")
```



Our variable for ballpark-specific factors (i.e. the dimensions of the ballpark) is `home_team`. The dimensions, weather conditions, air quality, surrounding topology of the park, etc. can impact a player's performance. To quantify the effect of a team's home park and how it varies across the league, we can calculate a park factor for each team. At its most basic level, a park factor for hits compares the total hit rate in a team's home and road games. Typically the ratio is adjusted to that a value of 100 is equal to a "neutral" ballpark, and any number above 100 indicates a park that increases the probability of getting a hit (and vice versa). Here is a table of the hit park factor for every team in descending order:

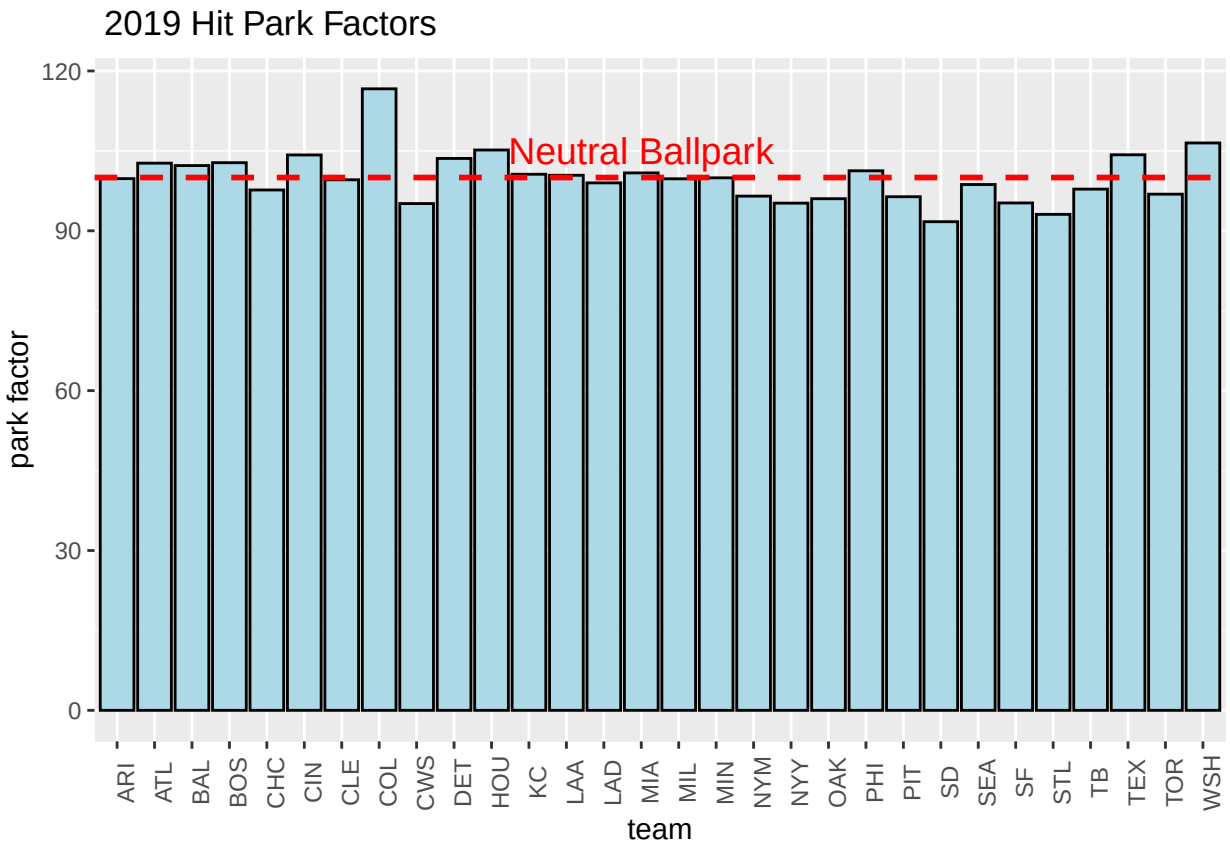
```
dat_away <- dat_ip %>%
  mutate(hit = as.numeric(as.character(hit))) %>%
  group_by(team_id = away_team) %>%
  summarise(hit = mean(hit)) %>%
  mutate(type = "away")

dat_home <- dat_ip %>%
  mutate(hit = as.numeric(as.character(hit))) %>%
  group_by(team_id = home_team) %>%
  summarise(hit = mean(hit)) %>%
  mutate(type = "home")

dat_pf <- dat_away %>%
  bind_rows(dat_home) %>%
  spread(key = type, value = hit) %>%
  mutate(pf = 100 * (home / away)) %>%
  select(team_id, pf) %>%
  arrange(desc(pf))
```

```r
print(dat_pf)
```

```
## # A tibble: 30 x 2
##    team_id    pf
##    <chr>   <dbl>
##  1 COL      117.
##  2 WSH      106.
##  3 HOU      105.
##  4 TEX      104.
##  5 CIN      104.
##  6 DET      104.
##  7 BOS      103.
##  8 ATL      103.
##  9 BAL      102.
## 10 PHI      101.
## # ... with 20 more rows
```

And here is a visualization of the hit park factor for all 30 teams. The dashed red line indicates a park factor of 100, which is for a neutral ballpark.

```r
dat_pf %>%
  ggplot(aes(x=team_id, y=pf)) +
  geom_bar(stat = "identity", fill = "light blue", color = "black") +
  geom_hline(yintercept=100, linetype="dashed", color = "red", size = 1) +
  annotate(geom="text", label="Neutral Ballpark", x=15, y=100, color = "red", size = 5, vjust = -.5) +
  theme(legend.title = element_blank()) +
  theme(axis.text.x = element_text(angle = 90)) +
  labs(y="park factor", x="team", title =" 2019 Hit Park Factors")
```



As you can see, the park factors do vary across the league, with Colorado as the most hitter-friendly park and San Diego at the bottom of the list. This appears to be a relationship that we should incorporate into our model.

Methods/Analysis

We will consider five evaluation metrics for our classification models: accuracy, precision, sensitivity, specificity and F1 score. **Accuracy** tells us how often our model makes correct predictions overall. It is good evaluation metric for classification problems that have balanced class distribution. **Precision (predicted positive value)** tells us how often our model is correct when it predicts positive. It is a measure of *exactness*, and is good choice to use when you want to be very sure of your prediction and the cost of false positives are high. **Sensitivity (recall)** tells us how often our model correctly predicts positive values. It is a measure of *completeness*, and is a good choice when you want to minimize false negatives. **Specificity** tells us how well the model classifies *negative cases*. This is useful for our problem because the accuracy of predicting an out is important information too. Finally, **F1 score** combines precision and recall (sensitivity) to tell us how well the model balances those metrics. A high F1 score means you have few false positives and few false negatives, so you are correctly identifying positive cases while also capturing as many positive cases possible – a model that is both exact and complete.

Before building and testing our models, we need to split the `dat_train` dataset into `train` and `test` datasets so we can train and test the performance of our models as we build them, before applying our final optimized model to the `validation` set we created earlier. We use an 80/20 split again, which is general rule of thumb to follow for datasets of our size (~100,000 observations) and will allow us to cross-validate on our train set.

Let's check again to make sure that our train and test sets are balanced, i.e. have a similar proportion of hits and outs.

```
prop.table(table(train$hit))
```

```
##
##         0         1
## 0.6445355 0.3554645
```

```
prop.table(table(test$hit))
```

```
##
##         0         1
## 0.6445141 0.3554859
```

Baseline model

We need to first establish a baseline result to compare the results from our other models. The **baseline_binomial** function creates a baseline evaluation of our test set that is better than simply guessing. According to the documentation, the function finds a range of likely values by evaluating multiple sets of random predictions and summarizing the related performance metrics. Below is a table with the output of the summarized metrics. It includes the mean, maximum and minimum values for our five metrics noted above; we also evaluate a set of all 0 predictions and all 1 predictions and report the associated metrics for those cases.

```
set.seed(123, sample.kind = "Rounding")
baseFit <- baseline_binomial(test_data = test, dependent_col = "hit",
                             metrics = list("all" = FALSE, "Accuracy" = TRUE,
                                            "Sensitivity" = TRUE, "Specificity" = TRUE,
                                            "Pos Pred Value" = TRUE, "F1" = TRUE))
base_model <- baseFit$summarized_metrics[c(1,5,6,9,10),c(1,2,4,5,6,3)]
print(base_model)
```

```
## # A tibble: 5 x 6
##   Measure Accuracy Sensitivity Specificity `Pos Pred Value`      F1
##   <chr>      <dbl>       <dbl>       <dbl>            <dbl>   <dbl>
## 1 Mean       0.500       0.500       0.500            0.356   0.416
## 2 Max        0.509       0.513       0.510            0.365   0.426
## 3 Min        0.489       0.479       0.490            0.344   0.401
## 4 All_0      0.645       0           1              NaN     NaN
## 5 All_1      0.355       1           0                0.355   0.525
```

NaiveBayes model

Our first algorithm is Naive Bayes, a simple classification model that is derived from Bayes Theorem and conditional probabilities. It is based on the idea that the predictor variables are independent of each other. While this likely isn't true for our real-world problem, it is easy to implement and good first step in our model-building process.

```
set.seed(123, sample.kind = "Rounding")
nbFIT <- naiveBayes(hit~., data = train)
nb_preds <- predict(nbFIT, newdata = test)
nb_cm <- confusionMatrix(nb_preds, test$hit, positive = "1")
```

```
nb_acc <- nb_cm$overall["Accuracy"]
nb_other <- nb_cm$byClass[c("Sensitivity", "Specificity", "Precision", "F1")]
nb_metrics <- c(nb_cm$overall["Accuracy"], nb_cm$byClass[c("Sensitivity", "Specificity", "Precision", "
results_nb <- matrix(as.numeric(nb_metrics), nrow = 1, ncol = 5)
metrics <- c("Accuracy", "Sensitivity", "Specificity", "Precision", "F1")
colnames(results_nb) <- metrics
rownames(results_nb) <- "Naive Bayes"
as.data.frame(results_nb)
```

```
##             Accuracy Sensitivity Specificity Precision        F1
## Naive Bayes 0.714054   0.5999392   0.7769947 0.5973947 0.5986642
```

We do see an improvement from our baseline model, but the values are still lower than we desire.

rPart (decision tree) model

The next step is to look at decision tree algorithms, which are popular for classification problems and work effectively with non-linear data. A basic decision tree can be built with the **rpart** package.

To tune the model (and subsequent models), we use the **mlr** package. A complete online tutorial for the package can be found here. In testing various models, I found the **mlr** package to be more flexible and faster than **caret** for tuning algorithms. The process is simple: 1) Create task = description of your machine learning task (classification, regression, etc.). You need to do this for both your test and train datasets. 2) Create learner = define machine learning algorithm to use (rpart, randomforest, etc.) 3) Define resample = set resampling strategy (cross-validation, etc.) and search method (grid, random, etc.). 4) Set hyperparameters = define ranges for parameters to tune 5) (optional) Set parallel backend = activate parallelization for faster computing 5) Tune parameters = get optimal parameters 6) Train (using optimal parameters) 7) Predict

First, make sure that the `mlr` package was loaded after the `caret` package (this will be true if the "libraries" code chunk at the top was executed). Then we proceed with the first few steps through tuning the parameters. I choose to tune the `cp`, `minsplit`, and `minbucket` parameters for the rpart algorithm. I also use a random search method because it has been shown to be more efficient (faster) and effective than grid search for hyperparameter tuning.

```
set.seed(123, sample.kind = "Rounding")
#create tasks
traintask_rp <- makeClassifTask(data = train, target = "hit", positive = "1")
testtask_rp <- makeClassifTask(data = test,target = "hit", positive = "1")
#create learner
rp.lrn <- makeLearner("classif.rpart", predict.type = "response")
#define resample
rdesc <- makeResampleDesc("CV",iters=4)
ctrl <- makeTuneControlRandom(maxit = 30)
#set hyperparameters
rp.parms <- makeParamSet(
  makeIntegerParam("minsplit",lower = 10,upper = 50),
  makeIntegerParam("minbucket",lower = 5,upper = 50),
  makeNumericParam("cp", lower = .001, upper = .1)
)
#set parallel backend
#on Mac OS and Linux you may want use parallelStartMulticore() instead
parallelStartSocket(cpus = detectCores())
```

```
#tune the parameters
rp.tune <- tuneParams(learner = rp.lrn, resampling = rdesc, task = traintask_rp,
                      par.set = rp.parms, control = ctrl,
                      measures = acc)
parallelStop()
```

Now that we have tuned the parameters, let's check the optimal parameters for our model.

```
rp.tune$x
```

```
## $minsplit
## [1] 43
##
## $minbucket
## [1] 23
##
## $cp
## [1] 0.001337206
```

Now we can train the model and make predictions with our optimal parameters. We also print our five evaluation metrics: Accuracy, Sensitivity, Specificity, Precision, and F1 score.

```
rp.opt_pars <- setHyperPars(rp.lrn, par.vals = rp.tune$x)
rp.train <- train(rp.opt_pars, traintask_rp)
rpart_preds <- predict(rp.train, testtask_rp)
ms <- list(acc, tpr, tnr, ppv, f1)
rpart_perf <- performance(rpart_preds, measures = ms)
results_rp <- matrix(as.numeric(rpart_perf), nrow = 1, ncol = 5)
colnames(results_rp) <- metrics
rownames(results_rp) <- "rPart"
as.data.frame(results_rp)
```

```
##       Accuracy Sensitivity Specificity Precision        F1
## rPart 0.822906   0.6612413   0.9120732 0.8057461 0.7263765
```

We have substantially improved our performance across all metrics. But we can do better, so let's build an entire forest of trees.

Random Forest model

By combining and averaging predictions from multiple decision trees, we should see an improvement in performance with the Random Forest model. Again, we will tune and train the model using the **mlr** package. We will tune three parameters using a random search: **ntree**, **mtry** and **nodesize**. (This may take 10-15 minutes.)

```
set.seed(123, sample.kind = "Rounding")
traintask_rf <- makeClassifTask(data = train, target = "hit", positive = "1")
testtask_rf <- makeClassifTask(data = test,target = "hit", positive = "1")
rf.lrn <- makeLearner("classif.randomForest", predict.type = "response")
rdesc <- makeResampleDesc("CV",iters=4)
```

```
ctrl <- makeTuneControlRandom(maxit = 30)
rf.pars <- makeParamSet(
  makeIntegerParam("mtry",lower = 1,upper = 8),
  makeIntegerParam("nodesize",lower = 10,upper = 50),
  makeIntegerParam("ntree", lower = 50, upper = 300)
)
parallelStartSocket(cpus = detectCores())
rf.tune <- tuneParams(learner = rf.lrn
                      ,task = traintask_rf
                      ,resampling = rdesc
                      ,measures = acc
                      ,par.set = rf.pars
                      ,control = ctrl)
parallelStop()
```

Here are the optimal parameters:

```
rf.tune$x
```

```
## $mtry
## [1] 6
##
## $nodesize
## [1] 31
##
## $ntree
## [1] 287
```

Now we can train the model and make predictions with our optimal parameters. We also print our five evaluation metrics: Accuracy, Sensitivity, Specificity, Precision, and F1 score.

```
rf.opt_pars <- setHyperPars(rf.lrn, par.vals = rf.tune$x)
rf.train <- train(rf.opt_pars, traintask_rf)
rf_preds <- predict(rf.train, testtask_rf)
rf_perf <- performance(rf_preds, measures = ms)
results_rf <- matrix(as.numeric(rf_perf), nrow = 1, ncol = 5)
colnames(results_rf) <- metrics
rownames(results_rf) <- "Random Forest"
as.data.frame(results_rf)
```

```
##                Accuracy Sensitivity Specificity Precision        F1
## Random Forest 0.8471854   0.7284758   0.9126605 0.8214408 0.7721703
```

XGBoost model

Our accuracy has improved again, but there still is room for improvement. The next step in our model progression is to implement a "boosting" algorithm to enhance the tree-growing process. This means trees are grown sequentially so that each successive tree is grown using information from previously grown trees. We implement a popular technique called Extreme Gradient Boosting (XGBoost). Again we use the **mlr** package to tune and train the model. We will tune the following parameters using a random search: `nrounds`, `max_depth`, `subsample`, `colsample_bytree`. These parameters were chosen thanks to guidance from this helpful tutorial. Note: This process may take 10-15 minutes.

```
set.seed(123, sample.kind = "Rounding")
traintask_xg <- makeClassifTask (data = train,target = "hit", positive = "1")
testtask_xg <- makeClassifTask (data = test,target = "hit", positive = "1")
traintask_xg <- createDummyFeatures (obj = traintask_xg)
testtask_xg <- createDummyFeatures (obj = testtask_xg)
xg.lrn <- makeLearner("classif.xgboost",predict.type = "response")
xg.lrn$par.vals <- list( objective="binary:logistic", eval_metric="error", eta = .1)
xg.pars <- makeParamSet(makeIntegerParam("nrounds", lower = 50, upper = 300),
                        makeIntegerParam("max_depth",lower = 3, upper = 12),
                        makeNumericParam("subsample", lower = .2, upper = .8),
                        makeNumericParam("colsample_bytree",lower = .2,upper = .8))
rdesc <- makeResampleDesc("CV",stratify = T,iters=4)
ctrl <- makeTuneControlRandom(maxit = 30)
parallelStartSocket(cpus = detectCores())
xg.tune <- tuneParams(learner = xg.lrn, task = traintask_xg, resampling = rdesc, measures = acc, par.se
parallelStop()
```

Here are the optimal parameters

```
xg.tune$x
```

```
## $nrounds
## [1] 153
##
## $max_depth
## [1] 10
##
## $subsample
## [1] 0.6136567
##
## $colsample_bytree
## [1] 0.7778657
```

Now we can train the model with the optimal parameters.

```
xg.opt_pars <- setHyperPars(xg.lrn, par.vals =xg.tune$x)
invisible(({capture.output({
xg.train <- train(xg.opt_pars, traintask_xg)})}))
```

Finally, we get our predictions and performance metrics:

```
xg_preds <- predict(xg.train, testtask_xg)
xg_perf <- performance(xg_preds, measures = ms)
results_xg <- matrix(as.numeric(xg_perf), nrow = 1, ncol = 5)
colnames(results_xg) <- metrics
rownames(results_xg) <- "XGBoost"
as.data.frame(results_xg)
```

```
##          Accuracy Sensitivity Specificity Precision        F1
## XGBoost 0.8585411   0.7477943   0.9196241 0.8369084 0.7898458
```

We have further improved all of our metrics. The improvements have been in small increments for the past couple models, so let's wrap up and pick our final model.

Results

The XGBoost model produced the best results across all metrics from our modeling process. Here is a table of our results from all the models.

```
models <- c("Baseline","Naive Bayes","rPart","Random Forest","XGBoost")
metrics <- c("Accuracy", "Sensitivity", "Specificity", "Precision", "F1")
results <- data.frame(rbind(as.numeric(base_model[1,2:6]), as.numeric(nb_metrics), as.numeric(rpart_per
rownames(results) <- models
colnames(results) <- metrics
print(results)
```

```
##                Accuracy Sensitivity Specificity Precision        F1
## Baseline      0.5001887   0.5000061   0.5002895 0.3556203 0.4156260
## Naive Bayes   0.7140540   0.5999392   0.7769947 0.5973947 0.5986642
## rPart         0.8229060   0.6612413   0.9120732 0.8057461 0.7263765
## Random Forest 0.8471854   0.7284758   0.9126605 0.8214408 0.7721703
## XGBoost       0.8585411   0.7477943   0.9196241 0.8369084 0.7898458
```

Now we will apply this XGBoost model to tune and train the entire `dat_train` dataset and then make our final predictions on the `validation` set. First we tune the parameters. This may take 10-15 minutes.

```
set.seed(123, sample.kind = "Rounding")
traintask <- makeClassifTask (data = dat_train,target = "hit", positive = "1")
testtask <- makeClassifTask (data = validation,target = "hit", positive = "1")
traintask <- createDummyFeatures (obj = traintask)
testtask <- createDummyFeatures (obj = testtask)
xgFinal.lrn <- makeLearner("classif.xgboost",predict.type = "response")
xgFinal.lrn$par.vals <- list(objective="binary:logistic", eval_metric="error", eta = .1)
xgFinal.pars <- makeParamSet(makeIntegerParam("nrounds", lower = 50, upper = 300),
                        makeIntegerParam("max_depth",lower = 3, upper = 12),
                        makeNumericParam("subsample", lower = .2, upper = .8),
                        makeNumericParam("colsample_bytree",lower = .2,upper = .8))
rdesc <- makeResampleDesc("CV",stratify = T,iters=4)
ctrl <- makeTuneControlRandom(maxit = 30)
parallelStartSocket(cpus = detectCores())
xgFinal.tune <- tuneParams(learner = xgFinal.lrn, task = traintask, resampling = rdesc, measures = acc,
parallelStop()
```

Here are the optimal parameters

```
xgFinal.tune$x
```

```
## $nrounds
## [1] 204
##
## $max_depth
## [1] 8
##
## $subsample
## [1] 0.5702709
##
## $colsample_bytree
## [1] 0.7068657
```

Now we can train the model with the optimal parameters.

```
xgFinal.lrn_opt <-setHyperPars(xgFinal.lrn, par.vals = xgFinal.tune$x)
invisible(({capture.output({
xgFinal.train <- train(xgFinal.lrn_opt, traintask)})}))
```

Finally, we get our predictions and performance metrics for the validation set:

```
xgFinal_preds <- predict(xgFinal.train, testtask)
xgFinal_perf <- performance(xgFinal_preds, measures = ms)
results_xgfinal <- matrix(as.numeric(xgFinal_perf), nrow = 1, ncol = 5)
colnames(results_xgfinal) <- metrics
rownames(results_xgfinal) <- "Final (XGBoost)"
as.data.frame(results_xgfinal)
```

```
##                   Accuracy Sensitivity Specificity Precision        F1
## Final (XGBoost) 0.8594047   0.7584276    0.915095 0.8312658 0.7931781
```

Model-building process

We started our model-building process with a naive algorithm that used conditional probabilities and assumed the predictor variables were independent. When this produced a sub-optimal result, we moved to decision trees, which work well with classification problems and non-linear decision boundaries. When our simple decision tree model (rPart) resulted in good – but not great – performance metrics, we decided to use Random Forest, an ensemble decision tree method. This algorithm improves upon a basic decision tree model because it combines information from the (weak) individual trees to produce a stronger learner (forest), and ultimately make better predictions. Finally, to increase the model's performance even further, we added a "boosting" technique. We choose XGBoost, a very popular machine learning algorithm which uses gradient boosting (GBM). GBM is a process that applies each previous model's error to build successive models, therefore continually reducing error until an optimal model is built. The final XGBoost model produced the highest values in all five performance metrics when compared to the other models we tested.

Feature importance

To better understand the importance of each feature and our model, let's take a look at the top 10 features ranked by importance from our trained model.

```
imp <- getFeatureImportance(xgFinal.train)
imp$res %>% top_n(10, importance) %>% arrange(desc(importance))
```

```
## # A tibble: 10 x 2
##    variable             importance
##    <chr>                     <dbl>
##  1 hit_distance_sc         0.275
##  2 launch_angle            0.221
##  3 adj_spray_angle         0.214
##  4 launch_speed            0.206
##  5 inf_pos.inf_shift       0.0160
##  6 stand.L                 0.0124
##  7 inf_pos.inf_standard    0.00341
```

```
##  8 stand.R                0.00321
##  9 home_team.BOS          0.00255
## 10 of_pos.of_shift        0.00245
```

The four "batted ball trajectory" metrics are the most important features and the only ones with a value of greater than .02. This shows how critical those variables are to predicting whether a ball becomes a hit or an out. (Perhaps a simpler model would incorporate only those features.)

Another key insight from the table is the relative importance of the infield positioning variable, specifically when a team used a shifted alignment (`inf_pos.inf_shift`). It is the highest of any of the "non-trajectory" variables. It is notable that the variable for left-handed batters (`stand.L`) is right below the infield shift variable in importance (and nearly four times as important as the variable for right-handed batters). This is related to the fact that left-handed batters face a higher percentage of shifted infield defenses than right-handed batters.

Another key insight is that exit velocity (`launch_speed`) was the least important of the "trajectory" features. Hitting the ball really hard doesn't automatically increase the probability of getting a hit; vertical and horizontal angle of the batted ball are equally and perhaps more important.

An interesting observation from this table is the inclusion of the `home_team.BOS` variable (the home ballpark for Boston Red Sox) in the top 10 list of important features. Fenway Park has some of the most unique features and extreme dimensions of any MLB ballpark, and this impacts whether a batted ball becomes a hit or an out, more so than any other ballpark according to our model.

Conclusion

The goal of this project was to create a robust model to predict the probability of a batted ball becoming a hit using the machine learning techniques and data analysis skills learned in previous courses. We incorporated features relating to both the trajectory of the batted ball and external factors such as defense and ballpark. We evaluated each test model using five performance metrics: Accuracy, Sensitivity, Specificity, Precision and F1 score. Our best-performing model utilized the XGBoost algorithm, a powerful machine learning tool based on decision trees and boosting.

Limitations

One limitation is that we used the fielding team as a very basic variable to model the effect of defensive quality. A better model might be achieved by incorporating actual team defensive metrics as a predictor. Also, we did not model a specific weather effect (which could greatly impact the hit probability), but merely used the ballpark as a proxy for weather conditions. Another limitation is that we did not model the batter's speed, which could have an effect on hit probability, especially for infield hits.

Future work

One interesting observation is that the model did a better job of predicting an out than a hit (the specificity is higher than precision). Future work could involve exploring why this occurred and if possible how to build a better model that predicted both with greater accuracy. While the goal of our project was to simply build a prediction model, this model has wide-ranging applications for teams, analysts and fans. For example, a team's research analyst could compute a "predicted batting average" for a player using the model, compare that to his actual batting average, and then look at reasons why those two might differ; also, if the predicted batting average is above the actual batting average, the player might be expected to perform better in the future (i.e. because of the law of regression). An extension of the model could involve using other response variables. Instead of simply predicting a hit or an out, we could use the model framework to predict the *type*

of hit (single, double, triple, home run). We could then apply this alternative model to develop a "predicted" metric that incorporates the type of hit, such as "predicted" OPS (on-base plus slugging) and "predicted" wOBA (weighted on-base average).

References

https://rafalab.github.io/dsbook
https://www.coverfoursports.com/post/working-with-data-getting-statcast-data-in-r
https://baseballsavant.mlb.com/csv-docs
http://m.mlb.com/glossary/statcast
http://rdrr.io/cran/cvms/man/baseline_binomial.html
http://rdocumentation.org/packages/e1071/versions/1.7-3/topics/naiveBayes
https://mlr.mlr-org.com/index.html
http://jmlr.csail.mit.edu/papers/volume13/bergstra12a/bergstra12a.pdf
https://cran.r-project.org/web/packages/xgboost/vignettes/xgboostPresentation.html
https://sites.google.com/view/lauraepp/parameters