Predicting On Base Percentage

Will Foote

2024-12-03

```
library(baseballr)
```

```
## Warning: package 'baseballr' was built under R version 4.2.3
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(caret)
```

```
## Loading required package: ggplot2
```

```
## Loading required package: lattice
```

```
library(xgboost)
```

```
##
## Attaching package: 'xgboost'
```

```
## The following object is masked from 'package:dplyr':
##
##     slice
```

Intro

I decided to use the widely available Fangraphs data using the baseballr package. I hope the OBP's are in alignment with the dataset provided. My process will be to extract and clean the data. Then, transform it if needed and add features that I think will be helpful for predicting next years OBP, such as last year's batting average, OBP, swinging strike rate, expected wOBA and more.

Loading in previous seasons data

Using the baseballr package, import data from Fangraphs.

```
df_2016 <- baseballr::fg_bat_leaders(startseason = '2016', endseason = '2016') %>%
  rename_with(.fn = ~paste0(., "_2016"))
df_2017 <- baseballr::fg_bat_leaders(startseason = '2017', endseason = '2017') %>%
  rename_with(.fn = ~paste0(., "_2017"))
df_2018 <- baseballr::fg_bat_leaders(startseason = '2018', endseason = '2018') %>%
  rename_with(.fn = ~paste0(., "_2018"))
df_2019 <- baseballr::fg_bat_leaders(startseason = '2019', endseason = '2019') %>%
  rename_with(.fn = ~paste0(., "_2019"))
df_2020 <- baseballr::fg_bat_leaders(startseason = '2020', endseason = '2020') %>%
  rename_with(.fn = ~paste0(., "_2020"))
df_2021 <- baseballr::fg_bat_leaders(startseason = '2021', endseason = '2021') %>%
  select(playerid, OBP)
```

Join together

```
joined = left_join(df_2021, df_2020, by = join_by('playerid' == 'playerid_2020'), suffix
= c('_2021', '_2020')) %>%
  left_join(df_2019, join_by('playerid' == 'playerid_2019'), suffix = c("_2020", "_201
9")) %>%
  left_join(df_2018, by = join_by('playerid' == 'playerid_2018'), suffix = c("", "_201
8")) %>%
  left_join(df_2017, by = join_by('playerid' == 'playerid_2017'), suffix = c("", "_201
7")) %>%
  left_join(df_2016, by = join_by('playerid' == 'playerid_2016'), suffix = c("", "_201
6"))
```

Exploratory Data Analysis

Find the categorical variables

```
# Identify categorical variables (factors or characters)
categorical_vars <- names(joined)[sapply(joined, function(col) is.factor(col) || is.char
acter(col))]
print(categorical_vars)
```

```
##  [1] "team_name_2020"       "Bats_2020"            "PlayerNameRoute_2020"
##  [4] "PlayerName_2020"      "AgeRng_2020"          "position_2020"
##  [7] "team_name_abb_2020"   "team_name_2019"       "Bats_2019"
## [10] "PlayerNameRoute_2019" "PlayerName_2019"      "AgeRng_2019"
## [13] "position_2019"        "team_name_abb_2019"   "team_name_2018"
## [16] "Bats_2018"            "PlayerNameRoute_2018" "PlayerName_2018"
## [19] "AgeRng_2018"          "position_2018"        "team_name_abb_2018"
## [22] "team_name_2017"       "Bats_2017"            "PlayerNameRoute_2017"
## [25] "PlayerName_2017"      "AgeRng_2017"          "position_2017"
## [28] "team_name_abb_2017"   "team_name_2016"       "Bats_2016"
## [31] "PlayerNameRoute_2016" "PlayerName_2016"      "AgeRng_2016"
## [34] "position_2016"        "team_name_abb_2016"
```

Identify columns

Below is a list of the features I would like to explore and why.

- OBP (TV)
  - OBP from 2016-2020 as features
- AVG
  - batting average is a big factor in on base percentage and likely would help predict future OBP. High-average players will be higher OBP typically.
- K_pct
  - If a player strikes out a lot, they're going to likely get on base less. Although three true outcome hitters are an exception.
- B_pct
  - Walks, another big way to get on base besides a hit. If a player walks a lot, they like have a higher OBP, even if their average is low.
- CStr_pct
  - Called strike percent. More called strikes, more strikeouts, less on base opportunities.
- SwStr_pct
  - Swinging strike percent. See above reasoning.
- C+SwStr_pct
  - Called strike plus swinging strike rate. See above reasoning.
- xAvg
  - Expected average (Statcast). Maybe a player has a worse average than they should because they were unlucky. Maybe xAVG is a better predictor than just AVG.
- xwOBA
  - Expected wOBA. wOBA, weighted on-base average, is an advanced stat that calculates a player's offensive contributions besides just OBP. Perhaps it helps predict OBP.
- Spd
  - Speed. Do faster players get on base at a higher clip?

Subset the data to only include the above features (and player id)

```
library(dplyr)

# Define the patterns to match (case-sensitive)
patterns <- c("OBP", "AVG", "K_pct", "BB_pct", "CStr_pct",
              "SwStr_pct", "C+SwStr_pct", "handedness",
              "xAvg", "xwOBA", "xAVG", "Spd")

# Use select() with matches() to filter columns
filtered_df <- joined %>%
  select(playerid, matches(paste(patterns, collapse = "|")))

names(filtered_df)[20] <- "OBP_2019"
# View the selected columns
colnames(filtered_df)
```

```
##  [1] "playerid"         "OBP"              "AVG_2020"         "BB_pct_2020"
##  [5] "K_pct_2020"       "OBP_2020"         "Spd_2020"         "SwStr_pct_2020"
##  [9] "CStr_pct_2020"    "C+SwStr_pct_2020" "AVG+_2020"        "BB_pct+_2020"
## [13] "K_pct+_2020"      "OBP+_2020"        "xwOBA_2020"       "xAVG_2020"
## [17] "AVG_2019"         "BB_pct_2019"      "K_pct_2019"       "OBP_2019"
## [21] "Spd_2019"         "SwStr_pct_2019"   "CStr_pct_2019"    "C+SwStr_pct_2019"
## [25] "AVG+_2019"        "BB_pct+_2019"     "K_pct+_2019"      "OBP+_2019"
## [29] "xwOBA_2019"       "xAVG_2019"        "AVG_2018"         "BB_pct_2018"
## [33] "K_pct_2018"       "OBP_2018"         "Spd_2018"         "SwStr_pct_2018"
## [37] "CStr_pct_2018"    "C+SwStr_pct_2018" "AVG+_2018"        "BB_pct+_2018"
## [41] "K_pct+_2018"      "OBP+_2018"        "xwOBA_2018"       "xAVG_2018"
## [45] "AVG_2017"         "BB_pct_2017"      "K_pct_2017"       "OBP_2017"
## [49] "Spd_2017"         "SwStr_pct_2017"   "CStr_pct_2017"    "C+SwStr_pct_2017"
## [53] "AVG+_2017"        "BB_pct+_2017"     "K_pct+_2017"      "OBP+_2017"
## [57] "xwOBA_2017"       "xAVG_2017"        "AVG_2016"         "BB_pct_2016"
## [61] "K_pct_2016"       "OBP_2016"         "Spd_2016"         "SwStr_pct_2016"
## [65] "CStr_pct_2016"    "C+SwStr_pct_2016" "AVG+_2016"        "BB_pct+_2016"
## [69] "K_pct+_2016"      "OBP+_2016"        "xwOBA_2016"       "xAVG_2016"
```

Calculate some year over year stats

```
library(dplyr)

# Select the specific OBP columns manually
obp_years <- c("OBP_2016", "OBP_2017", "OBP_2018", "OBP_2019", "OBP_2020")

# Calculate the row-wise average OBP across these columns
filtered_df1 <- filtered_df %>%
  mutate(Avg_OBP = rowMeans(select(., all_of(obp_years)), na.rm = TRUE))
```

Functionalize this

Note: Thanks ChatGPT for making my dreams a reality.

How does a player's performance over multiple stats change through and over time? I wanted to see if it's possible to quantify a player's change in hitting profile over time. Maybe there was a recent decrease in OBP from 2019 to 2020. Is their OBP lower than their 5-year average? Will this decline continue?

Hopefully this provides some signal for the model to learn the patterns.

```
# Load required library
library(dplyr)

# Define the function
calculate_multiple_stat_metrics <- function(data, stat_names, years = 2016:2020) {
  # Loop over each stat category
  for (stat_name in stat_names) {
    # Create column names for the years
    year_columns <- paste0(stat_name, "_", years)

    # Check if all columns exist in the dataset
    missing_columns <- setdiff(year_columns, colnames(data))
    if (length(missing_columns) > 0) {
      stop(paste("Missing columns in the dataset for", stat_name, ":", paste(missing_col
umns, collapse = ", ")))
    }

    # Calculate the mean of the statistic for the given years
    data <- data %>%
      mutate(!!paste0("mean_", stat_name) := rowMeans(select(., all_of(year_columns)), n
a.rm = TRUE))

    # Calculate the differential between 2020 and the mean
    data <- data %>%
      mutate(!!paste0(stat_name, "_diff_from_mean") := .[[paste0(stat_name, "_2020")]] -
.[[paste0("mean_", stat_name)]])

    # Calculate the change from 2019 to 2020
    data <- data %>%
      mutate(!!paste0(stat_name, "_change_2019_to_2020") := .[[paste0(stat_name, "_202
0")]] - .[[paste0(stat_name, "_2019")]])
  }

  # Return the modified dataframe
  return(data)
}

# Example usage
# Assuming 'joined' is your dataframe
stat_categories <- c("OBP", "AVG", "K_pct", "BB_pct")  # Add all relevant stats here
temp1 <- calculate_multiple_stat_metrics(filtered_df, stat_categories, years = 2016:202
0)

# View the resulting columns for one of the stat categories
head(temp1)
```

```
## ── MLB Player Batting Leaders data from FanGraphs.com ──────── baseballr 1.6.0 ──
```

```
## ℹ Data updated: 2024-12-06 01:07:04 PST
```

```
## # A tibble: 6 × 84
##   playerid   OBP AVG_2…¹ BB_pc…² K_pct…³ OBP_2…⁴ Spd_2…⁵ SwStr…⁶ CStr_…⁷ C+SwS…⁸
##      <int> <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1    16252 0.375   0.335  0.0849   0.139   0.394    7.25  0.0881   0.172   0.260
## 2    19709 0.364   0.277  0.105    0.237   0.366    7.34  0.126    0.144   0.269
## 3    20123 0.465   0.351  0.209    0.143   0.490    3.97  0.0653   0.185   0.250
## 4    11579 0.429   0.268  0.201    0.176   0.420    6.02  0.125    0.11    0.235
## 5    13510 0.355   0.292  0.122    0.169   0.386    5.80  0.0593   0.170   0.230
## 6    19611 0.401   0.262  0.0823   0.156   0.329    4.51  0.114    0.134   0.248
## # … with 74 more variables: `AVG+_2020` <dbl>, `BB_pct+_2020` <dbl>,
## #   `K_pct+_2020` <dbl>, `OBP+_2020` <dbl>, xwOBA_2020 <dbl>, xAVG_2020 <dbl>,
## #   AVG_2019 <dbl>, BB_pct_2019 <dbl>, K_pct_2019 <dbl>, OBP_2019 <dbl>,
## #   Spd_2019 <dbl>, SwStr_pct_2019 <dbl>, CStr_pct_2019 <dbl>,
## #   `C+SwStr_pct_2019` <dbl>, `AVG+_2019` <dbl>, `BB_pct+_2019` <dbl>,
## #   `K_pct+_2019` <dbl>, `OBP+_2019` <dbl>, xwOBA_2019 <dbl>, xAVG_2019 <dbl>,
## #   AVG_2018 <dbl>, BB_pct_2018 <dbl>, K_pct_2018 <dbl>, OBP_2018 <dbl>, …
```

Make Model

I chose to do XGBoost because that's what I've been working with at work. But, other options like linear regression could be helpful for interpretability although performance might be worse. Random Forest is another good decision-tree-based model and has a good amount of predictive power without overfitting as much as XGBoost can sometimes.

I will also run a hyperparameter grid search with the data to improve performance. This iterates through different settings of the model, so we can find the one that fits this data best/better. This can sometimes be computationally expensive, but the dataset is small enough that it is rather easy.

I will also later prune off some of the features. This is because too many features can be bad for multiple reasons. For one, it could lead to overfitting. If the model has 80 columns as input, but only ten or fifteen of them are actually helpful, with new data the model could perform very poorly or have all-over-the-place results (i.e. high variance).

This is in part because of the curse of dimensionality. The distance between points in a one-dimensional line or two-dimensional space or even three-dimensional space is relatively small. But with each dimension (baseball statistic in our case) that we add to the model, the distance becomes points becomes further and further from each other. Such that with a huge amount of features, and proportionally not enough players to look at the stats of, the model can have difficulty being trained correctly. When it encounters new data it could do very poorly.

Moreover, it's also computationally inefficient to include features in a model that are not actually helpful (or have diminishing returns when added to the model).

We'll define how good a model is by its root mean squared error (RMSE). This essentially will tell us on average how far off our predictions are from the actual 2021 OBP's. Note, a smaller number is better here. Later, I will compare my results to that of other industry models to see how mine does.

```
# Step 1: Prepare the data
data <- temp1 %>%
  select(-playerid) %>%  # Drop non-numeric columns
  na.omit()              # Remove rows with missing values

# Define features (X) and target (Y)
X <- data %>%
  select(-OBP)  # Exclude target column
Y <- data$OBP

# Step 2: Train-test split
set.seed(123)
train_index <- createDataPartition(Y, p = 0.8, list = FALSE)
X_train <- X[train_index, ]
Y_train <- Y[train_index]
X_test <- X[-train_index, ]
Y_test <- Y[-train_index]

# Step 3: Create a caret-compatible grid for hyperparameter tuning
xgb_grid <- expand.grid(
  nrounds = c(50, 100),          # Number of boosting iterations
  eta = c(0.01, 0.1, 0.3),       # Learning rate
  max_depth = c(4, 6, 8),        # Tree depth
  gamma = c(0, 1),               # Minimum loss reduction
  colsample_bytree = c(0.5, 1),  # Feature sampling ratio
  min_child_weight = c(1, 3),    # Minimum child weight
  subsample = c(0.5, 1)          # Subsampling ratio
)

# Step 4: Set up caret trainControl
train_control <- trainControl(
  method = "cv",                 # Cross-validation
  number = 3,                    # Number of folds
  verboseIter = FALSE,            # Show training progress
  allowParallel = TRUE           # Allow parallel processing
)

# Step 5: Train the model using caret and grid search
set.seed(123)
xgb_tuned_model <- train(
  x = as.matrix(X_train),
  y = Y_train,
  method = "xgbTree",
  trControl = train_control,
  tuneGrid = xgb_grid,
  metric = "RMSE",
  verbosity = 0
)

# Step 6: View the best model and hyperparameters
print(xgb_tuned_model$bestTune)
```

```
##     nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 169      50         8 0.1     0                1                1       0.5
```

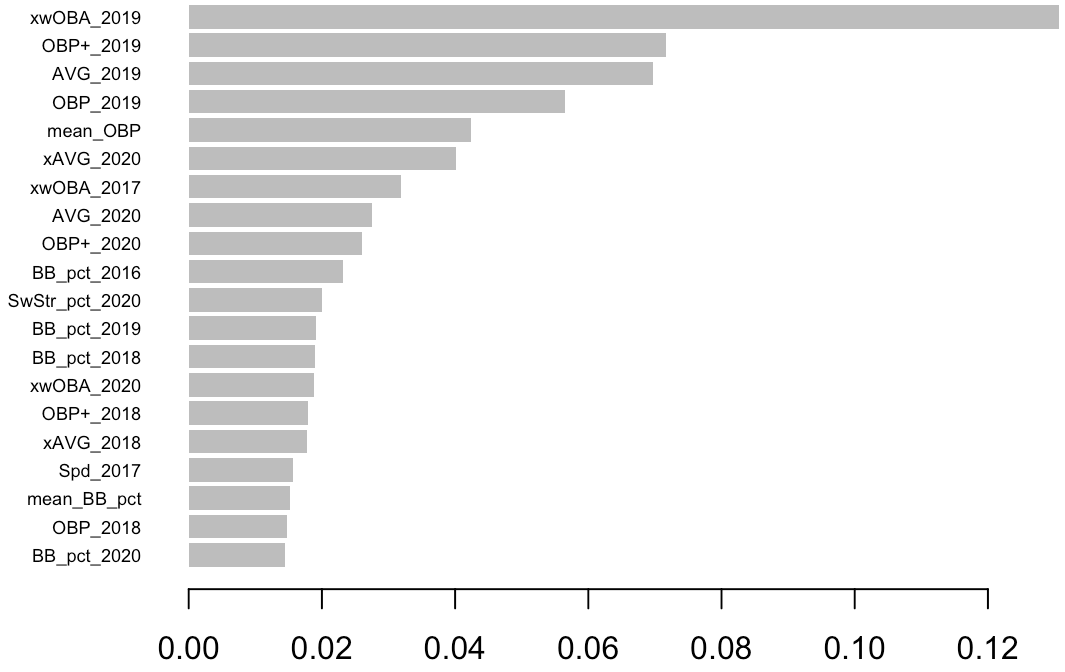
```
# Step 7: Evaluate on the test set
predictions <- predict(xgb_tuned_model, newdata = as.matrix(X_test))
rmse <- sqrt(mean((predictions - Y_test)^2))
print(paste("Test RMSE:", round(rmse, 4)))
```

```
## [1] "Test RMSE: 0.0579"
```

Use feature importance to define smaller model - Less overfitting

The feature importance chart can be a good place to start when trying to find the best statistics to include in the model. It doesn't take in domain knowledge, but it can mathematically give a general idea of how the model uses different features to predict OBP in 2021.

```
# Calculate feature importance from the XGBoost model
importance_matrix <- xgb.importance(model = xgb_tuned_model$finalModel)

# Plot the top features
xgb.plot.importance(importance_matrix, top_n = 20)
```



From the above results, we see that expected wOBA from 2019 is very important. This is likely because the Statcast expected wOBA formula accounts for external factors like ballpark and defense, and gives an idea of how a player should be performing based on underlying metrics like quality of contact. If we were to predict how a player will perform in 2021, it's a better idea to look at how they made contact with the ball and how often, rather than just the outcomes. If they got their stats inflated by playing at Coors Field or Great American Ball Park or against a bad defense in 2019 for example, their wOBA for 2019 could be inflated. The expected wOBA would then give us a metric that is more "true" to the player's actual skill.

OBP, AVG, and OBP+ (OBP standardized against league performance, where 100 is average) from 2019 are also all helpful it seems. It's interesting that 2020 isn't as helpful, which one might think would be counterintuitive because it was the last season before the target variable. But this could be explained by the shortened COVID season. That smaller sample size could be less representative of players' actual skills.

Another interesting factor that is important is mean_OBP which is a player's On Base Percentage from the past 5 years (2016-2020) averaged. This could suggest that a sufficiently large sample size of a player's performance in terms of OBP could be a good predictor of future OBP. This logically makes sense, as a player who has had a high OBP for the past five seasons likely will have high OBP again.

I included some time-based variables to see how performance changed over the seasons, but it didn't prove totally helpful. More depth on this analysis could prove fruitful (like looking at the change from two seasons or three seasons ago), so I haven't fully lost hope. But, the features we have now seem to be a good start.

Multicollinearity?

If features are looking at the same aspect (like AVG and AVG+), we probably don't want to include them all.

```
# Compute correlation matrix
cor_matrix <- cor(X_train, use = "complete.obs")

# Find highly correlated pairs (threshold: 0.9)
high_corr <- which(abs(cor_matrix) > 0.9 & abs(cor_matrix) < 1, arr.ind = TRUE)
high_corr_pairs <- data.frame(
  Feature1 = rownames(cor_matrix)[high_corr[, 1]],
  Feature2 = colnames(cor_matrix)[high_corr[, 2]]
)
print(high_corr_pairs)
```

```
##                     Feature1                 Feature2
## 1                  AVG+_2020                AVG_2020
## 2                BB_pct+_2020              BB_pct_2020
## 3                 K_pct+_2020               K_pct_2020
## 4                  OBP+_2020                OBP_2020
## 5                   AVG_2020               AVG+_2020
## 6                BB_pct_2020              BB_pct+_2020
## 7                 K_pct_2020               K_pct+_2020
## 8                  OBP_2020                OBP+_2020
## 9                  AVG+_2019                AVG_2019
## 10               BB_pct+_2019              BB_pct_2019
## 11                K_pct+_2019               K_pct_2019
## 12                 OBP+_2019                OBP_2019
## 13                  AVG_2019               AVG+_2019
## 14               BB_pct_2019              BB_pct+_2019
## 15                K_pct_2019               K_pct+_2019
## 16                 OBP_2019                OBP+_2019
## 17                 AVG+_2018                AVG_2018
## 18               BB_pct+_2018              BB_pct_2018
## 19                K_pct+_2018               K_pct_2018
## 20                 OBP+_2018                OBP_2018
## 21                  AVG_2018               AVG+_2018
## 22               BB_pct_2018              BB_pct+_2018
## 23                K_pct_2018               K_pct+_2018
## 24                 OBP_2018                OBP+_2018
## 25                 AVG+_2017                AVG_2017
## 26               BB_pct+_2017              BB_pct_2017
## 27                K_pct+_2017               K_pct_2017
## 28                 OBP+_2017                OBP_2017
## 29                  AVG_2017               AVG+_2017
## 30               BB_pct_2017              BB_pct+_2017
## 31                K_pct_2017               K_pct+_2017
## 32                 OBP_2017                OBP+_2017
## 33                 AVG+_2016                AVG_2016
## 34               BB_pct+_2016              BB_pct_2016
## 35                K_pct+_2016               K_pct_2016
## 36                 OBP+_2016                OBP_2016
## 37                  AVG_2016               AVG+_2016
## 38               BB_pct_2016              BB_pct+_2016
## 39                K_pct_2016               K_pct+_2016
## 40                 OBP_2016                OBP+_2016
## 41 K_pct_change_2019_to_2020      K_pct_diff_from_mean
## 42      K_pct_diff_from_mean K_pct_change_2019_to_2020
```

```
# Decide which features to drop (e.g., keep one feature from correlated pairs)
```

Systematic Pruning - Recursive Feature Elimination (RFE)

Using recursive feature elimination, we can systematically find the most important features by removing less significant features iteratively until we are left with a subset that is optimal.

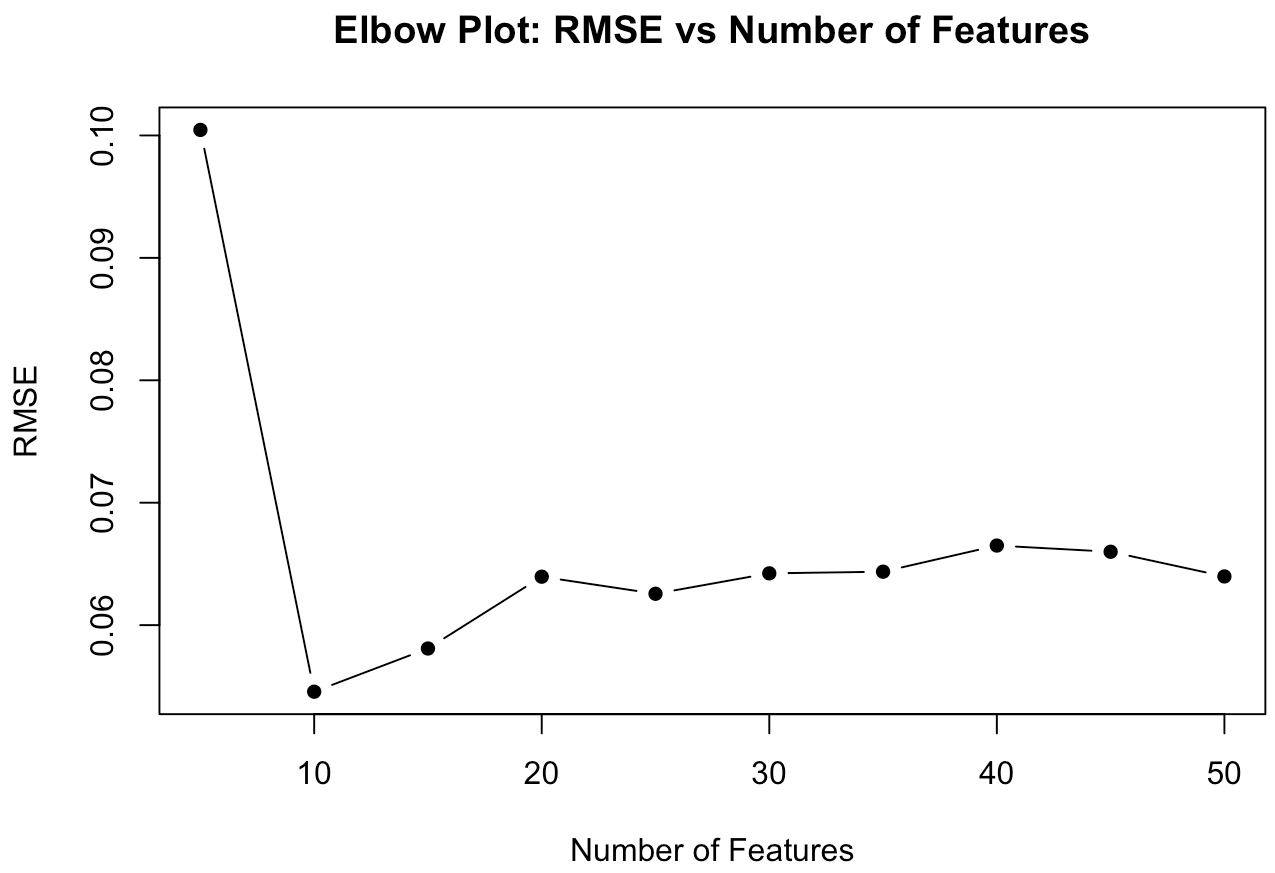
```
rfe_xgboost <- function(X, Y, max_features = 10, nrounds = 50, eta = 0.1) {
  # Ensure the input data and parameters are properly scoped
  feature_names <- colnames(X)  # Extract feature names
  selected_features <- feature_names  # Start with all features

  # Convert X to a matrix explicitly if it is a tibble
  X_matrix <- as.matrix(X)

  # RFE loop: remove least important features until reaching max_features
  while (length(selected_features) > max_features) {
    # Train XGBoost model with current selected features
    dtrain <- xgb.DMatrix(data = X_matrix[, selected_features, drop = FALSE], label = Y)
    xgb_model <- xgboost(data = dtrain, nrounds = nrounds, eta = eta, verbose = 0)

    # Get feature importance
    importance_matrix <- xgb.importance(model = xgb_model, feature_names = selected_feat
ures)

    # Identify the least important feature
    least_important <- importance_matrix[which.min(importance_matrix$Gain), "Feature"]

    # Remove the least important feature
    selected_features <- setdiff(selected_features, least_important)

    # Debugging: Print remaining features
    # print(paste("Remaining features:", length(selected_features)))
  }

  return(selected_features)  # Return the final set of selected features
}

# Example usage
set.seed(123)
selected_features_5 <- rfe_xgboost(X, Y, max_features = 5)  # Get top 5 features
selected_features_10 <- rfe_xgboost(X, Y, max_features = 10)  # Get top 10 features
selected_features_15 <- rfe_xgboost(X, Y, max_features = 15)  # Get top 10 features
selected_features_20 <- rfe_xgboost(X, Y, max_features = 20)  # Get top 10 features
selected_features_25 <- rfe_xgboost(X, Y, max_features = 25)  # Get top 10 features
selected_features_30 <- rfe_xgboost(X, Y, max_features = 30)  # Get top 10 features
selected_features_35 <- rfe_xgboost(X, Y, max_features = 35)  # Get top 10 features
selected_features_40 <- rfe_xgboost(X, Y, max_features = 40)  # Get top 10 features
selected_features_45 <- rfe_xgboost(X, Y, max_features = 45)  # Get top 10 features
selected_features_50 <- rfe_xgboost(X, Y, max_features = 50)  # Get top 10 features
```

See how these model performs

We can test a model and hold the model settings the same and train one model per feature subset. Then we can plot the performance for each subsequent model. This is called an “elbow plot” and the method can help determine what the optimal number of features to select is. Wherever the elbow point is indicates diminishing returns on adding more features.

```
# Step 1: Define a function to train a model and calculate RMSE for a given subset of features
train_xgb_model <- function(selected_features, X, Y) {
  # Subset the data to include only the selected features
  data <- temp1 %>%
    select(-playerid) %>%  # Drop non-numeric columns
    select(OBP, all_of(selected_features)) %>%
    na.omit()              # Remove rows with missing values

  # Define features (X) and target (Y)
  X <- data %>% select(-OBP)  # Exclude target column
  Y <- data$OBP

  # Train-test split
  set.seed(123)
  train_index <- createDataPartition(Y, p = 0.8, list = FALSE)
  X_train <- as.matrix(X[train_index, ])
  Y_train <- Y[train_index]
  X_test <- as.matrix(X[-train_index, ])
  Y_test <- Y[-train_index]

  # Train the XGBoost model
  dtrain <- xgb.DMatrix(data = X_train, label = Y_train)
  dtest <- xgb.DMatrix(data = X_test, label = Y_test)

  params <- list(
    objective = "reg:squarederror",
    eval_metric = "rmse",
    max_depth = 6,
    eta = 0.1,
    subsample = 0.8,
    colsample_bytree = 0.8
  )

  model <- xgboost(
    data = dtrain,
    params = params,
    nrounds = 100,
    verbose = 0
  )

  # Evaluate the model
  predictions <- predict(model, dtest)
  rmse <- sqrt(mean((predictions - Y_test)^2))

  return(rmse)
}

# Step 2: Train models for each feature subset and calculate RMSE
feature_sets <- list(
  "5 Features" = selected_features_5,
  "10 Features" = selected_features_10,
```

```
  "15 Features" = selected_features_15,
  "20 Features" = selected_features_20,
  "25 Features" = selected_features_25,
  "30 Features" = selected_features_30,
  "35 Features" = selected_features_35,
  "40 Features" = selected_features_40,
  "45 Features" = selected_features_45,
  "50 Features" = selected_features_50
)

rmse_results <- data.frame(
  num_features = integer(),
  rmse = numeric()
)

for (name in names(feature_sets)) {
  selected_features <- feature_sets[[name]]
  num_features <- length(selected_features)
  rmse <- train_xgb_model(selected_features, X, Y)
  rmse_results <- rbind(rmse_results, data.frame(num_features = num_features, rmse = rms
e))
}

# Step 3: Create an elbow plot
plot(
  rmse_results$num_features, rmse_results$rmse, type = "b", pch = 16,
  xlab = "Number of Features", ylab = "RMSE",
  main = "Elbow Plot: RMSE vs Number of Features"
)
```



Based on this elbow plot, it looks like ten features is the best model. What are these ten features?

```
selected_features_10
```

```
##  [1] "AVG_2020"      "xwOBA_2020"    "OBP_2019"      "Spd_2019"
##  [5] "CStr_pct_2019" "OBP+_2019"     "xwOBA_2019"    "AVG_2017"
##  [9] "CStr_pct_2016" "mean_OBP"
```

At first glance, this might seem like it's all over the place in terms of which seasons are helpful for predicting OBP.

There are a few explanations for this. As we might predict, most of the features are from 2019 or 2020, with half of them coming from 2019. This is likely because it's the most recent full season's worth of data and gives a more full picture on a player's true profile. 2020 comes in second because it can still be helpful.

Note there are overlap in some statistics (like xwOBA_2020 and xwOBA_2019). And also some stragglers from 2017 and 2016 for average and called strike percent. This is likely because while the 2019 and 2020 features are doing the heavy lifting (they're the most helpful to predicting OBP), there is still some non-zero amount of pattern that can be explained by these older features.

However, it would be important to test this model on other year's of data to see if the trend holds up that 4- or 5-years' prior stats are actually helpful for predicting OBP in 2022, 2023, and beyond.

Another interesting note is that like I hypothesized, speed is included in the important features. This could suggest that a speed is an important determining factor of OBP, which baseball-knowers likely already know.

Mean_OBP or a players average OBP in the 5 years we look at is also included in the top 10 features. This makes sense as a player's historical performance over a large sample size is likely helpful to predict their future OBP.

Hyperparameter Grid Search for 10

Now, do a hyperparameter grid search again to see which model settings are the most conducive to getting good results.

```
# Step 1: Prepare the data with the top 10 features
data <- temp1 %>%
  select(-playerid) %>%  # Drop non-numeric columns
  select(OBP, all_of(selected_features_10)) %>%
  na.omit()              # Remove rows with missing values

# Define features (X) and target (Y)
X <- data %>% select(-OBP)  # Exclude target column
Y <- data$OBP

# Train-test split
set.seed(123)
train_index <- createDataPartition(Y, p = 0.8, list = FALSE)
X_train <- as.matrix(X[train_index, ])
Y_train <- Y[train_index]
X_test <- as.matrix(X[-train_index, ])
Y_test <- Y[-train_index]

# Step 2: Define the grid for hyperparameter tuning
xgb_grid <- expand.grid(
  nrounds = c(50, 100, 150),            # Number of boosting iterations
  eta = c(0.01, 0.1, 0.3),              # Learning rate
  max_depth = c(3, 6, 9),               # Maximum tree depth
  gamma = c(0, 1, 5),                   # Minimum loss reduction to split
  colsample_bytree = c(0.5, 0.8, 1),    # Fraction of features sampled per tree
  min_child_weight = c(1, 3, 5),        # Minimum sum of instance weights per child
  subsample = c(0.5, 0.8, 1)            # Fraction of samples used for training
)

# Step 3: Define caret trainControl
train_control <- trainControl(
  method = "cv",                        # Cross-validation
  number = 3,                           # Number of folds
  verboseIter = FALSE,                   # Display training progress
  allowParallel = TRUE                  # Allow parallel processing
)

# Step 4: Train the model using caret and grid search
set.seed(123)
xgb_tuned_model <- train(
  x = X_train,
  y = Y_train,
  method = "xgbTree",
  trControl = train_control,
  tuneGrid = xgb_grid,
  metric = "RMSE",                    # Optimize for RMSE,
  verbosity = 0
)

# Step 5: View the best model and hyperparameters
print(xgb_tuned_model$bestTune)
```

```
##     nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 979      50         6 0.1     0              0.5                1         1
```

```
# Step 6: Evaluate the tuned model on the test set
predictions <- predict(xgb_tuned_model, newdata = X_test)
rmse <- sqrt(mean((predictions - Y_test)^2))
print(paste("Test RMSE with Tuned Hyperparameters:", round(rmse, 4)))
```

```
## [1] "Test RMSE with Tuned Hyperparameters: 0.0539"
```

Compare to Industry Models - ATC

I have the projections from Ariel Cohen's ATC model from 2022 and 2023. Let's compare their projections to the actual OBP's in that year by RMSE, the same performance metric we evaluated the above models with.

Load in ATC and Fangraphs Data

```
batc_2022 <- read.csv("~/Documents/GitHub/obp_prediction_model/bat_atc_22.csv")
batc_2023 <- read.csv("~/Documents/GitHub/obp_prediction_model/bat_atc_23.csv")
fg_2022 <- baseballr::fg_bat_leaders(startseason = '2022', endseason = '2022')
fg_2023 <- baseballr::fg_bat_leaders(startseason = '2023', endseason = '2023')

fg_2022$playerid <-as.character(fg_2022$playerid)
fg_2023$playerid <-as.character(fg_2023$playerid)
```

Calculate RMSE from ATC Projections and

Fangraphs Data

```
joined_2022 <- batc_2022 %>%
  select(Name, playerid, OBP) %>%
  inner_join(fg_2022[, c('playerid', 'OBP')], suffix = c('_projected','_actual'), by =
'playerid')
joined_2022 <- joined_2022 %>%
  mutate(diff = OBP_projected - OBP_actual)

rmse_2022 <- sqrt(mean(joined_2022$diff)^2)

joined_2023 <- batc_2023 %>%
  select(Name, PlayerId, OBP) %>%
  inner_join(fg_2023[, c('playerid', 'OBP')], suffix = c('_projected','_actual'), by = j
oin_by('PlayerId' == 'playerid'))
joined_2023 <- joined_2023 %>%
  mutate(diff = OBP_projected - OBP_actual)

rmse_2023 <- sqrt(mean(joined_2023$diff)^2)

paste('RMSE of ATC in 2022 is', rmse_2022)
```

```
## [1] "RMSE of ATC in 2022 is 0.0242784274193548"
```

```
paste('RMSE of ATC in 2023 is', rmse_2023)
```

```
## [1] "RMSE of ATC in 2023 is 0.005296107421875"
```

Result

Overall, ATC does perform significantly better, especially in 2023, when they were off by .005 OBP points on average with their predictions. This isn't a direct comparison as I only looked at the 2021 dataset. I wish I had access to the 2021 predictions to make a direct comparison, but I think I took it too easily that year in fantasy (I got last place that year).

Note, I could also code the columns as years back rather than the specific year for each statistic (e.g. last_year_avg for AVG_2020 or two_years_ago_OBP for OBP_2019). This would make it easie to make predictions in subsequent years and could also give more rows to build the model on. This would be because one row in the data would be one player per one year, rather than one player and all of their stats.

The way that I set up the data in this case, which I believe aligns more with the assignment's asking, limits the scope to only predicting 2021 OBP and isn't super reusable for years ahead.