Baseball Value

Matthew Helbig

2019-09-28

Introduction

The goal of this project is to identify the attributes of the players with the most value and the least value, as well as predicting which players will have the most and least value in the future.

Getting started

The first step we'll take is to identify which datasets we'll be working with. The Lahman Database has a lot of good information that we'll need, namely Player Value data, salary, and dates of birth.

We'll download that from http://www.seanlahman.com/baseball-archive/statistics/

It's possible to download the database as a .CSV file, and you could technically do everything needed for this project without using SQL, but it's a good practice to get used to working with databases, especially since a lot of the datasets you'd see in the real world will be bigger than the one we're currently working with.

Sean Lahman has posted the entire .sql file on his website, so importing it into MySQL is as simple as hitting 'Data Import' and loading that database into its own schema.

Once the Lahman database is all settled in its own schema, we want to start to work with that data. There are packages that let you run SQL commands inside R, and there is definitely some value in not having to hop back and forth between R and your SQL environments. However, I'm more comfortable working in MySQL directly, so we won't worry too much about those R-SQL packages (namely RMySQL).

Setting things up in SQL

The three tables that we're most interested in right now are "Batting," "Pitching," and "Salaries." The database has a ton of information that's interesting, but ultimately not super useful for our experiment. We're going to be mostly concerned with salary information from 2006 to 2016, since a lot of salary data before 2006 is incomplete and we're going to be running some predictive models on 2017 and 2018 data later.

"Subsetting" our Batting data

We'll need to create a new table in SQL based on our criteria of only needing 2006 and later (this database ends in 2016).

```
CREATE TABLE Batting_post_2006
LIKE Batting
```

Now we've created a table with the same containers as our Batting table, and now we need to load the data into those containers.

```
INSERT INTO Batting_post_2006
SELECT *
FROM Batting;
```

Now we're going to trim our data from our new table in order to limit ourselves to only 2006 to 2016. This is where SQL is beneficial, since our original dataset of "Batting" is complete safe and untouched.

```
DELETE FROM Batting_post_2006
WHERE yearID < 2006
```

We're going to do the same thing with our Salaries table, and eventually our Pitching table. We won't do the Pitching table just yet, but we can follow these same steps in order to set that table up for export to R. For now, we'll work on the Salaries table:

```
CREATE TABLE Salaries_post_2006
LIKE Salaries
```

```
INSERT INTO Salaries_post_2006
SELECT *
FROM Salaries;
```

```
DELETE FROM Salaries_post_2006
WHERE yearID < 2006
```

Here, we created a new table to put our data in, moved all of our data from the Salaries table into it, and then limited ourselves to data after 2016. Once again, the original Salaries table is untouched.

Joining Birth Year and Salary

One important step that hasn't happened yet is getting birth year statistics from the Master table in the database. This will allow us to create an Age variable in R later, which will be a potentially key attribute for determining who is overvalued and undervalued. In order to add birth year to our Salary data, we'll need to join the two tables.

First, we create a new container for our combined data:

```
CREATE TABLE Player_Salary_Info (
    AutoNum int NOT NULL AUTO_INCREMENT,
    playerID varchar(255),
    birthyear int,
    yearID int,
    salary int,
    PRIMARY KEY (AutoNum)
);
```

The AutoNum variable is important here, because we need something as the primary key, and if we were to use playerID then we'd have duplicate values between Salary and Master (since the playerIDs are the same between the two tables).

Next, we'll populate our new table with data:

```
INSERT INTO Player_Salary_Info
SELECT 0, Salaries_post_2006.playerID, Master.birthYear,
Salaries_post_2006.yearID, Salaries_post_2006.salary
FROM Salaries_post_2006
INNER JOIN Master ON Salaries_post_2006.playerID = Master.playerID
```

So now we know that we've managed to get playerID, birthyear, yearID, and salary in the same spot. Our next task will be combining this information with our Batting table, at which point we'll be able to move this new table over to R.

Joining Birth Year, Salary, and Batting

One hiccup we face is that several of our variables for some reason are indicated as strings despite being numeric values. I've encountered issues with this before, where a value of 0 would return an empty string (like " ") instead of actually being 0. To solve this, we can change the string columns into integer columns using the following lines of code:

```
ALTER TABLE example
ADD newCol int;

UPDATE example SET newCol = CAST(column_name AS UNSIGNED);

ALTER TABLE example
DROP COLUMN column_name;

ALTER TABLE example
CHANGE COLUMN newCol column_name int;
```

Now that our data is in a format we can work with that won't give us (hopefully) any errors, we can create and populate our Batting table with salary added. We create the parameters for the new table:

```
CREATE TABLE Batting_Salary (
    AutoNum int NOT NULL AUTO_INCREMENT,
    playerID varchar(255),
    birthyear int,
    yearID int,
    salary int,
    stint int,
    teamID varchar(255),
    lgID varchar(255),
    G int,
    AB int,
    R int,
    H int,
    2B int,
    3B int,
    HR int,
    RBI int,
    SB int,
    CS int,
    BB int,
    SO int,
```

```
    GIDP int,
    SF int,
    SH int,
    HBP int,
    IBB int,
    PRIMARY KEY (AutoNum)
);
```

Now we populate our new table:

```
INSERT INTO Batting_Salary
SELECT 0, Batting_post_2006.playerID, Player_Salary_Info.birthyear,
Player_Salary_Info.yearID, Player_Salary_Info.salary, Batting_post_2006.stint,
Batting_post_2006.teamID, Batting_post_2006.lgID, Batting_post_2006.G,
Batting_post_2006.AB, Batting_post_2006.R, Batting_post_2006.H,
Batting_post_2006.2B, Batting_post_2006.3B, Batting_post_2006.HR,
Batting_post_2006.RBI, Batting_post_2006.SB, Batting_post_2006.CS,
Batting_post_2006.BB, Batting_post_2006.SO, Batting_post_2006.GIDP,
Batting_post_2006.SF, Batting_post_2006.SH, Batting_post_2006.HBP,
Batting_post_2006.IBB
FROM Batting_post_2006
INNER JOIN Player_Salary_Info ON Player_Salary_Info.playerID = Batting_post_2006.playerID
AND Player_Salary_Info.yearID = Batting_post_2006.yearID
```

So now we've successfully combined our tables with salary information, birth year information, and batting statistics into one table called Batting_Salary. In the process, we also already shaped the data so that it's only dealing with the years we're concerned with (2006 to 2016). We're just one step away from having our Batter data ready for analysis in R. That step is getting our player value data from Baseball-Reference.

Getting player value data from Baseball-Reference

Our first step is to head to https://www.baseball-reference.com/data/, where they keep a daily log of player value data for every player in baseball history. Player value is measured in a lot of different ways, and a constant source of contention in the baseball community is which metrics provide the most accurate measurement of a player's true value.

The player value metric we'll be focusing on the most is Wins Above Replacement (abbreviated as WAR), which measures how many wins a player provided to his team compared to a random player that you could find in the minor leagues. As a rough translation, consider WAR to be the "stock price" of MLB players, as it gives a simple and quick approximation of the value of a player (or stock).

The information we're looking for is all the way at the bottom of the page, under "war_daily_bat.txt." We'll download that, and convert it to .csv using a spreadhseet program (I'm using Open Office). Just follow the steps for converting everything, and we'll save it to our active folder.

Now we need to load this data into SQL. However, if we look at this data in R:

```
Daily_bat <- read.csv("war_daily_bat.csv")
names(Daily_bat)
```

```
##  [1] "name_common"       "age"               "mlb_ID"
##  [4] "player_ID"         "year_ID"           "team_ID"
##  [7] "stint_ID"          "lg_ID"             "PA"
```

```
## [10] "G"                  "Inn"                "runs_bat"
## [13] "runs_br"            "runs_dp"            "runs_field"
## [16] "runs_infield"       "runs_outfield"      "runs_catcher"
## [19] "runs_good_plays"    "runs_defense"       "runs_position"
## [22] "runs_position_p"    "runs_replacement"   "runs_above_rep"
## [25] "runs_above_avg"     "runs_above_avg_off" "runs_above_avg_def"
## [28] "WAA"                "WAA_off"            "WAA_def"
## [31] "WAR"                "WAR_def"            "WAR_off"
## [34] "WAR_rep"            "salary"             "pitcher"
## [37] "teamRpG"            "oppRpG"             "oppRpPA_rep"
## [40] "oppRpG_rep"         "pyth_exponent"      "pyth_exponent_rep"
## [43] "waa_win_perc"       "waa_win_perc_off"   "waa_win_perc_def"
## [46] "waa_win_perc_rep"   "OPS_plus"           "TOB_lg"
## [49] "TB_lg"
```

We can see that we have a ton of variables that we don't necessarily want. Remember when we created a container in SQL above? If we were to try to read this into SQL now, we'd have to create a container for every one of these variables. We don't necessarily need "oppRpG," which measures how many runs per game that player's opponent scored. Similarly, we don't need a lot of other variables. We're going to limit ourselves to just the ones we want to keep by running the following (Note: you'll need the "dplyr" library for this function):

```
kept_Columns = select(Daily_bat, 1, 2, 4, 5, 6, 31, 32, 33, 36, 47)
```

Which is going to keep only the columns we're interested in, namely:

```
names(kept_Columns)
```

```
##  [1] "name_common" "age"         "player_ID"   "year_ID"     "team_ID"
##  [6] "WAR"         "WAR_def"     "WAR_off"     "pitcher"     "OPS_plus"
```

Next we're going to subset our data to keep it within the year range we're investigating (2006 to 2016), as well as limiting our data to be only batters, since we really aren't interested in how pitchers hit. Pitchers aren't paid based on their ability to hit, so batting statistics for pitchers aren't relevant to what we're studying.

```
years_pitchers_Subset <- subset(kept_Columns, 2005 < year_ID & year_ID < 2017 & pitcher == "N")
```

Finally, we'll remove the pitcher column, since all of our kept players are going to be batters:

```
Batting_value <- select(years_pitchers_Subset, -9)
```

After all this, we're ready to export this data from R and into SQL so we can join it with our Batting_Salary table.

```
write.csv(Batting_value, "Batting_Value.csv")
```

We create our container in SQL by running the following:

```
CREATE TABLE Batting_Value (
    AutoNum int NOT NULL AUTO_INCREMENT,
    name_common varchar(255),
    age int,
    player_ID varchar(255),
    year_ID int,
    team_ID varchar(255),
    WAR DECIMAL(4,2),
    WAR_def DECIMAL(4,2),
    WAR_off DECIMAL(4,2),
    OPS_plus int,
    PRIMARY KEY (AutoNum)
);
```

Then we'll load our data into the container using the following command:

```
LOAD DATA LOCAL INFILE '/filepath/Batting_Value.csv'
INTO TABLE Batting_Value FIELDS TERMINATED BY ','
ENCLOSED BY '"' LINES TERMINATED BY '\n'
IGNORE 1 LINES
```

Now we want to combine this data with our Batting_Salary data so we can have our final dataset to work with in R.

We'll create the container we'll use for our combined data:

```
CREATE TABLE Value_Salary (
    AutoNum int NOT NULL AUTO_INCREMENT,
    playerID varchar(255),
    name_common varchar(255),
    yearID int,
    age int,
    salary int,
    stint int,
    teamID varchar(255),
    lgID varchar(255),
    G int,
    AB int,
    R int,
    H int,
    2B int,
    3B int,
    HR int,
    RBI int,
    SB int,
    CS int,
    BB int,
    SO int,
    GIDP int,
    SF int,
    SH int,
    HBP int,
    IBB int,
```

```
    WAR DECIMAL(4,2),
    WAR_def DECIMAL(4,2),
    WAR_Off DECIMAL(4,2),
    OPS_plus int,
    PRIMARY KEY (AutoNum)
);
```

Then we load in our data:

```
INSERT INTO Value_Salary
SELECT DISTINCT 0,
Batting_Salary.playerID,
Batting_Value.name_common,
Batting_Salary.yearID,
Batting_Value.age,
Batting_Salary.salary,
Batting_Salary.stint,
Batting_Salary.teamID,
Batting_Salary.lgID,
Batting_Salary.G,
Batting_Salary.AB,
Batting_Salary.R,
Batting_Salary.H,
Batting_Salary.2B,
Batting_Salary.3B,
Batting_Salary.HR,
Batting_Salary.RBI,
Batting_Salary.SB,
Batting_Salary.CS,
Batting_Salary.BB,
Batting_Salary.SO,
Batting_Salary.GIDP,
Batting_Salary.SF,
Batting_Salary.SH,
Batting_Salary.HBP,
Batting_Salary.IBB,
Batting_Value.WAR,
Batting_Value.WAR_def,
Batting_Value.WAR_off,
Batting_Value.OPS_plus
FROM Batting_Salary
INNER JOIN Batting_Value ON Batting_Salary.playerID = Batting_Value.player_ID AND Batting_Salary.yearID
```

As a note, this table works very well until it comes to the players who were traded in the middle of the season. When a player hasn't been traded, there is one row per year. However, when a player has been traded, we would expecto see two columns that year (one for the old team, one for the new team). However, we see four:

```
##   AutoNum  playerID        name_common yearID age   salary stint
## 1       1  abadan01          Andy Abad   2006  33   327000     1
## 2       2 abercre01 Reggie Abercrombie   2006  25   327000     1
## 3       3 abreubo01        Bobby Abreu   2006  32 13600000     1
## 4       4 abreubo01        Bobby Abreu   2006  32 13600000     1
```

```
## 5        5 abreubo01        Bobby Abreu   2006  32 13600000     2
## 6        6 abreubo01        Bobby Abreu   2006  32 13600000     2
```

Split, apply, combine?

We need to remove those duplicates, and one very common way of doing so is via a process often used in data analysis called "Split, apply, combine."

This process involves splitting the data into subsets, applying a function to each subset, and then combining the data back together. Which is what we want to do here. We want to remove the duplicate players from every year, but we don't want to remove duplicates from the dataset as a whole. For example, there are multiple "Bobby Abreu" rows in our 2006 data, which we would like to reduce to a single row. However, we don't want to remove all of the "Bobby Abreu" rows from our master dataset, because that would remove Bobby Abreu's stats for 2007, 2008, and so on.

We could subset our data by year, write a function that removes duplicates for each year, and then join all of our subsets back together. Or, we could use a library that we've already called, the 'dplyr' library, and remove the duplicates by year from our main dataset.

If we execute the following command:

```
Batting_Value_Salary <- Pre_Batting_Value_Salary %>%
  distinct(playerID, yearID, .keep_all = TRUE)
```

We can see that we've successfully removed the duplicates:

```
##   AutoNum  playerID        name_common yearID age   salary stint
## 1       1  abadan01          Andy Abad   2006  33   327000     1
## 2       2 abercre01 Reggie Abercrombie   2006  25   327000     1
## 3       3 abreubo01        Bobby Abreu   2006  32 13600000     1
## 4       7 adamsru01         Russ Adams   2006  25   343000     1
## 5       8 aguilch01       Chris Aguila   2006  27   327000     1
## 6       9 alfoned01    Edgardo Alfonzo   2006  32  8000000     1
```

This is a nice way of using pre-built libraries to limit the amount of code that you need to write. The dplyr library has a ton of functions like this, and we'll make use of them when we further analyze our data in R. #Reading in our pitching data

Subsetting our pitching data

First we create our container in SQL:

```
CREATE TABLE Pitching_post_2006
LIKE Pitching
```

Then we load our data into our table:

```
INSERT INTO Pitching_post_2006
SELECT *
FROM Pitching;
```

Then we do what we did with Batting and Salary and limit our data to just 2006-2016:

```
DELETE FROM Pitching_post_2006
WHERE yearID < 2006
```

We create our Pitching_Salary table:

```
CREATE TABLE Pitching_Salary (
    AutoNum int NOT NULL AUTO_INCREMENT,
    playerID varchar(255),
    birthyear int,
    yearID int,
    salary int,
    stint int,
    teamID varchar(255),
    lgID varchar(255),
    W int,
    L int,
    G int,
    GS int,
    CG int,
    SHO int,
    SV int,
    IPouts int,
    H int,
    ER int,
    HR int,
    BB int,
    SO int,
    ERA float,
    BK int,
    R int,
    PRIMARY KEY (AutoNum)
);
```

Then we load our Pitching and Salary data into our new table:

```
INSERT INTO Pitching_Salary
SELECT 0,
Pitching_post_2006.playerID,
Player_Salary_Info.birthyear,
Player_Salary_Info.yearID,
Player_Salary_Info.salary,
Pitching_post_2006.stint,
Pitching_post_2006.teamID,
Pitching_post_2006.lgID,
Pitching_post_2006.W,
Pitching_post_2006.L,
Pitching_post_2006.G,
Pitching_post_2006.GS,
Pitching_post_2006.CG,
Pitching_post_2006.SHO,
Pitching_post_2006.SV,
Pitching_post_2006.IPouts,
Pitching_post_2006.H,
```

```
Pitching_post_2006.ER,
Pitching_post_2006.HR,
Pitching_post_2006.BB,
Pitching_post_2006.SO,
Pitching_post_2006.ERA,
Pitching_post_2006.BK,
Pitching_post_2006.R
FROM Pitching_post_2006
INNER JOIN Player_Salary_Info ON Player_Salary_Info.playerID = Pitching_post_2006.playerID
AND Player_Salary_Info.yearID = Pitching_post_2006.yearID
```

So now we're at the point where we need to go back to Baseball-Reference and get our value data. For batters, it was called "war_daily_bat," and for pitchers

Notes to self

Tomorrow, we should read in the pitcher data from Baseball-Reference, and see how far we can get on setting up our data to be similar to our Value_Salary data