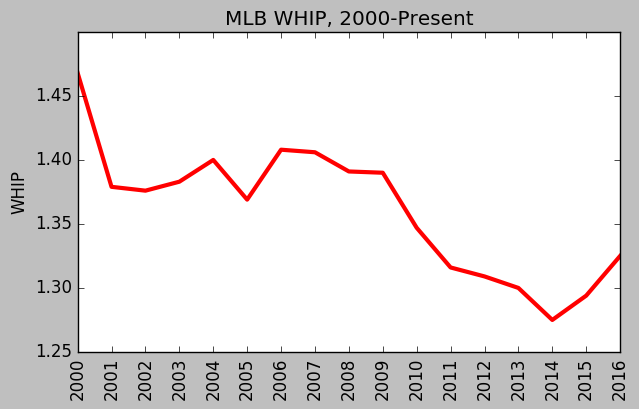
“[God gets you to the plate, but once you're there you're on your own.](http://www.azquotes.com/quote/1306902)”

*-Ted Williams[[1]](#footnote-1)*

Business Understanding

There have been two noticeable trends in Major League Baseball over the last five to ten years: a marked decline in the overall level of hitting, from both the standpoint of average and power[[2]](#footnote-2), as well as a similar degree of improvement in the quality of pitching. When trying to explain reasons for the former, experts point to the end of the “Steroid Era”, brought on by the introduction of league-wide testing for performance-enhancing drugs.[[3]](#footnote-3) The subsequent fall in batting performance can then be attributed to a reversion to the mean from artificially inflated numbers.

On the flip side, a number of factors such as advanced metrics and modern improvements in conditioning and preventive arm care have led to a completely new set of rules governing how pitchers are deployed in the game. Whether it is through a conscious reduction of their workloads to decrease stress on their throwing arms, or employing statistically-driven fielding shifts to strengthen defenses behind them, men on the mound are being managed in an increasingly systematic fashion. Recent results speak for themselves, as demonstrated by the decade-long downward trend in league average WHIP[[4]](#footnote-4) (an advanced benchmark designed to gauge a pitcher’s performance against batters- lower is better).



Regardless of whether it is driven by weaker batters, more dominant pitchers or a combination of the two, the issue facing major league franchises and their management is clear: the task of hitting a baseball has become increasingly difficult. Given the abundance of data currently available and the wide acceptance of advanced metrics, isn’t it time to provide hitters with a robust analytical tool that can be used in real time to boost performance? Why not attempt to design a predictive model that can tell them what pitches they can expect to face during every at bat?

The most critical component to any modeling process is a reliable dataset, and MLB already has one in place. Pitchfx is a pitch tracking system that records multiple pitch features including velocity, movement and spin rate for every pitch thrown in a game.[[5]](#footnote-5) It is currently installed in every MLB Stadium and has been in use since 2006. If utilized correctly, it should provide more than enough empirical data to construct reliable, actionable indicators that can be deployed in a real-time game environment.

Data Understanding and Preparation

To build a model to predict the next pitch type, we downloaded the MLB’s Pitchfx data from 2014, 2015, and 2016. The dataset includes every pitch from every regular season and playoff game during these three seasons. Each instance in the dataset is a single pitch. There are roughly 2.1 million records in the entire dataset.

Dataset Compilation/Web Scraping

Given the project aim, Pitchfx data is the perfect source for building a prediction model. There are several online sources for scraping Pitchfx data (Brooks Baseball[[6]](#footnote-6), Baseball Heat Maps[[7]](#footnote-7), PitchRx) that employ various methods and formats to return some or all of the desired information. The challenge was to find an option that combined ease of use with the most comprehensive output possible.

Initially, we tried using PitchRx[[8]](#footnote-8), an R-based package designed specifically for Pitchfx. Unfortunately, the scripts were designed in a way that prevented the retrieval of one comprehensive dataset. Even if we had been able to concatenate the **pitch** and **atbat** subsets, they still lacked a sizeable amount of the features we had earmarked for potential analysis.

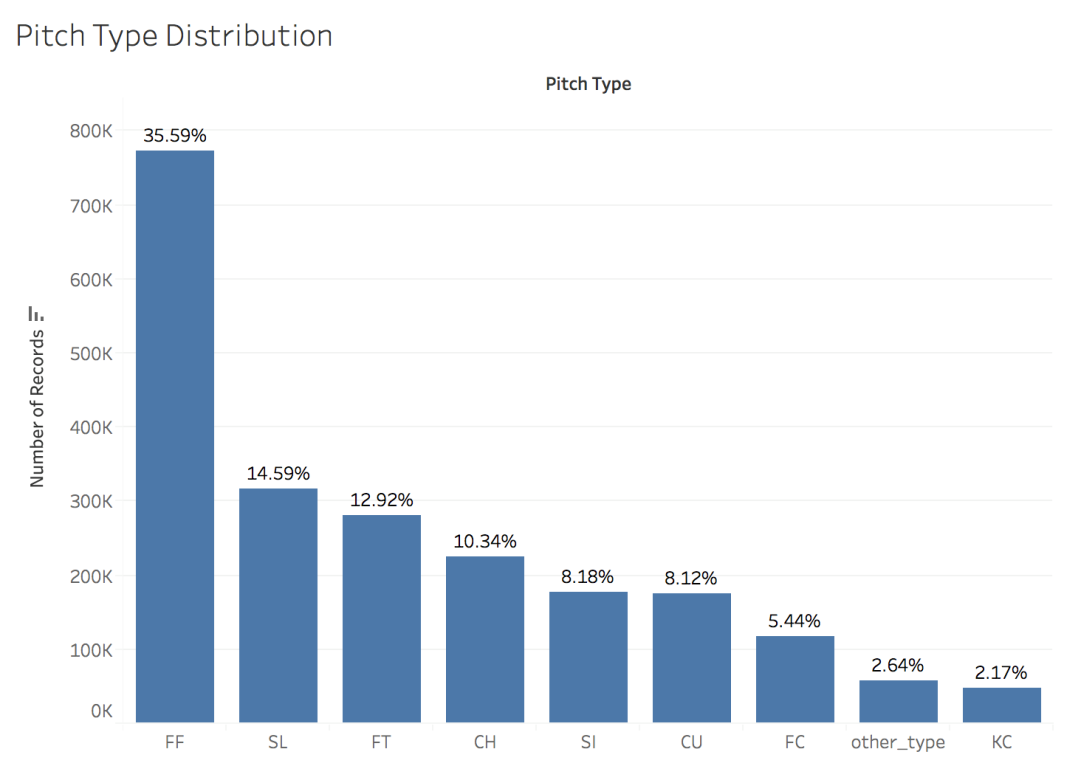
Our second approach was to entertain the idea of a customized scraper. This led us to look directly at the master archive itself, which is publicly available through MLB Gameday Directory[[9]](#footnote-9) and consists of a large number of sub-directories, the majority of which are organized by individual year, providing a hierarchical structure for each season’s data, all the way down to the pitch-by-pitch level. Information is stored in XML files, which most often represent a single inning of every game played. Navigating this complex environment would require designing a script to locate and then scrape all desired samples using BeautifulSoup, a Python library for pulling data from select file types (XML, HTML).

Before committing the time and effort to code a custom solution, we conducted one last round of research that ultimately led us to a Python-based parser/scraper[[10]](#footnote-10) that fortunately provided exactly what we needed. The scraper retrieves the features listed below and outputs them via two files (**pitch\_table.csv** and **atbat.csv**) that catalog the information on a per pitch basis:

* Game ID
* Flags
  + Spring Training
  + Regular Season
  + Playoffs etc.
* MLB Game ID
* Game Location Data
* Batter/Pitcher ID data
* Game Situation Data (balls and strikes, number of outs, inning)
* Pitch outcome sequence up to that point in the plate appearance
* Flag to designate if the pitch is the last pitch in the plate appearance
* Retrosheet-style event code

Target Variable and Features

The target variable (“pitch type”) was already given to us in the form of a two-letter string. Overall, there are eight major pitch types recorded in the MLB’s data. They are: 1) Four-Seam Fastball (FF), 2) Two-Seam Fastball (FT), 3) Cut Fastball (FC), 4) Sinker (SI), 5) Curveball (CU), 6) Change-up (CH), 7) Slider (SL), and 8) Knuckle-Curve (KC). A distribution of the classes of the target variable is shown below:



Four-seam fastballs (FF) appear much more frequently than the other classes, the majority of which range from 5% to 15%. Even though knuckle-curves comprise only 2.17% of total pitches, we decided to keep KC as a distinct class because there are a handful of pitchers who throw a lot of knuckle-curves. In other words, KC is a rare pitch, but if a particular pitcher is on the mound, KCs can occur quite frequently.

Below is a chart illustrating the features we used in our model:

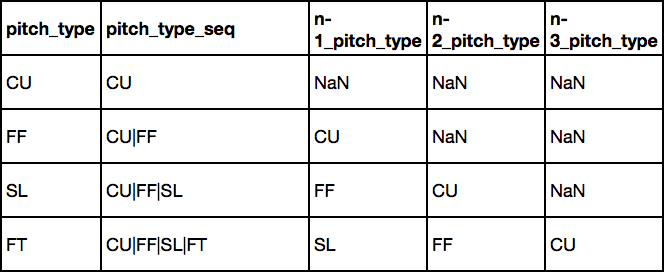
|  |  |
| --- | --- |
| **Target Variable** | Pitch Type (FF, FT, FC, SI, CU, CH, SL, and KC) |
| Game-Specific Features | * Balls and Strikes Count * Number of Outs * Pitch type of previous three pitches (n-1, n-2, and n-3) * Outcome of previous three pitches (n-1, n-2, and n-3) * Scoring Position: binary variable indicating if a baserunner is on second base or beyond |
| Pitcher-Specific Features  \*Note: for most pitching features, we used the player’s statistics **from the previous season** to avoid leakage problems. | * Pitcher-handedness (right- or left-handed) * Pitcher age * Cumulative pitch count (per game) * Strikeout Rate (Strikeouts/Batters Faced) * Walk Rate (BBs/Batters Faced) * Strikeout-to-Walk ratio * Earned Run Average (ERA) * Walks + Hits per Inning Pitched (WHIP) * Home Runs Allowed per 9 innings (HR9) * % of Batted Balls in each of the 4 categories: Ground Balls (GB%), Fly Balls (FB%), Line Drives (LD%), and Pop-Ups (POP%) * % of Pitches in each of the 8 pitch type categories: FF, FT, FC, SI, CU, CH, SL, and KC. |
| Batter-Specific Features  \*Note: for most batting features, we used the player’s statistics **from the previous season** to avoid leakage problems. | * Batter-handedness (right- or left-handed) * Position in Lineup (1 to 9) * Batting Average * On-Base Percentage * Slugging Percentage * Swing Rate (% of pitches swung at) * Contact Rate (% of swings where batter made contact) * Out-of-Zone Swing Rate (% of pitches outside the strike zone the batter swung at) * Out-of-Zone Contact Rate (% of swings outside strike zone where batter made contact) * Pitches per Plate Appearance (proxy for a batter’s “patience”) |

The features of our dataset were obtained in three ways: 1) our core dataset from the MLB; 2) feature engineering, and 3) pitching and batting statistics from the website Baseball Prospectus.

Feature “Engineering”

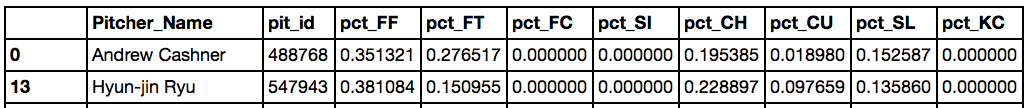
Once the dataset was downloaded, we engineered a number of features. In many cases, creating new features was difficult because of the scale of our dataset, which largely prevented us from using “for” loops. However, we still managed to create the following features:

* *Pitch Type of Previous Three Pitches*: we created three separate variables (n-1, n-2, and n-3) denoting the pitch type of the previous three pitches. To create these variables, we extracted two-letter strings from the column “pitch\_type\_sequence” provided by the MLB. Pitch\_type\_sequence tracks the sequence of pitch types for each at-bat. An illustration is provided below.

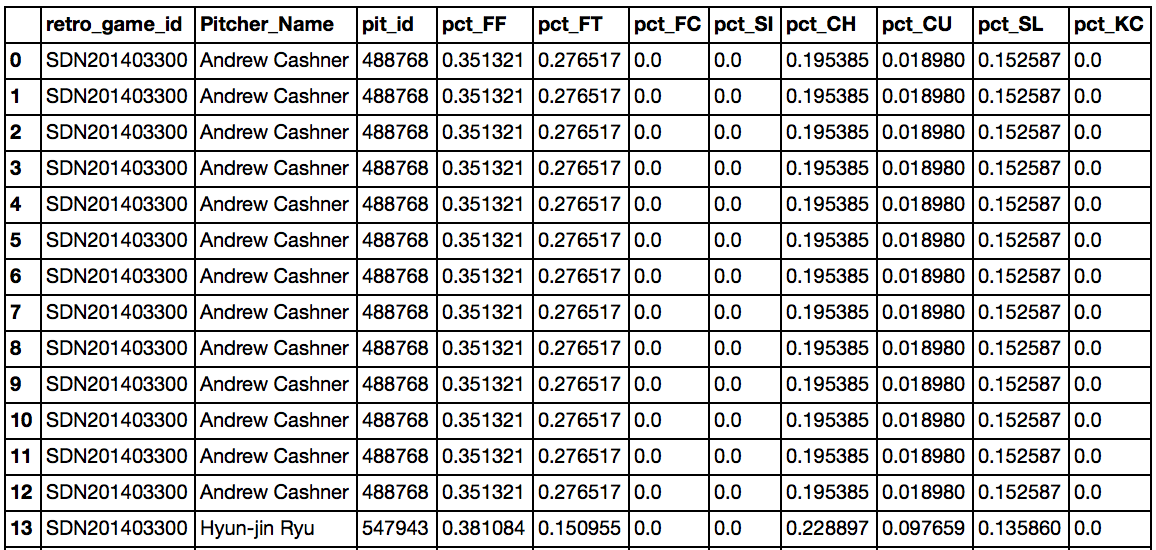


To deal with the missing values, we plugged in the pitch type “Other” – which includes all pitches that are not in the 8 majority classes. We also created three “missing” indicator variables to specify when we had missing data.

* *Pitch Result of Previous Three Pitches:* we followed a similar procedure as above, creating three separate variables (n-1, n-2, n-3) to denote the pitch result of the previous three pitches. To clarify, “pitch result” indicates whether the pitch was a ball (B), swinging strike (S), called strike (C), foul tip (F), or batted ball (X). We extracted characters from a column called “pitch\_seq”.
* *Scoring Position*: the MLB’s dataset used a bizarre metric (‘start\_bases’) for the presence of baserunners. Effectively, the MLB assigned 1 point if a runner was on first base, 2 points for a runner on second base, and 3 points for a runner on third base. The values thus ranged from 0 (nobody on) to 6 (bases loaded). To simplify matters, we decided to convert “start\_bases” into a binary indicator variable “scoring position”. In baseball, a runner is considered to be in scoring position if he is on second base or beyond. As such, our binary variable equaled “0” if “start\_bases” < 2, and “1” if start\_bases >= 2.
* *Cumulative Pitch Count:* this variable measures the number of pitches thrown by the pitcher *at the time* of his next pitch. A pitcher might demonstrate different tendencies depending on how many pitches he has already thrown in a game. To create the variable, we used the *groupby* function to partition the dataset based on ‘game\_id’ and ‘pitcher\_id’, then used the *cumcount()* function in the same line.
* *Batter Position:* This feature is a proxy for batter quality: good players are generally in the 1-5 spots, while bad players are in the 6-9 spots. To create this feature, we first created another variable (“first\_pitch”) to signal whether the pitch was the first pitch in an at-bat (introducing the start of a new at-bat). We then used the *groupby* function to partition the data based on ‘game\_id’ and ‘bat\_home\_id’ (a binary variable indicating which team was at bat). Afterwards, we used the *cumsum()* function on the “first\_pitch” column and “divided” the result by modulo 9. (e.g. 12 % 9 = 3).
* *Percent of Pitches for each Pitch Class:* For each pitcher in the dataset, we calculated the percentage of his pitches that were FF, FT, FC, SI, SL, CU, CH, and KC *in the previous year*. To build these eight variables (pct\_FF, pct\_FT, etc.), we divided our large dataset (of 2 million records) into three smaller data frames: 2014 pitch records, 2015 pitch records, and 2016 pitch records. We also downloaded the MLB’s 2013 Pitch F/x data (“2013 pitch records”). The next step was to take the unique ‘Pitcher\_ID’ codes from the 2014, 2015, and 2016 datasets. Using 2014 as an example, we took each ‘Pitcher\_ID’ in 2014 and obtained: i) the number of pitches in the 2013 data thrown by that pitcher in each pitch class (FF, FT,…); and ii) the total number of pitches thrown by the pitcher in 2013. An example of the output is below:

**

Once we had the pitch distribution for each pitcher\_ID in 2014 (using 2013 data), we merged this information to the 2014 pitch records, using an “inner” merge with “pit\_id” as the key. Effectively, we did a one-to-many merge. An example below:



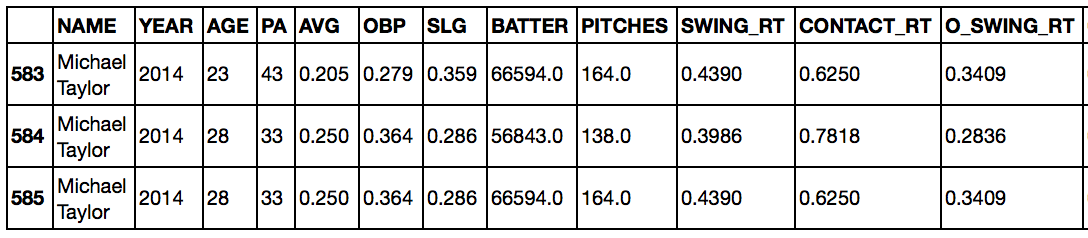
Finally, we concatenated the 2014, 2015 and 2016 pitch records to “re-create” our large dataset.

Other Pitching and Batter Statistics

We obtained several batting and pitching features from a baseball sabermetrics website called Baseball Prospectus.[[11]](#footnote-11) The site has a variety of tables for batting and pitching statistics. On the pitching side, we used the “Pitcher Season – Rates”[[12]](#footnote-12) and “Pitcher Season – BIP” (Balls-in-Play)[[13]](#footnote-13) tables. For batting, we used the “Batter Season - Standard“[[14]](#footnote-14) and “Batter Plate Discipline”[[15]](#footnote-15) tables. We collected data from the 2013, 2014, and 2015 seasons.

* *Pitcher Season – Rates*: This table contained features on the overall effectiveness of each pitcher, such as ERA, WHIP, UBBr (walks/batters faced), SOr (strikeouts/batters faced), SO/BB (strikeout-to-walk ratio), and HR9 (home runs allowed per 9 innings). We also obtained the pitcher’s age. We thought pitcher age could potentially be a significant feature, with the idea that older pitchers tend to rely on slower pitches as they lose arm strength. We were not exactly sure how the “performance features” in this table would influence our target variable (pitch type), but we had the vague hypothesis that high-performing pitchers might exhibit similar characteristics in their pitch type selection. For example, elite pitchers might be more clever in how they mix up their pitches than below-average pitchers.
* *Pitcher Season – Balls-in-Play:* This table included features on the percentage of batted balls against each pitcher that were: 1) ground balls; 2) fly balls; 3) line drives; and 4) pop-ups. We were motivated to obtain these features because in baseball parlance, pitchers are frequently described as “ground ball pitchers” or “fly ball pitchers”. Ground ball pitchers tend to rely on pitches with substantial downward movement, such as sinkers (SI) and two-seam fastballs (FTs). Fly ball pitchers, on the other hand, tend to throw high fastballs to induce weak upward contact.
* *Batter Season – Standard*: This table contained basic batting statistics such as batting average, on-base percentage, and slugging percentage. We speculated that pitchers might use different pitch types against good hitters and bad hitters.
* *Batter Plate Discipline:* This table included features on the hitter’s tendencies during an at-bat. Important features included Swing\_Rate (% of pitches swung at), Contact\_Rate (% of swings that result in contact), O\_Swing\_Rate (for pitches outside the strike zone), and O\_Contact Rate (for pitches outside the strike zone). Pitchers are likely to pitch more aggressively to hitters with a high swing rate. An “aggressive” pitching strategy usually involves throwing pitches outside the strike zone to lure impatient hitters into swinging at an errant pitch. Against hitters with a high swing rate, pitchers might rely on pitches that are difficult to control (such as a knuckle-curve (KC)). However, if the batter rarely swings at pitches outside the zone, the pitcher will probably choose pitch types with higher levels control (such as a regular four-seam fastball).

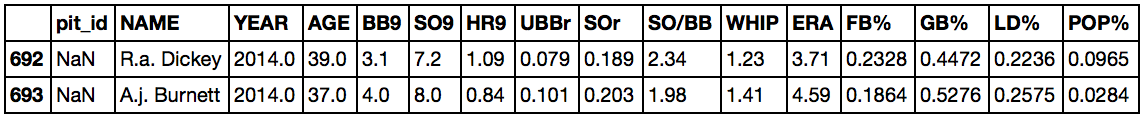
We wish we had obtained the data using an advanced scraping method, but we relied on brute force here, copying and pasting all of the information into Excel. We had three years’ worth of data (2013-2015) and four tables for each year, meaning 12 Excel files altogether. Next, for each year, we merged the two batting tables and we merged the two pitching tables. This step was tricky because the only available key to merge on was ‘NAME’, and there were several players who shared the same name. (One of the batting tables had playerID, but the other batting table did not). To resolve this issue, we first conducted an “outer” merge to see if there were any duplicated names.[[16]](#footnote-16) An example from 2014 is below.



After finding the duplicated names, we had to go back and designate one of the duplicated names with an underscore (e.g. Michael Taylor\_1) in the Excel file to distinguish between the two players. Luckily, there weren’t too many cases where this needed to be done, and we were extremely careful while making these modifications.

Once this step was complete, we had to link the Baseball Prospectus player IDs (denoted as ‘BATTER’ in the example above) to the MLB player IDs used in our original dataset. To accomplish this, we downloaded an online csv file mapping the Baseball Prospectus to the MLB IDs.[[17]](#footnote-17) Linking the Baseball Prospectus batting data to the MLB IDs was relatively simple because we had player IDs for the batters.

However, linking the Baseball Prospectus pitching data to the MLB IDs was much more difficult, because Baseball Prospectus did not include PlayerIDs in the particular pitching tables we chose. As a result, we needed to merge with “Name” as the key. This was problematic, however, because the MLB and Baseball Prospectus often provided slightly different player names. For example, the MLB uses “R.A. Dickey” while Baseball Prospectus uses “R.a. Dickey”. To resolve this issue, we employed a similar strategy as above, using an “outer merge” to find examples where the MLB and Baseball Prospectus names were not identical – in particular, when the column “pit\_id” was empty.



In this case, we would have gone back to the original Excel files and edited the names above as “R.A. Dickey” and “A.J. Burnett”. This editing process was tedious but necessary.

Finally, once this process was done, we split our large, 2-million record dataset into 2014, 2015, and 2016 data frames. We then took the 2013 batting data and merged it to the 2014 records using an inner merge with “bat\_id” as the key. We employed a similar merge between the 2013 pitching data and the 2014 records, this time using “pit\_id” as the key. We repeated this procedure for the other two years, and then concatenated the 2014, 2015, and 2016 records back into one large data frame.

Other Data Manipulation

**Game Type Description**

The raw dataset includes a feature to describe the exact type of game that produced each individual sample: *Regular Season, Wild-Card Game, Divisional Series, League Championship Series (LCS), World Series, Spring Training* and *Unknown*. In order to keep our analysis limited to official games and lower the number of NaNs, we disregarded the pitches thrown in the *Spring Training* and *Unknown* categories*.*

**Creating Dummy Variables**

After cleaning the dataset, it was necessary to perform some feature engineering on a group of seven k-class variables in order to run our baseline Logistic Regression model:

* Pitch Type
* Batter Position
* Months
* Year
* Number of Outs
* Ball Count
* Strike Count

The required k-dummy variables were generated using the **One Hot Encoding** module.

**Normalization**

While many of our features were between 0 and 1 (mostly percentages), we had seven additional variables that needed be expressed on a scale of 0 to 1. The features below were normalized using **minmax**. Compared to other normalization techniques, minmax provides a result with smaller standard deviations, which can suppress the effect of outliers.

* Pitcher Age
* Cumulative Pitch Count
* Home Runs Surrendered Per 9 Innings (HR9)
* Strikeout to Walk Ratio (SO/BB)
* Walks + Hits per Innings Pitched (WHIP)
* Earned Run Average (ERA)
* Pitches Per Plate Appearance

Modeling & Evaluation

For our project, we considered three types of algorithms: logistic regression, decision trees, and random forests. The pros and cons of each algorithm are listed below:

*Logistic Regression*

Pros:

* Efficient dealing with our large dataset; LR is the fastest algorithm we used.
* Fewer parameters to be tuned which make it easier to improve.
* Low variance, so less prone to overfitting

Cons:

* Since we have a mix of multi-class and numerical features and our target variable is also multi-class, we needed to create several dummy variables
* LR requires that each data point be independent of all other data points. If observations are related to one another, then the model will tend to overweight the significance of those observations.
* LR is highly biased
* Assumes all features are linearly related to the log odds of the target variable.

*Decision Trees*

Pros:

* DT is unbiased
* Decision trees can automatically detect non-linear features and interactions between features, without having to make explicit variables
* Decision trees are fairly intuitive compared to other models

Cons:

* Prone to overfitting, especially if class sizes are small/imbalanced
* High variance
* More parameters to be tuned

*Random Forests:*

Pros:

* Almost always performs better than DT
* Reduces the variance of decision trees
* If done correctly, individual trees are independent
* RF, like DT, can handle very well high dimensional spaces as well as large number of training examples

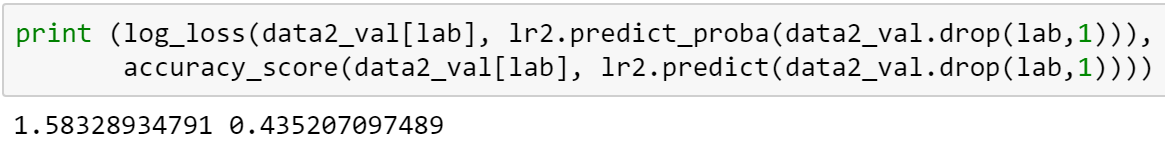
Cons:

* Very slow to run
* Difficult to interpret as a “black box” method; an aggregation of many different decision trees
* Even more parameters than DT to be tuned

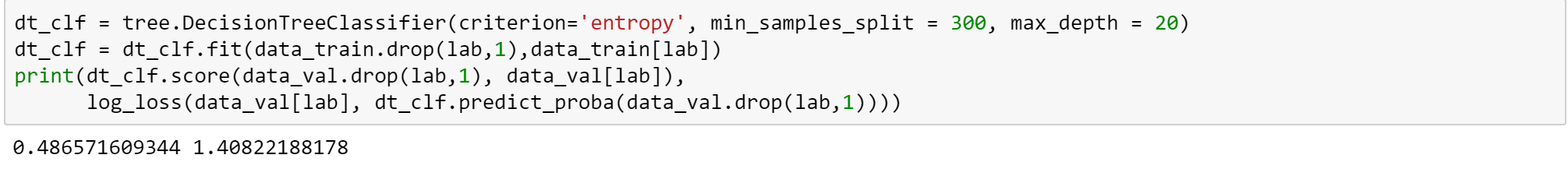
We designated the 2014-2015 data as our training set and the 2016 data as our test set. We felt this approach was the most practical because it makes more sense for test instances to occur after training instances. Given the size of our dataset, we did not feel that cross-validation was necessary. Instead, we split the training set into training and validation subsets using an 80/20 random split. Ultimately, we sought to evaluate three metrics: 1) accuracy, 2) log-loss, and 3) the Expected Value an MLB team would receive by making a series of predictions above a certain confidence threshold.

Logistic Regression and Baseline Model

To identify a heuristic baseline for accuracy, we can simply take the percentage of pitches in the dominant class (‘FF’) of our target variable. As illustrated earlier, this value is 0.3593. Our baseline model was an L2-regularized logistic regression with normalized data. The log-loss was 1.5832 and the accuracy was 0.4352.

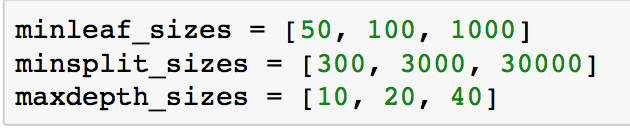


We want to improve the accuracy and decrease the Log-loss of our model. To improve the baseline model, we will try different models with several combinations of hyper parameter settings. The model with lowest log-loss and highest accuracy will be our optimal choice. First, however, we created dummy variables for multi-class features and then ran the L2-regularized logistic regression, from which we have following result: log-loss decreased to 1.56502965799 and accuracy increased to 0.437028310956.



Decision Trees

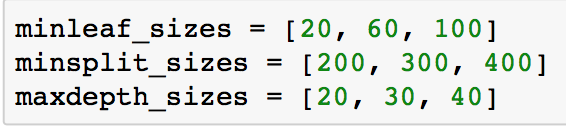
In an attempt to improve upon the performance of our baseline model using Logistic Regression, we decided to run several **decision tree** instances, employing different hyper-parameters that included *maximum depth, minimum leaf size* and *minimum split size*. A cross-section of 3 values was selected for each attribute and individual trees were then fit and run for each possible permutation, yielding the following accuracy scores:



**

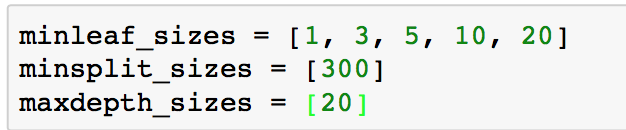
For each *min leaf size,* the smaller the *min split size*,the better the score. Moreover, higher *max depths* tend to give higher scores. Lastly, for each *min split size,* the smaller the *min leaf size* the better the score. Given the values above, the highest score was **0.485**.

Using information gained from the results above, we adjust the hyper-parameters to improve our model selection. The output below shows that the highest score from this round was **0.486** (a slight improvement)with a *max depth* of 30 and *min split size* of 300.





After settling on *max depth* and *min split size* as 20 and 300, respectively, we need to determine the optimal *min leaf size*. The additional tests show that the smaller *min leaf size* has performed better:



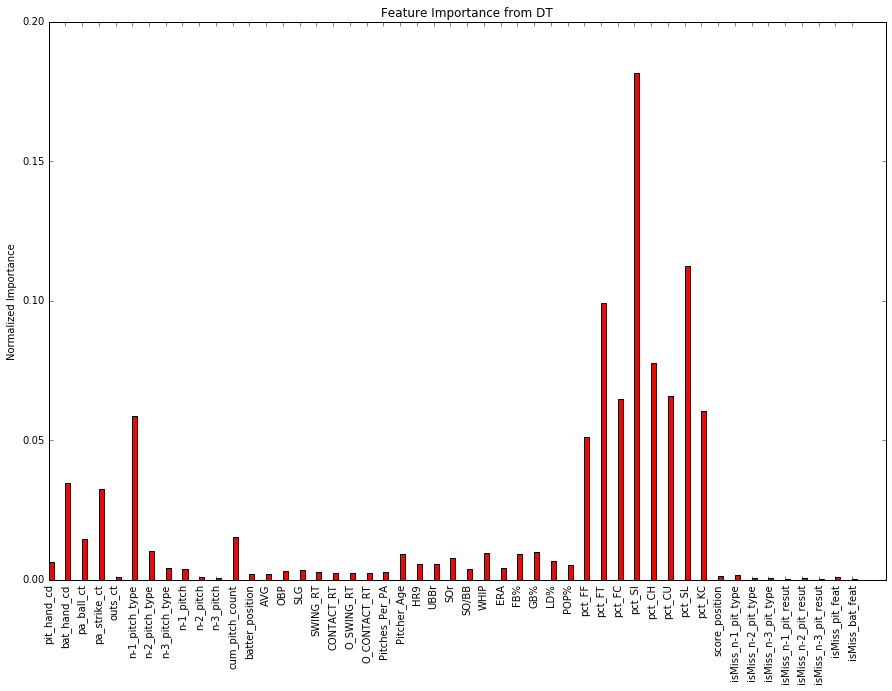


To conclude, the best result **0.486** (which outperforms the Logistic Regression Model) and log-loss **1.409**, is achieved using:

* *max depth size* = 20
* *min leaf size* = 5
* *min split size* = 300

Although, 1 and 3 perform slightly better than *min leaf size 5,* in order to reduce the complexity of the model, we used *min leaf size* 5 in our final model. The below figure summarizes the feature importance for our final decision tree models. The top three most important features are:

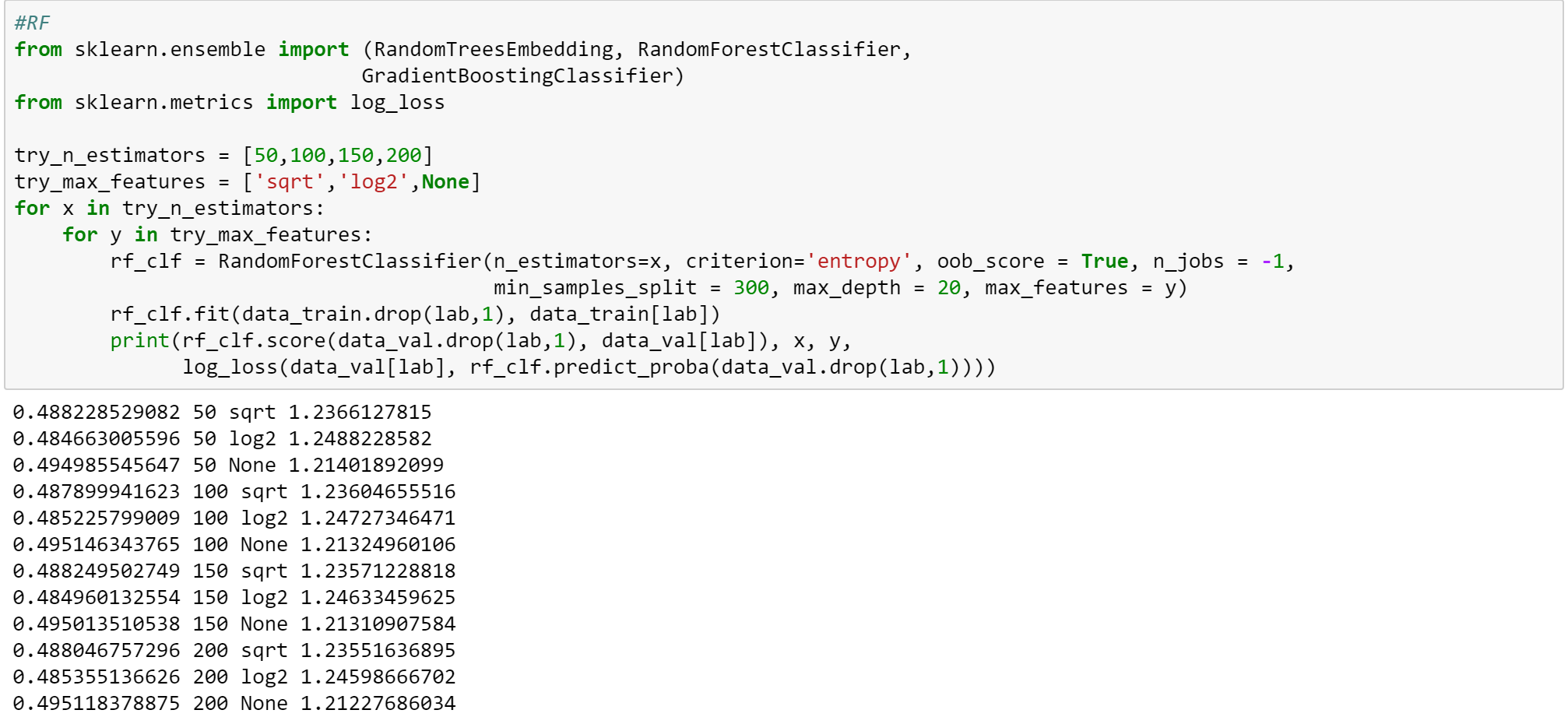
* **pct\_SI** = % of Sinkers thrown by pitcher in previous year (Sinkers/Total Pitches)
* **pct\_SL** = % of Sliders thrown by pitcher in previous year (Sliders/Total Pitches)
* **pct\_FT** = % of 2-Seam Fastballs thrown by pitcher in previous year (2-Seam Fast/Total Pitches)



It should come as no surprise that the most important features are the pitcher’s distribution of pitch types from the previous season. For example, if a pitcher threw many sinkers in 2013, he will probably throw a lot of sinkers in 2014 as well. But why is the feature importance for pct\_SI higher than the others? Sinkers are sometimes used to induce ground balls; perhaps the model predicts a higher likelihood of a sinker when there are runners on base and the pitcher wants to get a double play. Besides the pitch distribution features, the variable “n-1 pitch type” (the class of the previous pitch) appears to have some degree of predictive power. If a pitcher throws a 96 mph four-seam fastball (FF) on the previous pitch, he might be more inclined to throw in an 85 mph changeup (CH) to keep the batter off-balance.

Random Forests

We tried to do feature selection and used various subsets of features to run our models. Without exception, using all the features led to better performance than using a subset of the features. The output below indicates that both accuracy increased and log-loss decreased when we set max\_features = None, rather than use the square root or log2 of the total number of features. Since our features have small correlation – most of the pairwise correlations are less than 0.02 – this could mean that we can improve our model by adding more useful features.

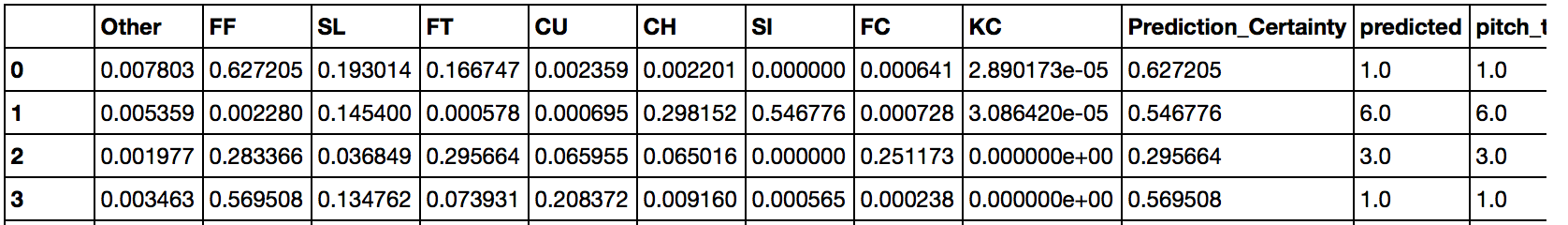


The best result we got is: log-loss decreased to 1.2015195355 and accuracy increased to 0.499554309564.

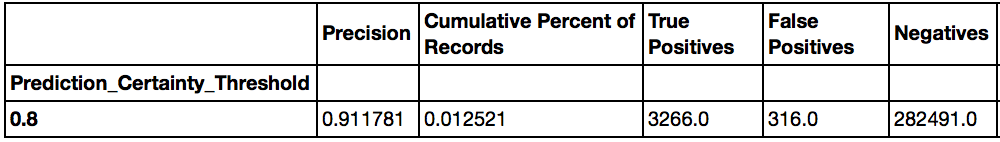
“Net Gain” or Expected Value of Predictions

In addition to calculating accuracy and log-loss, we also attempted to develop a metric more tailored to the needs of MLB teams, called “Net Gain”. At the end of the day, a baseball team wants to know whether the model will deliver positive results. One obstacle that arises with a multi-class target variable is that none of the classes have especially high probability, meaning that the exact type of the next pitch can be hard to predict. Furthermore, providing only a probability distribution for the next pitch (e.g. 25% chance of FF, 40% chance of CU, 35% chance of SI) is useless for a hitter in the batter’s box. Humans are notoriously bad at interpreting probability. A direct prediction (e.g. “the next pitch will be a CU”) is much more effective. However, this brings up the question: when should a prediction be given to the batter? It may be misleading to instruct the hitter to expect a CU when we are only 40% sure of the outcome.

With this problem in mind, we developed the idea of a “Prediction Certainty Threshold.” For example, at a threshold of 0.8, an MLB team would instruct the hitter to expect a certain type of pitch *only* when the team was at least 80% sure of its prediction. To calculate the “Prediction Certainty” for each pitch, we used the *predict\_proba()* function to predict the probabilities of class membership. There were nine total classes (the main eight classes, plus an “Other” class). We then found the maximum probability in each row (ignoring the “Other” column) to derive the “prediction certainty” of a given pitch. We then added columns indicating: 1) the predicted outcome and 2) the actual pitch type. A sample of the subsequent data frame is illustrated below. The final column is the actual pitch type:

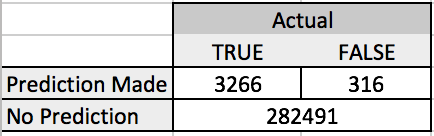


Once we created this data frame, we evaluated the accuracy of the model at different “prediction certainty” thresholds. For example, if we set the threshold = 0.8, we would collect the subset of records where the “prediction certainty” exceeded 0.8. We then calculated the percentage of records in this subset that were predicted correctly. We also counted the number of true positives (correct predictions), false positives (wrong predictions), and negatives (cases where no prediction was made). Example output from a random forests model is below. (Notice that we named the column “Precision” as opposed to “Accuracy”, because we are measuring the accuracy of *only those records that meet the 0.8 threshold*):



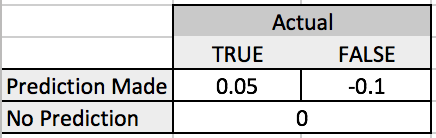
Essentially, the output tells us that for all records with “prediction certainty” above 0.8, the correct prediction was made 91% of the time. However, only 1.25% of the records in the validation set met this threshold, according to the “Cumulative Percent of Records” column above.

However, we still haven’t answered our initial question: will the model benefit or hurt an MLB team? To solve this problem, we used the Expected Value framework described in *Data Science for Business.[[18]](#footnote-18)* This framework entails gathering a confusion matrix and multiplying it cell-wise against a cost-benefit matrix.Given a threshold of 0.8, the confusion matrix is:



The cost-benefit matrix is more difficult to pin down. Let’s define the “net outcome” of a prediction as the change in batting average that occurs if a prediction is right (or wrong). If a prediction is correct, for example, the information might enable a .250 career hitter to perform at a .300 level. In this case, the “net outcome” would be 0.05. Conversely, if the prediction is incorrect, the .250 hitter might perform much worse, perhaps at a .150 level. The “net outcome” in this case would be -0.10.

We designated the “net outcome” of a prediction as 0.05 for a true positive, and -0.10 for a true negative. Our choices were somewhat arbitrary, but our basic intuition was that the loss from a false positive was far greater than the benefit of a true positive. We felt it would be safer to take a conservative/pessimistic view in our cost-benefit estimates. If we were to pursue this project in greater detail, we could try reaching out to current MLB players and ask for their opinion on the matter. In any event, this hypothesis yields the following cost-benefit matrix:



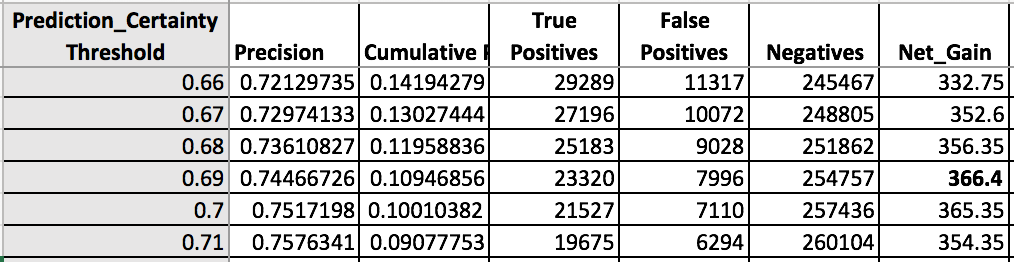
Using these two matrices, the “Expected Value” (or “Total Net Gain”) of making a prediction when the threshold is 0.8 is:

***Net Gain*** = True Positives (benefit of good prediction) + False Positives (cost of bad prediction)

3266(0.05) + 316(-0.10) = **131.7**

Because “Net Gain” is above 0, giving predictions to players when the prediction threshold is above 0.8 appears to add some value to an MLB team. One of the problems with “Net Gain”, however, is that the number itself does not convey significant meaning. Perhaps it would have been appropriate to divide Net Gain by 286,000 – the approximate number of records in the validation set. Net Gain divided by 286,000 could be construed as the “average gain in batting percentage per pitch”. In the example above, 131.7/286,000 = 0.00046, which is miniscule. For purposes of simplicity, let’s refrain from normalizing the values of Net Gain.

Our next question becomes: at what prediction threshold is “Net Gain” maximized? If we set our threshold too high, we will rarely issue any predictions, so the number of True Positives remains low. However, if we set our threshold too low, our precision will suffer. To find the optimal value of “Prediction Certainty Threshold”, we looked at a variety of thresholds from 0.4 to 1, using increments of 0.01. In the example below, “Net Gain” peaks at 366.4 when the threshold is 0.69:



We then used Net Gain as a metric to evaluate the performance of our models, in addition to looking at accuracy and log-loss. We adopted this metric relatively late in the process, and so we only tested a handful of models. Regardless, the results for Net Gain roughly mirrored our earlier results for accuracy.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model**  **Type** | **Parameters** | **Accuracy** | **Net\_Gain**  **Maximum** | **Threshold** (where max. occurs) | **Coverage**  (where max. occurs) |
| DT | MinSplit=300, MaxDepth=20, MinLeaf=5 | 0.4868 | 355.15 | 0.70 | 0.105 |
| DT | MinSplit=300, MaxDepth=20, MinLeaf=50 | 0.4850 | 366.40 | 0.69 | 0.109 |
| RF | N\_Est=100, MinSplit=300, MaxDepth=20 | 0.4951 | 360.60 | 0.64 | 0.096 |
| RF | N\_Est=200, MinSplit=300, MaxDepth=20 | 0.4951 | 362.25 | 0.64 | 0.096 |
| RF | N\_Est=100, MinSplit=100, MaxDepth=20 | 0.4995 | 467.10 | 0.64 | 0.113 |

According to the “Net Gain” metric, the final random forests model (with parameters n\_estimators = 100, min\_split\_size = 100, and max\_depth = 20) performed substantially better than the other models, with a Net Gain of 467.10. This result corroborated our earlier analysis, which found that this particular model had the highest accuracy (0.4995) and lowest log-loss (1.201). Generally speaking, if we were to give this model to an MLB team, we would likely advise them to only provide predictions to a batter if they were at least 64-70% sure of the next pitch type, given our cost-benefit estimates above. As illustrated in the appendix, the distribution of Net Gain plunges sharply downward as the probability threshold decreases. Therefore, we would probably recommend that MLB teams err on the side of caution when selecting a probability threshold, even if it means giving predictions for only 10% of all pitches (“coverage”). For a display of charts and graphs for the final RF model, please refer to the appendix.

Test Results

After testing a wide variety of models, we chose to apply a random forests algorithm with parameters n\_estimators = 100, min\_split\_size = 100, and max\_depth = 20. Our results were a mixed bag:

Test Results

|  |  |  |
| --- | --- | --- |
| **Accuracy** | **Log-Loss** | **Net Gain** |
| 0.4634 | 1.470 | 526.95 |

The maximum value for Net Gain increased to 526.95, corresponding to a probability certainty threshold of 0.60. While the result seems promising, it’s important to note that only 6.29% of records in our test set had a probability certainty over 0.6 (see appendix); because the model applies to fewer than 7% of total pitches, it would not be a very useful tool for MLB teams. Furthermore, if we were to normalize the Net Gain from our validation set (467.10) and the test set (526.95) by the number of records in the validation set (286,000) and the test set (739,000), respectively, the Net Gain for the validation set would look more impressive. Therefore, our test value for Net Gain is probably inferior to what we calculated earlier. Even more troubling, the accuracy plunged from 0.4995 to 0.4634, while log-loss increased from 1.201 to 1.470.

What accounts for the significant decrease in performance? It’s very possible that we over-fit the model in 2014 and 2015. When we employed random forests during the validation stage, we found that accuracy increased when we ignored the max\_features parameter. While this decision helped boost performance for the particular validation set, it may have damaged the RF model’s ability to generalize to new data, since the individual decision trees in the forest were likely heavily correlated.

Deployment

The model would be used primarily to help MLB teams instruct hitters in the batters’ box about what pitch type to expect.

* The idea would be for someone in the team’s analytics department to input the values of the 20-30 most important features in our model before every pitch. While our dataset has around 50 features, many of the variables provide little predictive power.
* For each pitch, if the analyst saw that the model’s “prediction certainty” exceeded a particular threshold, say 0.7, the analyst would relay the model’s output to the manager, who in turn would reach out to the third- or first-base coach, who in turn would give signals to the batter.
* The system would be infinitely easier to operate if first- and third-base coaches were allowed to have earpieces, but in the MLB, coaches on the field must rely on hand signals.
* Regardless, given that the time between pitches is usually 20-30 seconds, there should be enough time for the model’s output to be communicated to the batter.

When using the model, an MLB team may want to consider collecting data on the model’s performance.

By collecting this data, a team could more accurately estimate the cost of a false positive and the benefit of a true positive prediction. In our model, we assumed that the benefit was 0.05 and the cost was -0.10, but these values were essentially guesswork.

Future Considerations

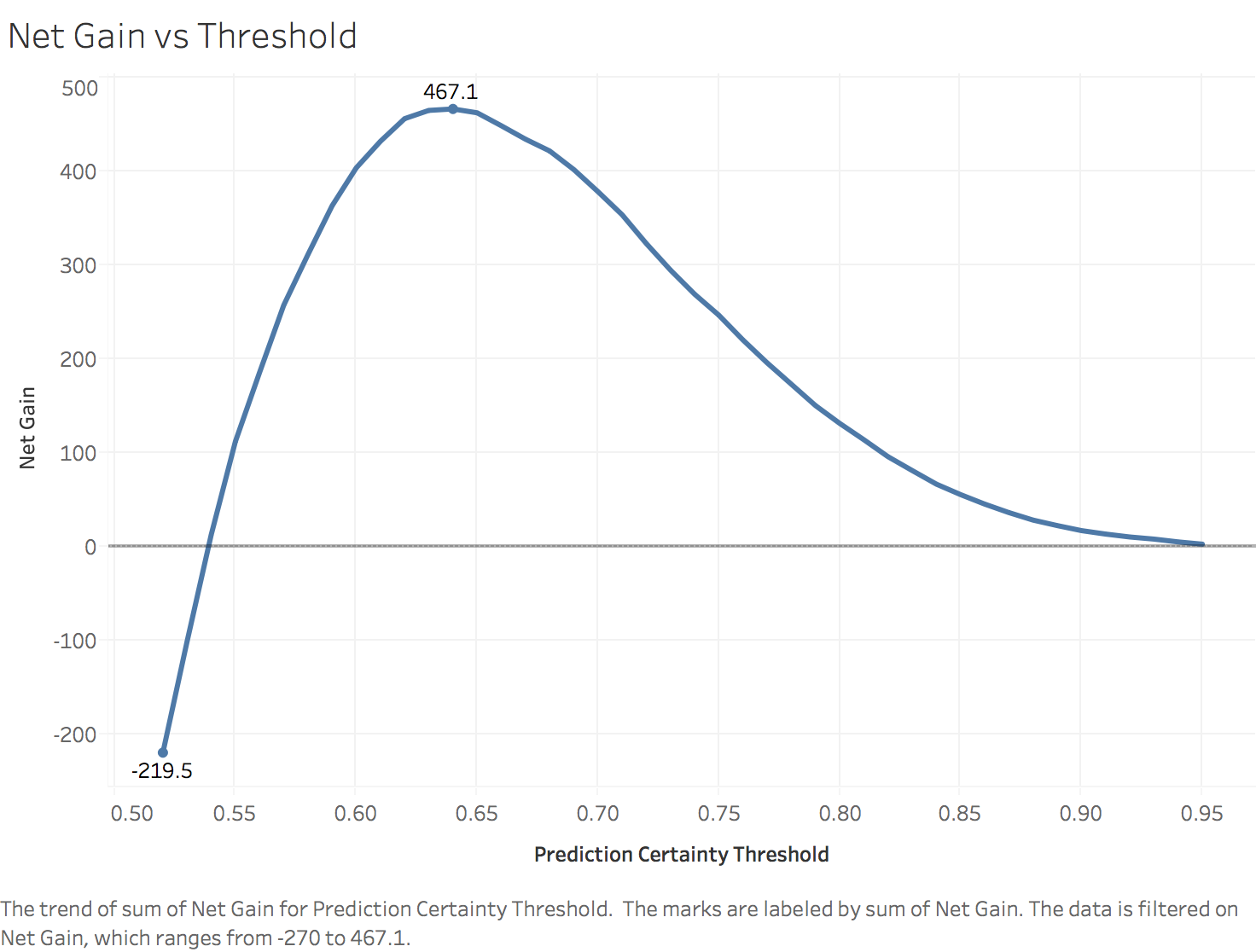
There are many ways in which we could have improved the model. First, we probably did not have enough features in our dataset. Our lack of meaningful features was probably exposed when we tried different options for the ‘max\_features’ parameter in random forests, and found that performance was optimized when we set no limit. One category of features we wanted to obtain was a hitter’s batting average against each of the 8 pitch types; however, we could not find any online data for this metric. Clearly, if the batter is a good fastball hitter, the pitcher is more likely to throw slower pitch types. In hindsight, we should have collected the *percentage of pitch types seen by the batter in the previous season*. This set of 8 features (one for each pitch class) would have likely improved our model significantly. To build these features, we could have used a similar procedure as the one we used to create the 8 features indicating the pitcher’s pitch type distribution from the prior season. Furthermore, there are a few situational batting statistics from Baseball-Reference.com that may have been helpful, such as “first pitch swinging percentage”, “swinging percentage with a 3-0 count”, and so on.

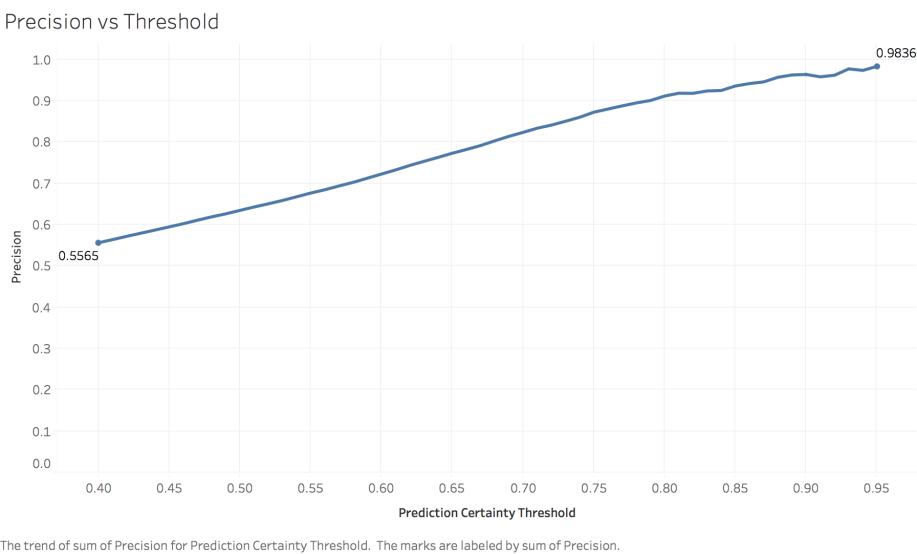
Aside from feature selection, we could have re-considered the sizes of our training, validation, and test sets. The test set (2016) covered more than 33% of our data, while our validation set (20% of 2014-2015) covered only about 13% of the total data. Furthermore, it probably would have made more intuitive sense to order the training and validation sets by time, such that all records in the training set occurred before records in the validation set. Moving forward, we could flesh out the “Net Gain” concept a bit more so that it could be easier to explain to MLB teams. Finally, we could have tried gradient boosted trees as another ensemble technique. There is no guarantee that our test results would have improved, but it would have been interesting to compare random forests and gradient boosted trees.

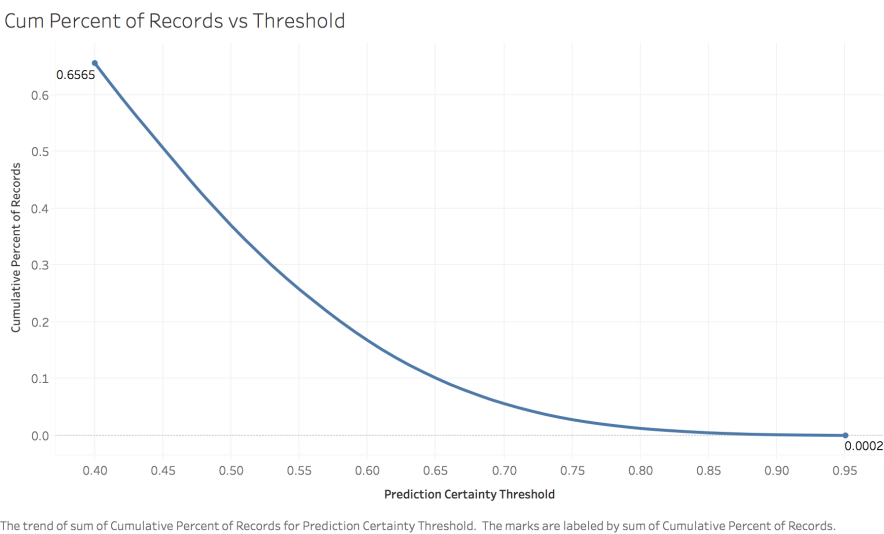
Appendix – Results for Net Gain using Random Forests Models

Validation Set (n\_estimators=100, min\_split\_size = 100, max\_depth = 50):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Prediction Certainty Threshold** | **Precision** | **Cumulative Percent of Records** | **True Positives** | **False Positives** | **Negatives** | **Net\_Gain** |
| 0.4 | 0.556467888 | 0.656465308 | 104503 | 83294 | 98276 | -3104.25 |
| 0.41 | 0.564052372 | 0.625011798 | 100852 | 77947 | 107274 | -2752.1 |
| 0.42 | 0.572036092 | 0.594295862 | 97253 | 72759 | 116061 | -2413.25 |
| 0.43 | 0.579664376 | 0.564506262 | 93610 | 67880 | 124583 | -2107.5 |
| 0.44 | 0.587233654 | 0.535800303 | 90010 | 63268 | 132795 | -1826.3 |
| 0.45 | 0.59490546 | 0.506926554 | 86272 | 58746 | 141055 | -1561 |
| 0.46 | 0.602624522 | 0.478419844 | 82477 | 54386 | 149210 | -1314.75 |
| 0.47 | 0.611095994 | 0.449616007 | 78601 | 50022 | 157450 | -1072.15 |
| 0.48 | 0.619178559 | 0.422056608 | 74759 | 45980 | 165334 | -860.05 |
| 0.49 | 0.626528036 | 0.39604926 | 70985 | 42314 | 172774 | -682.15 |
| 0.5 | 0.634850617 | 0.370076868 | 67211 | 38658 | 180204 | -505.25 |
| 0.51 | 0.643148281 | 0.345621572 | 63590 | 35283 | 187200 | -348.8 |
| 0.52 | 0.650803461 | 0.322386244 | 60021 | 32205 | 193847 | -219.45 |
| 0.53 | 0.65880773 | 0.29934667 | 56417 | 29218 | 200438 | -100.95 |
| 0.54 | 0.667760676 | 0.27798499 | 53103 | 26421 | 206549 | 13.05 |
| 0.55 | 0.676845578 | 0.257448274 | 49849 | 23800 | 212424 | 112.45 |
| 0.56 | 0.68487099 | 0.238169978 | 46663 | 21471 | 217939 | 186.05 |
| 0.57 | 0.694083303 | 0.219129383 | 43510 | 19177 | 223386 | 257.8 |
| 0.58 | 0.702810337 | 0.201004639 | 40413 | 17089 | 228571 | 311.75 |
| 0.59 | 0.712801978 | 0.183764284 | 37472 | 15098 | 233503 | 363.8 |
| 0.6 | 0.722895588 | 0.167562126 | 34652 | 13283 | 238138 | 404.3 |
| 0.61 | 0.732924392 | 0.152153471 | 31902 | 11625 | 242546 | 432.6 |
| 0.62 | 0.743776748 | 0.138038193 | 29371 | 10118 | 246584 | 456.75 |
| 0.63 | 0.753643089 | 0.124737392 | 26893 | 8791 | 250389 | 465.55 |
| **0.64** | 0.763233789 | 0.112722976 | 24612 | 7635 | 253826 | **467.1** |
| 0.65 | 0.773279912 | 0.101204937 | 22388 | 6564 | 257121 | 463 |
| 0.66 | 0.782353396 | 0.090487393 | 20252 | 5634 | 260187 | 449.2 |
| 0.67 | 0.791904268 | 0.080916409 | 18331 | 4817 | 262925 | 434.85 |
| 0.68 | 0.803260395 | 0.072048044 | 16556 | 4055 | 265462 | 422.3 |
| 0.69 | 0.814197735 | 0.063571186 | 14807 | 3379 | 267887 | 402.45 |
| 0.7 | 0.823774571 | 0.056195447 | 13243 | 2833 | 269997 | 378.85 |
| 0.71 | 0.83391175 | 0.049354535 | 11774 | 2345 | 271954 | 354.2 |
| 0.72 | 0.841422899 | 0.043139339 | 10384 | 1957 | 273732 | 323.5 |
| 0.73 | 0.850729517 | 0.037375076 | 9096 | 1596 | 275381 | 295.2 |
| 0.74 | 0.860424981 | 0.032407812 | 7977 | 1294 | 276802 | 269.45 |
| 0.75 | 0.872686343 | 0.027950908 | 6978 | 1018 | 278077 | 247.1 |
| 0.76 | 0.88030215 | 0.024063788 | 6060 | 824 | 279189 | 220.6 |
| 0.77 | 0.887986464 | 0.020659063 | 5248 | 662 | 280163 | 196.2 |
| 0.78 | 0.895100751 | 0.017694784 | 4531 | 531 | 281011 | 173.45 |
| 0.79 | 0.901050175 | 0.014978694 | 3861 | 424 | 281788 | 150.65 |
| 0.8 | 0.911781128 | 0.01252128 | 3266 | 316 | 282491 | 131.7 |
| 0.81 | 0.918623884 | 0.010567233 | 2777 | 246 | 283050 | 114.25 |
| 0.82 | 0.918335296 | 0.008903322 | 2339 | 208 | 283526 | 96.15 |
| 0.83 | 0.924098672 | 0.007368749 | 1948 | 160 | 283965 | 81.4 |
| 0.84 | 0.925217391 | 0.006029929 | 1596 | 129 | 284348 | 66.9 |
| 0.85 | 0.936185642 | 0.004820448 | 1291 | 88 | 284694 | 55.75 |
| 0.86 | 0.942028986 | 0.003859155 | 1040 | 64 | 284969 | 45.6 |
| 0.87 | 0.946100917 | 0.003048173 | 825 | 47 | 285201 | 36.55 |
| 0.88 | 0.957251908 | 0.002289625 | 627 | 28 | 285418 | 28.55 |
| 0.89 | 0.962818004 | 0.001786257 | 492 | 19 | 285562 | 22.7 |
| 0.9 | 0.964010283 | 0.001359793 | 375 | 14 | 285684 | 17.35 |
| 0.91 | 0.958333333 | 0.001090631 | 299 | 13 | 285761 | 13.65 |
| 0.92 | 0.962184874 | 0.000831955 | 229 | 9 | 285835 | 10.55 |
| 0.93 | 0.97752809 | 0.000622219 | 174 | 4 | 285895 | 8.3 |
| 0.94 | 0.973913043 | 0.000401995 | 112 | 3 | 285958 | 5.3 |
| 0.95 | 0.983606557 | 0.000213232 | 60 | 1 | 286012 | 2.9 |







Test Set (n\_estimators=100, min\_split\_size = 100, max\_depth = 50):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Prediction Certainty Threshold** | **Precision** | **Cumulative Percent of Records** | **True Positives** | **False Positives** | **Negatives** | **Net\_Gain** |
| 0.4 | 0.540276552 | 0.573187842 | 228886 | 194760 | 315459 | -8031.7 |
| 0.41 | 0.546719543 | 0.540418479 | 218374 | 181052 | 339679 | -7186.5 |
| 0.42 | 0.552846676 | 0.508010364 | 207579 | 167894 | 363632 | -6410.45 |
| 0.43 | 0.559307759 | 0.477133831 | 197241 | 155411 | 386453 | -5679.05 |
| 0.44 | 0.566589005 | 0.445990759 | 186767 | 142867 | 409471 | -4948.35 |
| 0.45 | 0.574634451 | 0.415280643 | 176376 | 130560 | 432169 | -4237.2 |
| 0.46 | 0.581830922 | 0.384232281 | 165233 | 118755 | 455117 | -3613.85 |
| 0.47 | 0.590811543 | 0.352424892 | 153894 | 106585 | 478626 | -2963.8 |
| 0.48 | 0.600669215 | 0.320649975 | 142355 | 94639 | 502111 | -2346.15 |
| 0.49 | 0.611267658 | 0.290010215 | 131024 | 83324 | 524757 | -1781.2 |
| 0.5 | 0.620821133 | 0.261524411 | 120001 | 73293 | 545811 | -1329.25 |
| 0.51 | 0.630895472 | 0.234631074 | 109408 | 64009 | 565688 | -930.5 |
| 0.52 | 0.640853213 | 0.20944656 | 99206 | 55597 | 584302 | -599.4 |
| 0.53 | 0.651562136 | 0.185868043 | 89509 | 47867 | 601729 | -311.25 |
| 0.54 | 0.663090217 | 0.163932053 | 80342 | 40821 | 617942 | -65 |
| 0.55 | 0.674179305 | 0.14371571 | 71612 | 34609 | 632884 | 119.7 |
| 0.56 | 0.686130356 | 0.124591229 | 63183 | 28903 | 647019 | 268.85 |
| 0.57 | 0.698216631 | 0.107124157 | 55282 | 23894 | 659929 | 374.7 |
| 0.58 | 0.711844239 | 0.090962718 | 47858 | 19373 | 671874 | 455.6 |
| 0.59 | 0.726511231 | 0.076077147 | 40851 | 15378 | 682876 | 504.75 |
| 0.6 | 0.742254067 | 0.062881458 | 34497 | 11979 | 692629 | **526.95** |
| 0.61 | 0.756562369 | 0.051543421 | 28822 | 9274 | 701009 | 513.7 |
| 0.62 | 0.771977933 | 0.041938561 | 23929 | 7068 | 708108 | 489.65 |
| 0.63 | 0.785305607 | 0.034288768 | 19902 | 5441 | 713762 | 451 |
| 0.64 | 0.798588963 | 0.027998728 | 16526 | 4168 | 718411 | 409.5 |
| 0.65 | 0.808317625 | 0.02296832 | 13722 | 3254 | 722129 | 360.7 |
| 0.66 | 0.819772977 | 0.018475048 | 11194 | 2461 | 725450 | 313.6 |
| 0.67 | 0.830621572 | 0.014801686 | 9087 | 1853 | 728165 | 269.05 |
| 0.68 | 0.837591241 | 0.011862996 | 7344 | 1424 | 730337 | 224.8 |
| 0.69 | 0.847616345 | 0.009535858 | 5974 | 1074 | 732057 | 191.3 |
| 0.7 | 0.85796949 | 0.007716089 | 4893 | 810 | 733402 | 163.65 |
| 0.71 | 0.863259065 | 0.006193978 | 3952 | 626 | 734527 | 135 |
| 0.72 | 0.874496103 | 0.005034467 | 3254 | 467 | 735384 | 116 |
| 0.73 | 0.886965044 | 0.004141495 | 2715 | 346 | 736044 | 101.15 |
| 0.74 | 0.897547468 | 0.003420353 | 2269 | 259 | 736577 | 87.55 |
| 0.75 | 0.909219191 | 0.002876452 | 1933 | 193 | 736979 | 77.35 |
| 0.76 | 0.914728682 | 0.002443496 | 1652 | 154 | 737299 | 67.2 |
| 0.77 | 0.918128655 | 0.002082248 | 1413 | 126 | 737566 | 58.05 |
| 0.78 | 0.924662966 | 0.001706118 | 1166 | 95 | 737844 | 48.8 |
| 0.79 | 0.936613056 | 0.001430108 | 990 | 67 | 738048 | 42.8 |
| 0.8 | 0.939038687 | 0.001154099 | 801 | 52 | 738252 | 34.85 |
| 0.81 | 0.941176471 | 0.000897031 | 624 | 39 | 738442 | 27.3 |
| 0.82 | 0.948207171 | 0.0006792 | 476 | 26 | 738603 | 21.2 |
| 0.83 | 0.948320413 | 0.000523606 | 367 | 20 | 738718 | 16.35 |
| 0.84 | 0.953736655 | 0.00038019 | 268 | 13 | 738824 | 12.1 |
| 0.85 | 0.961352657 | 0.000280068 | 199 | 8 | 738898 | 9.15 |
| 0.86 | 0.968553459 | 0.000215125 | 154 | 5 | 738946 | 7.2 |
| 0.87 | 0.974137931 | 0.000156947 | 113 | 3 | 738989 | 5.35 |
| 0.88 | 0.987804878 | 0.000110945 | 81 | 1 | 739023 | 3.95 |
| 0.89 | 1 | 7.31E-05 | 54 | 0 | 739051 | 2.7 |
| 0.9 | 1 | 3.52E-05 | 26 | 0 | 739079 | 1.3 |

Insert Ali’s new graphs her

1. <http://www.azquotes.com/author/15725-Ted_Williams> [↑](#footnote-ref-1)
2. <http://bleacherreport.com/articles/1648362-proof-that-the-steroid-era-power-surge-in-baseball-has-been-stopped> [↑](#footnote-ref-2)
3. <http://www.espn.com/mlb/topics/_/page/the-steroids-era> [↑](#footnote-ref-3)
4. <https://en.wikipedia.org/wiki/Walks_plus_hits_per_inning_pitched> [↑](#footnote-ref-4)
5. <http://www.fangraphs.com/library/misc/pitch-fx/> [↑](#footnote-ref-5)
6. <http://www.brooksbaseball.net/> [↑](#footnote-ref-6)
7. <http://www.baseballheatmaps.com/> [↑](#footnote-ref-7)
8. <http://cpsievert.github.io/pitchRx/> [↑](#footnote-ref-8)
9. <http://gd2.mlb.com/components/game/mlb/> [↑](#footnote-ref-9)
10. <http://www.beyondtheboxscore.com/2015/9/24/9374949/a-new-python-based-pitchf-x-parser-scraper> [↑](#footnote-ref-10)
11. <http://www.baseballprospectus.com/sortable/> [↑](#footnote-ref-11)
12. Pitcher Rates: http://www.baseballprospectus.com/sortable/index.php?cid=1928886 [↑](#footnote-ref-12)
13. Pitcher BIP Stats: http://www.baseballprospectus.com/sortable/index.php?cid=1819106 [↑](#footnote-ref-13)
14. Batter Standard: http://www.baseballprospectus.com/sortable/index.php?cid=1918875 [↑](#footnote-ref-14)
15. Batter Plate Discipline: http://www.baseballprospectus.com/sortable/index.php?cid=1858217 [↑](#footnote-ref-15)
16. When you do an outer merge, and there are two players with the same name, you get some weird results. Say you have player A (‘Tom Jones’) and player B (‘Tom Jones’), and two data frames X and Y. The merged dataset will have three records with the name ‘Tom Jones’. One record will have player A’s records from X and Y, a second record will have player B’s records from X and Y. The third record, however, will combine player B’s records from X and player A’s records from Y. [↑](#footnote-ref-16)
17. 'http://www.baseballprospectus.com/sortable/playerids/playerid\_list.csv.

    We then used the *pd.io.parsers.read\_csv* function to download it into iPython. [↑](#footnote-ref-17)
18. Foster Provost and Tom Fawcett, *Data Science for Business: What You Need to Know About Data Mining and Data Analytic Thinking,* Chapter 7. [↑](#footnote-ref-18)