Major Qualifying Project: Advanced Baseball Statistics

Matthew Boros, Elijah Ellis, Leah Mitchell
Advisors: Jon Abraham and Barry Posterro

December 13, 2019

Contents

Abstract

We developed an expansive database collected from multiple sources that contains the outcome of each at bat in every baseball game since 1921, and the Las Vegas betting odds on every game since 2006. With 13 million records, we created new statistics and develop enhanced prediction models. Our thinking has always been centered around probabilities and sustainable production on the baseball field. This sustainable production is what will lead to player's and team's future successes. As a result, we have tried to quantify that success and implemented it into our models. We have created different models, and have had varying success with predicting the outcomes of baseball games. Lastly, we compared our models to Las Vegas book odds to see if any teams represented an arbitrage opportunity.

Chapter 1

Background

This section is comprised of three subsections we give an overview of the history of baseball, important people who paved the way in the world of baseball sabermetrics. Then we will go into detail on the development and formulas on some of the statistics pivotal to our project.

1.1 The History of Baseball

The origins of baseball have been highly debated as it is unknown who invented baseball, where, and when. Many speculate that the now-American sport has its roots in England and other European countries. Early versions of the game were recorded being played as early as the 1791, with teams, or 'social clubs', beginning to form in 1845.

It was in these early days that the beginning of box scores were started to be written down. Henry Chadwick is claimed to be the first to keep records in 1959. He started with only runs, hits, put-outs (now strikes), assists, and errors. It was these 10 years later,in 1869, that Major League of Baseball, MLB, was formed.

For the purposes of this paper, we will classify the beginning modern baseball to be at the turn of the 20th century. Since then, baseball has seen many different eras, some defined by historic events, like wars and segregation, and some defined by major rule changes and developments within the league.

The first era of what we will define as 'modern baseball' was the Deadball Era, which lasted from about 1901 to 1920. This era was defined by the lack of power hitting due to the focus on pitching and defense. Also, the balls were not switched out after each pitch making the balls harder to hit when dirty. During this time the pitchers mound was raised to 15 inches in 1904. After an unfortunate death, it was mandated that only clean balls were used and the pitchers could no longer tamper with the ball by spitting on it.

Baseball took a turn in the 1920's with the emergence of Babe Ruth as a power hitter. AS Ruth joined the league, the leafue as a whole saw a sharp increase in runs scored from 1918 to 1921, increasing from 50 to 177 runs. Ruth was one of the first true power hitters and soon became a headliner of the era and baseball as a whole. Game scores sky rocketed to an average of 10 runs scored per game. Records for both batting and pitching were shattered. During this era mainly cork and rubber centered balls were used and the minimum home run distance was set to 250 feet. This era continued on until 1941, when WWII started to have its impact on baseball.

The 1940's saw many changes in baseball, with effects from WWII to the end of segregation and the Yankees emerging as a powerhouse. The WWII era (1941-1945) saw most of its better players,

like Ted Willliams, go to war. Players were either putting their careers on hold or ending them to enlist in the military. Unlike WWI where the season ended early in 1918, baseball continued strong through this time. Roosevelt wrote his "Green Light" letter to Commissioner Kenesaw M. Landis explaining the importance of baseball to the people during this time. Roosevelt wrote:

> *"I honestly feel that it would be best for the country to keep baseball going. There will be fewer people unemployed and everybody will work longer hours and harder than ever before. And that means that they ought to have a chance for recreation and for taking their minds off their work even more than before"*

Thus, baseball prospered through the tough times. With the loss of some of its star players, the league saw a lot of unusual wins and awards. Next in the 1940's, the Segregation Era that had been around for all of baseball's history finally ended in 1947 with the signing of Jackie Robinson to the Brooklyn Dodgers. Robinson went on to have an amazing career, with awards like Rookie of the Year ('47), six consecutive All Star awards ('49-'54), a championship with the Dodgers ('55), and an induction into the Hall of Fame ('62). Lastly, the 1940's began the Post-War/Yankees Era that lasted till the late 50's to the early 60's. The Yankees were a powerhouse during this time, winning 10 out of the 13 World Series.

From 1953 to 1961, many eastern based teams moved to the west of the country, categorizing the Westward Expansion in baseball. Very much like the American Westward Expansion, the availability of transportation, population growth, and market availability were all key factors to the growth of baseball in the western part of the country. The Boston Braves moved to Milwaukee('53), the Brooklyn Dodgers moved to LA ('58), and the New York Giants moved to San Francisco ('58).

When the 1960's rolled around, baseball fell into the second coming of the Deadball Era. The focus of the game play switched back to pitching, such that Carl Yastrzemski was able to win the season batting title with the lowest batting average of 0.301 and Bob Gibson led the league with his 1.12 ERA. At the end of the decade, there were multiple rule changes, including the pitching mound lowered 5 inches, the strike zone was reduced, and the anti-spit ball law reworked, all in 1969.

The next two 'eras' are different than earlier ones, since they are categorized mainly by major rule changes and continue till now. In 1973, the American League created the Designated Hitter. The DH for short, was a player that would bat instead of the team's pitcher, since pitchers tended to have weak batting skills. This rule made lineups in the American League much more formidable, because the worst hitter, usually the pitcher, was now replaced with one of the best hitters on the team who did not have to worry about taking fielding and could focus solely on hitting.

The second era was the beginning of player free agency in the Arbitration Era. Before this era, teams had complete control of the players, with the ability to trade players however they wanted. St. Louis' Curt Flood fueled this change with his 1970 lawsuit to Commissioner Kuhn and MLB, due to the ideology that a team should not own a player. Flood even said that it was like slavery. The case was brought to the Supreme Court and was ultimately lost by Flood. Even though the case was so controversial that it ended Flood's career, in 1975 the league had its first two free agents, Andy Messersmith (LA) and Dave McNally (Montreal). Today players reach free agency after 6 years in MLB or once a team decides to release the player. This allows players more freedom to where they play and the salary they are paid.

Starting in the 1980's, offense started to become powerful with MLB not understanding why. Players home run totals grew from an average of 50 to above 90. Players started to admit to using Performance Enhancing Drugs (PED's). For example, players like Mark McGwire and Sammy Sosa in 1998 were part of the battle for most home runs that season. MLB started their research in the minor leagues, with the intentions of collecting data with no punishment. The league found that The Bay Area Laboratory Co-Operative (BALCO) was responsible for most of the PED's players

were taking. The products were advertised as nutrition supplements and were often prescribed by trainors. By 2004, MLB was testing every player randomly throughout the year. In 2005, MLB negotiated drug regulation into player contracts.

During the Steroid Era, baseball also saw the Longball Era which lasted from 1994 to 2005. Many developments throughout this era favored batters, with changes in strength training, batting techniques, and bat styles. Parks often change layouts to favor the hitters, while pitchers started throwing less inside pitches and the number of relief pitchers increased.

The last named era is the Wild Card Era. Similar to the Designated Hitter and Free Agency Era, the Wild Card Era was the beginning of a new playoff structure. As an attempt to increase the number of teams in the playoffs, this rule allows the best team (and later fifth) not winning their division to join in postseason play. This rule was enacted in 1994, the same year as the Player Strike and cancellation of the championship due to labor and pay debates.

Throughout the last century, baseball has developed into the game we know and love today. With the evolution of players, rules, and game play techniques, there is a lot to study about baseball.

1.2 Key Historical Figures

The next section will detail some of the important names in baseball that have shaped the field of baseball, baseball statistics, and our project.

1.2.1 Jerome Holtzman

Jerome Holtzman started his career in 1943 with his hometown paper, the Chicago Daily, where he stayed for over 50 years as he became one of the most influential sports writers. After two years in the military Holtzman moved to writing for the Chicago Tribune. Throughout his writing career, Holtzman focused mainly on the Cubs and White Sox. He wrote the entry on baseball for the Encyclopedia Britannica. In 1999, Holtzman finished his career as a writer and became the official MLB historian.

Holtzman is mostly known for his creation of the Save statistic. He argued that the relief wins and loses statistics were inadequate. Jerome started the statistics in the early 1960's and throughout the decade the statistic began to take shape and was officially started to be recorded in 1969, the first new statistic since the 1920's. The rule continued to change and be modified through the coming years until the current rule first in 1975. Many of the saves recorded in earlier years would not have been recorded as saves by today's current standards. We will discuss today's standards later in the paper.

1.2.2 Bill James

Bill James is one of the most influential baseball statisticians. James got his start writing during his shifts as a security guard. His writings were different from other authors of the time. While other writers were focused on interviews and the most amazing games, James was focused on asking and answering questions with statistics. James struggled to get his works published as editors thought his complicated writing style wouldn't appeal to their audience. James decided to start publishing his work on his own and published his first ever *Baseball Abstract* in 1977. While the first edition only sold 75 copies, by his second edition he sold 250 copies. From there his popularity grew and he began to grow in support too, getting a job at Esquire to write an annual preview and publish his own works. In 1988 James stopped publishing his yearly abstract, but did not stop his work in the baseball field. He published works like *The Baseball Book, The Player Ratings Book, The Bill*

James Golf Mind and the still published today, *The Bill James Handbook*. James used to work for the Boston Red Sox as a Senior Advisor on Baseball Operations, where he was hired in 2002.

Throughout his career, James had many different ideas that shaped the way baseball statistics were thought about. Most notably his development of runs created, win shares, Pythagorean winning percentages, and the coining of the term 'sabermetric', for the Society for American Baseball Research, or SABR for short. It was because of James' work and discoveries that lead Billy Beane (famous from *Moneyball*) to use sabermetrics to build the Oakland Athletics team, which ultimately lead to the acceptance and use of sabermetrics throughout all of MLB.

1.2.3 Nate Silver

Nate Silver is a dual focused statistician who has gained huge popularity for his work in both baseball and political elections. In the world of politics, Silver is known for his scarily accurate predictions of the 2008 U.S. President Elections, where he predicted 49 out of 50 states correctly. In baseball, he is known for the development of the Player Empirical Comparison and Optimization Test Algorithm, or PECOTA for short, while working as a writer for Baseball Prospectus. He used this algorithm for predicting the future performance of hitters and pitchers.

Silver began his career in 2000 as an economic consultant, but knew his true passion was baseball statistics as he had begun his work on PECOTA on the side.. He moved to Baseball Prospectus in 2003, for which he worked as a writer and continued his development of PECOTA. In 2007, Silver began his work in political statistics by publishing articles under the pen name "Poblano" and it was under this name that he published his 2008 election predictions. Silver decided to begin his own blog and website under the name FiveThirtyEight, named after the number of electors in the electoral college. FiveThirtyEight skyrocketed with Silvers own popularity growth, so Silver decided to leave Baseball Prospectus. FiveThirtyEight is now owned by ABC news and covers everything from sports, to politics, to pop culture. Lastly, in 2012, Silver published his book *The Signal and The Noise* that details methods of mathematical model building using a variety of techniques and example problems from baseball to elections to climate change.

1.2.4 Joe Peta

Joe Peta originally worked on Wall Street as a trader for a long/short equity hedge fund, until he was struck by an ambulance and was left immobile and out of a job for months. With his new free time, Peta began exploring the intersection of the stock market, betting, and baseball. Peta published the book *Trading Bases* that detailed the development and results of his work.

Peta used his skills in market data to find trends within baseball sabermetrics and with his discoveries, he was able to make hundreds of thousands of dollars betting on sports games. He developed formulas that were able to predict team wins and losses throughout the season based on individual player performances. He used his own original ideas of assessing a team's true talent along with the runs Pythagorean winning percentage and advanced projection statistics like PECOTA. Using his model, he found teams that were over and under valued by the odds makers and he would bet to exploit that arbitrage.

Around the same time that Joe Peta published Trading Bases, the Houston Astros hired former Wall Street executive, Jeff Luhnow, to be the teams General Manager. A notable stock trader and sports better, Luhnow now focused his strength of using data to reinvent the Astros organization. Starting in 2013, the Astros began revamping their minor-league affiliates with data backed young talent. Four years later, the Astros won the World Series while having 16 of their 25 rostered players

coming from the 2013 rehaul of the organization. This idea of using analytics to build a strong foundation of young talent is now sweeping across MLB.

1.3 Explanation of Baseball Statistics

Here we will go into detail explaining a few baseball statistics. The statistics range from normal calculations to advanced sabermetrics created by the people we discussed in the previous section.

1.3.1 Save

The Save statistic was invented in 1969 with a goal of rewarding relief pitchers who enter the game under certain conditions and help earn their team a win. After 1969, when a pitcher came into a game with his team winning and he was able to hold the lead for the rest of the game, as long as he did not get credit for the win, we would be credited with a save. The pitcher could only get the save and be removed if he was replaced by a pinch hitter or pinch runner. Under these conditions, there could be more than one pitcher in a game who qualified for a save, but there could only be one save awarded per game. Therefore, it was up to the discretion of the official scorer to determine who was the "more effective pitcher." This was an extremely simple version of the save compared to what is has evolved into today.

After the 1973 season, baseball expanded on their previous definition and identified two specific circumstances where a relief pitcher could earn a save. The first is that if the pitcher either entered the game with the potential tying or winning run on base or at the plate, he needed to preserve the lead for his team. The second was that the pitcher needed to pitch at least three or more effective innings and keep the lead. The same rules applied that the pitcher had to finish the game unless he removed for a pinch hitter or pinch runner.

Shortly before the 1975 season, MLB updated the qualifications yet again. The pitcher needed to be the last pitcher in a game won by his team, but not the winning pitcher. Then, at the same time that pitcher needed to meet at least one of the next three qualifications:

1. He entered the game with a lead of no more than three runs and pitched for at least one inning
2. He entered the game, regardless of the score, with the potential tying run either on base, at bat, or on deck
3. He pitched for at least three innings (official scorers still have some discretion as to whether or not to award the save).

Any pitcher who enters a game with one of the circumstances listed above qualifies for a Save Opportunity. If the pitcher allows the opposing team to tie the game, the pitcher will then get what is called a blown save. A blown save is a negative statistic because it means when the pitcher came into the game his team was winning and then he allowed the opposing team to come back. The save percentage is the number of times the pitcher gets a save divided by the total number of save opportunities. The higher the save percentage, the more effective the relief pitcher is considered to be.

Since 1975, the save has become such an integral part of baseball that there now existed a specialized pitcher, the closer, who is usually the pitcher responsible for entering the game in save situations. Closers are commonly compared by the amount of saves they have in a season and their save percentage. Some baseball writers have been skeptical of the importance of closers and the save statistic because they usually only pitch one inning per appearance and come into the game when

their team is leading. The save also only applies to closers, so the statistic almost ignores middle relievers and cannot be used to quantify their performance. However, this statistic is continued to be used for comparing closers and their abilities to help get their team the win.

1.3.2 OBP,SLG,ISO

On base percentage (OBP) is used to measure how often a hitter reaches base per plate appearances. This statistic was invented in the 1940s by Branch Rickey and Allan Roth, however, it was not an official statistic until 1984. The formula is:

$$OBP = \frac{\text{Hits + Walks + Hit by Pitch}}{\text{Plate Apperances}}$$

OBP is considered to be one of the best basic evaluative tools because, while it is simple and easy to calculate, it shows exactly when hitters avoid making outs which is always their number one priority. Evidently, studies have found that OBP tends to correlate with runs scored because higher OBP means fewer outs created and more opportunities for the offense to score. For those reasons, OBP is a very traditional "go to" statistic for measuring a hitters performance and effectiveness.

Another common statistic for batters is called Slugging Percentage (SLG). Slugging percentage is the average number of bases a hitter records per at-bat. As opposed to OBP, SLG only includes hits with a double counting as one more base than a single, and a triple counting one more than a double, and so on. The statistic represents the amount of 'power' hitters possess because hitters who hit more extra base hits than singles will have a higher SLG. SLG was created to balance out the traditional batting average metric (AVG) where every hit counts the same. However, a flaw with SLG is that it assumes a double is worth exactly two times a single (and a triple is three times a single and so on). Later we will talk about more advanced sabermetrics that take into account exactly how much more valuable a double is than a single. Nonetheless, SLG is easy to calculate, understandable, and is still one of the most traditional hitting metrics out there to measure a hitters extra base hitting ability. A related statistic is Isolated Power (ISO). This is found by:

$$ISO = SLG - AVG$$

So unlike SLG, it measures a double as one, a triple as two, and a home run as three while not factoring in singles. So this only measures extra base hits. It also is a way to combine two extremely common hitting statistics in slugging percentage and average.

1.3.3 Earned Run Estimators

The goal of earned run estimators is to quantify the talent of pitchers. One of the most common baseball statistics for measuring a pictures success is Earned Run Average (ERA). However, there is a growing trend away from using ERA because people feel it does not accurately portray who pitched better and cannot predict who will pitch better in the future. The formula for ERA is the fraction of earned runs over innings pitched scaled up to nine innings:

$$ERA = \frac{9 \cdot \text{Earned Runs Allowed}}{\text{Innings Pitched}}$$

It is meant to only include earned runs because these are the runs which the pitcher is held accountable. However, this metric, because of how it is scored with earned runs, introduces a subjective concept that makes the ERA less reflective of true talent. Also, pitchers can allow different number of runs depending on how good the fielders are behind them. Therefore, that

introduces a bias into the pitchers statistic that is reliant on how good the fielders are and not the pitcher. That is where other run estimators come in like FIP and SIERA.

Fielding Independent Pitching (FIP) only cares about the outcomes the pitcher can control. The thinking is that pitchers can control walks, hit by pitch, strikeouts, and home runs. This ignores all balls hit in play because it was revealed by Voros McCracken that of balls hit in play, those that fell for hits and those that resulted in outs did not correlate from season to season. So that means pitchers have little control over balls hit in play as to whtether they are outs or hits. Therefore, a method like FIP is better for estimating their overall talent. If a player has a low earned run average, but high FIP, progression to his FIP number is likely because it is a better estimation of the pitchers talent. This is the calculation for FIP:

$$FIP = \frac{(13 \cdot HR) + (3 \cdot (BB + HBP)) - (s \cdot K)}{IP} + \text{constant}$$

The second main ERA estimator is Skill-Interactive Earned Run Average (SIERA). Like FIP, SIERA tries to eliminate factors that pitchers cannot control. However, they do take into account some balls that are hit in play which is different than FIP. The thinking is that certain balls in play like ground balls, fly-balls, and line drives, are somewhat controllable by pitchers. This added complexity is to better explain and represent what makes certain pitchers more successful. SIERA holds strikeouts in an even higher esteem than FIP because pitchers with high strikeout rates allow weaker contact. Because of that, they have lower home run rates and higher ground ball rates. SIERA also found that walks hurt all pitchers, but hurt pitchers more that allow a high rate of walks much more than pitchers who do not walk a lot of batters. This phenomenon is because pitchers with a low walk rate can manage baserunners better than their counterparts. For balls hit in play, things get more complicated. That's because ground balls translate to hits more than fly balls do, but fly balls lead to more extra base hits. SIERA is able to look at all of these scenarios and come up with a predicted runs number.

1.3.4 Probability Based Statistics

These statistics take nothing into account except the game state. The game state does not care about the talents of the players on the field, but it consdiers how many runners are on base, the amount of outs, the inning, and what team is home and away. Given a situation, there were some number of games in the history to baseball that had that exact same situation and there is a smaller number of times that the team batting went on the win the game. We can use these empirical probabilities for the development of a few different statistics.

Win Probability Added (WPA) is used to assess how much players increase or decrease their team's chances of winning that game. WPA does this by looking at the probability of a team winning from one event to the next. Therefore, big plays in closer games, where the probability before the big play is closer to 50%, can impact the game more so the WPA will be higher. WPA is done by taking the difference of the chances the team will win before an event, such as a plate appearance, and the chances the team will win after an event. This makes WPA a fun "story" statistic which people can use to look back at a game or season to see who had the biggest impact for their team's wins and loses. The statistic,Leverage Index (LI), further quantifies the importance of particular events using winning probability. It measures the importance of game states with a neutral situation being 1 and more pivotal situations being higher. Leverage index looks at the win probability of the current situation and the range of possibilities for win probability after the next event. LI is essentially the expected value of all of the possible out comes during one at bat.

1.3.5 wOBA

Weighted On-Base Average (wOBA) is a statistic that attempts to encapsulate the entirety of a hitters offensive value into one number. It was created by Tom Tango and was heavily featured in The Book. It has a similar concept to OBP, but uses a linear weighting technique to give all the outcomes varying levels of impact. The idea is to see just exactly how much value there is in hitting a triple. The value is measured in terms of runs so when a player hits a triple, how much closer did his team get to scoring a run. Common thinking would suggest the team is 75% closer, but because hitters tend to hit better when players on base, players can score off of sacrifice flys and errors, and pitchers need to change their style of pitching when runners are on base, Tom found that he could be more accurate than the previously assumed 75%. These percentages are known as linear weights and are used to determine how likely a walk or a hit will lead to a run. The linear weights are created by looking at historical trends and updated for the upcoming year. Using these weights, wOBA is able to take into account the different levels of importance of different events and, therefore, is considered one of the best measures of a players all around batting talent. The general range of wOBA is shown below.

wOBA Rules of Thumb	
Rating	wOBA
Excellent	.400
Great	.370
Above Average	.340
Average	.320
Below Average	.310
Poor	.300
Awful	.290

Figure 1.3.5 shows ratings of different weighted On Base Averages for players.

1.3.6 WAR

WAR is another advanced saber metric that has a goal to estimate a players overall talent on one statistic. WAR takes into account fielding and base running along with hitting. Additionally, pitchers have their own version of WAR. WAR stands for Wins Above Replacement. The goal of this statistic is say how many more wins does this player gets you compared to the next readily available replacement. The formula is very complex as it tries to summarize every impact a player has into wins. WAR is not meant to be used as a precise indicator, but an approximation of a player's performance to date. There are two different ways to calculate WAR. For position players, WAR looks at runs, base running runs, and fielding runs along with adjustments for position, their league, and 'replacement' runs or the amount of runs for an average player. Then it is divided by runs per win. A simple form of the equation is:

$$WAR = (\text{Batting Runs} + \text{Base Running Runs} + \text{Fielding Runs} + \text{Positional Adjustment} + \text{League Adjustment} + \text{Replacement Runs})/\text{Runs Per Win}$$

The theme of WAR is all centered around runs created. So WAR analyzes every aspect in which a position player can either gain runs for his team, batting and base running, or save runs for his team (fielding). The position adjustment exists because some positions have different expectations for their ability to field and hit. It is possible for players to have a negative WAR. In that case, the statistic thinks that that player is worse than a player the team could just pick up from the minors or an average free agent.

Pitching WAR is a little more confusing and complicated. It uses a type of FIP that is adjusted for the differences in every ballpark and scales the number depending on how many innings the pitcher has pitched. The FIP is then converted into runs, and then, from runs into wins. Which is different from hitters in their calculation for WAR because pitchers do not have full control of what happens when the ball is hit in play. Thus, that is the reason to use FIP because it only includes home runs, strikeouts, and walks. Similar to the hitters WAR, there is a replacement component to be able to have what the next readily available player could be. This is a simplified version of the formula:

$$X = \frac{\text{League “FIP” – “FIP”}}{\text{Pitcher Specific Runs Per Win}} + \text{Replacement Level}$$

$$WAR = [X \cdot \frac{IP}{9}] \cdot \text{Leverage Multiplier for Relievers}$$

WAR is able to be compared from player to player regardless of position. However, it is best used to compare people of the same position across MLB. It is often used by writers and general managers for roster decisions such as trades and free agent signings. Although, it is important to remember that WAR can be used to separate classes of players, but it isn't reliable to distinguish players within their classes. For instance, if two players have a 5.5 WAR and a 4.3 WAR, then the player with the 5.5 can be said to be more effective. If the two players had a 5.5 and 5.2 WAR, though, then we could not say one is better than the other with certainty. We would need to look at other statistics and determine that from them.

1.3.7 Projection Systems

Since the mid- 2000s there has been a heavy emphasis on predicting outcomes in baseball by looking at the past. The projections they produce are true talent estimators which means that their stat projections are estimating what statistics players and teams should get based on their skill set, but cannot take into account luck or things that happen by chance. Every projection formula uses historical data of the players, and then each one has their own technique for bringing that historical data forward to the next year.

PECOTA sands for Player Empirical Comparison and Optimization Test Algorithm and is published by Baseball Prospectus. Founded in 2003 by Nate Silver, it functions is a projection tool that predicts player and team performance. PECOTA uses and combines several different methods when trying to predict outcomes. One method that they are especially proud of is their comparable players component. They use previous year's time series data and match current players with historical players who have similar curves and characteristics. This idea allows Baseball Prospectus to see what direction the current player will go in based on age and where they are compared to their prime. They also factor in play time, injury history, ballpark, and quality of competition. For younger players, they even use Minor League statistics. All of these aspects come together for individual players and then are combined into total team numbers. From there, the team number of runs scored and allowed are converted into predicted wins based on the team's talent levels.

The sZymborski Projection System (ZiPS) is another extremely common tool used to predict baseball statistics. It was created by Dan Szumborski while he was at ESPN and Baseball Think Factory. ZiPS takes historical data from individual players and assigns growth and decline curves based on various player types to try to find trends. For players age 24-38, they use data from the last four years and weight the recent years higher. For players younger or older than that range, they only use the previous three years. The weights they use depend on the age curve assigned to that player and whether or not they think he will improve or decline going into next year. Additionally, they also factor in ball velocities, injury data, and play-by-play data into their calculations.

Steamer is another projection system that was created by Jared Cross, a former high school science teacher in Brooklyn. He had the help of two former students named Dash Davidson and Peter Rosenbloom. "Steamers" is the nickname of St. Ann's High School where Cross worked. Similar to PECOTA and ZiPS, Steamer uses past performance with aging trends for their projections for players. However, where Steamer is different is that they have separate regressions for each individual statistic. Essentially, this system is made up of several different smaller projections. To help the pitchers, they use pitch tracking data and update the prediction daily during the season. They then combine all of these projections and produce their own team statistics based on the players each team has.

Chapter 2

Aggregated Baseball Database

To conduct our research, we needed to develop a database that aggregates baseball data from several sources. We required data about each at-bat in every major league baseball game on record, which is an extremely large amount of records. We also needed data about the rosters for every baseball game, which was difficult to find. Later on in our project, we added data on the betting odds for recent games, and up-to-date injury reports.

Our team used Python, SQLite, and several Python packages to create this database. The most prevalent package we used is SQL Alchemy, which enabled us to write queries in ordinary Python instead of having strings of foreign SQL code in our files. Our team also utilized Pandas for data manipulation, LXML and BeautifulSoup for HTML parsing, Matplotlib and Seaborn for graphing, and NumPy, SciPy, StatsModels, and ScikitLearn for calculations.

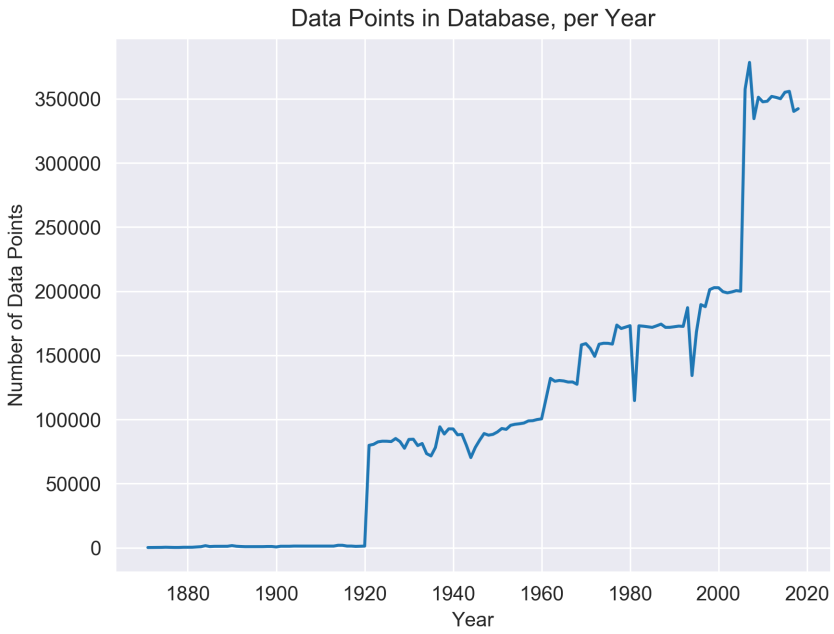

Figure 2 shows the number of data points in the database, split by year.

2.1 Data Sources

In the following section, we will break down the data sources we pulled from to create the aggregated database.

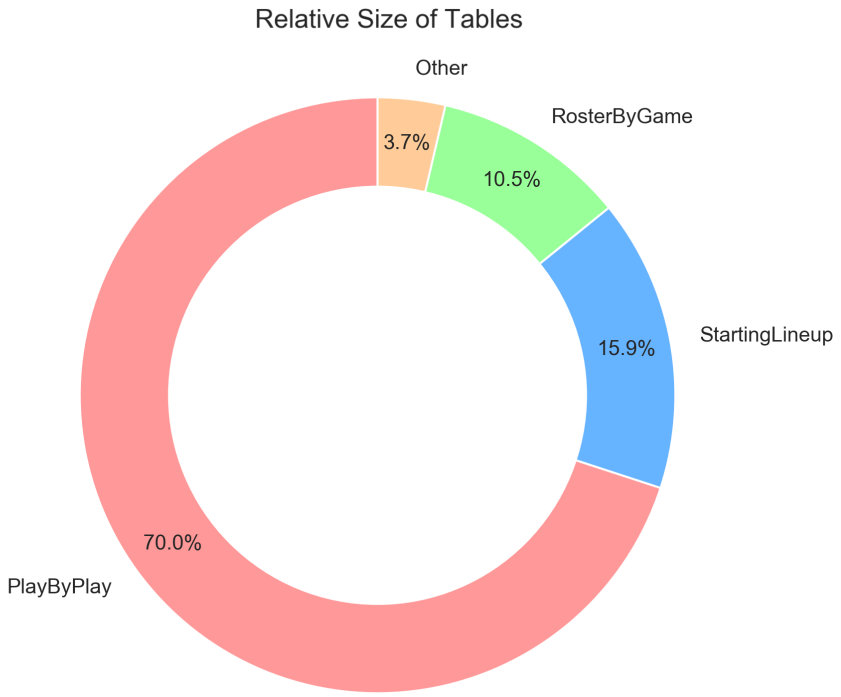

Figure 2.1 displays of the size of the largest tables in the database, by number of rows.

2.1.1 Retrosheet

Retrosheet was our primary source of data for general baseball game records. The Retrosheet records are extensive, containing data on the outcome of games, but also information on each specific at-bat in every MLB game since 1921. Retrosheet also provides us with seasonal team information, since team names have changed, or teams have been relocated to different cities.

The database contains several tables derived from the Retrosheet data. For example, a table for the starting lineup in each game has been created by analyzing the at-bat records from Retrosheet. We have also derived several tables worth of statistics for each player and team in every season of baseball available. Our team has purposely calculated these statistics ourselves instead of relying on an outside source, so that we can implement our custom methods of calculating statistics more easily.

With Retrosheet data we have compiled a list of game states, the different combinations of possible situations in a game, and their historical probability of winning. For example, the first row of the Excel document indicates that at the top of the first inning, with 0 outs and no one on base, and the visiting team winning by at least 5, there is a 13% chance of the home team winning, with a sample size of 54. We use these game state probabilities later in our calculations for garbage time.

2.1.2 MLB.com

MLB.com provides player rosters for each game since 1900. It was necessary to include this data in our research since Retrosheet does not include players who do not often participate in games in their database. MLB.com allows us to see the entire roster for each game, instead of just the players who participated in the game. Our team wrote scripts to scrape the MLB website for rosters. Since MLB game IDs and Retrosheet game IDs are different, we have written code to deduce which Retrosheet game ID corresponds with a given MLB game ID.

2.1.3 PECOTA

PECOTA, created by Nate Silver, is a series of annual predictions of what statistics will look like for most baseball players. Our team used PECOTA in several different models, as it provides a diverse range of predictions.

2.1.4 CBS Sports

To create pre-season predictions, we needed access to MLB injury reports. Our team developed code to scrape the CBS Sports website for injury reports and the expected date of return of the injured players. The code then analyzes the team schedules to see how many games a player should participate in given their expected return date from injury. This data is used to help calculate the preseason predictions for each teams win percentage outcome.

2.2 Table Structure

Each baseball player, team, and game have unique identifiers in the database. We chose to use the same identifiers used by Retrosheet, since the majority of our data is from Retrosheet.org. The following is an explanation of the purpose of each table in the database.

2.2.1 Game Logs

Each Major League Baseball game has an entry in the game log table. The table provides information on general game data, such as the date it occurred, the teams that participated, and the final score. Other tables in the database often reference the game identifier column in this table, to indicate a relationship with a specific baseball game. This table contains data since 1871.

2.2.2 Play-by-Play

The play-by-play table contains information on each plate appearance in games since 1921. This is 13,088,915 plate appearances in total. To indicate which plate appearance each record is, the table references a game ID in the game log table, and also a play index. The play index indicates what number the play is in the game, starting from zero. The first plate appearance in the game has a play index of zero, the second appearance has a play index of one, and so on.

The play-by-play table is incredibly detailed. It indicates what inning it is, how many outs, balls, and strikes there were, the current score of the game, who the batter is, and who all of the players in the field are. It provides dozens of columns containing extra information, such as who the batter on deck is, or the lineup position of the runner on second.

In addition to the data provided by Retrosheet, our group has used Chadwick Baseball Tools to extend the play-by-play table, adding addition columns containing useful information. The software provided by Chadwick has helped us avoid calculating information derived from Retrosheet ourselves.

2.2.3 Starting Lineups

Derived directly from the play-by-play table, the starting lineup table contains data on the starting lineup of each team during a particular game. It is convenient to store this data in a separate table instead of recalculating it each time, since we access starting lineup data often. The table has fields for the game ID, the team that the batter played for, the batter's position in the starting lineup, and the batter ID.

2.2.4 Team Schedules

The team schedule table contains data on scheduled games that have happened and are scheduled to happen. This table is similar to the game log table, except the schedules are released before the games occur.

2.2.5 General Team Information

The general team information table contains data on where a particular team is from, what league they are in, and what their full team name is.

2.2.6 Player - Game Participation

We found that our analysis requires obtaining the IDs of every player in a particular game repeatedly. The player-game participation table lists every player that participated in every game we have in the database. This allows us to query for player IDs more easily.

2.2.7 Roster by Game

The roster by game table provides a listing of every player that was on each teams roster for a particular game. These rosters are used in our preseason prediction model.

2.2.8 Seasonal Rosters

Retrosheet provides a list of players that were on a particular team for each season. This data is stored in the seasonal rosters table.

2.2.9 General Team Statistics

The general team statistics table contains data on the outcome of a teams season. It includes data on the number of wins and losses, total number of runs scored for and against the team, number of bases scored for and against the team, and statistics about runs and bases that exclude garbage time.

2.2.10 Player and Team Specific Statistics Tables

Our group has developed eight tables to hold data on the performance of every player and every team for each baseball season. These tables are derived from Retrosheet play-by-play data. The tables are as follows:

- Team Batting Statistics
- Team Pitching Statistics
- Player Batting Statistics
- Player Pitching Statistics
- Non-Garbage Team Batting Statistics
- Non-Garbage Team Pitching Statistics
- Non-Garbage Player Batting Statistics
- Non-Garbage Player Pitching Statistics

A non-garbage table indicates that in each statistic, any plays that are considered garbage time are excluded from the calculation. Each of these eight tables contains the following fields:

- Plate appearances
- At-bats
- Hits
- Singles, doubles, triples, and home runs
- Walks
- Number of hit by pitch
- Sacrifice flies
- On base percentage
- Slugging
- Batting average

We chose to store each of these tables separately to allow drop-in replacements of normal baseball statistics with non-garbage baseball statistics. For example, a block of code might query Player Batting Statistics for a player's performance, execute a calculation in an attempt to predict the player's performance next year, and then print the results. If the normal baseball statistics and non-garbage baseball statistics were stored in the same table, and the programmer wanted to use non-garbage statistics in the calculation instead of normal statistics, they would have to rename each column name in the SQL query. Since the tables have the same fields and are stored separately, the programmer can replace the table name "PlayerBattingStats" with "NonGbgPlayerBattingStats", and the program would then query for non-garbage statistics, since the columns have the same name.

This data organization allowed us to write functions with more dynamic capabilities. We have structured many of our functions to add the ability to exclude garbage time as keyword argument. Because of this, comparing models that use normal statistics to models that use non-garbage statistics has become much smoother.

2.2.11 PECOTA Batting and Pitching

Our database contains two tables of PECOTA batting predictions and pitching predictions for most players. These tables contain data on a player's projected batting average, slugging average, WAR, and other statistics.

2.2.12 Game State Counts by Year

The game state counts by year table contains data on the number of wins and the number of losses in a particular baseball scenario, split by year. This table is useful to help us calculate the probability of winning in scenarios, while being able to choose which years we take data from.

2.2.13 Game State Counts

The game state counts table contains aggregated information from the game state counts by year table. The total number of wins and losses for every available scenario is computed using all years past 1973. We have used this table to estimate the probability of winning in a given situation.

2.3 Conclusion

We have successfully created an aggregated database from several sources to fit the needs of our project. The database is almost 11 gigabytes in size and contains 18.7 million records of data, and over 12 thousand lines of code. The vast amount of information made available to us through this software has been very useful in our analysis, and has allowed us to do research very quickly.

Chapter 3

Cluster Luck

Cluster luck is a baseball phenomenon defined by Joe Peta in his book Trading Bases. He explains how cluster luck allows teams to appear better or worse than they are. Hits and offensive success can be clustered which means teams can score more runs than they really deserve. For example, let's look at this game from June 28, 2015. The Cleveland Indians were visiting the Baltimore Orioles for the second game of a double header. Both teams ended the game with the same number of hits, but the Orioles ended up winning 8-0.

Team	Runs	Hits
Orioles	8	8
Indians	0	8

Table 3 shows the score of the June 28^{th}, 2015 Orioles-Indians game.

We could say that the Orioles were 'lucky' to win this game by as much as they did because their hits were clustered in just two different innings. A series of consecutive hits and home runs lead to 6 runs being scored in the 3rd inning. People may look at the scoreboard and think because the Orioles won that they are better than the Indians. However, it can be said that the Indains's offensive performance is just as sustainable as the Orioles. So while public opinion may have the Indians being really bad, they could be just as good as the Orioles going forward.

So now if cluster luck is expanded from a single game to a full season teams can accumulate wins and runs that contrast with their overall hitting statistics. Over the course of a season, teams can get lucky or unlucky and eventually this can lead to more or less wins than they deserve. In order to capture a team's true runs deserved, we looked at different batting statistics that encompass every aspect of hitting a team needs to be successful. They can be broken down into the ability to get on base and hitting for extra bases. We look at statistics like slugging, batting average, and on base percentage.

The 2014 Los Angeles Angels, who had a league best 98 wins, are an example of a team that was lucky, they scored excess runs compared to their true talent.

2014	Wins	Runs	BA	OBP	SLG
The Angels	98	773	0.259	0.322	0.406
League Average	81	659	0.251	0.314	0.386
League Rank	1^{st}	1^{st}	5^{th}	7^{th}	6^{th}

Table 3 shows the Angles record for various team statistics for 2014.

As the table above shows, the Angels had a very potent offense in 2014 as they finished first in runs. However, there might be evidence that they did not "deserve" all of these runs and they might have benefited from some cluster luck. Looking at the individual statistics, we can see the Angels were higher than the league average, about 6^{th} place, in every case. Then, they were 1^{st} in runs which is much higher than the other categories. Judging the other statistics, it would make more sense that they would have finished 6^{th} place in runs. In 2014, 6^{th} place in runs would have put them around 710 runs for the season. So based off of their hitting metrics, cluster luck lead to the Angels scoring an extra 63 runs over the season. Using the rate of 9 runs per win (which teams had from 2013-2018), we can estimate that the Angels benefited from luck that played a role in winning seven extra games. However, Las Vegas odds may not reflect this luckiness because they may weigh the number of wins from last year more than they should in creating their odds the following year. We can assume the Angels success is not as repeatable as Vegas will anticipate, which means Vegas might bet too high on the Angels. We needed to find a way to better quantify this cluster luck and determine how we can use it to exploit this finding. This way we will be able to be more accurate in finding a team's true talent which means we could predict who wins more games.

3.1 Quantifying Cluster Luck

Our base regression model was

$$\text{runs} = a \cdot BA + b \cdot OBP + c \cdot SLG$$

We choose these three statistics because they were similar to the ones Joe Peta discussed in his book. Additionally, we felt that they encompassed the true talent of offense because they take into account how often the team hits the ball, when they get on base, and how many bases they get in one hit. However, we wanted to test the theoretical side to test the significance of these different variables. Here are the results from running various regressions using the statistics Batting Average (BA; $\frac{\text{Hits}}{\text{At Bats}}$), On Base Percentage (OBP; $\frac{\text{Times on Base}}{\text{At Bats}}$), Slugging Percentage (SLG; $\frac{\text{Bases}}{\text{At Bats}}$), Isolated Hitting (ISO; $\frac{\text{Extra Bases}}{\text{At Bats}}$), and weighted On Base Percentage (wOBA). wOBA weighs each possible outcome for the batter in terms of runs produced. Then combines that into a per at bat statistic.

Runs Regression (2008-2018)			
R1	Coefficient	T-Stat	P-Value
AVG	-1478.56	-4.655	$4.71 \cdot 10^{-6}$
SLG	2027.55	17.13	$2.22 \cdot 10^{-47}$
OBP	821.39	2.98	0.0030
R2			
SLG	526.30	1.62	0.11
OBP	838.13	3.04	0.003
ISO	1504.28	4.73	$3.37 \cdot 10^{-6}$
R3			
wOBA	2241.92	280.70	0
R4			
AVG	2631.10	0.63	0.53
SLG	-1397.92	-0.33	0.74
OBP	2078.99	3.74	$2.15 \cdot 10^{-4}$
ISO	4037.57	0.96	0.34
wOBA	-2114.51	-2.57	0.01
R5			
OBP	1276.56	23.27	$1.99 \cdot 10^{-71}$
ISO	1986.16	17.21	$9.69 \cdot 10^{-48}$

Table 3.1: Details the different regressions run using various baseball statistics, including coefficinets, t-statistics, and p-values.

Figure 3.1 shows the different regressions (labeled R1-R5) that we studied to predict runs. We used overall team statistics from 2008 to 2018. The second group is a smaller sub-section of the entire sample to ensure our results are consistent. Under each regression are the offensive statistics that we used for that trial. In R1, every statistic is significant based on the t-statistic and the p-value, but batting average is negatively correlated while the other two were positive. Intuitively, it does not make sense for average to be negatively correlated with runs since they should increase together. We hypothesize the negative result comes from the three statistics being too correlated with each other.

Then, when we look at the subgroup of the data may we see it may be a bit too robust and the OBP falls out of significance in the regression. There are similar problems with R2 and R4. R3 uses just wOBA which is the most sophisticated of the statistics investigated. It has an extremely low p-value for both the entire data set and the sub sections, but due to the sophisticated nature of wOBA it is too difficult to predict going in the year so we cannot use it to predict runs for the next year. We also cannot only use one variable in our model because it would be too reliant on one statistic. That leaves R5 which gives us the best result because there are only two variables and they are both positively correlated in both scenarios in each timeframe. ISO is a combination of BA and SLG so it condenses to one variable from our original base regression. Also, ISO and OBP are not as correlated because they measure different aspects of offensive talent. ISO shows the amount of extra base hits and then OBP shows how often players get on base which are two pivotal aspects of hitting when it comes to scoring runs. That is why we believe these regression results make sense and why these two statistics could identify true hitting talent.

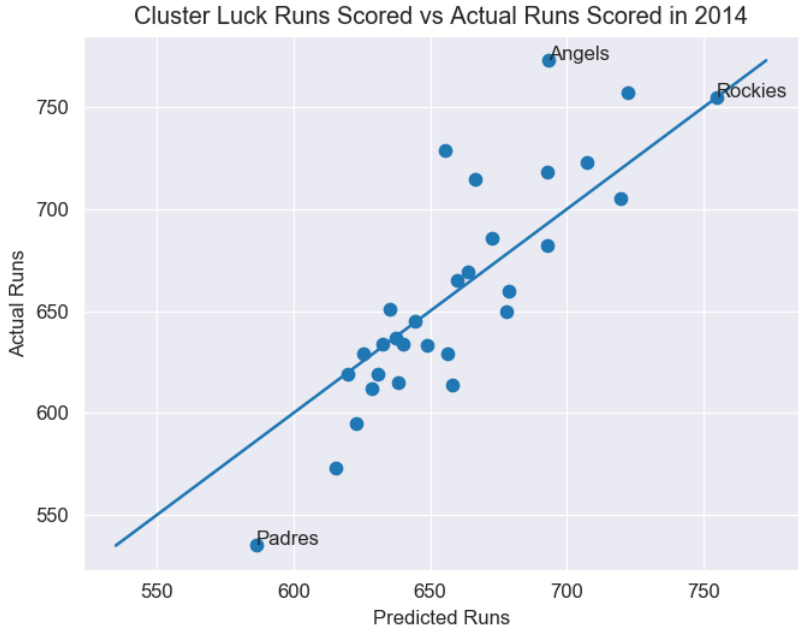

Figure 3.1 shows the predicted runs using cluster luck regression versus actual runs during the 2014 season.

Returning to the Angels example, we used our baseline regression model to quantify every teams' cluster luck runs. We used the model that included BA, OBP, and SLG using statistics from 2010-2018 from every team we ran a regression. With the given coefficients and the Angels statistics from 2014, we would have predicted them to score 699 runs which is 74 runs less than they actually scored. After we adjusted every team's runs scored for 2014, we found the Angles would have ranked 6^{th} overall, which makes sense when compared to the statistics we used with the Angels ranking 5^{th}, 6^{th}, and 7^{th} overall.

The Angels pitching allowed 630 runs. Based on their performance they deserved to only allow 594 run which means they actually got slightly unlucky by 36 runs. Which, using the Bill James Pythagorean, leads to runs model formula to a .5739 win-percentage or 92.9 wins. Which means when we account for their advantageous cluster luck offensively and their pitching 'unluckiness' we find that based on their actual sustainable success they should have about 93 games instead of the 98 that they won. That presents an opportunity where we have a chance to catch Las Vegas overrating the Angels. We may have a different winning percentage created from their cluster luck runs while most people rate the Angels on their normal runs scored and allowed.

3.2 Circumventing Cluster Luck with Total Bases

Above, we explain why our accumulated runs measurement needs to be adjusted because of the possibility of clustered hits. However, we wondered if there was a way to measure the true talent of a team without needing to run a regression. And therefore, making the prediction be more accurate because of the ability to use the actual statistics rather than the regression results. The statistic we chose to quantify this is Total Bases. This is simply a measure of the amount bases a team obtains

with every hit (for example, a single being worth 1 and a home run being worth 4). If we use a measurement that strictly relies on total bases, cluster luck is innately taken into account. This is because the total amount of bases a team hits, is a way to measure the amount of runs a team deserves. The amount of bases a team earns factors in the amount the team gets on base and the amount of bases they obtain with every hit. This is similar thinking to how we chose our statistics for the runs regression. Earning runs usually relies on other players performing well earlier in the inning, while earning bases has no such condition. For this reason, our group chose to pursue the use of total bases to circumvent cluster luck.

Our total bases model is simpler than our runs model, since we do not need to regress batting statistics with total bases to adjust bases for cluster luck. We add up the number of bases a team earned and bases the team gave up, and plug the totals into our Pythagorean runs formula.

$$\text{Expected Wins Bases} = \frac{\text{Bases For}^m}{\text{Bases For}^m + \text{Bases Against}^m}$$

where m is the exponent. This provides a cluster luck adjusted estimate of the number of runs a team truly deserved in a given season. We optimized the exponent in the Pythagorean formula to minimize the residuals between the actual wins teams and the expected wins (bases) each team had for seasons 2015 to 2018. The optimal exponent was 2.52. Below is a graph comparing our bases model with cluster luck adjusted runs to calculate each teams expected wins in 2018. The total bases model has a mean absolute error of 4.35 wins, while the runs model model has a mean absolute error of 5.69 wins.

The total bases model is highly correlated with actual wins, but still has enough of a difference from the actual wins that it provides a meaningful adjustment to remove cluster luck. Going forward, we will consider our total bases model in addition to our cluster luck adjusted runs model when attempting to predict seasonal outcomes.

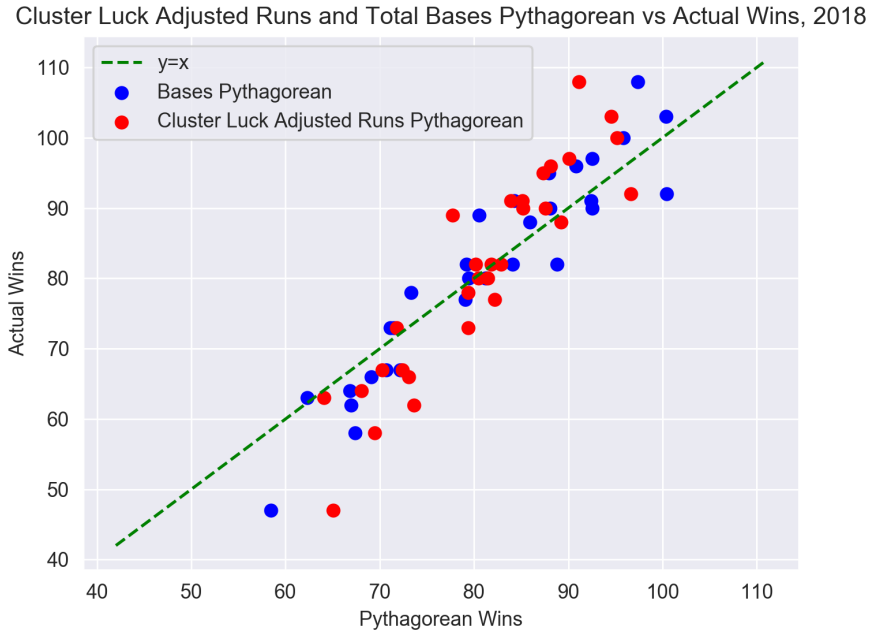

Figure 3.2 shows the cluster luck adjusted Pythagorean for both runs and bases versus the actual team wins for 2018.

3.3 Conclusion

Adjusting for cluster luck is a staple of how we evaluate teams. These evaluation methods will be used in a predictive model that determines a given team's winning percentage. The goal is that by taking cluster luck into account, the winning percentage will be more accurate than using traditional runs scored. We will continue to utilize our two methods of quantifying cluster luck and measuring them against each other in different situations to assess which is better.

Chapter 4

Garbage Time

Let's go back to Oakland Coliseum on September 20, 2018, a rivalry game where the Los Angeles Angels played the Oakland A's. In the top of the third inning the Angels took the lead 1-0. However, the A's were not phased in the slightest and went on to score 12 runs in the next two innings. Fast forwarding to the 6^{th} inning, there was more of the same as the score was 12-2 and the A's seemed dominant. Imagine the Angels players, away from home, playing only one game out of 162, and now severely behind. Obviously, the Angels players might not be trying their hardest in this game. Maybe they are already thinking about other things and are no longer focused on the outcome of the game. Additionally, the teams may be switching up players to give them an opportunity and resting resting their better players. Regardless, are the statistics during this time worth the same as close games? We do not feel like these count the same. Going back to the previous example, In the bottom of the 6^{th} inning the A's went on to score 6 more runs and then 3 more in the 7^{th}. The game finished 21-3, but did those 9 runs really impact the game at all? We believe that the Angel's pitcher was trying less, which lead to the four runs that were scored by home runs. We would not consider that success, or the other team's downfall, repeatable. Therefore, we would want to not include any statistics from garbage time in our model. Knowing this, we need to determine the specifics of garbage time.

In the example above, the A's had a 98.34% chance of winning by the end of the 4^{th} inning after they went up 12-1. That percentage of winning is what we think is the most important aspect of specifying garbage time. It is the best measure of when games are seemingly over and players will most likely give up. Obviously, percent chance of winning is one of the stipulations for our garbage time definition. However, we cannot just have a percent chance as the definition because early in games, even if there are big leads, players may still have hope their team can come back because there will be enough time to come back later in the game. Therefore, having a minimum inning requirement for garbage time is necessary. Another consideration is the size of the lead because the games that are in the 9^{th} inning may have higher winning percentages because the team is only a few outs away from winning. Therefore, we should include a lead requirement for garbage time so these late inning at bats are still included in non-garbage time statistics. Do to the nature of calculating percent chance of winning, we need that situation to have happened enough times to make sure the empirical probability is representative of the situation. With all of those considerations, we developed the following definition for garbage time:

A game is considered to be in garbage time if all of the following are true:

1. One team has more than a 97% chance of winning the game
2. The situation that generates that percent chance has happened 200 times or more in our database
3. The game is in the 6^{th} inning or later
4. The lead is at least 4 runs

We will now discuss how we arrive at these winning percentages and the specifics of this definition of garbage time.

4.1 Calculating Situational Odds of Winning

One of the most important aspects of our garbage time definition requires an approximation of the percent chance of winning for a team in a given game situation. For example, if a team is batting in the bottom of the seventh inning, has runners on first and second, and is losing by 5, do they have a three percent chance of winning or less? The answer to this question is key to determining whether a plate appearance is in a garbage time state or not. We chose to use historical data to calculate these probabilities of winning. There have been close to 220,000 MLB games played, leading to a sample size of over 13 million plate appearances for us to analyze. To find the probability of winning for a given game situation, we first had to define the properties of a game situation. The unique characteristics that define a game situation are normally:

Characteristics	Values
The pitching team's score	Runs
The batting team's score	Runs
The inning number	1-10
Top or bottom of the inning	Top/Bottom
Number of outs	0,1,2
If there is a runner on first	Yes/No
If there is a runner on second	Yes/No
If there is a runner on third	Yes/No

Table 4.1 details the different characteristics that a game state can have and the values for each characteristics.

We found that by the seventieth plate appearance in a game, a significant amount of game situations had a low sample size, which was insufficient for our needs since the average length of a game is about 78 plate appearances. To increase this sample size, we chose to combine certain features of the game state definition.

Our first adjustment was to use the difference in the team's scores instead of their actual scores to define a game state. We were able to make this change because a game where the score is 6 to 3 has identical probabilities to a game where the score is 10 to 7, given that other properties of the game situation are the same. The value of the scores is less important than the difference in the scores, or the lead. Once we began using the difference in scores, we chose the differential to be a maximum of 5.

Another change we made to the game situation definition was collapsing all extra innings into a game situation where the inning was 10. Game situations in the 12th inning are similar to game

situations in the 10th inning because regardless of the inning number, the game is conceptually in the last inning. Finally, we chose to use data only for 1973 and on, due to rule changes made by Major League Baseball.

Our final game situation definition consisted of:

- Pitching team's score minus batting team's score, minimized at -5, maximized at 5.
- The inning number, maximized at 10
- Top or bottom of the inning
- Number of outs
- If there is a runner on first
- If there is a runner on second
- If there is a runner on third

These changes yields a much better sample size for most game situations. Without the collapsed game situations, there about 20,400 game situations with a sample size less than 100. If we include the changes we made, this number drops to only 17 game situations with a sample size of less than 100. The distribution of collapsed game situation sample sizes has a much longer distribution tail, indicating that more game situations have a reasonable sample size.

Using our new game-state definition, we calculated an empirical probabilities of each team's chance of winning for every possible game situation. For example, the Mets and Nationals were playing on September 3, 2019 where the Mets were winning 10-4 at the beginning of the 9th inning. This scenario has happened 22,688 times before in our database. Only 86 times the team losing has comeback to win, so the Nationals had a 86 out of 22,688 chance to win this game based on the empirical probability. On this day, that Nationals were able to defy the odds and comeback and win the game.

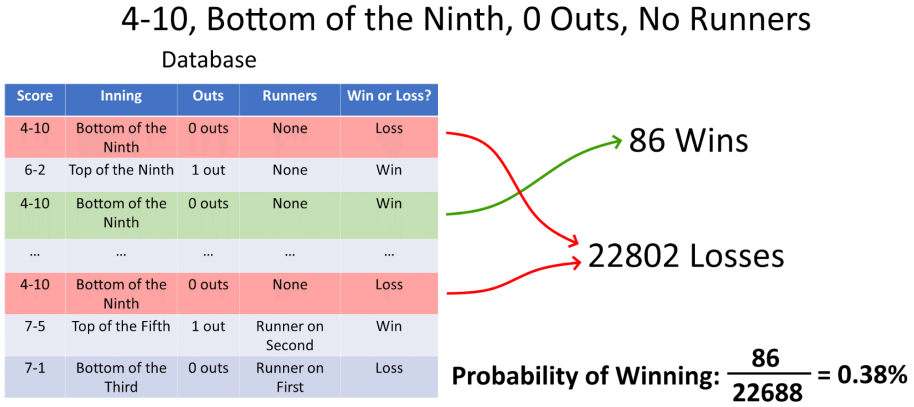

Figure 4.1 shows the process of calculating the empirical winning odds for a game situation.

The result of these calculations was a table in our database containing information about the probability of winning for either team and the sample size used for the probability estimate. We also created a database consisting of every game situation that has occurred in baseball, which breaks down which decade each sample came from (available in hidden columns).

Sample size is a very important aspect of these probabilities due to the large variation in game outcomes. Most often, one team will not completely dominate another team in repeated games. Therefore, a larger sample size is necessary to ensure that we have an accurate measurement of the probability of winning in a game situation.

4.2 Quantifying Garbage Time

After calculating the game situation probabilities, we could use this information to determine a definition of garbage time. First, we picked a variety of stipulations to test from chance of winning, minimum innings, minimum sample sizes, minimum score difference. The table below outlines the different values tested for each of the categories. The range of chance percentages were chosen such that there was a low chance of the team winning. We tested every inning and a large range of score differences so we could capture as many different scenarios possible. The minimum sample size was set to insure that the final definition was not easily manipulated by the data. With the numerous different possible scenarios, a small sample size has the possibility of being influenced more by outcomes that are outliers.

Chances	Min. Inning	Min. Sample Size	Min. Score Differential
1-20%	Innings 1-9	20,50,75,100,125,150,174,200	0-11 Runs

Table 4.2 shows the different game state characteristics used to solve for the Garbage Time definition.

After determining the range of stipulations, we needed to test the 17,280 different combinations. For each scenario, we filtered the database to include only non-garbage time data, so only the plays that could possibly happen before one of the restrictions. With this filtered data set, we optimally fit the predicted runs equation for quantifying cluster luck:

$$\text{Runs excluding garbage time} = a * OBP + b * SLG + c * AVG$$

and found the optimal exponent for the equation:

$$\text{Predicted Wins} = \frac{(\text{C.L. Runs})^m}{\text{C.L. Runs})^m + \text{C.L. Runs Against})^m}$$

where m the exponent value. The equation was fit through an ordinary least squares function and the exponent was found through testing exponents ranging from 0.1 to 15 with steps of 0.01 to find the lowest residual. Using the final optimal results, we were able to get an average absolute residual value for each of the possible garbage time scenarios by finding the difference between the predicted wins versus the actual wins. By optimizing the regression and exponent value for each scenario, we could be sure that we were comparing the scenarios themselves versus how well each garbage time scenario fit with the original equation and exponent rather than the runs regression. Below is a table of the best two and worst two combinations of parameters. As seen in the table, with the implementation of the final definition of garbage time the residual was 3.56 games, which is 2.2% difference within a season.

Chance	Min. Inning	Min. Sample Size	Min. Score Difference	Residual
3%	6	200	4	3.558
3%	6	125	4	3.562
20%	4	125	0	4.337
20%	4	150	0	4.358

Table 4.2 shows the two best and worst possible garbage time definitions.

We can use our definition to visualize what garbage time looks like in any game. In the game below, the Nationals lost to the Braves by 9 runs. A significant portion of the plays past the 6^{th} inning were in a garbage time state, demonstrating how difficult it can be to get out of garbage time. Note that right before the 7^{th} inning, the nationals had greater than a 3% chance of winning, so it was no longer considered a garbage time play.

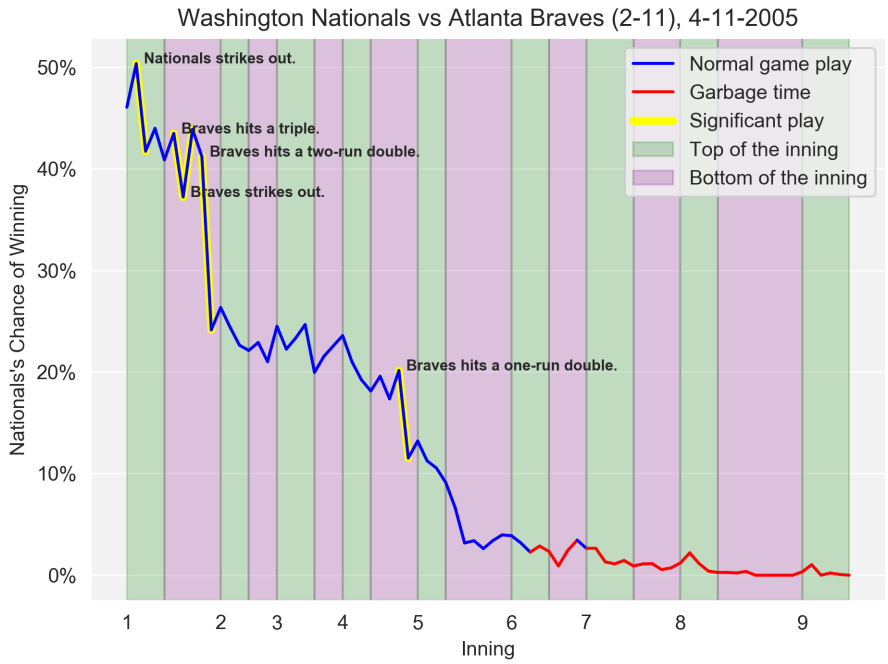

Figure 4.2 demonstrates what garbage time can look like in a game.

4.3 Garbage-Adjusted Statistics

Using our method to quantify garbage, we have created an array of new statistics that we call garbage-adjusted. A garbage-adjusted statistic is a traditional measurement of a baseball player, except that only plate appearances that are not considered garbage time are considered. We believe that plate appearances in a garbage time state often do not have the player's utilizing their full potential, since the game has already been decided.

An example could be batting average. The traditional batting average formula is

$$\text{Batting Avg} = \frac{\text{Hits}}{\text{At-Bats}}$$

While this statistic can be incredibly useful, we believe that including garbage time plays could be misleading and distort a player's batting average away from their true level of skill. A garbage-adjusted statistic that takes out at-bats in garbage time would be calculated as

$$\text{Garbage-Adj. Batting Avg.} = \frac{\text{Non-Garbage-Time Hits}}{\text{Non-Garbage-Time At-Bats}}$$

This formula is rooted in the same basic idea as the traditional batting average statistic: Measuring the percent of time that the player hits the ball. However our adjusted statistic ensures that we only use plays where all players are trying their best, since both teams still have a chance of winning.

While this may seem like a small change, the results can be significant. In 2018, 9.51% of all plate appearances were in a garbage time state. In 2017, 10.42% of plate appearances were in garbage time. As a result, many garbage-adjusted statistics vary greatly from traditional statistics. For example, Ryan Braun in 2015 had a large difference in his traditional slugging average and garbage-adjusted slugging average.

Ryan Braun, 2015	Traditional Stat	Garbage-Adj Stat	Difference	Percent Difference
PA	568	521	-47	-8.27%
AB	506	464	-42	-8.30%
H	144	122	-22	-15.28%
2B	27	21	-6	-22.22%
3B	3	3	0	0%
HR	25	20	-5	-20%
BB	54	52	-2	-3.70%

Table 4.3 details traditional and garbage-adjusted statistics for more simple baseball statistics for Ryan Braun in 2015.

Ryan Braun, 2015	Traditional Stat	Garbage-Adj Stat	Difference	Percent Difference
BA	0.285	0.263	-0.022	-7.72%
OBP	0.356	0.339	-0.017	-4.78%
SLG	0.498	0.45	-0.048	-9.64%
OPS	0.854	0.790	-0.064	-7.49%

Table 4.3 details more complex statistics for Ryan Braun with both traditional and garbage-adjusted.

Since we are excluding a number of plate appearances, non-scaled fields such as at-bats or home runs will never increase. The more important fields to analyze are scaled, such as batting average slugging average, and on-base percentage. Typically, even scaled statistics will decrease, because in garbage time at-bats the pitcher will not try as hard, resulting in inflated statistics for the batter. Removing this inflation will lower the statistics. However, a .0964 decrease in slugging average is significant even compared to most players.

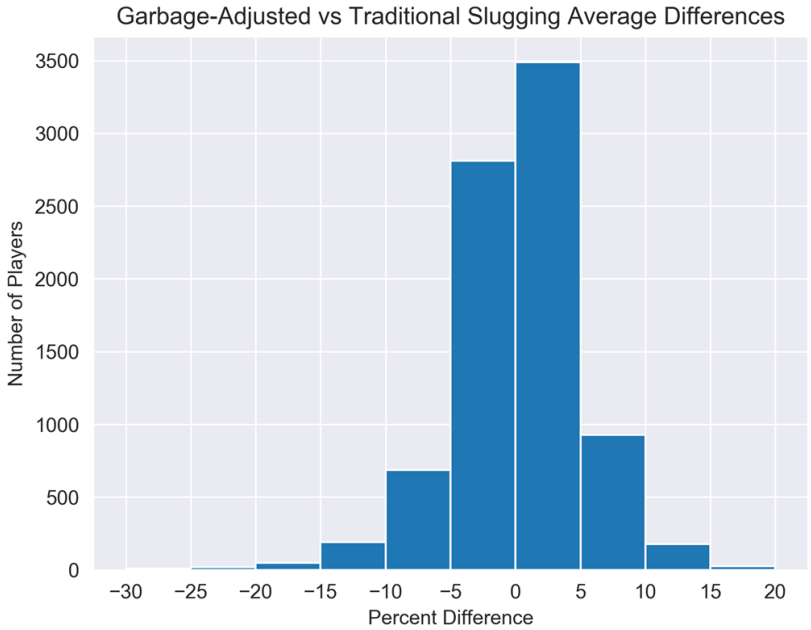

Figure 4.3 is a histogram of the differences between garbage-adjusted and traditional slugging statistics.

Above is a histogram of the differences in traditional and garbage-adjusted slugging average for players in each season since 2000. We only include players with more than one hundred plate appearances to avoid misleading differences, such as if a player had one at-bat in garbage time that was a home run, resulting in a difference of 100%. Of the 8400 players with more than one hundred plate appearances, 2100 of them, or about 25%, have a slugging difference of more than .050. These differences can be important in determining the true talent of a player as compared to others in the major leagues.

Chapter 5

Preseason Model

As we have talked about before, we found regression to be a valuable method in determining runs that were gained or lost due to cluster luck. We discussed the different hitting metrics we found to be the most indicative of true hitting talent. Additionally, we have talked about the phenomenon we call Garbage Time and why we eliminate statistics that are accrued in that game state. If we combine these ideas to create and adjust statistics that we use to value teams after a whole season, we may be able to predict a teams total number of wins going into the next year. In this chapter we will talk about how we create an expected seasonal winning percentage for each team before the first pitch is thrown. We call this our **preseason model**.

5.1 Calculating Cluster Luck Adjusted Wins

As we said back in the previous chapter, if we wanted to accumulate the cluster luck adjusted runs for every team, that we would need batting average, on-base percentage, slugging percentage, and isolated hitting for every player on the team to generate the team statistics. Additionally, we would need all of the opponents statistics for the pitchers to calculate runs against. Once we have calculated our adjusted runs scored and runs against, we can then calculate the win percentage using the Bill James winning percentage pythagorean formula:

$$\text{Win Percentage} = \frac{RS^{1.83}}{RS^{1.83} + RA^{1.83}}$$

Normally, this formula is used to predict how many wins a team should have in a given season. With cluster luck adjusted runs number, we can show how many games a team deserved to have won the previous year and adjust that number to roll it forward onto the next year.

Using historical data, we have calculated the necessary batting and pitching statistics for every team in MLB, and every year since 2000. To do this, we iterate through each plate appearance a team has, and use the given values, such as doubles or home runs, to calculate the statistic. For example, batting average is created by dividing hits by at-bats, which means we would accumulate of the number of hits and number of at-bats a team had in a given year. We have finish checking every plate appearance, divide the total count of hits by the total count of at-bats, and then have our aggregated team batting average statistics. Using our aggregated team statistics, we performed our cluster luck analysis, yielding our cluster luck adjusted runs scored and runs against, which then led to our cluster luck adjusted wins after applying the Bill James Pythagorean formula.

Below Figure 5.1 is of cluster luck adjusted wins compared to actual wins for teams in the 2018 season. The green dotted line indicates points where adjusted wins and actual wins are equal, creating a baseline for analysis. A team above the line had higher actual wins than adjusted wins, indicating that they experienced a net of positive cluster luck in their season. The red data point is the Boston Red Sox, which were a prime example of significant, positive cluster luck. A team on the green line, such as the St. Louis Cardinals, in purple, experienced almost no cluster luck or both positive and negative cluster luck that canceled out over the course of the season. Teams below the line had negative cluster luck, or unluckiness, like the Baltimore Orioles in orange.

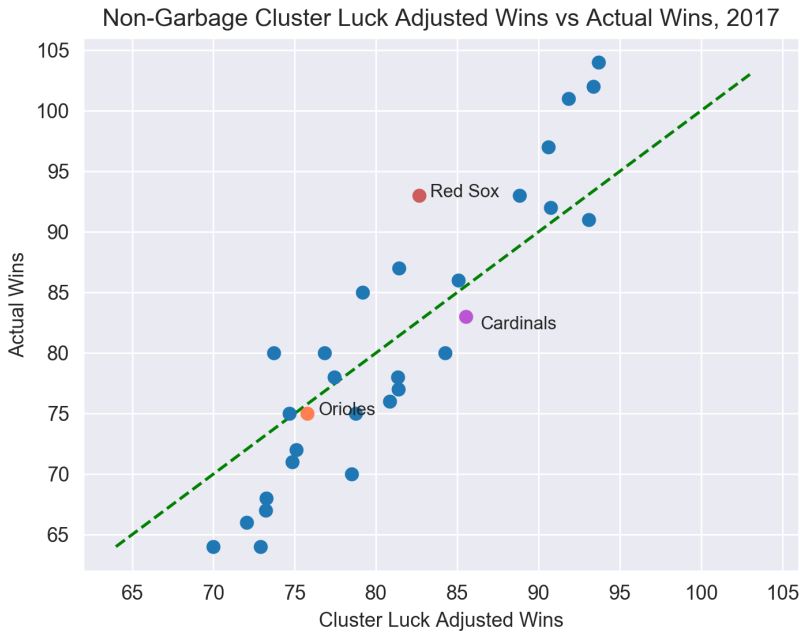

Figure 5.1: A graph of adjusted cluster luck wins versus actual wins for 2018.

A team's adjusted wins serves as a baseline for the next season's total wins. By stripping out any luck the team experienced during the year, we create a solid foundation for prediction. It is important to note that luck experienced this year does not indicate luck or success experienced in years going forward. This is why this measure is more concerned with true talent,which in theory is a better predictive metric with luck stripped out.

For each of the statistics we calculated, aggregated batting average, on-base percentage, and slugging average for teams, we also calculated equivalent garbage-adjusted statistics to try to improve the accuracy of the statistics we use. For example, when calculating garbage-adjusted batting average, we count the hits and at-bats a team participated in during a season, but only with plate appearances that do not occur in garbage time. With non-garbage hits amd non-garbage at-bats we create a statistic for a garbage time-adjusted batting average. This way, we can be sure our regression is using a more indicative sample of data which theoretically should give us the best results.

This created two different methods of determining a total team's deserved runs based off of their production the previous year. Now we can project these results forward and attempt to look at the next year using our new statistics. If there were no adjustments to rosters or player's abilities over

the offseason, this model would be a true predictor of wins for the next year. However rosters and talent levels do change, so these are considerations we need to take into account.

5.2 Roster Changes

In the off-season, players are traded, cut, or signed. Each move creates a need for recalibration so our predictions still accurately represent the teams we are predicting. We need to account for the runs lost or gained based on the roster that ended the previous season and project them onwards to the next season. This forces a change in our thinking from a total team's talent to individual player talent. We need to use individual player statistics to determine how many runs each player is worth to their team.

Recall from the cluster luck section where we discuss the 2014 Angels. We discussed that the Angels benefited from extra runs due to cluster luck and we were projecting that towards the 2015 season. The Angels had led the league with 773 runs scored and then had only allowed 630. After running our cluster luck regressions, we found their adjusted numbers to be 699 runs scored and 594 runs allowed. However, the Angels had some slight changes to their both their rotation and their lineup going from 2014 to 2015.

In 2014, starting pitcher Tyler Skaggs had Tommy John surgery, which is a deteremental surgery for a pitcher on their elbow. As a result, Skaggs would be out for the entire 2015 season, so to replace him the Angels got Andrew Heaney. They also lost Kevin Jepsen who was an important middle reliever for the Angels throughout the 2014 season. Unfortunately, the Angels also lost Josh Hamilton and Howie Kendrick who were two big stars in their lineup throughout 2014. They were replaced in the lineup with Johnny Giavotella and Matt Joyce. After losing Hamilton and Kendrick the Angels lost out on seven wins and were only able to replace them with an expected two wins. The difference in their lineup is rather noticeable.

Clearly the Angles' talent was depleted from the previous year. Especially in their projected runs scored, we need to use some sort of way to measure this. To account for these roster changes, we use the WAR, Wins Above Replacement, statistic that gives each player a value in terms of wins. The WAR metric gives each player a number of wins they are worth above a "readily available replacement." Essentially, WAR is the number of wins a team could expect to lose if a player gets hurt in spring training or leaves the team and they need to call up a player from AAA for the rest of the season. Now, in order to be used in our pythagorean formula, we need to convert the number in terms of wins into runs. From 2010 to 2018 we found that the conversion of one win translates to roughly 9 runs. For example, we can take the WAR of Josh Hamilton and equate it to 9*WAR to find the number of runs scored the team has lost with the departure of Hamilton. Pitcher's runs are calculated the same way except we need to subtract runs if there is an improvement in WAR because that pitcher is allowing less runs for his team.

Below in table 5.2 are the off-season transactions for the Angels and the associated impact on expected runs.

Player Name	War(2014)	Runs Allowed Gained/Lost
Kevin Jepsen(Pitcher)	1.3	+11.7 RA
Tyler Skaggs (Pitcher)	0.2	+1.8 RA
Andrew Heaney (Pitcher)	-0.3	+2.7 RA
Josh Hamilton	1.4	-12.6 RS
Howie Kendrick	5.4	-48.6 RS
Johnny Giavotella	0.1	+0.9 RS
Matt Joyce	1.3	+11.7 RS

Table 5.2: shows the converting of various players WAR to their runs allowed value.

In table 5.2 we look at all of the roster changes for the Angels in the 2014-2015 offseason. With the roster changes taken into account we change the runs scored from 699 to 650 and runs allowed from 594 to 610. We can now calculate the wins expected using the 650 for runs scored and 610 for runs allowed. These statistics result in a .529 winning percentage which results in a prediction of 85.5 wins in the 2015 season.

5.3 Injury Adjustments

Along with roster changes, injuries are another major factor that affects a teams performance. Injuries are difficult to account for, since the amount of time a player is injured for can be difficult to predict. Additionally, when the player returns to the game their performance can be vastly different. Players are taken off the roster if they are on the Injured List which means they are not counted as being on the team, so their associated runs number is taken off the team as if they had been traded. These players could come back and play for the team when they are healthy so we need to account for that possible increase in production when they come back and replace someone else. In our model we use a simple approach for injury adjustments.

We were able to obtain injury information using CBS Sports MLB Injury Reports. CBS Sports provides information on each players name, position, injury, and often an expected return date. Our preseason algorithm scrapes the necessary injury data from CBS Sports and deduces the player ID of each injury listing using the name and player position given. The algorithm then analyzes the statement containing the expected return date, and cross-references this date with team schedules. Using this data, the expected percent of games the player participates in is calculated.

We continue to use the WAR statistic to calculate a players contribution to their team in the form of runs. However, instead of adding WAR multiplied by 9 to the teams runs scored and runs against (as we did in roster adjustments), we will first multiply by the percent of games the player is expected to participate in. We can do this because WAR is already considered in terms of wins per season.

For example, suppose a player is currently injured, and expected to return on April 9^{th}. The team they play for has 30 games scheduled before April 9^{th}, meaning the player will miss the first 30 games. There are still 132 games remaining in the season, so the player is expected to participate in 81% of games. Without an injury, the player is expected to have a WAR of 2, meaning he would contribute 18 runs to the team's expected season total for runs scored. However, the player is missing 19% of games, so using 18 runs would be inaccurate. Instead, we would use 18*0.81 = 14.58 runs, which is the estimated number of runs the player is expected to score in the expected 81% of the games they are participating in.

Player	Games Expected to Play	Percent of Games Expected to Play	Expected Runs Scored (WAR*9)	Injury-Adjusted Runs Scored
Jed Lowrie	133	82.10%	43.2	35.47
Todd Frazier	151	93.21%	16.92	15.77
Yoenis Cespedes	68	41.98%	8.28	3.48
Total				**54.72**

Table 5.6: Mets injured list adjustments for the 2019 season.

The final expected number of runs from injured players on the Mets sums to an additional 54.72 runs. These runs get added to the cluster luck adjusted runs scored for the Mets because they were not listed on the Opening Day roster, but came back and played that season. We also total the injured pitchers expected runs allowed, and add their total to the teams cluster luck adjusted runs allowed.

Without injury adjustments, our model predicts the Mets to have 77.25 wins, but accounting for injuries increases their expected wins to 83.57. The actual number wins the Mets had in 2019 was 86, leading to a residual of only 2.43 wins.

5.4 Alternative Attempts at Creating the Preseason Model

Some concerns arise when we just take last years data and use that to project onto the next year. For example, players get a year older which can affect players differently depending on age. Older players can get worse with age, but younger players approach their prime and so they might get better, and in some cases, a lot better. To try to predict this, there are different projection software that predict individual player performance. The most popular are PECOTA and ZIPs projections. Each system uses different advanced statistical and analytical methods to produce their projections.

We tried to think of ways to incorporate these projections into our models and one good opportunity was using a player's projected WAR statistic for offseason transactions. Overall, we found that PECOTA was more accurate of the two for our needs so we continued to use PECOTA for our trials. PECOTA uses statistical techniques that compare current players' performance curves and finds relationships to former players' curves. With this method, they can try to predict when players will enter and leave their primes. For those reasons it made sense to use PECOTA's projected WAR for our player transactions. Below are the mean absolute value of the preseason residuals using previous years WAR and PECOTA projected WAR.

Year	Mean Average Error (MAE) Using Previous Year	MAE Using PECOTA
2013	8.84	9.22
2014	6.46	6.90
2015	9.22	8.93
2016	6.96	7.16
2017	9.39	8.97
2018	9.43	9.37
Avg	8.38	8.43

Table 5.4 details the mean average error for 2013 to 2018 for our preseason model and for Silver's PECOTA.

We obtained PECOTA data through the Baseball Prospectus website, which is now part of the statistics in our database. The two residuals are very close as a different method 'wins' every year. We see recently PECOTA has been performing better four out of the last five years. PECOTA has

improved over the years with new developments and improvements in personal player modeling and curves since their start in 2013. Therefore, looking into 2019 it might be better by an even greater margin and over time using PECOTA may prove to be the better option.

5.5 Alternative Methods to Creating Preseason Models

Over the course of developing this model we tried many different things that did not produce results. We will detail some of those attempted methods

5.5.1 PECOTA and ZIPS

First we will talk about alternate uses of PECOTA and uses for ZIPS. We thought to use PECOTA to pull their projections for BA and the other statistics we utilized in our regression equation and plugging them in to get a projected runs number. When we used the projected seasons WAR from PECOTA or ZIPs, we did not see an improvement over just using last year's WAR. When we attempted to use the projected statistics we could only find them for runs scored. PECOTA and ZIPs do not try to predict onbase average or slugging against for pitchers so we could not create a runs allowed. It would not be consistent if we used last year's data for runs allowed and the projections for runs scored so we could not do that option.

5.5.2 Total Bases

We also explored to incorporate our total bases pythagorean model for the preseason similarly to our runs model. However, this did not perform as well as our cluster luck adjusted runs model. We thought that total bases itself would be a good statistic to use because it accounts for cluster luck in the nature of how it is calculated. On the other hand, over the course of one season and rolling it forward to the next, there is most likely too much noise to have a consistent result. Additionally, we tried to take the predicted values of total bases and incorporate them into our model. Often times these statistics were not provided for pitchers which meant there was no way to get total bases allowed for pitchers which is needed to calculate half of the Pythagorean winning percentage. Our total bases model for preseason data was also not successful so it makes sense that using the projected total bases would also be too inconsistent.

5.5.3 Fractional Rosters

Fractional rosters is a method to improve our adjustments for traded players by taking into account how long the player was part of the team. As long as there are trades and injuries, a team's true roster in a season is not the roster at the start of the season or the roster at the end of the season. A better representation of the team's true roster might be the fractional roster.

The fractional roster for a team is calculated by taking the number of games every player in MLB has played for that team in the season, and dividing by 162. Which allows for fractions of a person to be on the roster. For example, if Mike Trout was traded to the Angels midway through the 2018 season, and ended up playing exactly 81 games for the Angels, the Angels would have an entry of 0.5 for Mike Trout in their fractional roster. For players who have never played a game for the Angels in 2018, they would have an entry of 0, or no entry.

Fractional rosters allows us to more accurately adjust our preseason prediction for new players who are traded midseason. For example, if we were using fractional rosters to adjust for Mike Trout in the Angels 2019 season, we would add Mike Trout's WAR multiplied by the difference in his

expected fraction of games played for the Angels in 2019 and his fraction of games played for the Angels in 2018. We do not expect the Angels to trade Mike Trout away in 2019, so his expected fraction of games played in 2019 is 1. Therefore our WAR adjustment is:

$$WAR \cdot (1 - 0.5) = 0.5 \cdot WAR$$

WAR is a non-standard statistic, meaning players accumulate WAR for a team during a season. Unlike other statistics, WAR is not divided by number of at-bats or number of games played to standardize it. In this example, Mike Trout only accumulated 81 games worth of WAR for the Angels in 2018, but he is expected to accumulate a full years worth of WAR in 2019. Therefore we add the difference, since the Angels will have 81 more games of Mike Trout on their fractional roster as compared to last year.

For this example, our standard method would look at the difference between the last game of the previous years roster, and compare with the first game of the current years roster, and see no difference. Since Mike Trout was traded to the Angels halfway through the 2018 season, and is still on the Angels at the start of 2019, our standard method would say there is no adjustment necessary.

Fractional rosters have been implemented as one of our possible models, and in tests of different preseason prediction methods, models using fractional rosters had the lowest average residual.

5.6 Regression Results

Year	MAE of Pre-Season Predictions	MAE of Pre-Season Predictions Excluding Garbage Time
2013	8.54	8.84
2014	6.39	6.46
2015	9.18	9.22
2016	7.17	6.96
2017	9.54	9.39
2018	9.14	9.43
Average	8.33	8.38

Table 5.6: shows the results of the regression from using BA, OBP, and SLG.

Year	MAE of Pre-Season Predictions	MAE of Pre-Season Predictions Excluding Garbage Time
2013	8.17	8.60
2014	6.56	6.74
2015	9.06	9.12
2016	6.92	6.60
2017	9.68	9.56
2018	9.02	9.46
Average	8.24	8.35

Table 5.6: shows the results of the ISO and OBP regression.

Table 5.6 contains the results using specifically the cluster luck definition using AVG, OBP, and SLG. On average we are within 8.33 games on every team's prediction at the beginning of the season which results in a 5% residual over the 162 game season. It was curious to us as to why the garbage

time residual was not an improvement on our other model. This may be because our definition for garbage time needs to be changed. It is also possible that it is more important to include the samples of garbage timeplays to increase our sample size rather then take them out in fear of them not being a good indicator of talent.

We see similar, but slightly better results with the ISO and OBP regression in table 5.6. This we assumed would happen because of the less variables in the cluster luck runs regression. With less variables, the statistics are less correlated with each other which means we do not need any coefficients in the regression to offset one another in the equation. This is what happened with the BA, OBP, and SLG model. Since all three statistics are correlated, they all went up with runs, which means at least one of the three needs to balance out the increase of the other two. We also see that the garbage time adjusted model is off by slightly more as with the other regression's model.

There could be a few reasons that garbage time does not benefit the model. Firstly, over the course of the season, the effect of measuring garbage time might lose its ability as a predictive measure. We see that with some players there are noticeable differences in player's production, but that might not be the case for all players. Therefore, taking out garbage time might not be beneficial for the majority of the players which is what we do in using garbage time with this model. It could also be, as similar to using total bases from the year before, there might just be too much noise and garbage time to take it from one season to the next.

5.7 Second Regression

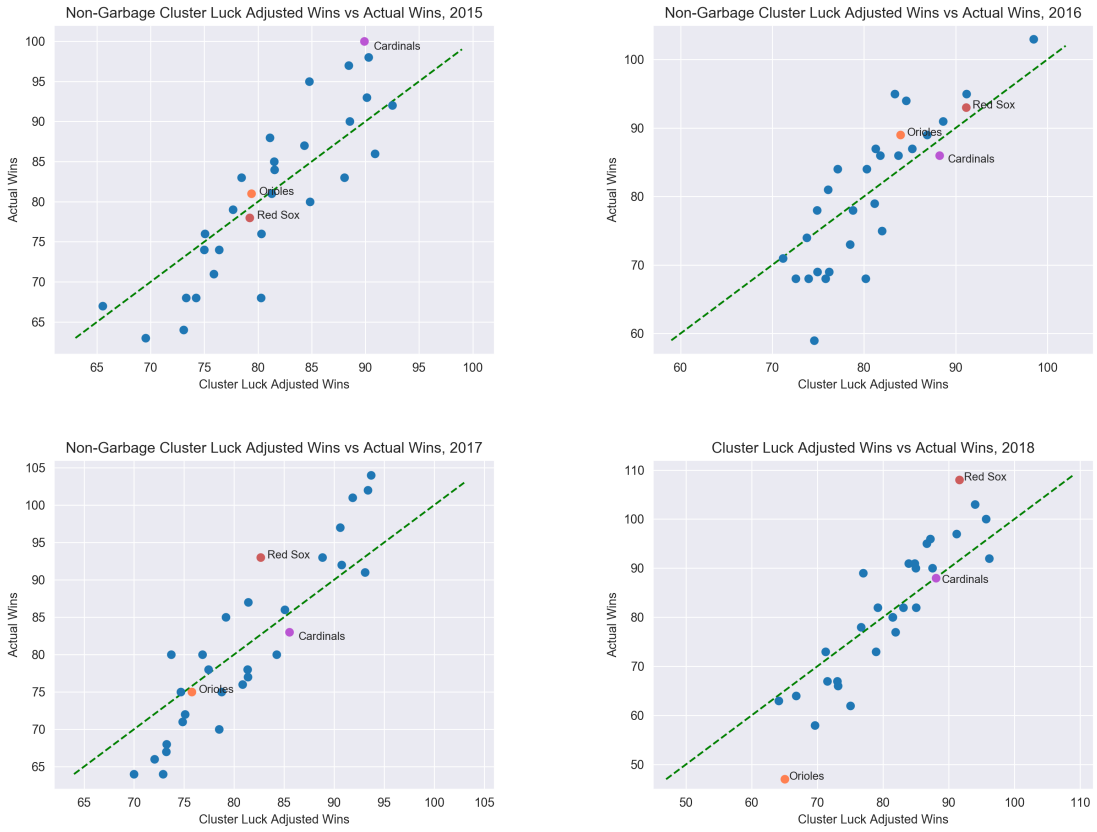

Figure 5.7 shows the attempts to fit a second regression for the preseason predictions for various years.

After studying the outcomes from our best models we were looking to see if there was any way to improve our results. Here we have graphs from 2015 to 2018. Recalling our average residual is about eight games per season, it is easy to see that some games on the extremes are usually off by more than eight games. Teams that we predict to win more, win even more than we expect and the really bad teams lose even more than we expect. We see this trend for every year of our model. If we can narrow down these higher residuals on both ends we can improve our overall residual tremendously. We thought the best way to do this was to take a second regression. This time, a regression between our expected wins against the actual wins. This way we should be able to shift and tilt our green projection line up, thus, fitting our data more. We can take this regression and input our original expected wins. However, we found that that tactic made the error increase so we never incorporated the second regression in our final model.

Chapter 6

In-Season Model

We have created a model that can predict games at the beginning of the season solely off of prior season and off-season data. However, the season is 162 games long and spans from March to September, so there is a countless amount of data that comes out over the course of the season that could enhance the model as we stray away from the start of the year. All of this data can be used to alter our winning percentages as the season progresses. Therefore, we have developed the **in-season model**, a model that changes as every game is played to update our predictions.

6.1 Runs

The first model we developed was based off Joe Peta's model from his book *Trading Bases*. Since we first started to base our work of his, we needed to fully understand how his model works. This model calculates a win percentage through a pythagorean equation:

$$\text{Win Percentage} = \frac{RS^{1.83}}{RS^{1.83} + RA^{1.83}}$$

where RS is the runs scored and RA is the runs allowed. To calculate a team's estimated number of total wins after each game, we determine from the database the total number of runs and runs against thus far and plug those values into the equation above to get the win percentage. After calculating the win percentage, we then multiply by 162 to get the teams' estimated total wins for the season. The full equation is:

$$\text{Wins} = 162 \cdot \frac{RS^{1.83}}{RS^{1.83} + RA^{1.83}}$$

We can repeat this calculation after every game by updating the number of runs and runs against, thus updating the prediction for total game won.

Figure 6.1 shows four examples of 2018 in-season runs models for Boston Red Sox, L.A. Dodgers, Toronto Blue Jays, and San Francisco Giants. For Boston, the model was able to predict 102.9 of the total of 108 games won, a residual of 5.1 games. The model over predicted for the Dodgers with a prediction of 100.84 wins for a 92 game season which is a residual of 8.84 games. Toronto won a total of 73 games, whereas the model predicted 69.23 games to result in a residual of 3.77 games. Lastly, the runs model predicted 94.79 total wins for Oakland, who won 97 total games for a residual of only 2.21 games. It is important to note the convergence of the model in all but the

last figure. By roughly game 60, the model was able to come close to the true number of wins. The Oakland A's graph shows that this convergence is not always true for all teams, but the model can still calculate a strong estimate on the scale of the whole team. However, this method does not take into account lucky runs that happened in the past and therefore might not be able to predict wins with as much accuracy as our talent based models.

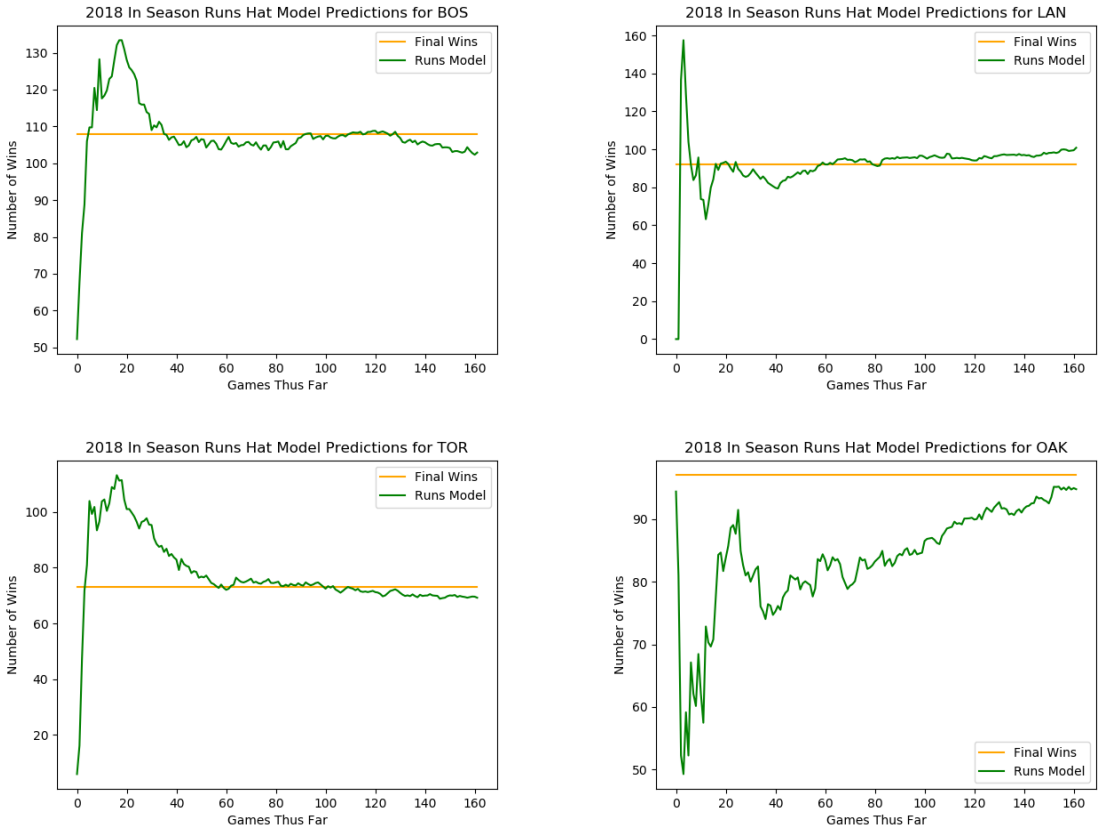

Figure 6.1 shows graphs the Runs Model for 4 teams in 2018.

6.2 Cluster Luck Runs

After developing the runs model, we wanted to incorporate the idea of cluster luck, as done in the preseason model. In this updated model, we used the same basic set-up of calculating the win percentage value using the Pythagorean equation and determining the final estimation by multiplying by 162. Instead of directly plugging in runs for and runs against into the Pythagorean equation, we now use our linear regression to estimate the runs for and runs against, using the teams Batting Average, OBP, and Slugging. The new equation is:

$$\text{Wins} = 162 \cdot \frac{CLRS^{1.83}}{CLRS^{1.83} + CLRA^{1.83}}$$

where CLRS and CLRA are the runs from the cluster luck regressions. We hope that the model will be able to encompass a teams deserved runs based on performance rather than just the runs

they have had. The model is built off of our idea from the preseason model, but allows us to incorporate the statistics that are accrued this season. This is important because if teams are performing differently than their current runs scored and earned, the in-season model will be able to offset that original prediction.

In figure 6.2 below, the same team in-season predictions as previously discussed are shown. Boston was predicted to win only 90.88 games, a large difference from their true wins of 108, a total residual of 17.12 games. The model came close for Dodgers, with a 4.68 game residual, but does not have the same converging tendencies as we saw in the runs model. Toronto saw this convergence, but with a prediction residual 6.2 games higher than the actual wins. Oakland has the same general shape as in the runs model, ending with a prediction residual of 5.56 games less than the teams' record.

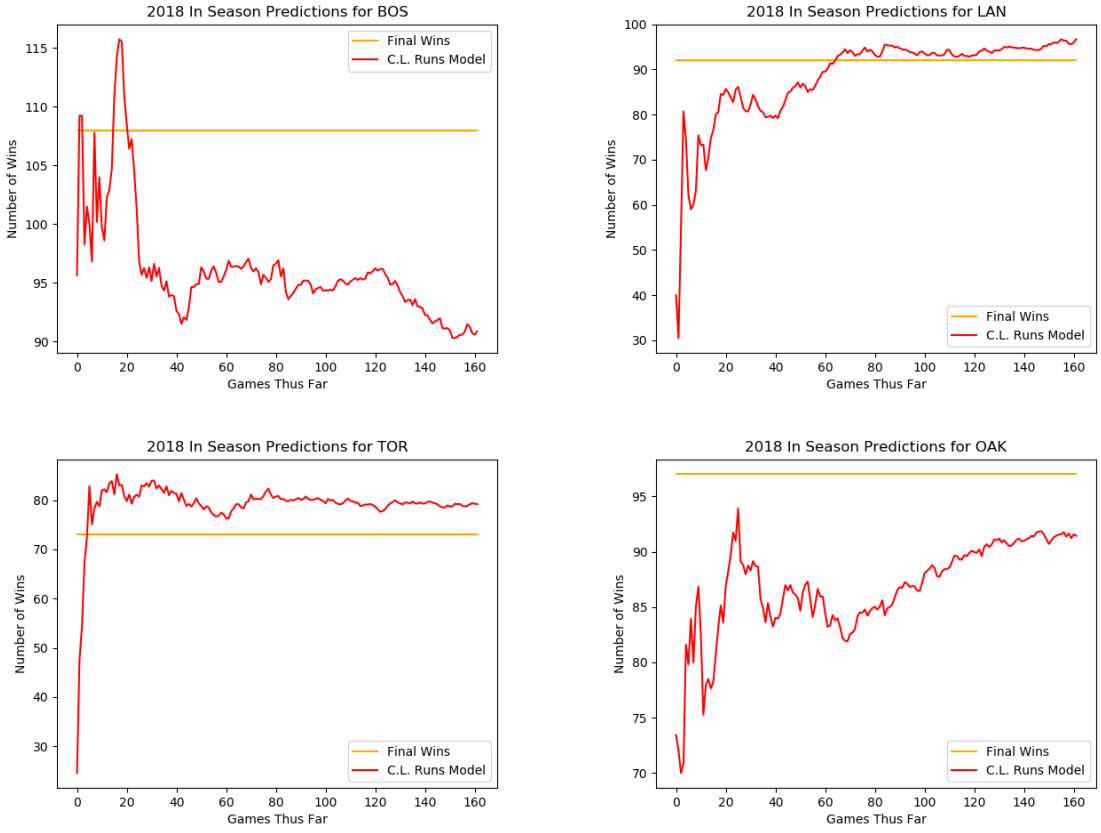

Figure 6.2 shows graphs the cluster luck Model for 4 teams in 2018.

6.3 Total Bases

Like our preseason model, we tried our developed total bases pythagorean formula for the in-season model. This works similar to the formulas above where we accumulate the total bases for and against as the season goes on, plugging them into the same Pythagorean formula we developed in the cluster luck section:

$$\text{Wins} = 162 \cdot \frac{TB^{2.5}}{TA^{2.5} + TB^{2.5}}$$

At every game, we have a number of wins we would expect the team to have at that point using the above formula. Below, in figure 6.3, are examples from the same four teams we have used in the previous sections. Similarly, there is a lot of fluctuation in the early games because there is so much noise with only using a few games of data for this model. We see the trend of around game 80 that the model moves much more smoothly. The models final predictions for the Red Sox had a residual of 14.01 games, a residual of 4.44 for the Dodgers, a residual of only 0.48 games for the Blue Jays, and a residual of 6.86 games for the A's.

Figure 6.3 shows graphs the Bases Model for 4 teams in 2018.

6.4 Wins Adapted In-Season

The three models we developed only incorporated team statistics at any given point in the year, but never accounted for the current wins of the team. In the wins adapted model, the current team standing was incorporated into the prediction to remove error from predicting previous games. At each game, the prediction is now:

$$\text{Wins} = \text{Current Wins} + (162 - n) \cdot \text{Wins Expected}$$

where wins expected can be calculated with any of the three base models: runs, cluster luck runs, or total bases. Now at each game step we only predict the number of wins a team will get in the rest of the season while adding in the wins that they already had using n as the number of games played. This model lets luck that has already happened get accounted for so the model can just focus on predicting the future games. Below, in figure 6.4, we have all four models we have created on the model. Obviously, the models all end being on the exact number of wins a team has because of how the formula is developed. However, we can also see that the model is much more accurate in predicting the end result of number of games won from a given point in the season, since the model converges faster. This is because we are removing the effects of when the model was incorrect earlier in the season and just using that data to look forward in the season. In about $\frac{3}{4}$ of the teams we come very close to their season ending number of wins around game 80.

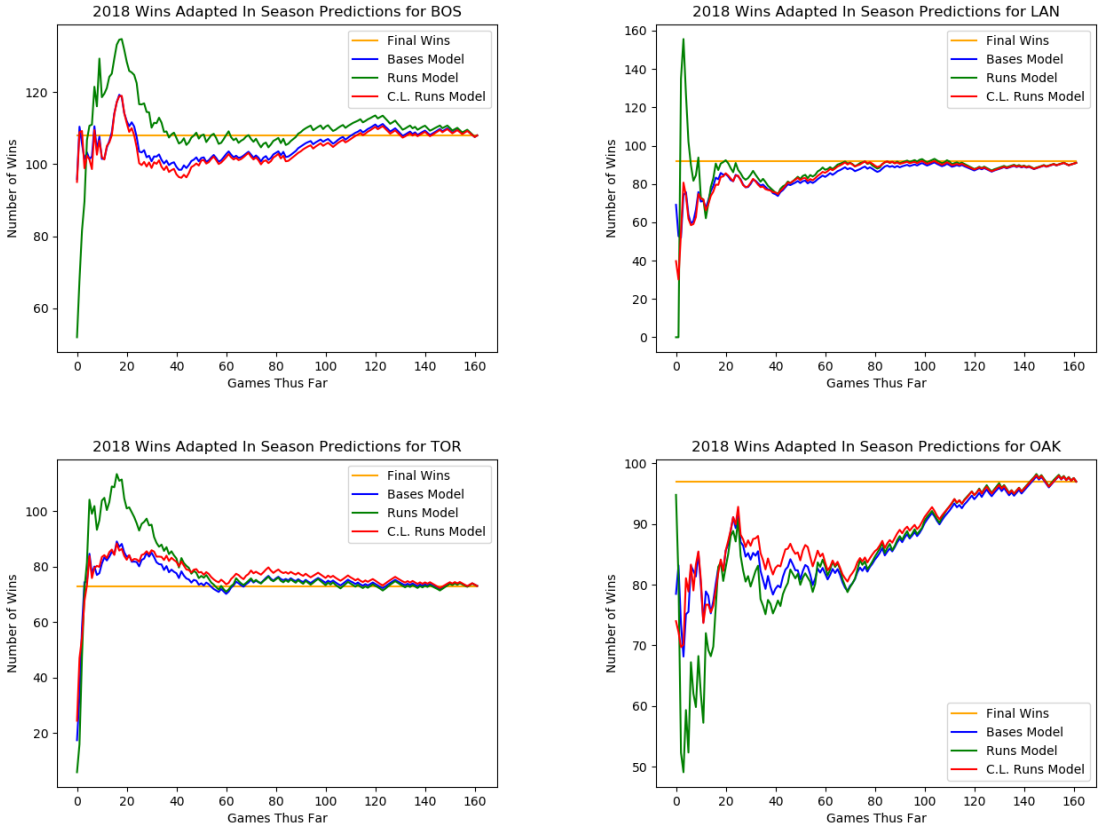

Figure 6.4 shows the Wins-Adapted models for the Red Sox, Dogders, Blue Jays, and the A's

6.5 Garbage Time Exlcuded

Lastly for the inseason models, we wanted to incorporate the previously discussed idea of garbage time. When calculating any of the models discussed above, we can ignore any bases, runs, or statistics for the cluster runs that may have happened in garbage time to factor out the variability of player performance during these times. For this method, the formulas stay the same, but before incorporating any statistic the algorithm determines when in the game it was from and only uses

those from non-garbage time plays. The results from this action can be seen below in the results section.

6.6 Results

We have already discussed the ways we look to predict a team's total wins. In figure ?, below you can see the average residual for every team, from the years 2015-2018. There are three basic models based off of runs, cluster luck adjusted runs, and total bases. These three models did well in their basic forms. For the years 2015-2018, the runs model had an average residual of 7.73, cluster luck runs had an average residual of 7.27, and bases had an average residual of 7.17 games. The base model did the best of these methods, but only by a fraction of a game.

Moving on to the wins-adjusted models, it is not surprising that the residuals are improved, since we are now accounting for, and removing, incorrect predictions from earlier in the season. We see bigger improvements in the runs model because there was more room to improve on the higher residual than the other models that already had lower residuals . However, even with the big improvement, the residual of 6.68 games for the runs model is still higher compared to the 5.54 and 5.87 game residuals for cluster luck runs and bases models respectively. We see the wins adjusted cluster luck pull ahead of the total bases model with the lowest average residual.

Taking out garbage time statistics did not greatly affect the results of both the basic and wins adjusted models. However it had a large effect on the the runs model. Taking into account garbage time for runs has a greater impact than on cluster luck runs because the runs taken out in garbage time has more impact than removing batting average, on-base percentage, and slugging percentage that are produced in garbage time. However, we expected to see total bases react the same way as runs because, thinking intuitively, these should be correlated and therefore we should see the same decline in residual for garbage time. The difference in the models' residual change from garbage time to not garbage time could come down to how we developed the the total bases pythagorean formula versus the runs pythagorean. We aligned the total bases exponent to cluster luck adjusted runs, but in future work we could explore if we aligned it to wins, if we would see a different, possibly improved, result.

Overall, we see the best results with the total base models, along improvements with wins-adjusted models, and very little difference with garbage time models. The total base models were able to predict within 7 games on average for all 30 teams over 2015-2018. As we intuitively thought, the wins-adjusted addition created significant improvement, with up to a 2.19 decrease in residual. In future work, we can look into our garbage time definition to study how we can adjust it to create a better than these results, with at most a 0.55 decrease in residual. We will look to see how we can optimally combine these models with our preseason models to create one model that can incorporate both sets of data.

Model	Avg. Residual	Garbage Time Residual
Runs	7.73	7.18
C.L. Runs	7.27	7.27
Total Bases	6.75	6.80
Wins-Adj Runs	6.68	6.25
Wins-Adj C.L. Runs	5.54	5.54
Wins-Adj Bases	5.43	5.53

Table 6.6 details the average residual for the in-season prediction averaged for the years 2015 to 2018.

Model	Avg. Residual	Garbage Time Residual
Runs	8.14	8.29
C.L. Runs	7.69	8.16
Total Bases	6.71	6.50
Wins-Adj Runs	7.09	6.34
Wins-Adj C.L. Runs	5.87	6.98
Wins-Adj Bases	5.40	5.40

Table 6.6 shows the average residual for 2018 for the in-seaon model.

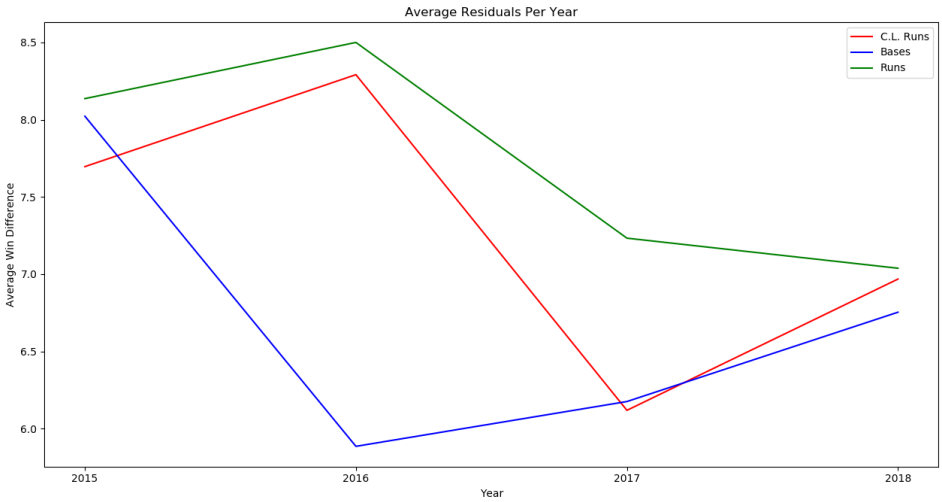

Figure 6.6 shows the average residuals for years for 2015 to 2018 for the regular models.

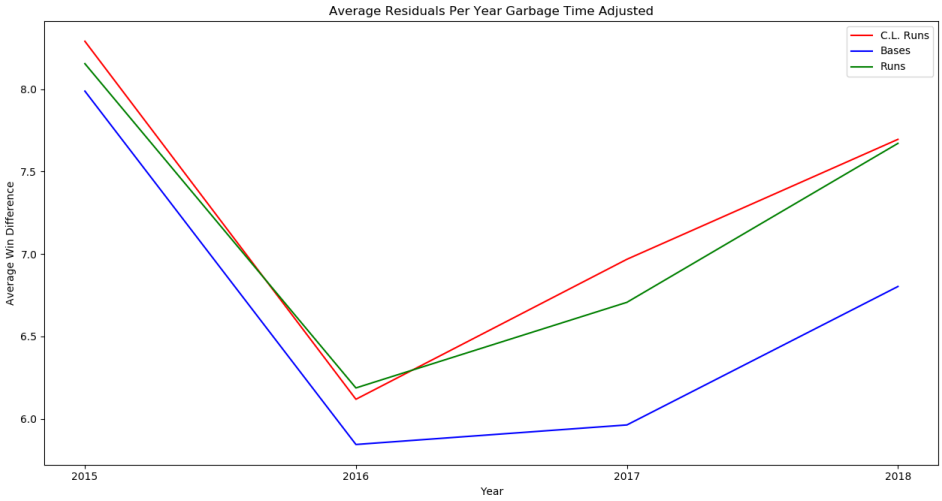

Figure 6.6 shows the average residuals for years for 2015 to 2018 for the garbage-adjusted models.

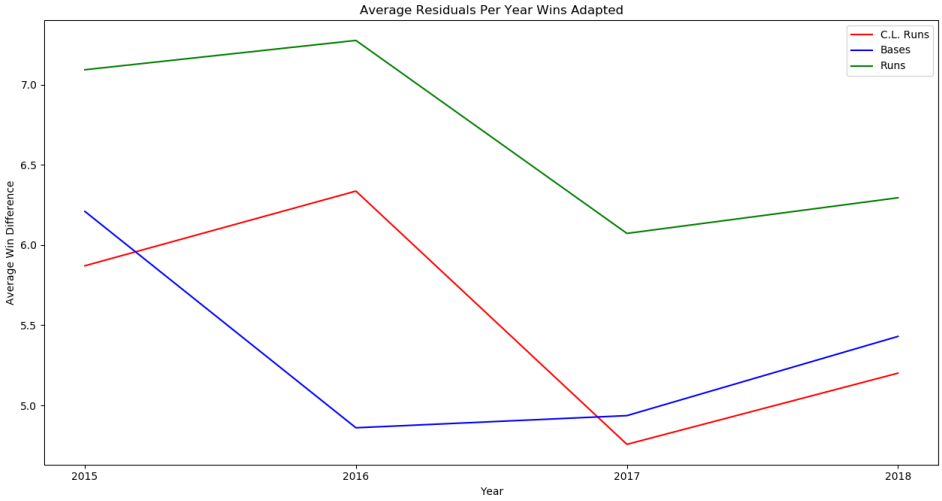

Figure 6.6 shows the average residuals for years for 2015 to 2018 for the wins-adapted models.

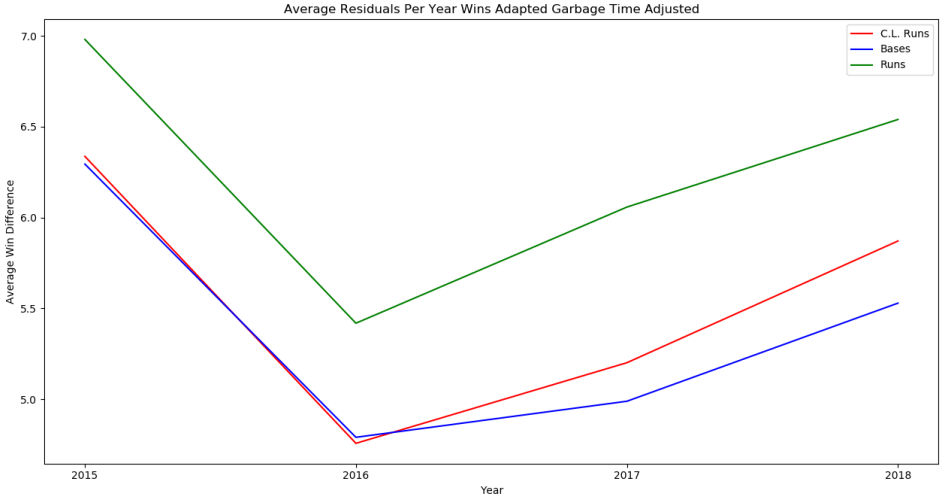

Figure 6.6 shows the average residuals for years for 2015 to 2018 for the wins-adapted garbage time adjusted models.

Chapter 7

Combination of the Two Models

After building the two different models, we needed to explore different ways to combine them optimally. While each method has its benefits, the preseason model is static and cannot take into account player performance throughout the season and the in-season model experiences a great deal of noise and unstable behavior at the beginning of the season. We set out to develop a method that would slowly switch the trust from the preseason model to the in-season model over the course of the beginning of the season, starting with complete trust in the preseason model and by mid-to-late-season having complete trust in the in-season model. In the next section we will detail the different methods we used to combine the two models, and which gave the lowest average residual at each game step.

7.1 Credibility Theory

Credibility Theory is an application from Actuarial Mathematics, typically used to make estimations of interest. The method creates a weighted combination of a prior mean from data or knowledge and a current mean from the most recent data collected. Combined the method creates the compromise estimator. The equation to calculate the compromise estimator is:

$$C = ZR + (1 - Z)H$$

where Z is the credibility factor, R is the mean of the current observations and H is the prior mean. When Z increases, the weighting favors the information from the current observation, thus putting more trust in our current data. It is this equation that we used to apply this idea to our research. We let R be the estimate from the in-season model and H be the estimate from the pre-season model.

For our research we used the Buhlmann's Approach, so the credibility factor Z is defined as:

$$Z = \frac{n}{n + K}$$

where n is the number of independent trials and K is:

$$K = \frac{EVPV}{VHM}$$

where EVPV is the expected value of the hypothetical mean and VHM is the variance of the hypothetical mean. In the context of our research we set n to be the current game. For calculating

K, we needed to determine what the expected value of the process variance and the variance of the hypothetical mean is in our context.

To explain the expectation of the process variance, we need to first define what the process variance is. The process variance is the variance of frequency or differences based on certain criteria. So for baseball, the process variance is the variation in possible wins left to gain in the season. Using the logic that for every team that wins, there has to be one that loses, we can say the process variance in our case is:

$$EVPV = \frac{162 - n}{2}$$

To calculate the variance of the hypothetical mean, we knew that the hypothetical mean in the context of our project would be the average wins of every team, or 81. The variance of this mean, or how far apart the actuals are from the hypothetical mean, would be the standard deviation of the number of wins of each team. To determine the variance, we tested two different statistics, the standard deviation and the variance of team wins at each game step using data from 2015 - 2018. Looking at the graphs below, we can see the slight, but important, difference in the slope and shape of the two different Z values over the seasons. For variance we can see that by game 20, we have 60% of our trust in the in-season model, but by game 20 for standard deviation we only have 40% trust in our in-season model. Both of these methods have pros and cons. The variance puts full trust in the in-season model sooner in the season, but also puts trust in the model even when there tends to be a lot of noise and variability in the estimations. The standard deviation takes a longer time to put full trust in the model, but doesn't put a lot of trust in the in-season model until after the noise reduces.

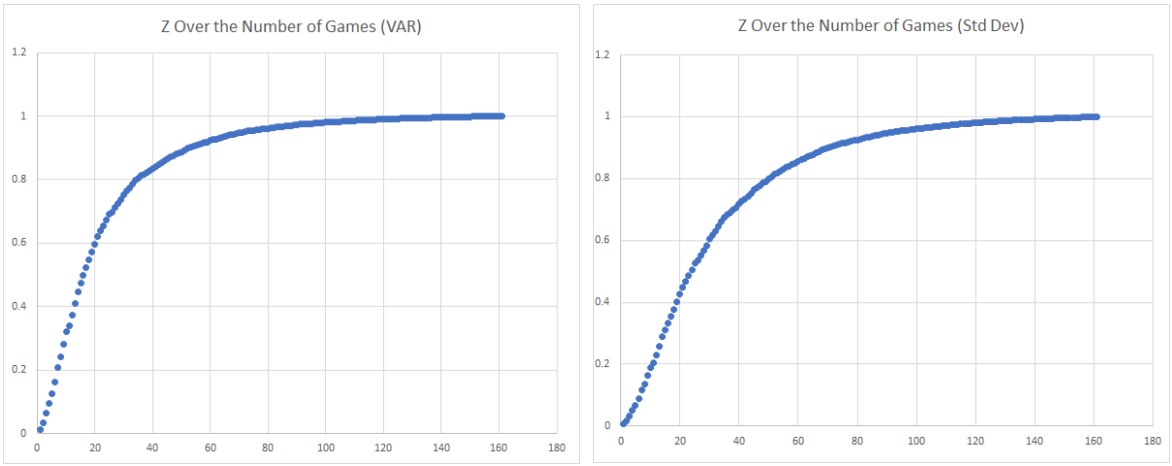

Figure 7.1 shows the credibility crossover Z value when using both variance and standard deviation.

7.2 Linear

A linear model was also developed with the use of the same compromise estimator equation but with a different Z. To create a linear switch, we defined Z as:

$$Z = \frac{(n - start)}{(end - start)}$$

where end and start are specified games to begin and end the crossover. To find the optimal end and start games, we ran this model with 10 different possible starting points (game 35 to 45) and 10 possible ending points (115 to 125). We choose these ranges based on graphs of the in-season model and where the model seemed to converge.

The graphs below show two different examples of Z for two different start and end game combinations.

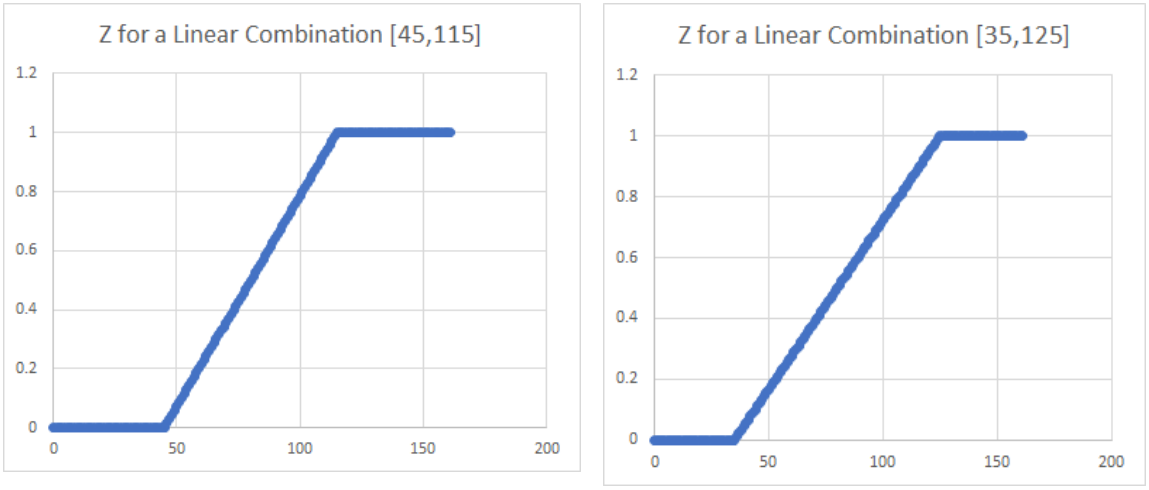

Figure 7.2 shows two examples of the linear crossover functions, one starting at game 45 and ending at 115 and another starting at game 35 and ending at 12.

7.3 Mixed

Our final model is a mix of the linear and credibility methods. In this method, we still use the credibility equation but with a defined start and end date like the linear method. While testing this method, we still used the same range of value to start and end the combination equation. Shown below are two examples of the mixed method. On the left we have the credibility equation with variance in use from games 45 to 125 and on the right, we have the standard deviation with a start game of 35 and end game of 115.

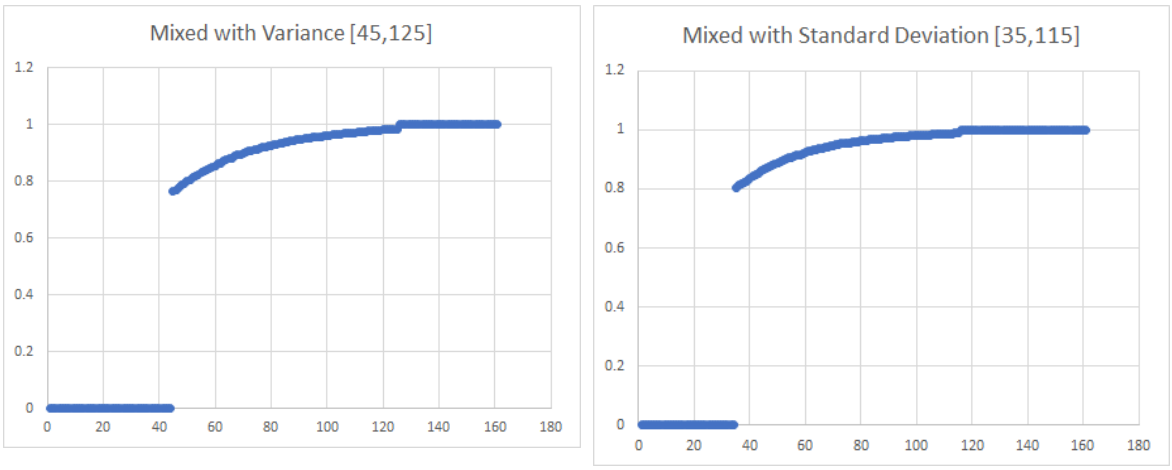

Figure 7.3 shows two examples of the mixed method crossover functions with the same starting and ending games of the linear crossover examples.

7.4 Results

After developing all the methods, we tested them against data from 2015-2018 to see which yielded the lowest average residual. We tested these methods with the regular, non-adapted in-season, as it has the higher residuals, thus more room for improvement. To calculate the average residual used, we calculated the absolute value residual at every game step for every team and averaged all the games for all the teams for all the years tested together. The below table shows the results of each method, with the most optimal start and end dates for those methods that require.

Method	Start and End Game	Avg. Residual
Linear	35, 115	6.003
Credibility (Var)	N/A	5.915
Credibility (Std. Dev.)	N/A	5.522
Mixed (Var)	35, 125	5.973
Mixed (Std. Dev.)	35, 125	5.778

Table 7.4 shows the average residual of the different combination methods.

As we can see from the results, the classic credibility equations with standard deviation had the lowest average residual of roughly a 5.5 game residual, a 3.4% percent difference. The mixed standard deviation method follows closely behind with a 5.8 game residual, showing that standard deviation is the better statistic in this application. This result is not surprising from both the previously discussed Z graph and the idea that standard deviation is not squared like variance, thus its units are games, instead of games-squared, and its magnitude is smaller overall. Now that we have a combined model that updates for every game, we may be able to convert our season predicting model to a more granular model that can bet on every game. A granular game-by-game model will give us the ability to find more opportunities to exploit oddsmakers.

Chapter 8

Single Game Betting

We have created a model that outputs a seasonal winning percentage for every team at any given point in the season. Using credibility theory, we were able to optimize data from the preseason and use the new current season depending where in the season we are trying to predict. With this model that changes for every game, we want to granularize our model to predict individual games rather than total season wins. However, in order to do that there are some adjustments we need to make to convert our model from a season winning percentage to a singular game percentage.

8.1 Starting Pitcher, Relief Pitcher, and Lineup

For our full season model, we use various ways of adjusting the roster's statistics to represent the team. It is important to take into account who is on the roster over the course of the season because eventually everyone on the roster should play. However, when predicting an individual game, we care less about the overall roster and more about the players playing in that game. In baseball, teams announce who the starting pitcher is going to be before the game so that is the information we will want to utilize. That starting pitcher then usually pitches about five to seven innings of the game. Teams rotate through five pitchers that are in their starting rotation for each game, but for single games we want to just look at the starting pitcher for that specific game. To calculate their impact, we utilize the WAR statistic to change the team's runs allowed, which is used to calculate the winning percentage. We do a very similar conversion for offseason transactions highlighted in the preseason section. In order to convert the winning percentage we would need to scale the percentage to be as if that pitcher played in every game all season. For example, we use the WAR the pitcher has for the appearances he made and scale it up to be as if he had played in all 162 games. We then convert that number from wins into our runs allowed we use for that game. Then, we calculate the hitting statistics like BA, OBP, and SLG for the entire lineup calculate the regression from their statistics. Using this method will give us what we believe to be the most accurate winning percentage for the players the team fields that day.

The last adjustment we need to make is for home field advantage. There are plenty of logical reasons we would want to implement this in our single game model. Teams and players are more comfortable playing in their home ball park where they play 81 of 162 games during the year. Additionally, when players need to travel for games their play can be affected for the negative because of the toll of transportation and not staying in their homes. We used empirical probability to determine that the home team has one 54% of the time since 1920. We also checked to make sure

that recently, since 2000, the winning percentage is still 54%. So for singular games we can add 4% for every home team's winning percentage and decrease the away's team by 4%.

8.2 Games

After these adjustments there is a winning percentage for every team for every game. Now when two teams play each other, their winning percentages are changed for the lineups they announce, and we compare the altered winning percentages. Both teams will have a winning percentage that is just for their team and we need to normalize it to pin the teams against each other using the formula:

$$\text{Winning Percentage for Team A} : A/(A+B)$$

We now have our odds for both teams playing in the game that add to one. We compare our odds with the implied odds from the Las Vegas betting book. For example, lets say the Red Sox are favorites at -120. We can convert that into implied probabilities by doing (120/(120+100)) for 54.4% change. If our model had a 60% chance for the Red Sox, because that is higher than 54.5%, we would decide to bet on the Red Sox in our model. To determine which team to bet on, we calculate the expected value of profit using the payoffs given by Vegas and the probabilities our model provides. We assume we have more accurate probabilities than Vegas, so that arbitrage opportunities are revealed. Assuming we bet one dollar on a team, we have the following formula for expected value of profit for betting on any arbitrary Team X.

$$\text{E(Profits from Betting on X)} = \text{[Payout if X Wins]} \cdot \text{P(X Wins} - 1 \cdot (1 - \text{P(X Wins))}$$

Our code analyzes the this formula for each team participating in a game. We choose to bet on a team if their expected value of profit is greater than zero. Note that it is impossible for both teams to have a positive expected value of profit, because of how the odds are made. The Vegas odds are made to be over 100% because that extra money over 100% is where they make some of their profit.

The amount we bet changes based on how much of a difference there exists between the two odds. The bigger the difference, the more confidence we have in our model, thus the more we would want to wager. Below is our betting gradient.

Difference of Bet	Percent of Fund
$X > 0.15$	2%
$0.13 < X < 0.15$	1.5 %
$0.11 < X < 0.13$	1 %
$0.09 < X < 0.11$	0.5 %
$0.06 < X < 0.09$	0.4 %

Table 8.2 shows the breakdown of percent difference of betting odd and corresponding betting percent of fund.

For example, let's say the Red Sox and Yankees were playing, and with our model we have the Red Sox's winning percentage to be 52% and the Yankees's winning percentage at 45%. Using the formula above we would have the Red Sox as favorites with a 54% chance while the Yankees would have 46%. Suppose we found that Vegas money lines have both teams at -105. This translates to implied probabilities of 51% for both teams. This is an example of how the odds makers keep the

excess amount over 100%. With our odds for the Red Sox being higher than the Las Vegas odds, we would bet on the Red Sox. If our expected value indicates we should not bet on a game, it means there is no opportunity for arbitrage.

8.3 Results

We tested our game by game model with the betting algorithm and methods discussed above. We ran it for every year from 2014 to 2018. Below we see the return on the fund and the winning percentage based on if we were correct on the bet.

Year	2014	2015	2016	2017	2018
Betting Record	338-340	336-303	295-339	371-419	328-408
Winning Percentage	49.85%	52.58%	46.53%	46.96%	44.57%
Profit Return	26.42%	25.94%	6.94%	23.88 %	-8.25%

Table 8.3 shows the results of our betting model for the years 2014 to 2018.

Our betting record seems to stay consistently around 47% to 49%. However, we see our percentage drop to 44% and our returns go negative at -8% in 2018. This is obviously a problem, but we can explore different aspects of our model such as looking into our records for different betting gradients and teams. Below we see the amount of games we bet on in each gradient categories. For a reminder, the more our odds were different from Vegas, the more of our fund we bet.

Normally, our model should not be different from Vegas odds very often so that is why the trend is that we bet on less games for the higher percentage of our fund. That seems to be true until we get up to the 2% gradient. We see a massive spike in the number or games that we bet in the 2% gradient compared to the others. That means we have more games where the difference between our odds and the implied probabilities from Vegas are over 15%. This is due to the fact that we are wrong on teams that are on the extreme ends of the standings. The teams that win more than we expect and lose more than we expect are these teams. We continue to bet on these teams even though our model could be wrong on them and that is where the difference comes in for Vegas. We also see a lower win percentage for the 2% than the other areas. It is okay that the win percentage drops as we increase the gradient because in those categories we are usually betting on more underdog games. These are more risky in a sense, but they have a higher payoff when we win, which allows us to be below 50 and still make a profit. However, the 2% win percentage drops off a lot faster than the other gradients, so if we were able to pinpoint teams that we were wrong on sooner and stop betting on them, we would be able to increase our win percentage by decreasing the number of games we bet on.

Percent of Fund	2%	1.5 %	1 %	0.5 %	0.4 %	0.2%
Total Games Bet from 2014 - 2018	457	222	308	392	874	1224
Win Percentage	41.58 %	45.50 %	48.6%	47.45 %	49.66 %	49.59%

Table 8.3 shows the different winning percentages for each of the different betting gradient value.

Here we look at the results specifically from the 2% gradient where we see some big losses earlier on, and then it gets better in the later years. However, this result was still our worst overall from any gradient category in terms of total return because of the massive losses earlier on.

2% Gradient by Year	2014	2015	2016	2017	2018
Betting Record	33-49	30-44	37-43	50-71	40-60
Winning Percentage	40.24%	40.54%	46.25%	41.32%	40.00%
Profit Return	-13.10%	-19.40 %	14.0 %	11.60%	8.80%

Table 8.3 details different year results for betting within the 2% gradient.

Now we will look at the breakdown of underdog and favorite bets. Betting on a favorite is when someone bets on a team when their implied probabilities give a team over a 50% chance of winning.When the team has below a 50% chance, that team is considered an underdog. We see the percentage of underdog versus favorite games bet stay about the same over the years at 73%. This is good to see that consistency because it means that our model assess the teams similarly each year. It also means that the amount of underdog bets is not the reason for our negative return in 2018. That does bring up to the winning percentage of underdog bets. We see an extremely low result in 2018 with 39.64%. This is the first data we found that correlated with our loss in 2018. This makes sense because we were betting on underdogs 76% and getting that bet wrong less than 40% of the time. In 2014 and 2015 we have our best years with underdog winning percentage which drives use to our best two years of profit. This could mean that underdog winning percentage if our best indicator for our model's return.

	2014	2015	2016	2017	2018
Underdog Games Bet	462 (68%)	452 (71%)	468 (74%)	621 (79%)	560 (76%)
Favorite Games Bet	216 (32%)	187 (29%)	166 (26%)	169 (21%)	176 (24%)
Underdog Winning Percentage	48.48%	49.56%	43.38%	43.96%	39.64%
Favorite Winning Percentage	52.78%	59.89%	55.42%	59.17%	60.23%

Table 8.3 shows the breakdown for betting on underdogs and favorites within the betting model.

8.4 Conclusion

Our model ended with an average annual return of 14%, but was still extremely volatile. We will need to wait for 2019 data to become publicly available so we can test our model on that season. These results will be a true test because we used years 2014-2018 to create our model which leaves 2019 as untouched data for us to experiment with. We have found that underdog winning percentage is indicative of our overall returns. Knowing this information will help us to go forward and alter the model to try to be more accurate in these scenarios in the future.

Chapter 9

2019 Testing

In any good algorithm development process it is important to have a training and testing data set for good practice. Throughout this paper we used a training set of data from the 2015-2018 seasons. It was our intention to then test our methods against the 2019 season. We expected to be able to get our data soon after the season ended, but at the time of writing this, it is not the case. Due to this set back, we can only test our preseason methods, discussed below. While it is unfortunate, there is no crying in baseball, so what can you do?

9.1 Preseason Testing

We ran the preseason mode with cluster luck runs regression with ISO and OBP to make our 2019 predictions. Overall, we had an average of 8.51 game residual, a 5.25% error. Our Chicago Cubs prediction had the lowest residual, predicting 84.38 wins in comparison to their 84 win season, a 0.38 residual. We had our highest residual of 24.38 games for the Detroit Tigers. We predicted a season record of 71.38 wins while the Tigers only won 47 games in total. It is important to note the low residual of 1.96 games for the Boston Red Sox. After having an impressively high season in 2018, we were able to understand that the same results were likely not to happen in 2019 and not overfit to excess of wins.

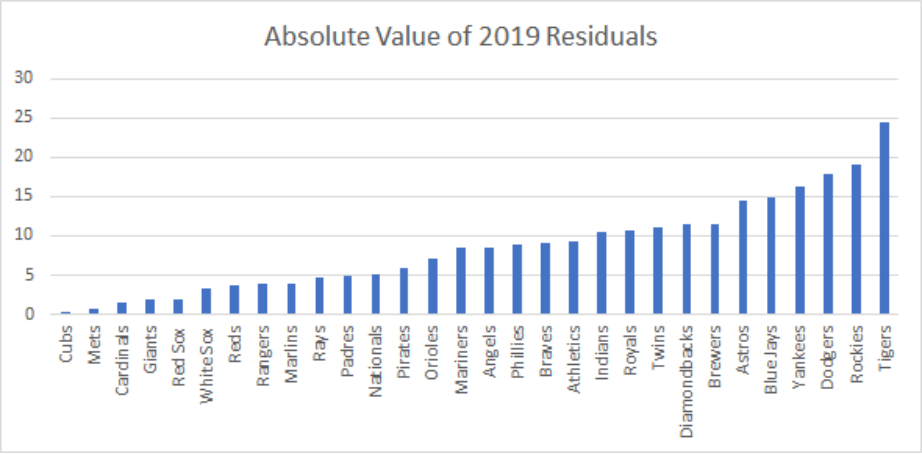

Figure 9.1 shows the residuals by team for our 2019 preseason predictions.

For our preseason predictions, we wanted to compare our results to that of Nate Silver's PECOTA. Silver had an average residual of 7.7 games, only 0.81 games lower than our predictions. Silver also shares the same minimum and maximum residual teams, having a 0.035 and a 20.97 game residual for the Cubs and Tigers respectfully. While we share the same max and min residuals, we can see that in the chart below our team residuals varry. Silver had high residuals for teams like the Mariners Rangers, and Rays where we had higher residuals for teams like Braves, Dodgers, and Yankees. Silver was not as successful in his prediction for the Red Sox, with a 10.96 game residual. Silver most likely fit to the extreme results of 2018, thus skewing his results.

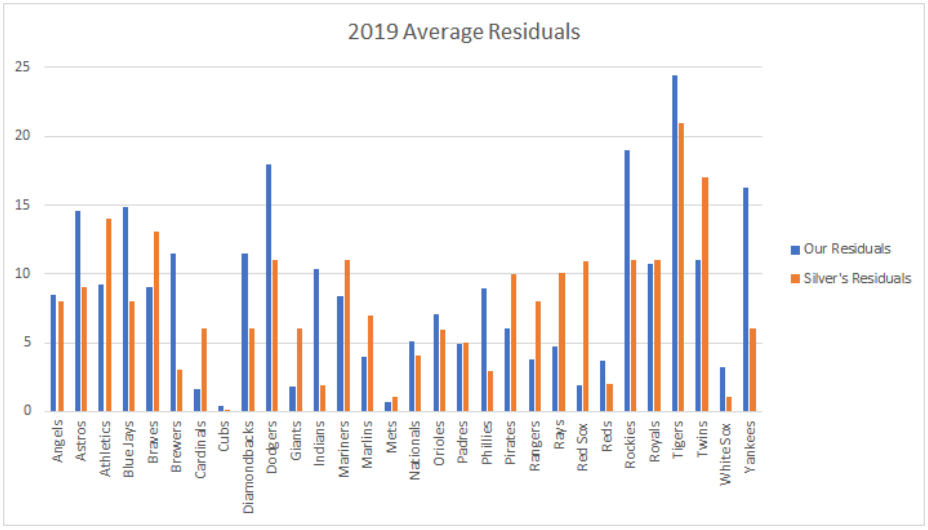

Figure 9.1 shows the residuals for both our methods (blue) and Nate Silvers (orange) preseason prediction models.

9.2 Conclusion

Overall, we are very happy with our preseason results. Only using historical data, we were able to predict teams full season records within 8.51 games. Comparing our results to Nate Silver, we were not far behind his PECOTA projections that he has been using for much longer. We hope to also test our in-season and combined models on the 2019 data once published and have the same success in results as the preseason.

Chapter 10

Conclusion

Although we explored many different paths for the project, there still is so much more that we wish we could have done. We had thoughts of exploring more baseball statistics. There is a vast range of different metrics that could all have been testing in our methods.There were some statistics that we could not use because we could not get them for both a hitter and pitcher's perspective. With more time we could have set up ways to some statistics for both positions.

We would also like to explore the idea of seasonal garbage time. In our cluster luck adjustments for seasonal wins, we notice that teams that have positive cluster luck are almost always the teams that perform very well, and teams that have negative cluster luck usually perform much worse. We believe this is caused by player mentality changing during the season if their team has lost too much, or if their team wins most of their games. A player could be encouraged to try harder, or to give up depending on their team's record. Adjusting this could result in our model being more accurate on the teams we are least accurate for. Therefore, a change in those team's residuals could lead in a significant drop of our overall residuals. Seasonal garbage time could also be created by team's making trades towards the end of a season. If a team is out of the playoffs, they could begin signing on new players with less experience, and giving away their more reliable players. By quantifying seasonal garbage time and adding it to our model, we could account for the cluster luck anomalies.

Data science and machine learning methods are all the rage these days with their great performance on a wide range of subjects and problems. Classification algorithms, such as k-nearest neighbors, could have been used to determine individual game wins and regression algorithms, such as support vector machines, could have been used to calculate win percentages. Neural Networks are another strong tool that could be used in this subject. Neural Networks have the ability to process and use a large amount of inputs and give highly accurate results.

Overall, we were able to develop many different ways to analyze and quantify baseball. We were able to describe the effects of team luck through cluster luck using regression models to appropriately describe sustainability. Using historical data, we were able to calculate empirical winning odds for any game situation in baseball. We then quantified true player talent by removing times of high variability through our garbage time definition. We built a prediction model using only preseason data, incorporating both injuries and roster changes. Our preseason model was able to predict full season wins, with an average residual of 8.35 games over 2015 to 2018. Using in-season data we build methods off of Bill James' Pythagorean formula that was able to predict full season games with an average residual as low as 5.36 games over 2015 to 2018. Lastly, we created a game by game predicting and gambling model that gave us a 93.52% return over four years, which converts to an average annual return of 14.12%.

Acknowledgements

We would like to thank those statisticians that came before us, from which we drew inspiration. This includes Bill James, Nate Silver, and Joe Peta among others. Thank you to Retrosheet for the majority of our data. Thank you to the WPI Mathematics Department for their support. Lastly, we would not have been able to complete this project without the help of Professors Jon Abraham and Barry Posterro. Thank you both for your guidance and support.

Appendix A

Database Technical Details

A.1 Design Decisions

Before building our database and populating it with data, our group had to make decisions on how we would design and implement it to maximize our productivity. These decisions were made carefully, as they could make a big difference in how easy it is to retrieve data.

A.1.1 Choosing SQLite and SQL Alchemy

Many databases (such as MySQL or PostgreSQL) are built for scalability in users, meaning that they are built to be accessed simultaneously by multiple users, each querying the database in various ways. For example, when you access a website, a database is often queried, and the results are sent back to your web browser and formatted properly. Thousands of users can use a website at once, so the underlying database is built for multiple queries at once. This feature is unnecessary in our project, since we will never be querying the database simultaneously. For our purposes, we decided that SQLite is the best choice. SQLite is a relational database that is stored with the application using it, rather than on a remote server. For this reason, it can often be much faster, as long as only a single user is working with it. We expected to have millions of records of data, so database performance is a significant concern. SQLite is often used by web browser applications to hold local data such as browsing history and bookmarks due to its speed.

SQL Alchemy is a Python library for object-relational mapping. Normally, when a database is queried, the data returned are records of the database, represented as tuples. With an object-relational mapper, the data returned is in the form of objects, making it easier to understand the code, and allowing us to code faster.

Given a table called GameLog which holds information about the outcome of MLB games, we can write code with and without SQL Alchemy to analyze games that were played in July of 2018.

```
1  # Example 1: Without SQL Alchemy
2  import sqlite3
3  conn = sqlite3.connect(DB_PATH)
4  q = conn.execute('''SELECT *
5                      FROM GAME_LOG
6                      WHERE YEAR = 2018 AND MONTH = 7;''')
7  rows = q.fetchall()
```

```
8       first_game = rows[0]
9       print(first_game[0])  # First column is the game ID
```

In the first example, we query the database using SQLite. We establish a connection, and have the connection execute some SQL code. The results are returned as records, which is a list of tuples. Each tuple represents a record. The first column in the GameLog table is the game ID, which we print out.

Without SQL Alchemy, our group would have to write SQL inside of a Python file, which can cause files to be long and difficult to read.

.

```
1       # Example 2: With SQL Alchemy
2       from db.models import get_session, GameLog
3       sess = get_session()
4       q = GameLog.query\
5                   .filter(GameLog.year == 2018)\
6                   .filter(GameLog.month == 7)
7       first_game = q.first()
8       print(first_game.game_id)
```

In the second example, we query the database using SQL Alchemy as our object-relational mapper. We can avoid writing an SQL code inside our Python script, allowing us to write code faster, and make it more readable. Each item returned has equivalent data to the records, but they are converted to be in the form of an object instead of a tuple. The first game ID is printed out.

Over time, we found that using SQL Alchemy was about twice as slow as raw SQL queries. This is because in Python, creating millions of instances of classes can be very slow, and SQL Alchemy relies on classes. Usually this difference is not noticeable, but our work involves querying for massive amounts of data. Because of this performance difference, we implemented features in our code to allow us to query faster, while retrieving less data. For example, a common query we made to the database was retrieving the play-by-play data for a given game ID. We factored out these repeated queries into one function that we could repeatedly call instead. The original function code is below.

.

```
1       '''
2       Returns the plays (in order) for a given game_id (a string).
3       To see the fields for the atbat objects, look at the PlayByPlay
4       class in db/models.py
5       '''
6       def step_through_game_play_by_play(game_id):
7           q = (PlayByPlay.query
8                           .filter(PlayByPlay.game_id == game_id)
9                           .order_by(PlayByPlay.play_index))
10          return q
```

After using this function multiple times and struggling with performance issues, we implemented a way to retrieve only certain columns from the table, by adding a "fields" argument to the function. The fields argument is a list of fields to retrieve from the database. We can use this argument if we need to speed up our queries, however in most cases this is unnecessary. Including this argument allows us to increase our code reusability. We found that only querying for certain columns can cut runtime in half.

.

```
    '''
    Returns the plays (in order) for a given game_id (a string).
    To see the fields for the atbat objects, look at the PlayByPlay
    class in db/models.py
    '''
    def step_through_game_play_by_play(game_id, fields=None):
        q = (PlayByPlay.query
                        .filter(PlayByPlay.game_id == game_id)
                        .order_by(PlayByPlay.play_index))
        if fields is not None:
            q = q.with_entities(*fields)
        return q
```

Our team used SQLite and SQL Alchemy to maximize our productivity. SQLite allows us to quickly retrieve data and avoid the use of a database server. SQL Alchemy allows us write all of our queries in Python, and retrieve our results as objects instead of records.

A.1.2 Table Design and Code Reusability

Our table design is highly influenced by Retrosheet, which is where the majority of our data is from. We created a table containing yearly data for each team, since the team's name and city could change from year to year. Additionally, we have created tables for each player, their team and their statistics for each season. We chose to split traditional statistics and garbage-adjusted statistics into separate tables, even though they could be grouped together. Separating these groups of statistics into two tables with identical column names allows us to easily write functions that will include or exclude garbage time depending on an input.

For example, below is part of a function that calculates cluster luck adjusted runs. It accepts the years used to calculate the regression, an argument to determine whether or not to include garbage time, and a function to retrieve the desired regressors from an entry of a statistics table.

.

```
    '''
    Returns the cluster luck-adjusted runs hat scored and against
    for all teams for given years. If include_gb=False, the
    non-garbage runs will be regressed on the non-garbage stats.
    The x_evaluator function accepts an entry from the TeamBattingStats
    and TeamPitchingStats, and returns the x values to use for the regression
    If x_evaluator is None, the default x values used are BA, SLG, and OBP.
    '''
    def get_runs_hat(years, include_gb=False, x_evaluator=None):
        if type(years) == int:  # one year was given
            years = [years]  # make it into a list
        years = list(years)

        batting_stats = []
        runs_for = []
        team_ids = []

        if include_gb:
```

```
19              BattingTable = TeamBattingStats
20          else:
21              BattingTable = NonGbgTeamBattingStats
22
23          for b_stat in BattingTable.query.filter(BattingTable.year.in_(years)):
24              if x_evaluator is None:
25                  batting_stats.append([b_stat.batting_avg,
26                                        b_stat.slugging,
27                                        b_stat.on_base_perc])
28              else:
29                  batting_stats.append(x_evaluator(b_stat))
30          ...
```

This function can serve many purposes because of how easy it is to switch from using traditional statistics to garbage-adjusted statistics. To exclude any garbage time plays, the keyword argument "include_gb" must be set to false, no other changes are necessary. This is possible and easy to implement because of our table design.

The argument "$x_evaluator$" $allows the independent variables for the cluster luck regression to be changed easily. For exa$ $BasePercentage statistics, which was created by reusing the \get_runs_hat" function.$

.

```
1
2       def get_runs_hat_ISO_OBP(years, include_gb=False):
3           def ISO_OBP_evaluator(b_stat):
4               iso = b_stat.slugging - b_stat.batting_avg
5               obp = b_stat.on_base_perc
6               return [iso, obp]
7           return get_runs_hat(years, include_gb, ISO_OBP_evaluator)
```

By designing our functions to be dynamic and have a broad set of capabilities, our group was able to maximize code reusability, and increase productivity.

A.2 Retrieving and Cleaning Data

A.2.1 Retrosheet

Game logs and play-by-play files were downloaded from the Retrosheet website. Additional fields, derived directly from the original Retrosheet fields given, were required to complete our work. The Chadwick [1] play-by-play tools allowed us to expand the fields in our Retrosheet data to give us exactly what we need. Code was written to call these command line tools from Python, using four processes at once to speed up the work. Only six fields were provided by Retrosheet originally, but the Chadwick tools expanded these to be more than 150 fields. Retrosheet provides a large amount of incredibly detailed and accurate data, so data cleaning was rarely necessary.

A.2.2 MLB.com Rosters

Our work required game-by-game rosters, which are not provided by Retrosheet. The information was found through searching online Sabermetrics forums, which eventually led to MLB.com. Al-

[1] https://github.com/chadwickbureau/chadwick

though not advertised, the website provides the names of each player[2] on each team for every MLB game in recent years.

After discovering the data, we wrote a web crawler to download the rosters for every game available. The crawler accesses and reads pages on MLB.com to find every possible game roster available, and then downloads the rosters. Hours of debugging time was put into this crawler due to peculiar behavior by the MLB website, including returning an error page if a forward slash is placed at the end of a link. Due to the massive amount of games played each year, it was necessary to use 16 simultaneous threads to download the game rosters in a reasonable timeframe. Close to 40,000 game rosters were downloaded, each as a separate XML file.

MLB.com and Retrosheet use different game and player IDs. We choose to only use Retrosheet IDs since the majority of our data is from Retrosheet. Some conversions are provided by Chadwick, but only for players. To insert the rosters into our database, we wrote code to parse each of the 40,000 game rosters and deduce which Retrosheet game ID each MLB.com game ID. Our deduction method compared the IDs of the players for games that were played on the same date, with the same game number.

This method of deduction required days worth of tuning, analysis and optimization. Tuning and analysis was necessary to ensure the accuracy of our results, since there are many small differences between the way Retrosheet and MLB.com handle their data. For example, Retrosheet indexes their game numbers from zero, while MLB.com game numbers start at one. This is an easy difference to overlook, which could result in every first game being identified as the second game in a double header. Optimization was required because querying our database multiple times for each of the 40,000 rosters takes hours. However, we were eventually successful in downloading and converting all available rosters.

A.2.3 CBS Sports Injury Reports

Our team needed a way to account for injuries in our preseason model. We wrote code to scrape CBS Sports Injury Reports using Python's lxml package. The expected return date for each player is written in the form of a sentence, meaning we had to write code to analyze the sentences and obtain an approximate end date.

.

```
1       '''
2       Takes an injury status description from CBS Sports, and
3       returns the expected date of return. If the player is out
4       for the season, returns December 1st of the current year.
5       '''
6       def interpret_injury_status(injury_status):
7           current_year = datetime.now().year
8           # Standardize the string so it is easily comparable.
9           injury_status = injury_status.strip().lower()
10          # If the injury status begins with any of these, then there will
11          # be a date we can read at the end of the string
12          if injury_status.startswith('expected to be out until at least ')\
13              or injury_status.startswith('probable for ')\
14              or injury_status.startswith('suspended until '):
15              # Month as a string
```

[2] http://gd2.mlb.com/components/game/mlb/year$_2$018/$month_0$7/$day_2$2/$gid_2$018$_0$7$_2$2$_a$tlmlb$_w$asmlb$_1$/players.xml

```
16              month_name = injury_status.split()[-2]
17              # Convert month name to an integer
18              month_int = month_name_to_month_int[month_name]
19              day = int(injury_status.split()[-1])
20              return current_year, month_int, day
21          elif injury_status == 'out for the season':
22              # Out for the season, so return a day past the end of the
23              # baseball season (December 1st is fine)
24              month = 12
25              day = 1
26              return current_year, month, day
27          else:
28              # Unrecognized injury status. This means we should change
29              # the code to be able to identify this case too.
30              raise Exception('Unidentified injury status : ' + injury_status)
```

An injury status starting with the words "Expected to be out until at least", "Probably for", or "Suspended until" indicates that there is a date later in the sentence, which we can extract and return. An injury status stating the player is "out for the season" means the player will miss all games in the season. In this case, we return December 1st as the return date, since this date is after the baseball season is over.

Once the expected date is found, the number of games for the team that occur after that date is calculated by querying the team Schedule table in the database, which contains data on the dates of the games a team is expected to play in a season.

.

```
1       '''
2       Accepts a date tuple as (year, month, day) and a team ID and
3       returns how many games that team had left to play in the season.
4       '''
5       def date_to_num_games_left_in_season(date, team):
6           year, month, day = date
7           date_str = '{:02}{:02}{:02}'.format(year, month, day)
8           q = (Schedule.query.filter((Schedule.home_team == team) |
9                                      (Schedule.visiting_team == team))
10                              .filter(Schedule.year == year)
11                              .filter(Schedule.game_date >= date_str))
12          return q.count()
```

The date is converted into a string that is directly comparable to Retrosheet game date strings. We add three filters to our query of the Schedule table. The first is to make sure one of the teams playing the game is the team that we are looking for. The second filter is to check that the years are the same, so that the schedules for other seasons are not included in the query. The third is to make sure the game date is further into the future or equal to the expected injured players return date. We then return how many scheduled games are in this query.

A.3 Using the Database

We have created an infrastructure that allows us to quickly query for player statistics, or retrieve information about a particular game or play for any year in baseball since 1921. For example, if you wanted the batting statistics for Mookie Betts in 2018, the following code will print out his statistics.

```
from db.models import PlayerBattingStats, get_session

sess = get_session()

q = (PlayerBattingStats.query
     .filter(PlayerBattingStats.player_id == 'bettm001')
     .filter(PlayerBattingStats.year == 2018))
print(q.first())
```

This code provides the following output, which can be confirmed by referencing Betts' Baseball Reference[3] page. Below is the output of this script.

```
  PA      AB      H     2B   3B    HR    BB       BA        OBP       SLG
14.0   520.0   180.0  47.0  5.0  32.0  81.0  0.346154  0.438111  0.640385
```

Garbage-adjusted statistics can be fetched easily, by changing the name of the table in the example above to "NonGbgPlayerBattingStats".

More specific data is available. For example, if you wanted to know who the batter in the tenth play of the fifteenth game that the Red Sox played in 2016, that information can be retrieved using the code below.

```
from db.models import get_session
from db.helpers import get_team_season_games, step_through_game_play_by_play, get_

sess = get_session()

# Retrieve all game logs for Red Sox in 2016
red_sox_games = get_team_season_games('BOS', 2016)
# Find the fifteenth game, and retrieve the plays for that game
fifteenth_game = red_sox_games[14]  # 0-indexed
plays = step_through_game_play_by_play(fifteenth_game.game_id)
# Find the tenth play, and print the batter's name
tenth_play = plays[9]  # 0-indexed
# The batter attribute is an ID, so convert to a name
print(get_player_name(tenth_play.batter))
```

The Red Sox's fifteenth game in 2016 was against the Tampa Bay Rays on April 21st[4], where the Rays won twelve to eight. The batter in the tenth play was David Ortiz, which is correctly printed out.

[3] https://www.baseball-reference.com/players/b/bettsmo01.shtml
[4] https://www.baseball-reference.com/boxes/BOS/BOS201604210.shtml

This code is concise because of its use of helper functions. Our group has developed a set of helper functions to factor out common queries into reusable functions. For example, querying the database for a players name is a common task, but rewriting the query multiple times is slow, so it was converted into a helper function. This allows us to do many tasks (such as the one above) without writing any queries to the database.

Through the use of SQLite and SQL Alchemy, a well designed structure, and multiple types of data collections, we were able to build a vast database to support our research. Overall, we have developed a reliable infrastructure for our work that allows for fast development and analysis.

A.4 Full Table List

A.4.1 GameLog

game_id: String, Primary Key
- game_date: String
- game_number: Integer
- day_of_week: String
- visiting_team: String
- visiting_team_league: String
- visiting_team_game_number: Integer
- home_team: String
- home_league: String
- home_team_game_number: Integer
- visiting_team_score: Integer
- home_team_score: Integer
- game_length_in_outs: Integer
- day_or_night: String
- year: Integer
- month: Integer
- day: Integer
- winning_team: String
- winner_int: Integer

A.4.2 PlayByPlay

game_id: String, Primary Key
- play_index: Integer, Primary Key
- visiting_team: String
- inning: Integer
- batting_team: Integer
- outs: Integer
- balls: Integer
- strikes: Integer
- pitch_sequence: String
- visitor_score: Integer
- home_score: Integer
- batter: String
- batter_hand: String

result_batter: String
result_batter_hand: String
pitcher: String
pitcher_hand: String
result_pitcher: String
result_pitcher_hand: String
catcher: String
first_baseman: String
second_baseman: String
third_baseman: String
shortstop: String
left_fielder: String
center_fielder: String
right_fielder: String
runner_on_first: String
runner_on_second: String
runner_on_third: String
event_text: String
leadoff_flag: String
pinch_hit_flag: String
defensive_position: String
lineup_position: String
event_type: Integer
batter_event_flag: String
official_time_at_bat_flag: String
hit_value: Integer
sacrifice_hit_flag: String
sacrifice_fly_flag: String
outs_on_play: String
double_play_flag: String
triple_play_flag: String
RBI_on_play: String
wild_pitch_flag: String
passed_ball_flag: String
fielded_by: String
batted_ball_type: String
bunt_flag: String
foul_flag: String
hit_location: String
number_of_errors: Integer
first_error_player: Integer
first_error_type: String
second_error_player: Integer
second_error_type: String
third_error_player: Integer
third_error_type: String
batter_destination: Integer
runner_on_first_destination: Integer

runner_on_second_destination: Integer
runner_on_third_destination: Integer
play_on_batter: Integer
play_on_runner_on_first: Integer
play_on_runner_on_second: Integer
play_on_runner_on_third: Integer
stolen_base_for_runner_on_first: String
stolen_base_for_runner_on_second: String
stolen_base_for_runner_on_third: String
caught_stealing_for_runner_on_first: String
caught_stealing_for_runner_on_second: String
caught_stealing_for_runner_on_third: String
pickoff_of_runner_on_first: String
pickoff_of_runner_on_second: String
pickoff_of_runner_on_third: String
pitcher_charged_with_runner_on_first: String
pitcher_charged_with_runner_on_second: String
pitcher_charged_with_runner_on_third: String
new_game_flag: String
end_game_flag: String
pinch__runner_on_first: String
pinch__runner_on_second: String
pinch__runner_on_third: String
runner_removed_for_pinch__runner_on_first: String
runner_removed_for_pinch__runner_on_second: String
runner_removed_for_pinch__runner_on_third: String
batter_removed_for_pinch__hitter: String
position_of_batter_removed_for_pinch__hitter: Integer
fielder_with_first_putout: Integer
fielder_with_second_putout: Integer
fielder_with_third_putout: Integer
fielder_with_first_assist: Integer
fielder_with_second_assist: Integer
fielder_with_third_assist: Integer
fielder_with_fourth_assist: Integer
fielder_with_fifth_assist: Integer
event_number: Integer
home_team_id: String
batting_team_id: String
pitching_team_id: String
fielding_team_id: String
half_inning: Integer
start_of_half_inning_flag: String
end_of_half_inning_flag: String
score_for_team_on_offense: Integer
score_for_team_on_defense: Integer
runs_scored_in_this_half_inning: Integer
number_of_plate_appearances_in_game_for_team_on_offense: Integer

number_of_plate_appearances_in_inning_for_team_on_offense: Integer
start_of_plate_appearance_flag: String
truncated_plate_appearance_flag: String
base_state_at_start_of_play: Integer
base_state_at_end_of_play: Integer
batter_is_starter_flag: String
result_batter_is_starter_flag: String
id_of_the_batter_on_deck: String
id_of_the_batter_in_the_hold: String
pitcher_is_starter_flag: String
result_pitcher_is_starter_flag: String
defensive_position_of_runner_on_first: Integer
lineup_position_of_runner_on_first: Integer
event_number_on_which_runner_on_first_reached_base: Integer
defensive_position_of_runner_on_second: Integer
lineup_position_of_runner_on_second: Integer
event_number_on_which_runner_on_second_reached_base: Integer
defensive_position_of_runner_on_third: Integer
lineup_position_of_runner_on_third: Integer
event_number_on_which_runner_on_third_reached_base: Integer
responsible_catcher_for_runner_on_first: String
responsible_catcher_for_runner_on_second: String
responsible_catcher_for_runner_on_third: String
number_of_balls_in_plate_appearance: Integer
number_of_called_balls_in_plate_appearance: Integer
number_of_intentional_balls_in_plate_appearance: Integer
number_of_pitchouts_in_plate_appearance: Integer
number_of_pitches_hitting_batter_in_plate_appearance: Integer
number_of_other_balls_in_plate_appearance: Integer
number_of_strikes_in_plate_appearance: Integer
number_of_called_strikes_in_plate_appearance: Integer
number_of_swinging_strikes_in_plate_appearance: Integer
number_of_foul_balls_in_plate_appearance: Integer
number_of_balls_in_play_in_plate_appearance: Integer
number_of_other_strikes_in_plate_appearance: Integer
number_of_runs_on_play: Integer
id_of_player_fielding_batted_ball: String
force_play_at_second_flag: String
force_play_at_third_flag: String
force_play_at_home_flag: String
batter_safe_on_error_flag: String
fate_of_batter: Integer
fate_of_runner_on_first: Integer
fate_of_runner_on_second: Integer
fate_of_runner_on_third: Integer
runs_scored_in_half_inning_after_this_event: Integer
fielder_with_sixth_assist: Integer
fielder_with_seventh_assist: Integer

fielder_with_eighth_assist: Integer
fielder_with_ninth_assist: Integer
fielder_with_tenth_assist: Integer
unknown_fielding_credit_flag: String
uncertain_play_flag: String
year: Integer

TeamInfo

year: Integer, Primary Key
team_id: String, Primary Key
league: String
city: String
team_name: String
team_city_and_name: String

A.4.3 SeasonalRoster

team_id: String, Primary Key
year: Integer, Primary Key
player_id: String, Primary Key
last_name: String
first_name: String

A.4.4 StartingLineup

game_id: String, Primary Key
team_int: Integer, Primary Key
lineup_num: Integer, Primary Key
batter_id: String

A.4.5 TeamStats

team_id: String, Primary Key
year: Integer, Primary Key
wins: Integer
losses: Integer
win_perc: Float
runs_for: Integer
runs_against: Integer
bases_for: Integer
bases_against: Integer
nongb_runs_for: Integer
nongb_runs_against: Integer
nongb_bases_for: Integer
nongb_bases_against: Integer

A.4.6 TeamBattingStats

team_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer
walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.7 TeamPitchingStats

team_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer
walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.8 NonGbgTeamBattingStats

team_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer

walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.9 NonGbgTeamPitchingStats

team_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer
walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.10 PlayerBattingStats

player_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer
walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.11 PlayerPitchingStats

player_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer
walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.12 NonGbgPlayerBattingStats

player_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer
walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.13 NonGbgPlayerPitchingStats

player_id: String, Primary Key
year: Integer, Primary Key
plate_apps: Integer
at_bats: Integer
hits: Integer
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer

walks: Integer
hit_by_pitch: Integer
sacrif_flies: Integer
on_base_perc: Float
slugging: Float
on_base_plus_slg: Float
batting_avg: Float

A.4.14 Game_State_Counts_By_Year

full_key: String, Primary Key
home_score: Integer
visitor_score: Integer
base_first: Boolean
base_second: Boolean
base_third: Boolean
inning: Integer
batting_team: Integer
outs: Integer
year: Integer
wins: Integer
losses: Integer
comparison_key: String

A.4.15 Game_State_Counts

comparison_key: String, Primary Key
wins: Integer
losses: Integer
sample_size: Integer
chance_winning: Float

A.4.16 RosterByGame

game_id: String, Primary Key
player_id: String, Primary Key
team_int: Integer

A.4.17 Schedule

game_id: String
game_date: String
game_number: Integer
year: Integer, Primary Key
month: Integer
day: Integer
day_of_week: String
visiting_team: String, Primary Key

visiting_league: String
visiting_season_game_num: Integer, Primary Key
home_team: String, Primary Key
home_league: String
home_season_game_num: Integer, Primary Key
time_of_day: String
makeup_date: String

A.4.18 BettingOdds

game_id: String, Primary Key
visiting_team: String
home_team: String
visiting_odds: Float
home_odds: Float

A.4.19 PecotaBatting

player_id: String, Primary Key
proj_year: Integer, Primary Key
singles: Integer
doubles: Integer
triples: Integer
home_runs: Integer
at_bats: Integer
plate_apps: Integer
batting_avg: Float
walks: Integer
caught_stealing: Integer
hits: Integer
hit_by_pitch: Integer
rbi: Float
stolen_bases: Integer
sacrifice_flies: Integer
sacrifice_hits: Integer
slugging: Float
strike_outs: Integer
total_bases: Integer
war: Float

A.4.20 PecotaPitching

player_id: String, Primary Key
proj_year: Integer, Primary Key
walks: Integer
era: Float
games: Integer
hits: Integer

hit_by_pitch: Integer
home_*Integer*
losses: Integer
strike_outs: Integer
throws: Integer
whip: Float
war: Float