BRIDGING THE GAP BETWEEN BASEBALL AND SOFTBALL ANALYTICS

A THESIS

Presented to the Department of Applied Statistics

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Applied Statistics

Committee Members:

Kagba N. Suaray, Ph.D. (Chair)
Seungjoon Lee, Ph.D.
Hojin Moon, Ph.D.
Roger Kirk

College Designee:

William Murray, Ph.D.

By Teah Thies

B.S., 2022, University of California, Santa Barbara

December 2025

ABSTRACT

This thesis bridges the gap between baseball and softball analytics by applying established baseball data and modeling techniques to the Cal State Long Beach Division I softball team. Using logistic regression, decision tree, random forest, and domain adaptation models optimized by F1 score, the analysis identified pitching strategies to reduce hits, on-base percentage, walks, and home runs while increasing strikeouts. Among the majority of outcomes, domain adaptation models—which transferred knowledge from Major League Baseball data—improved performance across most categories, demonstrating that despite mechanical pitching differences, baseball and softball pitches influence hitters in comparable ways. The best overall model was the walks domain adaptation model (F1 = 0.73), which effectively identified conditions leading to walks, highlighting the impact of situational pressure and pitch location. These findings show that leveraging baseball data can accelerate the development of softball analytics, and as softball data becomes more detailed and consistent, predictive models will continue to mature and enhance data-driven coaching and game strategy.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to the Long Beach Softball coaching staff—Coach Kim Sowder, Panita Thanatharn, Kevin Castillo, and Mariah Ramirez. When I transferred in 2023 as a player, you welcomed me with open arms. In 2024, you allowed me to finish my studies while still being a part of the program as a manager. This thesis is my heartfelt thank you for allowing me to finish my softball career with the best possible memories and experiences. Happy retirement, Coach Kim!

In addition, I would like to offer a special thank you to my advisor, Dr. Kagba Suaray, whose guidance was invaluable throughout this research. Your support and encouragement made the process as smooth as possible, and you continuously pushed me to challenge myself both in the classroom and in the development of this paper. You have been an absolute pleasure to work with, and I am deeply grateful for your mentorship throughout the entire program. I would also like to thank my committee members, Dr. Seungjoon Lee, Dr. Hojin Moon, and Roger Kirk, for your feedback and contributions.

All of this couldn't have been possible without my parents Jeff and Laura Thies. From driving me to countless practices and games to encouraging me to follow my passions and pursue the paths I choose, you have always been my biggest supporters. Your belief in me has given me the freedom to grow and find my own way in life. I want to extend my deepest appreciation to all my past teammates, especially my sisters—Cali, Riley, and Lilah Thies. The fun, laughter and memories we share made the journey just as meaningful as the destination.

Finally, thank you goes to all the women athletes who came before me and fought tirelessly for equal playing opportunities. Your perseverance has paved the way for so many, and your legacy continues to inspire me and countless others.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

LBSU	Long Beach State University
WCWS	Women's College World Series
ASU	Arizona State University
CWS	College World Series
NCAA	National Collegiate Athletic Association
AIAW	Association for Intercollegiate Athletics for Women
MLB	Major League Baseball
AU	Athletes Unlimited
SO	Strikeout
BB	Walk
HR	Home Runs
BAA	Batting Average Against
ERA	Earned Run Average
OBP	On Base Percentage
BA	Batting Average
NBA	National Basketball League
NFL	National Football League
NHL	National Hockey League
wOBA	Weighted On Base Average
SLG	Slugging Percentage
EDA	Exploratory Data Analysis
RHH	Right-Handed Hitters

LHH	Left-Handed Hitters
SEE	Sum of Squared Errors
MLE	Maximum Likelihood Estimation
AIC	Akaike Information Criterion
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
AUC	Area Under the Curve

CHAPTER ONE

INTRODUCTION

In the age of data-driven decision-making, baseball has long led the way in utilizing advanced analytics to optimize team performance. This highlights a significant opportunity to expand research in softball, particularly in pitching. It is found that "[P]itching-related softball articles are nearly 8 time less frequent compared to baseball pitching articles and are published in journals with lower impact factor" (Gilmer et al. 2023, 499). This thesis seeks to bridge the gap between baseball and softball analytics by utilizing advanced statistical techniques and mature baseball data to improve the strategic decision-making of the Long Beach State University (LBSU) softball team, with a focus on pitching.

It is first important to understand the historical context that shaped the development of college softball. It is believed that softball was invented in 1887 in Chicago with little mention of women playing the sport until the 1930s. It wasn't until 1972, after women had to fight to play at the extra-mural level, when the first inaugural Women's College World Series (WCWS) made its debut in Omaha. Arizona State University (ASU) attended the championship tournament and found themselves sleeping on the floor and putting on the same dirty uniform for every game that week (Baker 2021). While the first WCWS was a monumental moment for women's softball, it was very quick to notice that they still had a long ways to go to catch up with men's baseball who were in their twenty-third year of the College World Series (CWS).

The Education Amendments of 1972 passed Title IX on June 23, 1972 which states, "No person in the U.S. shall, on the basis of sex be excluded from participation in, or denied the benefits of, or be subjected to discrimination under any educational program or activity receiving federal aid." Title IX mandates gender equity in athletic opportunities but includes an exception

for football, recognizing the unique roster size and financial demands of the sport. This allows schools to offer scholarships to male athletes in football without having to eliminate other men's programs. This exception is crucial to maintain the purpose of Title IX being gender equality. Title IX remains one of the most important milestones in gender equality. Therefore, at every National Collegiate Athletic Association (NCAA) school, there is a men's team paired with a women's team that get equal amounts of funding. This results in the same technology distributed to baseball and softball teams on the same campus. It was after Title IX, that the first WCWS was sponsored by Association for Intercollegiate Athletics for Women (AIAW) in 1973. ASU was deemed as champions in the finals.

Fast-forward to 2021, "[V]iewership for the WCWS finals reached a record 1.85 million viewers in 2021 and notably passed the Men's CWS championship with 1.6 million viewers in 2022" (West, Deitsch, and Bardahl 2024). The past 2024 season, the WCWS averaged at 2 million views in the final games. Thanks to the foundation laid by Title IX, college softball teams enjoy proper opportunities, resources, and recognition reflecting the incredible progress the sport has made.

The leading professional baseball and softball leagues are Major League Baseball (MLB) and Athletes Unlimited (AU) respectively. AU, founded in 2020, features a unique scoring system where individual players earn points based on their performance, and champions are determined by individual rather than team success.

While the goal of gender equality in government-funded high school and college sports is crucial, the similar conversation surrounding pay in professional baseball and softball should be rooted in equal proportionality, rather than equal outcome across the board. MLB had about 15.8 million views for the 2024 championship series and average regular season viewership between

1.3 to 3 million per game. These numbers vastly outpace AU, whose games stream on more niche platforms like ESPNU and ESPN2, reaching a significantly smaller audience. In this context, the financial disparity between the two leagues is not a matter of gender equality, but of market realities and audience demand. As of 2023, the MLB is valued at approximately $16 billion whereas AU is valued at $1 billion which may explain the disparity in the availability of data analytics resources. Given the MLB's much larger financial backing, they are able to invest heavily in advanced data analytics infrastructure, whereas AU, with its smaller budget, does not have the same level of funding to prioritize or implement data analytics to the same extent. The financial difference potentially contributes to the greater availability of public data for baseball compared to softball, which in tune may lead to higher volume of baseball-related articles that incorporate data analytics.

From softball players sleeping on the floor before a championship game to reaching 2 million views, the future for leagues like AU, founded only in 2020, is incredibly exciting. Women's sports, while growing in popularity, still need to invest in attracting larger audiences and building a lasting presence in mainstream culture. There is great potential for the sport to flourish, creating new opportunity for players and expanding the data analytics landscape within softball.

The central aim of this study is to provide the LBSU coaching staff with actionable insights to enhance pitching strategies by leveraging a suite of statistical models. These models include logistic regression analysis to examine the relationships between variables like pitch types and performance outcomes, as well as decision trees and random forests for capturing complex patterns, and domain adaptation techniques to improve model transferability across publicly available MLB baseball datasets. These predictive models will provide a deeper

understanding of performance dynamics, enabling more effective game strategies and pitcher development.

Three datasets will be utilized in this investigation. The first, compiled from LBSU's play-by-play stats, game day lineups, and Synergy Sports tracking, provides detailed pitch-by-pitch and play-by-play data for LBSU pitchers, including features such as pitch type, location, and outcome. The second merges publicly available NCAA reports to offer team-level pitching performance across all Division I programs, supporting cluster analysis to benchmark LBSU and identify key metrics tied to winning. The third uses publicly available MLB pitch-by-pitch data from the baseballr package in R, adapted for transfer learning by matching baseball pitch types to softball equivalents based on speed and movement, enabling cross-sport comparisons and model training.

Analytics plays an increasingly important role in modern sports by providing coaches with deeper insights to support decision-making. The process typically begins with data management—collecting, organizing, and cleaning large volumes of performance data. This data is then processed through analytic models that identify patterns, trends, and potential strategies. The outputs are delivered through information systems that translate complex analyses into accessible insights for coaches.

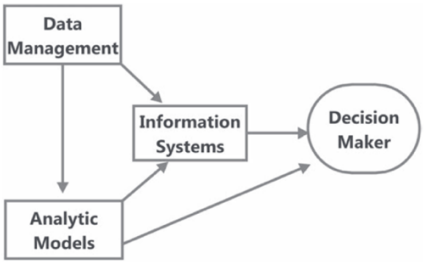


FIGURE 1. The flow of sports analytics from data to decision making (Alamar 2013, 5).

While analytics offers valuable context, it is ultimately just one tool among many—alongside film review, practice observations, and traditional statistics. Coaches remain the final decision-makers, using their expertise to weigh analytical input alongside other sources to make the best choices for their team.

This research will use descriptive and inferential statistics to analyze key pitching outcomes, including strikeouts (SO), walks (BB), hits, home runs (HR), and reached base, with the goal of developing strategies to improve pitching efficacy. These outcome variables contribute to broader pitching metrics such as batting average against (BAA), earned run average (ERA), and on-base percentage (OBP).

Ultimately, the findings from this study are intended to empower the LBSU coaching staff with more information that goes beyond traditional instinct. While analytics is not a replacement for coaching experience or on-field observation, it serves as a powerful complementary tool—helping to sharpen competitive edges, reduce inefficiencies, and support more confident in-game decisions. As the sport of softball continues to evolve, embracing the full potential of data analytics will be essential for programs that aspire to compete and win at the highest level.

CHAPTER 2

REVIEW OF LITERATURE

2.1 Disparity Between Softball and Baseball Analytics

The integration of data analytics into sports has revolutionized how teams approach strategy, player development, and performance evaluation. While baseball has long been at the forefront of sports analytics, softball has been slower to adopt similar techniques despite many overlapping aspects between the two sports. Gilmer (2023) found that, between 1990 and 2020,

the body of literature on baseball analytics was not only substantially larger than that on softball analytics, but also had a significantly higher average journal impact factor (Gilmer et al. 2023, 500). Moreover, baseball articles are much more established.

The movie *Moneyball* popularized the concept of sabermetrics — the empirical analysis of baseball statistics to evaluate and predict player performance more accurately than traditional scouting methods. Sabermetrics focuses on undervalued metrics such as OBP and slugging percentage rather than conventional stats like batting average (BA), enabling teams to optimize lineups and roster construction through data-driven decision-making. While tools like Synergy Softball which we will discuss in chapter 3 and general performance statistics are utilized by LBSU, these methods lack the inferential depth that more advanced data analytics could provide.

MLB has acquired very detailed data that is made available to them using Sportradar and other like technologies. Sportradar is the official provider of real-time MLB statistics, including Statcast data which consists of ball tracking technologies and is made publicly available. This feature provides quick and massive datasets in real time which not only track basic statistics such as BA, ERA, etc., but it can also track each pitch information—such as ball, strike, location, spin rate, ball launch angle. Data is collected in real time making the models more and more accurate after each day.

2.2 Predicting Wins and Losses

Predicting wins and losses in baseball is uniquely difficult, as the MLB exhibits significantly more randomness in game outcomes compared to other major sports leagues such as the National Basketball Association (NBA), National Football League (NFL), and National Hockey League (NHL). Even the strongest teams in MLB can lose a substantial portion of their games, highlighting how unpredictable wins and losses can be. Because of this inherent

variability, it is more insightful to focus on "small wins" within games—such as strikeouts for pitchers and other game variables—that, while individually minor, can cumulatively increase the likelihood of success over time. This micro-level approach becomes essential in a sport where macro-level predictions often fail due to randomness.

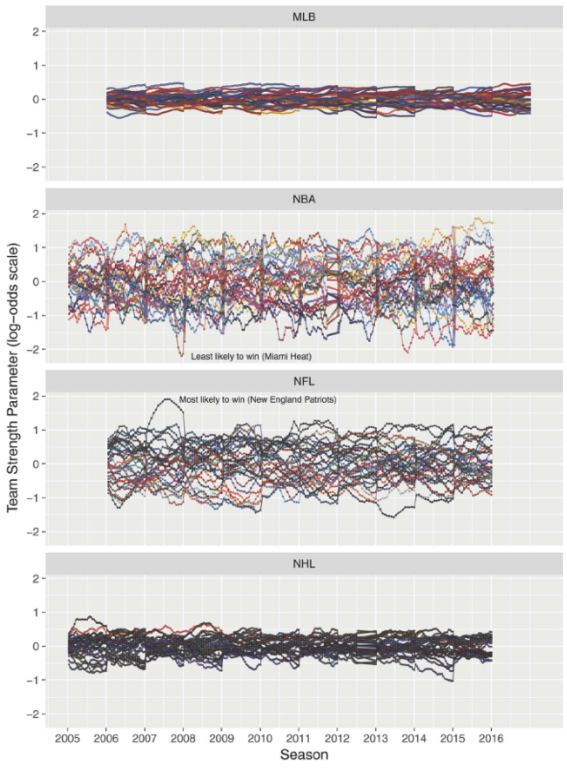


FIGURE 2. Mean team strength parameters (Lopez, Matthews, and Baumer 2018, 2508).

While baseball outcomes may often appear unpredictable, the value of analytics lies not in forecasting final scores, but in uncovering the small decisions that steadily tilt the odds in a team's favor. An article notes that "at a single point in time, team strength estimates diverge substantially more in the NBA and NFL than in the NHL and MLB. In the latter two leagues contests between two randomly chosen teams are closer to a coin flip, in which each team has a reasonable shot at winning" (Lopez et al. 2018, 2508). The article also shows that predictive models of team strength outperform win–loss records or point differential in forecasting future

performance. Finally, predictive accuracy was quantified with area under the ROC curve (AUC) and Brier scores: model predictions were well-calibrated across all four leagues, with strongest performance in the NBA (AUC ≈ 0.76) and NFL (≈ 0.68), moderate in the NHL (≈ 0.59), and weaker in MLB (≈ 0.57), likely reflecting the additional influence of starting pitchers. Thus, even in a sport dominated by randomness, analytics can identify controllable factors—pitch selection, defensive alignments, base-running, or lineup construction—that consistently offer teams a measurable competitive edge.

2.3 Clustering Use in Baseball

A 2020 honors thesis by Charlie Marcou at Grand Valley State University used unsupervised learning via cluster analysis to explore MLB pitchers and their ERA's based on what pitch types they throw. By analyzing Statcast data from 2017–2019, the study grouped pitchers based on variables such as pitch type usage, exit velocity, and batted ball outcomes. The results identified distinct clusters—such as high strikeout/high walk pitchers and extreme groundball pitchers—with meaningful differences in ERA, weighted on base average (wOBA), and quality of contact metrics like barrel rate and exit velocity. Some clusters performed better despite similar contact profiles, suggesting that variables like strikeout rate or pitch specialization play a key role in run prevention (Marcou 2020).

Marcou's analysis revealed that certain pitch-type clusters were associated with significantly better outcomes. For example, Cluster 1 (cutter balls) had the best overall performance, with the lowest average exit velocity, hard hit percentage, ERA, wOBA, and slugging percentage (SLG). In contrast, Cluster 6 (low SO, high BB) performed the worst across most metrics. The study notes that "Cluster 1 (high SO, high BB) gets the best results on average with the lowest mean ERA, wOBA and SLG," while "Cluster 6 (low SO, high BB) has the worst

mean ERA, wOBA, isolated power and SLG" (Marcou 2020, 25). These results suggest that combining high strikeout rates with certain pitch profiles may offer a competitive advantage in limiting offensive production.

FIGURE 3. MLB pitchers ERA by cluster.

Similarly, this thesis will apply unsupervised clustering methods during exploratory data analysis (EDA) to NCAA Division I team pitching statistics to examine how different combinations of statistics relate to team's winning percentages. Based off of those outcomes it will narrow down our target outcomes throughout the rest of the study. Rather than focusing on contact quality, this study aims to identify which clusters of pitching statistics (ERA, SO, BB, BAA, etc.) are most effective in producing favorable winning percentages, offering information for pitcher development.

2.5 Decision Tree

In Noah Woodward's investigation, decision trees are used for baseball pitching analytics to uncover pitch selection patterns by modeling relationships between pitch types and contextual variables such as count, base-runner state, previous pitch outcomes, and batter traits. For instance, a decision tree built for Justin Verlander revealed that count, batter handedness, and hitter patience were key predictors of pitch type. He was more likely to throw sliders to

aggressive right-handed hitters in 0-2 counts, while patient left-handed hitters often saw change-ups—even consecutively. Notably, Verlander also showed a strong bias toward fastballs when runners were on base, starting 75% of such at-bats with a fastball—significantly above his baseline—demonstrating how decision trees can expose potentially exploitable tendencies. Refer to Appendix A for an example of the decision tree figure.

Clayton Kershaw's tree, on the other hand, included no minimum sample threshold and generated a highly complex model with over 500 branches. While this depth allowed for more accurate prediction of rare pitch types, it struggled with slider prediction, emphasizing that model complexity must be tailored to the pitcher. Variable importance analysis showed that count, base-state, and previous pitch information were the most predictive inputs, while hitter attributes played a smaller role.

Fastball (FF)	Slider (SL)	Curve (CU)	Change (CH)	<-classified as	% Correct	Random Guess		Difference
827	144	65	2	FF	79.67%	60.79%	+	18.88%
255	101	49	8	SL	24.46%	25.91%	-	1.45%
78	44	54	10	CU	29.03%	15.05%	+	13.98%
33	9	4	4	CH	8.00%	2.00%	+	6.00%
*Trained with 3935 cases, evaluated on 1687 cases								

FIGURE 4. Testing metrics for Clayton Kershaw decision tree (Woodward 2014).

These examples show how decision trees can not only model pitch prediction, but also reveal behavioral patterns and strategic habits. A decision tree model trained to classify four pitch types (fastball, slider, curve, change-up) achieved an overall accuracy of 58.4%. Performance varied by pitch type, with fastballs classified most reliably (F1 = 74.1%), while off-speed pitches such as sliders, curves, and change-ups showed substantially lower precision, recall, and F1 scores. In my thesis, I will apply similar decision tree methodology to softball pitch-level data to recommend optimal pitch types in specific game situations, providing practical, data-driven guidance for pitch calling (Woodward 2014).

2.6 Binary Classification

An investigation was conducted a machine learning study aimed at predicting whether a baseball pitch would be called a ball or a strike by the home plate umpire. Using pitch-level data from Retrosheet and Baseball Savant covering MLB seasons from 2014 to 2017, the authors merged detailed pitch metrics—such as velocity, release point, spin rate, and spatial position—with umpire identifiers, producing a dataset of over 842,000 labeled instances.

The researchers evaluated logistic regression, decision tree, and random forest classifiers across three progressively larger feature sets ("tiny," "small," and "medium"). Among the tested models, the logistic regression trained on the small feature set performed best, achieving high predictive accuracy while maintaining model simplicity. The most influential variables in determining umpire calls were the strike zone location coordinates (plate_x and plate_z) and the zone category, indicating that spatial positioning is the dominant factor influencing pitch classification outcomes.

The strong results observed in the confusion matrix can be attributed to the use of a continuous variable to represent pitch location. In the dataset used for this study, locations were originally encoded as discrete categories (1–9). However, modeling the location as a continuous feature provides a more precise representation of spatial variation, leading to significantly improved test metrics. Additionally, the predictions focus on whether a pitch is called a ball or a strike by the umpire, rather than the batter's response. The data only includes pitches taken by the batters. This setup reduces the influence of the batter's hitting tendencies and instead captures the umpire's decision-making patterns, which are inherently more consistent and thus easier to model accurately. The confusion gives us testing metrics for his random forest model.

TABLE 1. Binary Classification Confusion Matrix

	Predicted Strike	Predicted Ball
Actual Strike	270,585	37,854
Actual Ball	51,353	153,205

TABLE 2. Testing Metrics for Binary Classification Confusion Matrix

Accuracy	82.6%
Precision	84.1%
Recall	87.7%
Specificity	74.9%
F1 Score	85.9%

These results demonstrate that a random forest can effectively model umpire decision patterns in pitch classification tasks, with spatial variables serving as the primary predictors. The study highlights both the consistency and the human variability inherent in called balls and strikes, providing insight into how borderline pitches are decided in practice (Feltey, Florence, and You 2017).

2.7 Random Forest

Ganeshapillai and Guttag (2012) established a baseline for evaluating pitch prediction models using a "naive guess" approach, where the most frequently thrown pitch in a training set is assumed to be the predicted pitch in the test set. This method provides a simple yet informative benchmark for comparison.

Building on this, a study by Sidle and Tran (2017) applying a random forest model demonstrated improved accuracy by identifying more nuanced patterns in pitch selection. For instance, Jake Arrieta's naive guess accuracy—based on his most frequent pitch—was 34.03%,

while the random forest model achieved 48.33%, outperforming the baseline by 14.3 percentage points. Across 287 pitchers, the average naive guess accuracy was 54.38%, providing a standard against which model performance was assessed. A detailed breakdown for Odrisamer Despaigne revealed that the random forest model exceeded naive accuracy across all pitch types, particularly excelling with fastballs and sinkers, though struggling with less common pitches like knuckleballs and sliders. These findings support the utility of random forest–based methods in modeling pitch selection, a framework that aligns with the present study's goal of using random forest to recommend optimal pitch types based on situational and batter-specific variables in softball.

		Predicted Pitch Type								
		FF	CT	SI	SL	CU	CH	KN	% Thrown	% of Each Correct
Actual Type	FF	330	5	77	2	17	2	0	28.60	76.21
	CT	55	42	19	2	1	8	3	8.59	32.31
	SI	108	5	312	2	19	10	0	30.12	68.42
	SL	31	2	23	10	2	6	2	5.02	13.16
	CU	36	2	43	3	54	3	1	9.38	38.03
	CH	72	1	34	0	2	33	2	9.51	22.92
	KN	63	3	30	3	5	3	26	8.78	19.55

FIGURE 5. Pitch-type random forest model predictions for Odrisamer Despaigne (Sidle and Tran 2017).

This article supports this investigation by showing how random forest models can identify patterns in pitch selection that go beyond simple frequency-based strategies. A random forest model trained to classify seven pitch types achieved an overall accuracy of 53.3%. Performance varied considerably by pitch: sinkers (F1 = 62.7%) and fastballs (F1 = 58.4%) were predicted with the highest reliability, while sliders and knuckleballs showed much lower recall

and F1 scores, indicating persistent challenges in distinguishing off-speed and specialty pitches (Sidle and Tran 2017).

While the study primarily focuses on predicting what pitch a pitcher will throw, the same modeling approach can be adapted to explore what pitch a pitcher should throw in a given situation to maximize effectiveness. By learning from contextual features such as count, previous pitches, and batter characteristics, random forests can help uncover optimal pitch choices in various scenarios. This is especially valuable in softball, where pitch strategy may differ from baseball but still benefits from data-driven decision-making. The study's use of pitch-specific accuracy metrics also highlights the importance of evaluating model output beyond overall accuracy—something I will incorporate when assessing pitch-type recommendations.

2.8 Transfer Learning – Domain Adaptation

Despite the close relationship between baseball and softball, a thorough review of existing literature revealed no substantial evidence of transfer learning applications between the two sports—particularly in leveraging baseball data to enhance softball analysis. While both games share foundational mechanics, such as pitch types and fielding dynamics, they also exhibit key differences in pitching style, field dimensions, and game strategy. As a result, many studies have remained siloed within their respective sports, with most softball research relying exclusively on sparse, sport-specific datasets. This absence of cross-domain learning indicates an untapped opportunity for computational modeling and predictive analysis.

One promising area of overlap lies in pitch behavior—specifically, the use of break, velocity, and change of speed to disrupt hitters' timing. Baseball analytics has long prioritized these metrics and has developed sophisticated tools to quantify and model them. Softball, on the other hand, is only beginning to integrate these variables at scale. By aligning softball pitch data

with comparable structures in baseball, researchers can harness the predictive strength of a more mature and abundant baseball dataset. This not only aids in building more robust softball models but also reduces the data burden on teams and researchers working with limited softball-specific information.

The lack of prior work in this domain strengthens the case for cross-sport domain adaptation as a novel and valuable direction. If successful, this approach could redefine how player performance and game strategy are analyzed in softball, potentially fast-tracking the sport's analytical maturity. Additionally, such transfer learning applications may inspire broader use of cross-domain models across other gendered or parallel sports, highlighting the importance of shared mechanics and transferable performance indicators in advancing equitable, data-driven insights.

CHAPTER 3

CURRENT LONG BEACH STATE STATISTICS

3.1 Athletic Department Scouting Reports

Statistical analysis provides a quantitative understanding of team performance, offering insights into strengths, weaknesses, and trends across different game situations. This section presents an example of what the coaches receive from Associate Athletic Director, Roger Kirk, each week before the weekend's games. This example is a piece of the last scouting report given to the coaches in the 2025 season. They allow the coaches to identify which outputs show some variation across different conditions, but they do not explain the underlying causes of these differences.

2025 Long Beach State Softball
Pitching Analysis for Long Beach State (as of May 08, 2025)
(All games Sorted by Player Name)

	vs left			vs right			w/runners			w/bases empty			vs leadoff			with 2 outs			fly	gnd	fly/	
Player	h	ab	avg	h	ab	avg	h	ab	avg	h	ab	avg	rch	ops	pct	h	ab	avg	out	out	gnd	gdp
4 Barnett, Kate	27	81	.333	44	172	.256	37	136	.272	34	117	.291	21	59	.356	21	80	.263	60	87	0.7	2
26 Cowans, Lindsey	15	41	.366	23	70	.329	27	70	.386	11	41	.268	14	27	.519	15	36	.417	17	39	0.4	2
9 Gonzales, Eryka	21	63	.333	38	155	.245	33	101	.327	26	117	.222	17	56	.304	21	73	.288	70	59	1.2	1
22 Haddad, Shannon	26	97	.268	49	225	.218	31	144	.215	44	178	.247	31	90	.344	25	109	.229	93	84	1.1	2
19 Martin, Maddy	15	56	.268	16	66	.242	21	71	.296	10	51	.196	9	26	.346	13	48	.271	41	31	1.3	3
5 Nally, Brynne	13	82	.159	55	196	.281	35	127	.276	33	151	.219	29	79	.367	20	86	.233	52	110	0.5	3
Totals	117	420	.279	225	884	.255	184	649	.284	158	655	.241	121	337	.359	115	432	.266	333	410	0.8	13
Opponents	120	398	.302	249	912	.273	198	698	.284	171	612	.279	137	331	.414	118	426	.277	405	385	1.1	11

Player	ap	gs	ip/app	*h/g	*r/g	*er/g	*bb/g	*so/g	k/bb	*2b/g	*3b/g	*hr/g	*fo/g	*go/g	*bf/g	sba	att	pct
4 Barnett, Kate	18	7	3.39	8.15	3.56	2.87	1.49	3.56	2.4	0.92	0.46	0.69	6.89	9.98	32.36	3	3	1.000
26 Cowans, Lindsey	13	3	2.00	10.23	7.00	6.46	5.38	4.58	0.9	1.35	0.00	0.54	4.58	10.50	36.08	4	5	.800
9 Gonzales, Eryka	15	11	3.76	7.33	4.35	3.85	2.98	3.73	1.3	1.24	0.00	0.50	8.70	7.33	31.69	10	12	.833
22 Haddad, Shannon	19	15	4.58	6.03	2.98	2.66	2.82	6.03	2.1	0.97	0.24	0.48	7.48	6.76	30.25	5	5	1.000
19 Martin, Maddy	13	1	2.38	7.00	6.32	4.29	3.61	3.61	1.0	1.13	0.23	0.68	9.26	7.00	33.42	5	5	1.000
5 Nally, Brynne	21	12	3.52	6.43	3.59	3.22	3.88	4.64	1.2	1.23	0.00	0.47	4.92	10.41	31.03	6	6	1.000
Totals	49	49	6.84	7.14	4.07	3.47	3.11	4.55	1.5	1.11	0.17	0.54	6.95	8.56	31.79	33	36	.917
Opponents	49	49	6.72	7.84	4.95	4.10	3.06	3.08	1.0	1.66	0.19	0.47	8.61	8.18	32.97	24	32	.750

* = average based on 7-inning game

FIGURE 6. An example of LBSU pitching analysis.

By analyzing these metrics, we can assess performance under various conditions and inform decisions moving forward with the investigation. However, it is important to note that these statistics are purely descriptive and do not provide any measure of statistical significance. They reflect outcomes rather than offering insight into the effectiveness of specific strategies or decision-making processes that take place in practice and/or games.

3.2 Challenges of Interpretation

Scouting reports for Long Beach State pitchers currently rely on extensive tabular statistics, with each pitcher evaluated across approximately 39 statistical categories in 8 different contexts. This creates more than 300 raw numbers per pitcher staff, which is an overwhelming amount of information to process. Although the table provides complete detail, it is difficult to quickly interpret or use in real-time decision making. Coaches must scan across rows and columns to piece together answers to key questions such as which pitcher is most effective against left-handed hitters, who best contains damage with runners in scoring position, or which pitcher is vulnerable when leading off an inning. In practice, the abundance of numbers often obscures meaningful differences rather than clarifying them.

To make this information more actionable, visual presentation of the same statistics is far more effective. Graphs highlight patterns and contrasts in ways that rows of numbers cannot. For example, dumbbell charts can illustrate each pitcher's strengths and weaknesses across different game situations, such as performance against left- versus right-handed hitters. A chart that illustrates this can allow coaches to see immediately where a pitcher excels and where vulnerabilities lie.

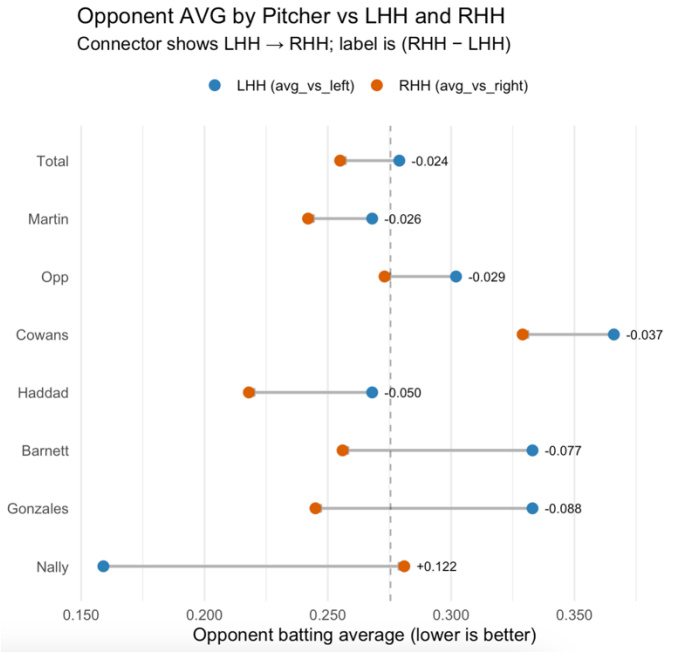


FIGURE 7. Dumbbell chart for pitchers vs RHH/LHH batting averages.

This chart shows pitcher splits vs left- and right-handed hitters. Haddad, Barnett, and Gonzales stand out as the strongest, holding opponents to consistently low averages. In contrast, Nally shows the largest gap with pitching about average to RHH and holding LHH to a battign average of only 0.159 which is very good.

Since performance outcomes differ between right-handed hitters and left-handed hitters, it is important to account for these variations when developing predictive models. Combining the two groups into a single model may obscure key performance patterns unique to each

handedness. Therefore, in order to improve model accuracy and ensure more reliable insights, separate models should be constructed for RHH and LHH.

Bar graphs are also effective for head-to-head comparisons. A simple set of bars can show which pitcher has the lowest opponent batting average with runners in scoring position. In Figure 9, one notable example is that Kate Barnett holds opponents to a .272 batting average with runners on base, while Shannon Haddad limits hitters to just .215 — both strong performances, but Haddad stands out as particularly dominant. In contrast, Lindsey Cowans opponent batting average of .386 with runners on shows a relative weakness that would be masked in the original table without visualization.

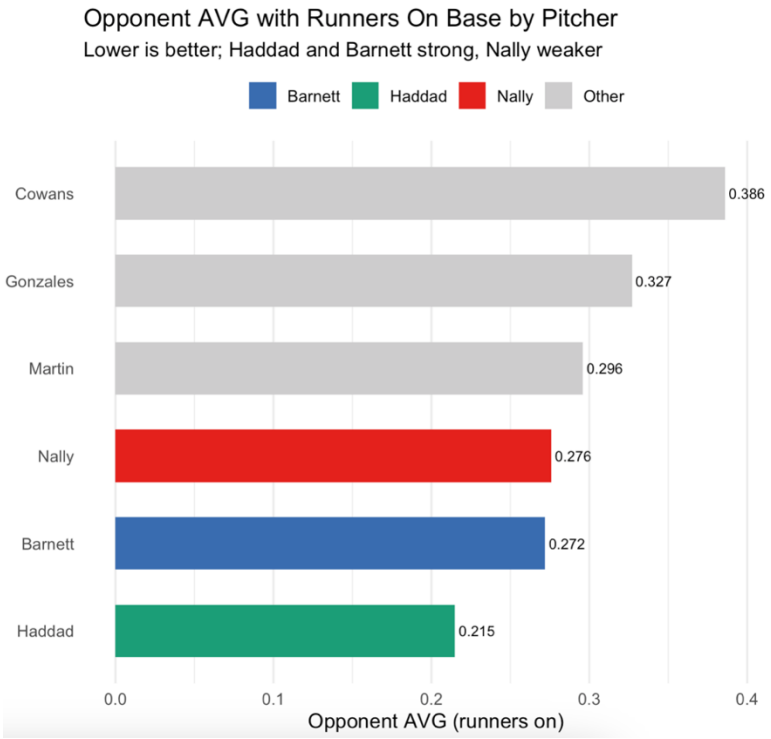


FIGURE 8. Bar graph of pitchers vs BAA with runners on base.

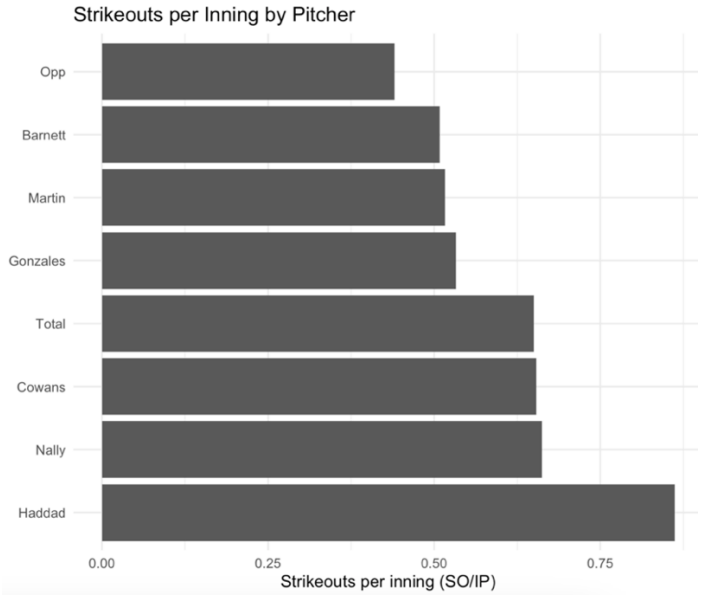


FIGURE 9. Bar graph of strikeouts per inning by pitcher.

The bar graph in Figure10 compares pitchers by their strikeouts per inning pitched (SO/IP). Each bar shows how many strikeouts a pitcher averages in a single inning, giving a quick sense of their strikeout effectiveness relative to teammates. For example, Haddad stands out with the highest rate, indicating he generates strikeouts more frequently per inning, while Barnett shows lower strikeout rates. The "Total" bar summarizes the team average for context.

By shifting from data-heavy tables to visuals such as radar charts, bar graphs, and heatmaps, scouting reports move from being collections of statistics to decision-ready tools. Instead of requiring time-consuming mental calculations, coaches can immediately grasp how pitchers compare across contexts and tailor game strategies more effectively. Now we can see how important not only the data is but also how we present it to a coaching staff.

3.3 Synergy Sport

Synergy Softball is an application widely used in college softball and baseball. The app offers advanced data filtering and video playback features, allowing coaches to generate detailed data with corresponding video for one or more teams, seasons, games, or players.

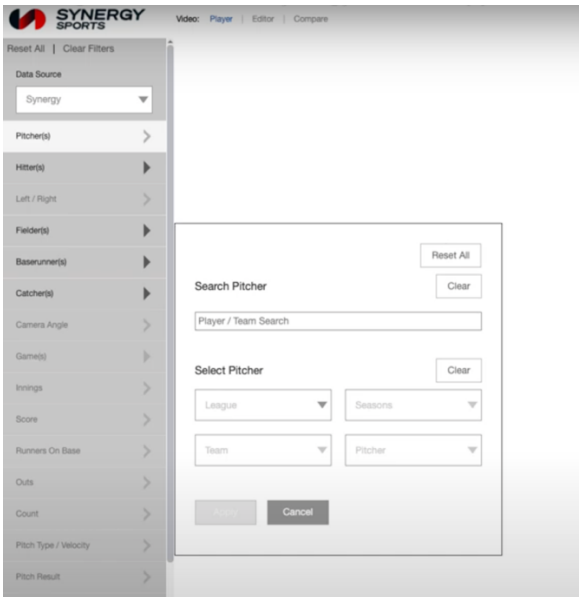


FIGURE 10. Data filtering in Synergy Sports (Synergy Sports n.d.).

Once data is selected for a specific player or team, the app enables users to visualize information through detailed pitch and spray charts, which can be printed as PDFs with additional filters. Users can also review basic statistics, watch associated game footage and generate multiple advanced reports for analysis and sharing.

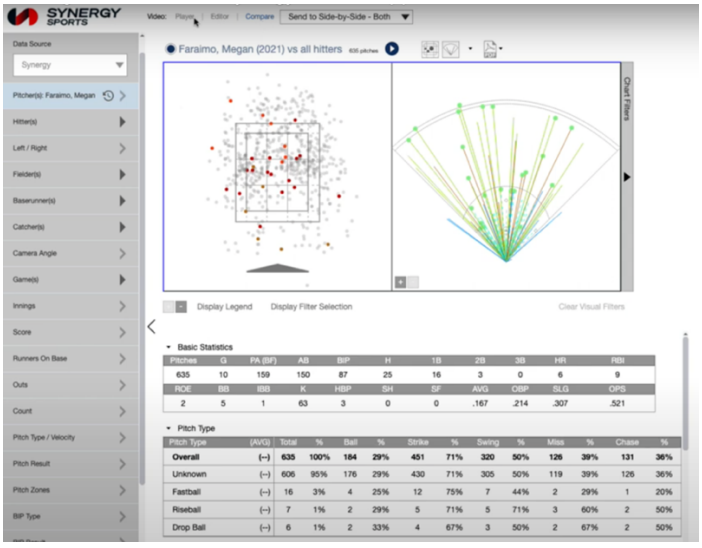


FIGURE 11. Visualization of pitches thrown and corresponding spray chart (Synergy Sports n.d.).

FIGURE 12. Examples of film review (Synergy Sports n.d.).

These reports can be saved for future reference, making them a valuable tool for the teams that utilize this feature. The single game view offers pitch-by-pitch details for individual games, with advanced filtering options that support in-depth review. This feature is commonly used to study upcoming opponents—both pitchers and hitters—in the days leading up to a game. Teams use it to scout tendencies, develop game plans, and structure practices accordingly. For example, they may design defensive plays based on opponent habits or focus hitting drills on specific pitch types frequently thrown by opposing pitchers. Additionally, the app provides visual representations that are later used in this thesis for collecting data on pitch type and location.

CHAPTER 4

DATA GATHERING AND PREPARATION

The data collection process aims to compile a comprehensive and structured dataset to facilitate robust statistical analysis. This involves aggregating information from multiple sources, ensuring consistency across seasons, and automating data extraction where possible to enhance accuracy and efficiency.

This investigation begins with gathering data from several sources to analyze the performance of the Long Beach State Softball Team. NCAA statistics from the 2024 season supply overall team statistics on all of the NCAA Division I teams, focusing on important pitching metrics such as hits (associated with BAA), walks given up, strikeouts, runs (associated with ERA), and reaching base (associated with OBP). The LBSU website, particularly focuses on play-by-play PDFs of each single game that includes the outcomes of each at-bat. The PDF also includes the game lineup which includes player name, bat handedness, and lineup position (i.e. 1 denotes the leadoff batter). An automated program in R was created to more efficiently process this data into an excel sheet. Pitch tracking data which can be found in Synergy Sports, an application used by collegiate teams to monitor pitch types and visualizations. The application was utilized in the this study's data collection process to manually input pitch type and location to supplement the play-by-play data given by the LBSU website (Long Beach State Softball n.d.).

4.1 LBSU Website: Compiled Play-By-Play Dataset

Data collected from play-by-play box scores—derived from PDF files from the LBSU website—is used to generate box score excel tables with consistent columns from one game to the next. This dataset will be used in all models in Chapter 7. These files contain key player performance metrics such as runs, hits, strikeouts, etc.. Pitch-by-pitch data is provided by Synergy for the 2023 and 2024 seasons and in real time game charting for the 2025 season, capturing pitch type and location which is crucial when studying in-game strategy and pitch calling objective. Lineup and defensive positions are also given in the box score which are manually recorded into an excel lineup sheet for each game, documenting batter name, batter's handedness, lineup position, and defensive alignment.

Hawai'i - Top of 1st	UH	LBS
Gin, X. struck out looking (3-2).	0	0
Keliinoi, K. flied out to rf (0-2).	0	0
Martinez, I. walked (3-2).	0	0
Johnson, H. popped up to 3b (2-0).	0	0

Runs: 0 Hits: 0 Errors: 0 Left On Base: 1

FIGURE 13. An example of a half inning on a play-by-play pdf (Long Beach State Softball n.d.).

To efficiently extract and structure the data, an automated script was developed to process the play-by-play PDFs. The script identified and categorized key events using predefined keywords, ensuring consistency across all games. Each PDF maintained a uniform terminology for strikeouts, walks, bunts, steals, singles, doubles, and other game events, allowing for reliable automated extraction. Through this process, 40 key variables were identified as relevant for analysis.

In compiling the dataset, I implemented several stop-and-check points within my code to ensure quality control throughout the process. At each checkpoint, I paused to review the compiled data and verify that all columns had been converted correctly and that the expected number of rows had transferred. Special attention was given to three key areas: (1) confirming the spelling of names in the lineup so they matched consistently across files, (2) ensuring that the event code was properly located and carried over, and (3) addressing cases where lines in the source PDF wrapped onto two lines—these had to be combined into a single line, since any loss of continuity would distort the play-by-play sequence and, in turn, the number of outs and baserunners recorded for a given inning. The specific stop-and-check procedures used to maintain these quality standards are integrated into the code provided in the Appendix C.

One of the most critical variables extracted is the event code, which systematically classifies game events. This feature is used to subset the data in order to study outcomes. A

corresponding event code table was created to standardize interpretation across the dataset. The event column was critical being that is is the phrase used in the box-score PDFs that would identify key game events and outcomes. Feature descriptions can be found in Appendix E.

The dataset includes games from the 2023 (54 games), 2024 (56 games), and 2025 (35 games) seasons, totaling 145 games. This resulted in a sample size of 10,401 pitches. Approximately 3,000 observations were recorded for each of the seasons. After separating pitches thrown by LBSU pitchers from opposing pitchers, there are 6,952 observations.

For the 2023 and 2024 seasons, only the final pitch of each at-bat was retained, linking the batter's result to the corresponding pitch. However, the 2025 season dataset includes every pitch thrown during an at-bat, providing a more detailed view of pitch sequences. It was later consolidated to just outcome pitch to keep the data consistent across all seasons.

The 2023 and 2024 datasets required extensive manual input to reconstruct missing pitch type and location data. This retrospective process significantly extended data preparation time, averaging 30 minutes per game. In contrast, 2025 data collection was streamlined by logging pitch details in real-time, allowing each game's dataset to be structured within 5 minutes. This underscores the importance of automated data integration or real-time recording for improved efficiency, consistency, and accuracy in data collection. The list of features can be found Appendix E.

There are several reasons for missing data in the pitch-level dataset, particularly across the 2023 and 2024 seasons. If an opposing team did not use Synergy, their pitch types and locations were not recorded. Additionally, plays that did not involve a pitch—such as stolen bases—do not have pitch data associated with them in '23/'24 seasons. In some games, Synergy footage was incomplete due to camera malfunctions, resulting in missing video and pitch

records. Pitch speed found in synergy was most often incomplete or very inconsistent. Thus, it was disregarded in the dataset. In earlier seasons, the pitch value column was sometimes removed too early in the process, preventing the capture of metrics like hit type, swing chase, or swing-and-miss rates. These swing-level details were not consistently available for 2023 and 2024. However, starting in 2025, pitch charting became more detailed, with manual tracking added for hitting data to improve overall completeness.

While the dataset provides valuable insights into pitch trends and hitter performance, the presence of missing or incomplete data limits the scope of some analyses. Variability in Synergy usage across teams, technical issues during games, and inconsistencies in pitch tracking—especially in earlier seasons—mean that certain metrics are not uniformly available. Despite these limitations, the addition of more detailed manual charting in 2025 has improved data quality moving forward. These improvements allow for more accurate and comprehensive evaluation, but it is important to interpret earlier data with an understanding of these gaps.

To ensure consistency across seasons, I filtered the 2025 and overall datasets to include only observations where bat_event = TRUE, matching the format of the 2023 and 2024 data. For strikeout prediction models, the data was subset to only include counts where the batter had two strikes. For walk models, I included only observations with three balls in the count. The datasets for hits, home runs, and OBP remained unchanged. After completing all data filtering and preparation steps, I split the dataset by testing on the last 20% of outcomes recorded in the 2025 season, and the rest were used to train the models.

4.2 2024 NCAA Overall Team Statistics

As part of this study, I utilize publicly available team-level data from the NCAA website (NCAA 2024) to analyze statistical trends among collegiate softball teams across the country in

the exploratory data analysis via clustering. This dataset includes comprehensive pitching statistics for all Division I programs and will serve as the foundation for understanding what differentiates winning and losing teams. By examining national trends, we aim to identify which pitch-level metrics are most predictive of success and which ones may be underutilized or overlooked in program-level decision-making.

I will consider team earned run average (ERA), fielding percentage, and strikeout-to-walk ratio. These stats serve as indicators of both pitching effectiveness and defensive efficiency. A low ERA and high fielding percentage traditionally signal strong defensive performance, while a favorable strikeout-to-walk ratio highlights control and command from the pitching staff.

By comparing these statistics between teams with high winning percentages and those with lower records, this study will assess which metrics are most consistently associated with success. If certain high-performing teams excel in areas that are not currently emphasized in our own team's evaluation criteria, it could indicate an opportunity to adjust what we value or measure internally.

4.3 Baseball Data for Transfer Learning

To gather the baseball data used in my domain adaptation model, I utilized the baseballr package in R, which provides programmatic access to MLB Statcast data. Specifically, I collected pitch-level data spanning from April 2024 to April 2025. This dataset includes detailed information on each pitch, such as pitch type, velocity, location, and outcomes. The data was used as the source domain in a transfer learning framework, where the goal was to apply insights from this source to a different but related target domain. Preprocessing steps included filtering for relevant variables, handling missing values, and formatting the data to align with the structure required for model training and evaluation.

After collecting the Statcast data, I needed to subset the columns to match the structure of the LBSU play-by-play dataset. This involved selecting only the relevant variables that were available in both datasets, such as pitch type, inning, etc. In addition to aligning the columns, I then ensured that the rows in LBSU softball data corresponded to the Statcast data correctly by matching on key identifiers and game context variables like inning and batting order. This step was essential to maintain consistency between the two data sources and ensure that the transfer learning model could accurately interpret patterns across domains. The features are described in Appendix E.

To ensure consistency between the Statcast and LBSU datasets, I recoded several Statcast columns to align with the variable names and formatting used in the LBSU play-by-play data. Specifically, I modified the columns events, zone, description, and game_type columns in the Statcast data to match their LBSU equivalents. On the LBSU side, I used mutate() to create a p_throws column based on the existing pitcher information, and I reformatted the bb_type column to match the structure used in Statcast. These adjustments allowed for a more seamless integration of the two datasets and improved the performance and interpretability of the domain adaptation model.

The zone variable in the Statcast data featured a more detailed pitch location system than the simplified version used in the LBSU dataset. Statcast defines zones 1–9 as areas within the strike zone (from the catcher's perspective) and zones 11–14 as out-of-zone areas (from the catcher's perspective). In contrast, the LBSU data used broader location categories and reported pitch locations from the pitcher's point of view. To align the two systems, I recoded the Statcast zones by reversing the perspective and grouping the detailed locations into simplified categories

that matched the LBSU play-by-play format. This step ensured that pitch location could be compared consistently across both datasets.

TABLE 3. Statcast Original Zones from the Catcher's Perspective (Major League Baseball 2025)

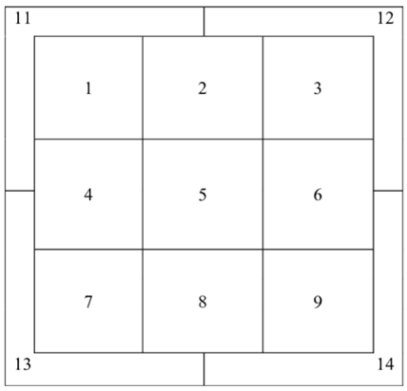


TABLE 4. LBSU Softball Zones from the Pitcher's Perspective

1	4	7
2	5	8
3	6	9

A major challenge in aligning the datasets was addressing the fundamental differences between baseball and softball pitch types, due to the distinct throwing mechanics—overhand in baseball versus underhand in softball. To bridge this gap, I analyzed the movement direction and velocity change of each baseball pitch relative to a fastball. Based on the pitch's break pattern and speed differential, I mapped each baseball pitch to its closest softball equivalent. This

matching process allowed for meaningful comparisons and model training across both sports, despite the inherent differences in pitch classification.

The most challenging pitch to match was the softball rise ball, as no baseball pitch replicates its upward trajectory due to the differences in throwing mechanics and mound elevation. In baseball, nearly all breaking pitches move downward, making direct comparison difficult. However, the four-seam fastball—when thrown with high velocity and backspin—tends to stay on a straight plane from the elevated mound, often causing hitters to swing underneath it or generate weak pop-ups. This outcome closely mirrors the effect of a rise ball in softball, making the four-seam fastball the most reasonable match despite the physical differences in pitch behavior.

TABLE 5. Baseball-to-Softball Pitch Type Equivalents

Baseball Pitch Type	Abbreviation	Matching Softball Pitch Type	Abbreviation
Change-up	CH	Change-up	X
Curveball	CU	Curveball	C
Cutter	FC	Drop ball	D
Four-Seam Fastball	FF	Rise ball	R
Forkball	FO	Change-up	X
Splitter	FS	Drop ball	D
Knuckle Curve	KC	Change-up	X
Knuckleball	KN	Change-up	X
No Pitch	NP	Unknown	UN
Screwball	SC	Screwball	S
Sinker	SI	Drop ball	D
Slider	SL	Curveball	C

Now that the baseball and softball datasets have been fully aligned—both in terms of variable structure and pitch type classification—they are ready to be used for transfer learning. With consistent formatting and matched features across domains, the model can effectively apply patterns learned from the source (baseball) data to the target (softball) context, enabling meaningful cross-sport insights.

CHAPTER 5

EXPLORATORY DATA ANALYSIS

Exploratory data analysis (EDA) is necessary to understand the structure within a dataset before applying advanced modeling techniques. This section examines the data using univariate analysis to explore individual variables, multivariate analysis to assess relationships between multiple variables, and unsupervised learning techniques to identify underlying clusters. The main dataset that is being investigated is the one complied of LBSU softball's play-by-play and pitch-by-pitch data from seasons 2023 to 2025 which consists of 6,952 rows of LBSU pitching data. To further enhance our analysis, we incorporate NCAA's 2024 overall team statistics to identify key pitching statistics associated with the winning percentages of all the division I teams in the nation. These methods collectively ensure a comprehensive understanding of the data, helping refine our approach for further analysis. Both right-handed hitter (RHH) and left-handed hitter (LHH) models are are included in the investigation; however, this report focuses exclusively on the analysis of RHH. To account for differences in pitch strategy based on batter handedness, all analyses in this study separate pitching against RHH and pitching against LHH. Batter handedness affects both pitch location and pitch selection, as pitchers typically adjust their approach when facing opposite-handed batters. All EDA graphs and models for LHH are provided in Appendix D.

Before beginning EDA, it is important to understand the meaning of each pitch type and location format used in the dataset. The location zones are recorded from the pitcher's perspective and are not classified by traditional strike or ball designations. Unlike systems such as Statcast, which overlay a defined strike zone, this dataset captures pitch locations using a 3x3 grid resembling a hashtag (#) format. Therefore, these zones provide spatial reference points rather than direct strike zone judgments. Additionally, pitch types are abbreviated using standard shorthand, which are detailed below.

TABLE 6. Location Key from Pitcher's View

1	4	7
2	5	8
3	6	9

TABLE 7. Pitch Type Key

F	Fastball
X	Change-up or Off-speed
C	Curveball
S	Screwball
D	Drop ball
R	Rise ball

5.1 Univariate Pitching Analysis

Through various graphs, this section aims to uncover the underlying structure of key variables such as pitch types, pitch counts, and performance metrics that our pitchers throw. The dataset contains more appearances against RHH (n=2,315) than LHH (n=1,360), reflecting the general distribution of hitter handedness in competition. This separation ensures that modeling results are not confounded by systematic differences in pitcher–batter matchups.

To better understand the pitch types LBSU pitchers throw, a series of visualizations highlighting pitch location and pitch type frequency is created. Specifically, this section includes three heat maps displaying the most common pitch locations the pitchers are throwing, along with tables summarizing the frequency of different pitch types thrown.

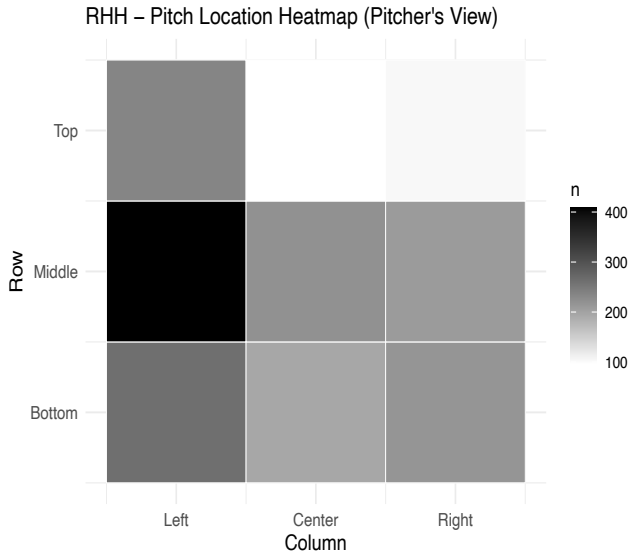


FIGURE 14. Heat map of pitch location.

These quadrants represent the location of pitches thrown by LBSU pitchers, categorized by height and width across the strike zone. According to the frequency graph, the most common pitch location in terms of width is inside to RHH is inside and outside at the waist height.

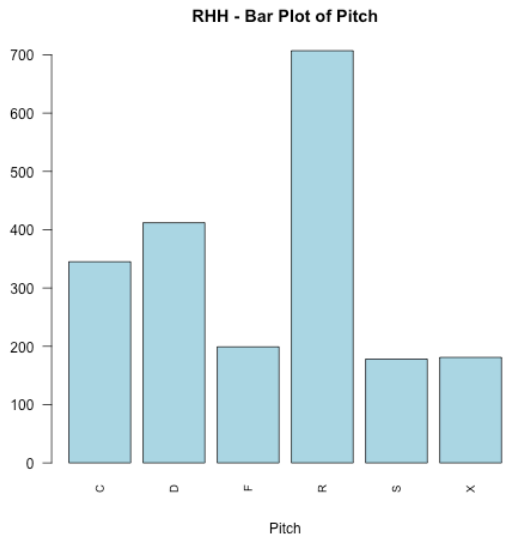


FIGURE 15. Pitch type frequency bar graph.

From these heat maps, a clear pattern emerges: LBSU pitchers most often throw rise and fastballs. The next two pitches most frequently thrown are curve and drop balls. Spins of the rise and drop ball focus on vertical movement—with the rise ball breaking up and the drop ball breaking down. Fastballs tend to stay on their line or slightly break down.

5.2 Multivariate Pitching Analysis

This analysis examines patterns in pitch type, location, and count to better understand how our pitcher generates successful outcomes. It explores which pitch types and locations are most effective at producing strikeouts, as well as tendencies when pitching in different counts. Strikeout percentages are shown by pitch type and location. The table below also defines the quadrant numbering system and pitch type abbreviations used in the analysis.

Batting performance by count reveals clear behavioral patterns for hitters. Hitters tend to struggle in early counts and two-strike situations, where strikeout rates are highest. As the count progresses to more favorable situations, such as three-ball counts, walk rates and on-base percentages increase significantly, indicating improved plate discipline. Home run rates tend to

peak in mid-count situations, where hitters are more likely to be aggressive. Overall, the data shows that hitter outcomes are heavily influenced by the count, with better results occurring in more advantageous situations.

count	batting_avg	SO_rate	HR_rate	BB_rate	on_base_rate	attempts
0-0	0.243	0.000	0.031	0.052	0.364	387
0-1	0.267	0.000	0.023	0.032	0.373	217
0-2	0.182	0.206	0.024	0.035	0.253	170
1-0	0.213	0.000	0.046	0.034	0.299	174
1-1	0.270	0.000	0.038	0.022	0.351	185
1-2	0.160	0.274	0.011	0.011	0.210	281
2-0	0.318	0.000	0.015	0.015	0.394	66
2-1	0.311	0.000	0.056	0.056	0.389	90
2-2	0.214	0.254	0.037	0.020	0.268	295
3-0	0.020	0.000	0.000	0.918	0.939	49
3-1	0.155	0.000	0.021	0.629	0.804	97
3-2	0.164	0.132	0.016	0.286	0.470	304

FIGURE 16. Count vs. outcome variables.

Pitch location has a clear impact on performance. The highest batting averages and on-base rates occur in the center of the zone, particularly in the middle-middle location, suggesting that RHH are most effective when pitches are left over the heart of the plate. Edges of the zone, especially the lower outside corner, show significantly lower batting success, reflecting difficulty handling pitches in those areas. Strikeout rates are generally higher in the upper and outer zones, while walk rates peak in the lower outside zone, possibly indicating pitches frequently missing just off the plate. Overall, hitters tend to produce the best outcomes when pitches are located in the middle or slightly inside parts of the strike zone.

Location	batting_avg	SO_rate	HR_rate	BB_rate	on_base_rate	attempts
1	0.141	0.224	0.023	0.137	0.300	263
2	0.235	0.093	0.022	0.044	0.324	408
3	0.191	0.106	0.008	0.131	0.381	236
4	0.194	0.117	0.056	0.117	0.337	196
5	0.356	0.014	0.059	0.000	0.392	222
6	0.173	0.112	0.051	0.143	0.459	98
7	0.212	0.092	0.014	0.157	0.396	217
8	0.214	0.048	0.029	0.095	0.390	210
9	0.104	0.151	0.019	0.387	0.519	106

FIGURE 17. Location vs. outcomes.

Hitters perform best against pitch types C and X, with higher batting averages and on-base rates. They struggle most against pitch type F, which shows the lowest offensive production. Pitch type D leads to the highest on-base rate, driven by a strong walk rate. In general, hitters are more effective against certain fastballs and off-speed pitches, while sharper breaking pitches tend to reduce their success.

Pitch	batting_avg	SO_rate	HR_rate	BB_rate	on_base_rate	attempts
C	0.243	0.072	0.029	0.070	0.348	345
D	0.175	0.083	0.032	0.192	0.437	412
F	0.246	0.030	0.015	0.146	0.492	199
R	0.184	0.147	0.030	0.098	0.313	707
S	0.258	0.118	0.039	0.096	0.382	178
X	0.243	0.105	0.017	0.044	0.320	181

FIGURE 18. Pitch vs. outcomes.

5.3 Unsupervised Learning – Data Clustering

To identify patterns among NCAA Division I softball teams, an unsupervised clustering analysis was conducted using key offensive, pitching, and scoring metrics from the 2024 season. An informal definition of data clustering is as follows:

> *Definition "Given a data matrix D (database D), partition its rows (records) into sets C1 . . . Ck , such that the rows (records) in each cluster are 'similar' to one another"* (Aggarwal, 2015, 20).

There are many different types of clustering techniques, but this analysis will focus on k-means clustering. In k-means, the goal is to minimize the sum of squared Euclidean distances between each data point and the nearest cluster center. This quantity, known as the sum of squared errors (SSE)**,** serves as the objective function and quantifies the quality of the clustering. Therefore, the objective can be written as:

$$Dist(\overline{X_i}, \overline{Y_j}) = ||\overline{X_i} - \overline{Y_j}||_2^2.$$

where $||\cdot||_p$ represents the L_p-norm (Aggarwal 2015, 162).

This approach groups teams' pitching staff by performance, revealing underlying profiles and competitive strengths beyond traditional rankings. Five clusters were selected to minimize group overlap and ensure each cluster represents a distinct pattern or behavior. This number strikes a balance between capturing data diversity and maintaining interpretability.

FIGURE 19. Pitching staff cluster visualization.

```
# A tibble: 5 × 12
  cluster  Rank win_perc    IP     R    ER   ERA     A    SO    BB KtoBB    PG
  <fct>    <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 1       181      0.421  339.  260.  209.  4.32  400.  196.  160. 1.24  0.841
2 2        44.6    0.708  381.  156.  125.  2.32  414.  321.  116. 2.83  1.31
3 3        77.9    0.607  367.  202.  166.  3.18  402.  277   160. 1.75  0.947
4 4       260.     0.241  290.  300.  245.  5.94  333.  141.  162. 0.904 0.966
5 5       192.     0.477  317.  199.  160.  3.54  344.  214.  119. 1.83  1.06
```

FIGURE 20. Pitching staff cluster analysis.

In 2024, LBSU's pitching staff fell into Cluster 2, which showed low walk rate but lacked strikeout power. In 2025, they moved to Cluster 3, showing improved effectiveness with a better ERA and K/BB ratio, but with moderate consistency. To further improve, LBSU should focus on a Cluster 1 strength: elite strikeout ability, which neither of their current clusters have. They should also avoid Cluster 4's weakness, which is high walk rates.

5.4 Correlation Analysis

As part of the exploratory data analysis, correlation analysis methods are applied to examine relationships among predictors and detect potential multicollinearity. High correlations between predictors distort model estimation, inflate variance, and obscure the relative importance of features (Dormann et al. 2013). To address this issue, pairwise correlations among continuous predictors are computed and evaluated against a multicollinearity threshold.

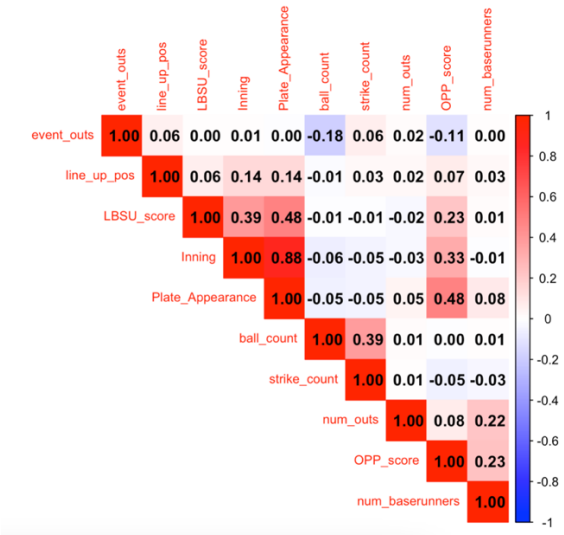


FIGURE 21. Feature correlation analysis.

To reduce redundancy and stabilize model estimation, only one variable from each highly correlated pair is retained in subsequent modeling. For example, Inning and Plate_Appearance with a correlation of 0.875 are not entered simultaneously in a given model. This procedure ensures that multicollinearity does not undermine model interpretability or predictive performance.

CHAPTER 6

METHODOLOGY

6.1 Models

In supervised machine learning, regression predicts numeric variables and classification predicts categories or classes. In binary classification, only two possible outcomes are modeled—such as success or failure, hit or no hit. This paper models binary outcomes—specifically, whether a batter achieved a hit, reached base, hit a home run, struck out, or walked. These outcomes serve as the foundational components of traditional statistics; thus, constraining these events directly impacts broader performance measures. This modeling approach allows for greater interpretability and flexibility, especially when working with limited or non-standardized data inputs. The following sections present each model's formulation and key assumptions.

In this thesis, four classification algorithms were used: logistic regression, decision tree, random forest, and domain adaptation through transfer learning. Domain adaptation is applied to baseball data to predict specific binary outcomes including whether a batter recorded a hit, reached base, or hit a home run based on pitch type and location. Rather than relying on high-level metrics like ERA or BAA, this analysis focused on the underlying events that contribute to those statistics. For example, because batting average is calculated using hits, modeling the likelihood of a hit directly provides insight into factors that influence batting success.

By predicting these individual outcomes, we can understand and influence broader performance trends. Each model brings a different approach to classification, and together they provide a comprehensive view of how various in-game variables can be used to anticipate offensive success in baseball.

6.1.1 Binary Logistic Regression

Binary logistic regression is used for categorical outcomes. In our case, we use binary logistic regression to model outcomes with two levels—for example, coding strikeouts as 1 and all other outcomes as 0. Rather than modeling the outcome directly, logistic regression models

the probability that *Y* belongs to a specific category. All models will take on the same binary outcomes.

$$Y = \begin{cases} 0, & \text{if no} \\ 1, & \text{if yes} \end{cases}$$

It is preferable to use a classification method such as logistic regression that is made to support qualitative response values (James et al. 2013, 130).

To avoid $E(y) < 0$ or $E(y) > 1$ in a linear approach, a function that gives $0 < E(y) < 1$ for all values must be used. The binary logistic regression model meets this criteria and is the following:

$$\pi = E(y) = \frac{\exp(\beta_0+\beta_1 X_1+...+\beta_k X_k)}{1+\exp(\beta_0+\beta_1 X_1+...+\beta_k X_k)}$$

where π denotes a probability that $E(y) = 1$. A different way to write this formula is the following:

$$\frac{\pi}{1-\pi} = \exp(\beta_0 + \beta_1 X_1 + \ldots + \beta_k X_k)$$

which is used to describe the odds of being in the current category of interest. By definition, the odds for an event is π / (1 - π) such that *P* is the probability of the event.

For a logistic regression model to be valid, several assumptions must be met. The dependent variable must be binary, and observations should be independent. The model assumes a linear relationship between the independent variables and the log-odds of the outcome. Multicollinearity among predictors should be low, and the data should be free of influential outliers. Lastly, a sufficiently large sample size is required to ensure stable and reliable estimates.

In Aggarwal's (2015), logistic regression is a probabilistic linear classifier that models the probability of class membership using a discriminative approach. Unlike the Bayes classifier, which assumes a specific form of the feature distribution per class, logistic regression directly models the conditional probability of a binary class variable $Y \in \{0,1\}$ given a feature vector $X = (x_1, x_2, \ldots, x_d)$.

Let $\Theta = (\beta_0, \beta_1, \ldots, \beta_d)$ represent the parameter vector, where β_0 is the intercept and each β_i corresponds to feature x_i. The class probabilities are given by the logistic (sigmoid) function:

$$P(Y = 1 \mid X) = \frac{1}{1 + e^{-(\beta_0 + \sum_{i=1}^{k} \beta_i x_i)}}$$

$$P(Y = 0 \mid X) = \frac{1}{1 + e^{\beta_0 + \sum_{i=1}^{k} \beta_i x_i}}$$

The expression $\beta_0 + \sum_{i=1}^{k} \beta_i x_i$ is a linear combination that defines a separating hyperplane between the two classes. Its sign determines the predicted class, while its magnitude corresponds to the confidence (or distance from the hyperplane).

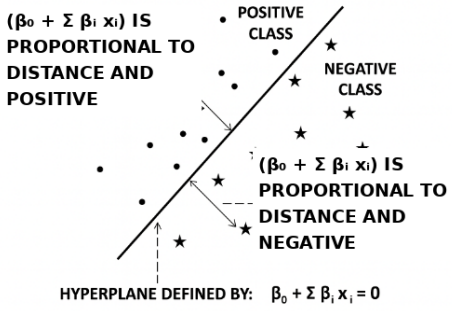


FIGURE 22. Illustration of logistic regression in terms of linear separators.

The parameters Θ are learned using maximum likelihood estimation (MLE).

Let $D^+$and D^- be the sets of positive and negative training examples, respectively.

The likelihood function for the training data is:

$$L(\Theta) = \prod_{X_k \in D^+} \frac{1}{1 + e^{-(\beta_0 + \sum_{i=1}^{k} \beta_i x_{ij})}} \cdot \prod_{X_k \in D^-} \frac{1}{1 + e^{\beta_0 + \sum_{i=1}^{k} \beta_i x_{ij}}}$$

To simplify computation, the log-likelihood is used:

$$LL(\Theta) = \sum_{X_k \in D^+} \log\left(\frac{1}{1 + e^{-(\beta_0 + \sum_{i=1}^{k} \beta_i x_{ij})}}\right) + \sum_{X_k \in D^-} \log\left(\frac{1}{1 + e^{\beta_0 + \sum_{i=1}^{k} \beta_i x_{ij}}}\right)$$

This function is maximized via gradient ascent, using the following gradient update rule:

$$\theta_i \leftarrow \theta_i + \alpha\left(\sum_{X_k \in D^+} x_{ij} \cdot P(\text{mistake on } X_k) - \sum_{X_k \in D^-} x_{ij} \cdot P(\text{mistake on } X_k)\right)$$

Here, α is the learning rate, and the update is driven by the model's current mistakes.

One common metric used to evaluate model fit is the Akaike Information Criterion (AIC). AIC assesses how well the model explains the data while applying a penalty for the number of parameters used, helping to balance goodness of fit with model simplicity. Lower AIC values indicate a better-fitting model with fewer unnecessary predictors (Penn State 2018).

6.1.2 Decision Trees

Classification decision trees operate by making a sequence of hierarchical, rule-based decisions using input features to predict a categorical outcome. The process begins at a root node and moves through a series of internal decision nodes, where the data is split based on conditions involving specific predictor variables. Each internal split partitions the data into more homogeneous subgroups. This continues recursively until an observation reaches a terminal, or leaf, node, where it is assigned a predicted class label. The overall structure is intuitive and mimics human decision-making processes, which makes decision trees particularly appealing in applied fields such as sports analytics, where visual interpretability and transparency are crucial.

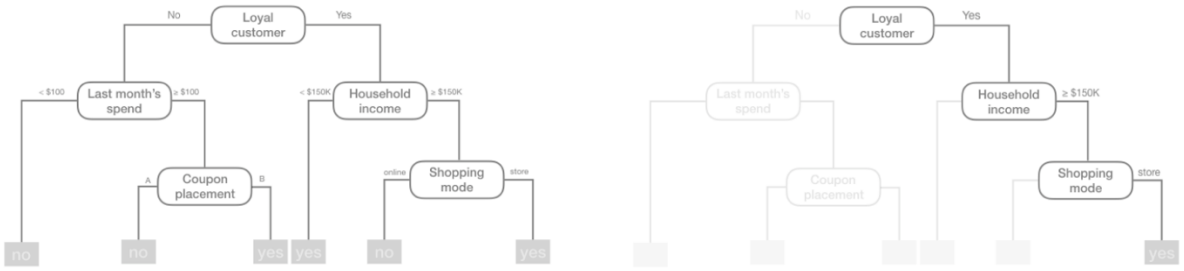


FIGURE 23. An example of a decision tree (Boehmke and Greenwell 2020).

Formally, a regression tree with J leaf nodes partitions the input space into regions $\{R_j\}_{j=1}^{J}$ with a constant prediction w_j in each region:

$$f(x;\theta) = \sum_{j=1}^{J} w_j \; I\big(x \in R_j\big), \quad w_j = \frac{\sum_{n=1}^{N} y_n I\big(x_n \in R_j\big)}{\sum_{n=1}^{N} I\big(x_n \in R_j\big)}$$

where $I(\cdot)$ is an indicator function, R_j is the region for leaf j, and w_j is the average response in that region. For classification, the leaf stores a distribution over the class labels rather than a single mean value.

To determine the best split at each node, classification trees employ recursive binary splitting—a process that searches over all variables and possible split points to find the one that minimizes node impurity. This can be expressed as:

$$(j^*, t^*) = \arg\min_{j,t} \frac{|D_L(j,t)|}{|D|} \; c\big(D_L(j,t)\big) + \frac{|D_R(j,t)|}{|D|} c\big(D_R(j,t)\big)$$

where $c(\cdot)$ is an an impurity measure (classification) (Murphy, 2022).

The classification error at a node m is:

$$E_m = 1 - \hat{p}_{mk*}$$

where $\hat{p}_{mk*}$ is the proportion of training observations in node m that belong to the most common class k^*.

More commonly, the Gini Index is used:

$$G_m = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^{K} \hat{p}^2{}_{mk}$$

where $\hat{p}_{mk}$ is the proportion of class k in node m. The Gini index reaches its minimum value of zero when the node is pure (i.e., all observations belong to a single class). A lower Gini value indicates higher purity, which the algorithm seeks to achieve at each split.

Though decision trees are non-parametric and do not assume a specific functional form between predictors and the outcome, certain practical considerations help improve their performance. For instance, decision trees generally perform better when predictor variables are relatively independent and exhibit low multicollinearity, as highly correlated variables can lead to unstable splitting behavior. Additionally, trees require a sufficient sample size to avoid overfitting and to ensure that patterns learned during training are generalizable. Categorical variables should also be handled with care; predictors with too many levels can cause overly specific splits that harm generalization.

Among their many advantages, decision trees are appreciated for their interpretability, their ability to handle both numerical and categorical features naturally (without requiring dummy variables), and their flexibility in modeling complex nonlinear interactions. These traits make them especially useful in practical settings where model transparency and ease of communication are important. However, a significant drawback is their tendency toward high variance—small changes in the training data can result in a very different tree. Moreover, single decision trees often have lower predictive accuracy compared to more sophisticated models.

To address these limitations, ensemble methods such as random forests are frequently used. Random forests improve upon individual decision trees by aggregating the predictions

from many trees, each trained on a different bootstrap sample of the data with a random subset of predictors considered at each split. This process reduces variance, increases model stability, and significantly boosts predictive performance, while still retaining some level of interpretability. As such, random forests are a powerful extension of decision trees, especially when the primary goal is not just explanation, but accurate prediction.

6.1.3 Random Forest

Random forests enhance the performance of decision trees by significantly reducing variance and improving prediction accuracy. Like bagging, random forests construct multiple decision trees using bootstrapped samples from the training dataset. However, at each split only a random subset of the predictors is considered. If the dataset has pp predictors, then at each split a random subset $S_i \subset \{1, \dots, p\}$ of size $m \approx p$ is drawn (James et al. 2013, 319). Murphy (2022) writes that the optimal split is chosen only from those variable.

$$(j^*, t^*) = \arg \min_{j \in S_i, t \in T_j} [\frac{|D_L(j,t)|}{|D|} \, c\big(D_L(j,t)\big) + \frac{|D_R(j,t)|}{|D|} c\big(D_R(j,t)\big)]$$

This modification prevents any single strong predictor from dominating the ensemble, thereby encouraging more diverse tree structures.

The full random forest predictor averages over *B* trees:

$$\hat{f}(x) = \frac{1}{B} \sum_{b=1}^{B} f_b(x)$$

where $f_B(x)$ is the prediction from the *b*-th tree. For classification, majority voting is used instead of averaging.

Unlike many models where increasing complexity harms generalization, adding more trees in a random forest does not lead to overfitting; instead, it stabilizes predictions.

In addition to predictive accuracy, random forests provide measures of variable importance. A common measure is the mean decrease in Gini impurity**,** defined as the total reduction in G_m attributed to splits on predictor j, averaged over all trees:

$$\text{VarImp}(j) = \frac{1}{B}\sum_{b=1}^{B}\sum_{m \in T_b:\text{ split on } j} \Delta G_m$$

where ΔG_m is the is the decrease in Gini index at node m. This provides insights into which predictors are most influential in the model.

The result is a model with reduced variance compared to both a single decision tree and a bagged ensemble, leading to more accurate and stable predictions. This is particularly valuable in high-dimensional settings or when the predictors are strongly correlated, as random forests can better manage redundant or overlapping information. Another key strength of random forests is their resistance to overfitting. Unlike many models where increasing complexity harms generalization, increasing the number of trees B in a random forest does not degrade performance. As long as the error rate converges, a larger B simply improves model stability without increasing the risk of overfitting (James et al. 2013, 319).

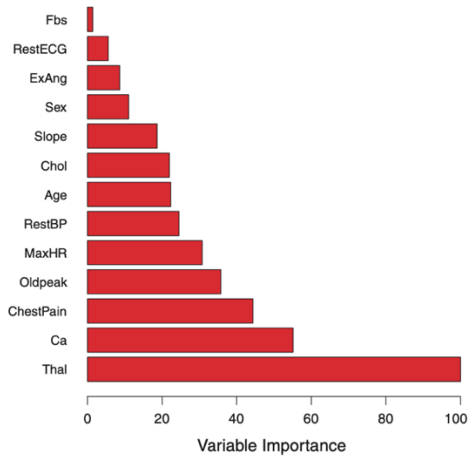


FIGURE 24. An example of variable of importance.

In addition to improved predictive performance, random forests offer a built-in method for assessing variable importance. This is typically computed based on the mean decrease in the Gini index across all trees, which quantifies how much each predictor contributes to reducing node impurity. Variables that consistently produce splits with large reductions in Gini index are deemed more important. This feature provides valuable insights into the relative influence of each predictor, making random forests not only powerful as predictive models but also useful for feature selection and interpretability (James et al. 2013, 319).

6.1.4 Domain Adaptation

Domain adaptation is beneficial when handling data that has low sample size or is unbalanced. The transfer learning technique consists of training the model on one's data set called the source domain to then test it to a different, but related data distribution.

Pan and Yang (2010) define transfer learning as a machine learning approach used when the training and test data do not share the same distribution or feature space. Instead of building new models from scratch for each task or dataset, transfer learning allows knowledge gained from one domain (the source) to be transferred to improve learning in another domain (the target). This method is particularly useful when labeled data in the target domain is limited or costly to obtain. The authors outline different types of transfer learning (inductive, transductive, and unsupervised) and highlight how models can be adapted across domains through shared representations or aligned distributions, while also addressing challenges like negative transfer, where the adaptation harms rather than helps model performance.

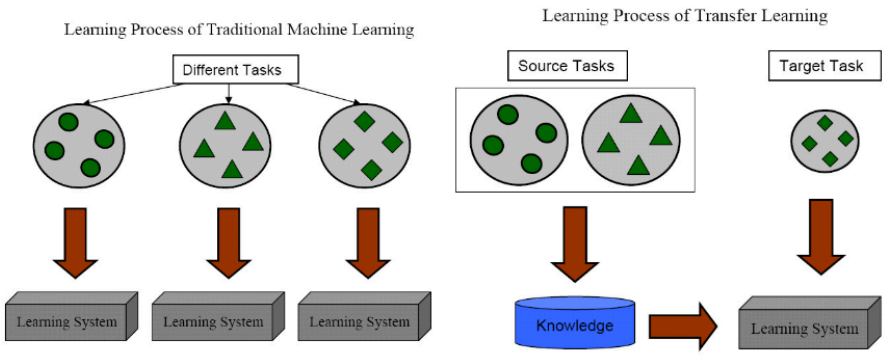


FIGURE 25. Traditional learning vs. transfer learning (Pan and Yang 2010, 2).

In this study, we consider the transfer learning scenario involving a single source domain D_S and a single target domain D_T , which is the most common setup in the literature (Pan & Yang, 2010). The source domain data is defined as:

$$D_S = \{(x_{S1}, y_{S1}), \dots, (x_{Sn}, y_{Sn})\}, \qquad x_{Si} \in X_S, \quad y_{Si} \in Y_S$$

where x_{Si} represents a data instance and y_{Si} is its corresponding class label. Similarly, the target domain data is:

$$D_T = \{(x_{T1}, y_{T1}), \dots, (x_{Tm}, y_{Tm})\}, \qquad x_{Ti} \in X_T, \quad y_{Ti} \in Y_T$$

Given a source domain D_S and learning task T_S and a target domain D_T and learning task T_T, transfer learning aims to improve the learning of the target predictive function $f_T(\cdot)$ in D_T by leveraging knowledge from D_S and T_S, where either $D_S \neq D_T$ or $T_S \neq T_T$.

A domain is defined as a pair:

$$D = \{X, P(X)\},$$

where X is the feature space and $P(X)$ is the marginal probability distribution over the features.

The condition $D_S \neq D_T$ implies either:

$$X_S \neq X_T \quad \text{or} \quad P_S(X) \neq P_T(X).$$

For example, in documentation classification, this could mean that the source and target documents use different languages (different X) or have different topical distributions (different $P(X)$).

Similarly, a task identifies as:

$$T = \{Y, P(Y|X)\},$$

where Y is the label space and $P(Y|X)$ is the conditional probability distribution of label given features. The condition $T_S \neq T_T$ means either:

$$Y_S \neq Y_T \quad \text{or} \quad P_S(Y|X) \neq P_T(Y|X).$$

For instance, the source task might classify documents into two classes, while the target task classifies into ten classes, or the class distributions might differ significantly. When $D_S = D_T$ and $T_S = T_T$, the problem reduces to traditional machine learning. Furthermore, if there exists some relationship—explicit or implicit—between the source and target feature spaces, the domains are said to be related, which facilitates transfer learning.

Transfer learning can be categorized into three main settings: inductive, where the source and target tasks differ; transductive, where the task remains the same but the domains differ; and unsupervised, where both source and target tasks are unsupervised and different. Since this investigation applies predictive modeling to both baseball and softball data with the same task—determining optimal pitch type given context—but from different domains (i.e., baseball as the source and softball as the target), transductive transfer learning is the most appropriate approach. Specifically, it aligns with the case where the feature spaces have been matched across domains ($X_S = X_T$), but their marginal distributions differ ($P_S(X) \neq P_T(X)$), making it a form of domain adaptation.

Transductive transfer learning, first formalized by Arnold, Nallapati, and Cohen (2007), applies when the source and target tasks are identical but the domains differ. Unlike traditional transductive learning—which requires all test data up front—Pan and Yang (2010) relax this to require only some unlabeled target data during training so the model can estimate the target-domain distribution. Formally, given a source domain D_S with task T_S and a target domain D_T with task T_T, transductive transfer learning seeks to improve the target predictor $f_T(\cdot)$ using knowledge from D_S and T_S under the constraints $D_S \neq D_T$ and $T_S = T_T$, with partial unlabeled data from D_T available. Two sub-cases exist: (a) different feature spaces $X_S \neq X_T$ or (b) identical feature spaces but different marginal distributions ($P_S(X) \neq P_T(X)$); the latter aligns with domain-adaptation and sample-selection-bias problems.

This study fits sub-case (b) of transductive transfer learning. After harmonizing column names and recoding events (e.g., "slider" → "drop ball"), baseball (source) and softball (target) share the same feature space while retaining different pitch-distribution profiles. By supplying unlabeled—or sparsely labeled—softball data during training, I will adapt decision-tree and random-forest models trained on abundant baseball data to recommend optimal pitch types in softball, leveraging the transductive framework to bridge the distribution gap without extensive new labeling.

Domain adaptation, a sub-field of transfer learning, addresses distribution shifts between source and target data. While deep domain adaptation using adversarial networks (e.g., Ganin et al. 2016; Hoffman et al. 2018) has gained popularity, non-deep approaches remain relevant. In particular, re-weighting methods—such as class-based reweighting (Saerens, Latinne, and Decaestecker 2002) and sample-based reweighting (Huang et al. 2006)—offer effective strategies for correcting sample selection bias in unlabeled data.

In this study, an X → Y is the assumed causal structure, where explanatory variables such as pitch count, pitch location, and pitch type influence the likelihood of an outcome. This reflects the intuitive understanding that game context drives outcomes. When transferring knowledge from baseball (source domain) to softball (target domain), it is unlikely that the marginal label distribution $P(Y)$ remains constant. Differences in rules, strike zones, and pitching styles between the two sports suggest that $P_S(Y) \neq P_T(Y)$. Similarly, although both domains share the same feature space (i.e., variable names and structure), the distribution of features $P(X)$ is also expected to shift due to inherent gameplay differences, implying $P_S(X) \neq P_T(X)$.

Since $P_S(X) \neq P_T(X)$, the marginal label distribution $P(Y)$ will typically also differ across domains.

$$P_S(Y) \neq P_T(Y)$$

However, it is reasonable to assume that the conditional distribution $P(Y|X)$—that is, the relationship between game context and the likelihood of a strike—remains relatively stable across domains: $P_S(Y|X) = P_T(Y|X)$. Under this assumption, the domain adaptation problem falls into the category of covariate shift, where the covariate distribution changes but the labeling function remains fixed. In this setting, the target risk $R_T(h)$ can be rewritten using importance weighting:

$$R_T(h) = \sum_{y \in Y} \int_X l(h(X), Y) \, \frac{P_T(X)}{P_S(X)} \, P_S(X, Y) dX$$

This leads to the practical formulation of the importance-weighted empirical risk:

$$\hat{R}_T(h) = \frac{1}{n} \sum_{i=1}^{n} \mathrm{l}(h(x_i), y_i) \cdot w(x_i), \quad \text{where} \quad w(x_i) = \frac{P_T(x_i)}{P_S(x_i)}$$

This reweighting adjusts for the differences in $P(X)$ between the domains while preserving the learning objective. The use of this method is theoretically supported by domain adaptation error bounds such as:

$$e_T(h) - \widehat{e_W}(h) \leq O\left(\sqrt{\frac{D_R^2(P_T||P_S)}{n}}\right)$$

where D_R^2 is the Rényi divergence between the source and target feature distributions, and n is the sample size. This shows that, under covariate shift, importance reweighting can yield unbiased risk estimates for the target domain and improve model transferability from baseball to softball.

To correct for distribution differences, importance weighting is applied using the adapt package in python. Each training instance is weighted by how representative it is of the target domain, using the ratio probability distribution of the target divided by probability distribution of the source. This adjustment ensures that the model trained on the source data better reflects the target domain's characteristics.

When you have some labeled data in your target domain**,** you can blend training from both the source and target. "Gamma weighting" is a what was used to control how much you rely on each data set.

$$Gamma = \frac{P_T(X)}{P_S(X)}$$

If gamma is closer to 0, you lean heavily on source data. If gamma is closer to 1, you rely more on the target. It gives you a continuous dial to adjust how much influence the target data has in training.

6.2 Model Comparison

A confusion matrix is a summary table used to evaluate the performance of a classification model by comparing the model's predicted labels to the true labels. It displays the counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), which form the foundation for several important evaluation metrics.

TABLE 8. Structure of a Confusion Matrix

	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

Accuracy: The measure the overall correctness of the model and is calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision: The quantification how often the model's positive predictions are actually correct:

$$Precision = \frac{TP}{TP + FP}$$

Recall (also known as sensitivity): The indication how many of the actual positives were correctly identified:

$$Recall = \frac{TP}{TP + FN}$$

F1: The harmonic mean of precision and recall, providing a single measure that balances both:

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

Specificity measures the proportion of actual negatives that were correctly predicted:

$$Specificity = \frac{TN}{TN + FP}$$

Lastly, the *Area Under the Curve (AUC)* reflects the model's ability to distinguish between positive and negative classes. A higher AUC (closer to 1) indicates better model performance in terms of separability.

Since the target variable in this analysis represents the minority class, I will prioritize evaluation metrics that best capture the model's ability to correctly identify these rare but important cases. Specifically, recall and precision will serve as the primary focus, as recall ensures that as many true positives as possible are detected, while precision minimizes false positives that could undermine the model's usefulness. The F1 score, as the harmonic mean of precision and recall, will also be emphasized because it balances both concerns and provides a more reliable measure of performance in imbalanced datasets. By focusing on these metrics first, I can ensure that the chosen model is not only accurate in general but also effective at correctly identifying the minority class, which is the main objective of this study.

These metrics provide a comprehensive view of classification performance. I will use them to evaluate and compare the effectiveness of each model—logistic regression, decision tree, random forest, and transfer learning—for selecting the best-performing classifier in this analysis.

6.3 Model Selection

In this section, I will walk through the model selection process, highlighting both the similarities and differences among the models used. While each model followed a comparable approach in terms of structure and methodology, there were key distinctions in how they handled the data and generated predictions. I will discuss each model individually to outline these variations, and then conclude by comparing their F1 scores to determine which performed best and is most suitable for further analysis.

For the logistic regression model, feature selection was first performed using the AIC criterion to identify key predictors. A reduced model was then run with the selected features, and test metric thresholds were adjusted to optimize F1 performance. F1 scores were compared between the full and reduced models to determine which performed best.

For the decision tree and random forest models, hyperparameters were manually tuned through a trial-and-error process to optimize the F1 score. The fully optimized parameters produced uninterpretable results, so manual adjustment was preferred. Test metric thresholds were adjusted to optimize F1 performance, but the models themselves were not tuned based on those thresholds. Both models were then reduced based on the hierarchy of feature significance, and the version with the highest F1 score was selected.

The domain adaptation process begins by training and fitting models on the baseball dataset, which serves as the source domain, before applying the trained models to the softball dataset, the target domain. The purpose of this approach is to leverage the larger and more mature baseball data to improve predictive accuracy for softball outcomes. All model tuning and experimentation were performed exclusively on the baseball data, while the softball data was reserved for final testing to ensure model integrity. The optimal model is identified and tuned in which hyperparameters were selected based on the best F1 score and accuracy achieved on the baseball data. The model's performance on baseball was comparable, and in some cases exceeded, its performance on the softball data after domain adaptation, demonstrating the effectiveness of this transfer-learning approach.

Because the dataset is unbalanced, meaning one outcome occurs far more often than the other, accuracy alone can be misleading since a model could perform well simply by predicting the majority class. The F1 score provides a more reliable evaluation by balancing precision and

recall, ensuring the model effectively identifies both positive and negative cases. Prioritizing the best F1 score therefore allows for a fairer assessment of performance and ensures that the model captures meaningful patterns in the minority class, which is critical for accurate prediction and practical decision-making in this context.

CHAPTER 7

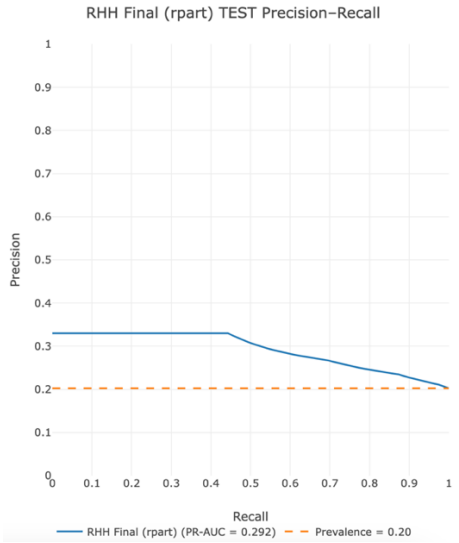
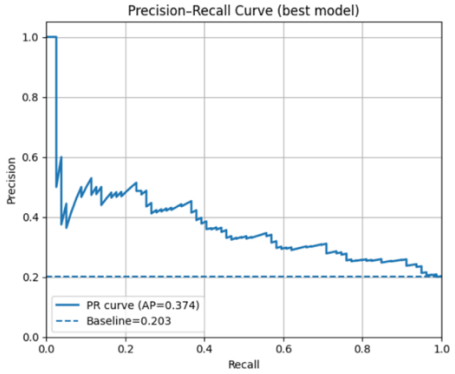
RESULTS

In the sections that follow, I will present the findings for each of the five key outcomes for RHH**:** hits, OBP, home runs, strikeouts, and walks. The LHH test results can be found in Appendix I. For each outcome, I will report the top two models identified using the model selection criteria outlined in Chapter 6. This consistent format allows for a clear comparison of predictive performance across outcomes, highlighting the strengths and limitations of each modeling approach.

Additionally, across all outcomes, models that separated left-handed hitters and right-handed hitters consistently outperformed those that combined them. This separation not only improved the models' ability to track pitch type and location, but also enhanced interpretability, allowing for more meaningful insights into hitter-pitcher matchups and zone-specific tendencies. To ensure robust evaluation, the models were trained on data from 2023, 2024, and 2025 non-conference games, while the 2025 conference games were held out as the test set. This temporal and contextual split was chosen to better simulate real-world prediction settings and evaluate model generalizability on high-stakes matchups.

TABLE 9. Hits Test Results

Test Metrics	Decision Tree				Domain Adaptation			
Confusion Matrix		Actual				Actual		
	Predicted		0	1	Predicted		0	1

	<table><tr><td></td><td>0</td><td>86</td><td>10</td></tr><tr><td></td><td>1</td><td>225</td><td>69</td></tr></table> N = 390	<table><tr><td></td><td>0</td><td>270</td><td>49</td></tr><tr><td></td><td>1</td><td>41</td><td>30</td></tr></table> N = 390
Precision	0.23	0.42
Recall	0.87	0.34
F1	0.37	0.40
Accuracy	0.40	0.77
AUC	0.64	0.69
PR-AUC	 PR-AUC = 0.292	 PR-AUC = 0.374
Gamma	NA	0.05

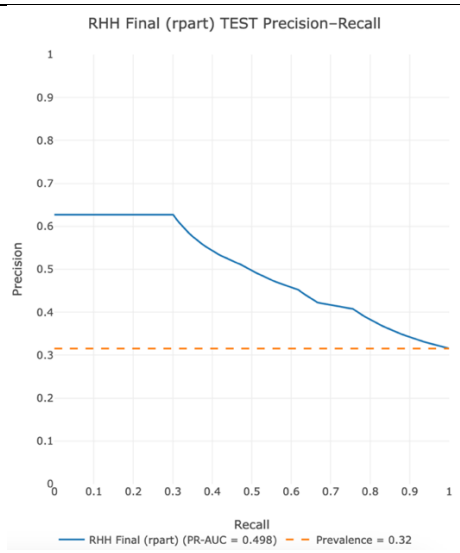
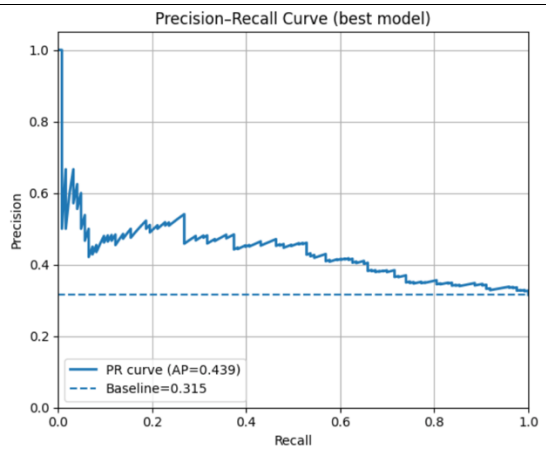
The performance results presented in the confusion matrices and summary metrics highlight key differences between the decision tree and domain adaptation models. The decision tree model correctly identified most of the positive cases, as shown by its high recall of 0.87, but it did so at the expense of precision, producing a large number of false positives. This means that while the model was effective at capturing nearly all true instances, it frequently misclassified negative cases as positive. As a result, its overall F1 score of 0.37 reflects an imbalance between sensitivity and accuracy in predictions. The relatively low accuracy of 0.40 and AUC of 0.64 further indicate that the decision tree struggled to make consistent, reliable classifications.

In contrast, the domain adaptation model demonstrated stronger overall performance and greater balance between precision and recall. Although its recall dropped to 0.34, meaning it identified fewer true positives, its precision improved to 0.42, resulting in fewer false alarms. This trade-off led to a higher F1 score of 0.40, showing that the model made more accurate and dependable predictions overall. The higher accuracy of 0.77 and AUC of 0.69 also suggest that the domain adaptation model was more effective in distinguishing between positive and negative outcomes.

Overall, the domain adaptation model outperformed the decision tree across most evaluation metrics, particularly in terms of accuracy, precision, and F1 score. While the decision tree was more aggressive in identifying positive cases, its tendency to overpredict reduced its reliability. The domain adaptation model achieved a more desirable balance, making it the better-performing model for this analysis.

TABLE 10. OBP Test Results

<table>
<tr><th>Test Metrics</th><th>Decision Tree</th><th>Domain Adaptation</th></tr>
<tr><td>Confusion Matrix</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>132</td><td>30</td></tr>
<tr><td>1</td><td>133</td><td>93</td></tr>
</table>
N = 390
</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>151</td><td>43</td></tr>
<tr><td>1</td><td>116</td><td>80</td></tr>
</table>
N = 390
</td></tr>
<tr><td>Precision</td><td>0.41</td><td>0.41</td></tr>
<tr><td>Recall</td><td>0.76</td><td>0.65</td></tr>
<tr><td>F1</td><td>0.53</td><td>0.50</td></tr>
<tr><td>Accuracy</td><td>0.58</td><td>0.59</td></tr>
<tr><td>AUC</td><td>0.68</td><td>0.63</td></tr>
</table>

PR-AUC	 PR-AUC = 0.498	 PR-AUC = 0.439
Gamma	NA	0.26

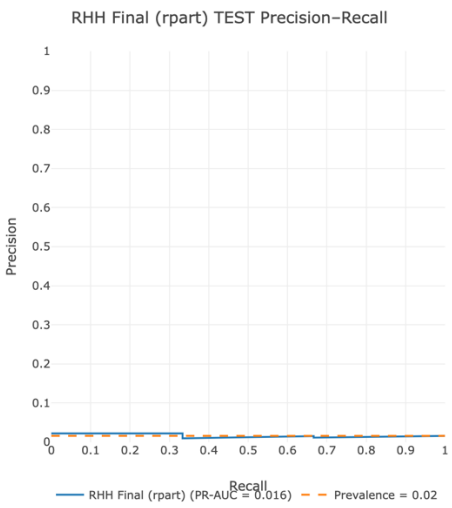
The confusion matrices and evaluation metrics provide a comparison between the decision tree and domain adaptation models in predicting the target outcome. The decision tree model shows stronger recall (0.76), meaning it correctly identified a larger portion of true positive cases, while maintaining a precision of 0.41. This suggests that the model was effective at capturing most of the positive instances but still produced a moderate number of false positives. Its F1 score of 0.53 indicates a balanced relationship between precision and recall, making it a relatively reliable model for detecting true outcomes. The decision tree also achieved an accuracy of 0.58 and an AUC of 0.68, showing that it performed reasonably well at distinguishing between classes.

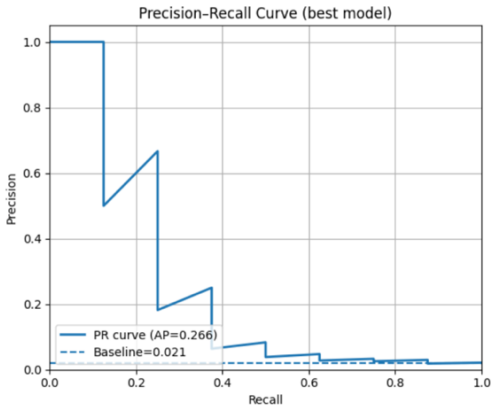
The domain adaptation model, by comparison, produced similar precision (0.41) but a lower recall of 0.65, meaning it missed a larger number of true positive cases. Its F1 score of 0.50 reflects a slight decline in overall performance compared to the decision tree. While the model achieved a slightly higher accuracy (0.59), its AUC of 0.63 and lower PR-AUC value

indicate that it was less consistent in correctly ranking or identifying positive cases across thresholds.

Overall, the decision tree model outperformed the domain adaptation model in this comparison, as shown by its higher recall, F1 score, and AUC. Although both models performed similarly in terms of precision and accuracy, the decision tree's stronger ability to detect true positives while maintaining balanced precision makes it the better-performing option.

TABLE 11. Home Runs Test Results

<table>
<tr><th>Test Metrics</th><th>Random Forest</th><th>Domain Adaptation</th></tr>
<tr><td>Confusion Matrix</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>394</td><td>7</td></tr>
<tr><td>1</td><td>4</td><td>2</td></tr>
</table>
N = 404
</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>381</td><td>6</td></tr>
<tr><td>1</td><td>1</td><td>2</td></tr>
</table>
N = 390
</td></tr>
<tr><td>Precision</td><td>0.33</td><td>0.67</td></tr>
<tr><td>Recall</td><td>0.22</td><td>0.25</td></tr>
<tr><td>F1</td><td>0.27</td><td>0.36</td></tr>
<tr><td>Accuracy</td><td>0.97</td><td>0.98</td></tr>
<tr><td>AUC</td><td>0.87</td><td>0.70</td></tr>
<tr><td>PR-AUC</td><td> 

PR-AUC = 0.016</td><td> 

PR-AUC = 0.266</td></tr>
</table>

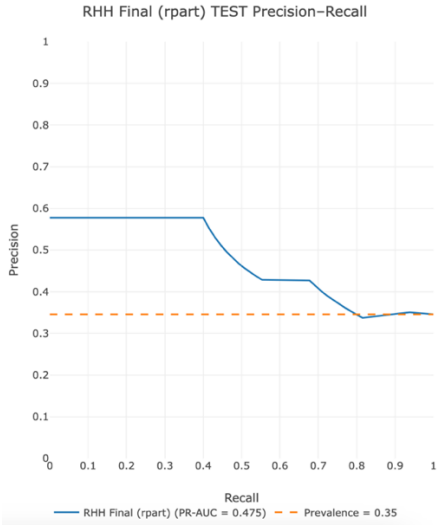
Gamma	NA	0.75

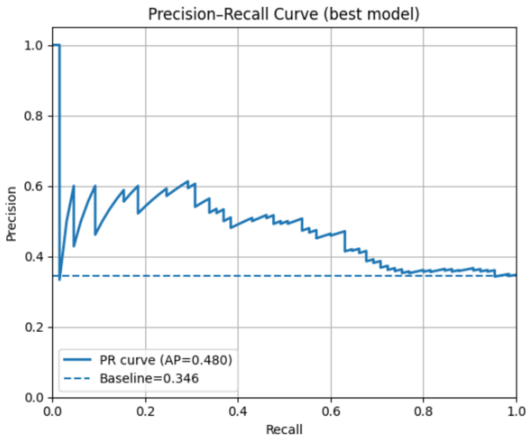
The comparison between the logistic regression and domain adaptation models shows clear differences in predictive behavior and overall performance. The logistic regression model achieved perfect recall (1.00), meaning it correctly identified all positive cases, but its precision was extremely low at 0.04. This indicates that while the model was highly sensitive, it classified nearly all observations as positive, resulting in many false positives. Consequently, its F1 score of 0.07 reflects poor balance between precision and recall, and its accuracy of 0.45 suggests that it often misclassified outcomes. Although the model's AUC of 0.89 shows it can separate classes well in theory, its poor calibration and high false-positive rate make it unreliable for practical use.

The domain adaptation model, on the other hand, demonstrated far more consistent and dependable performance. It achieved much higher precision (0.67), meaning most of its positive predictions were correct, though its recall dropped to 0.25, indicating it missed some true positives. This trade-off produced a stronger F1 score of 0.36, reflecting a better overall balance between identifying true cases and avoiding false alarms. Additionally, the domain adaptation model reached an accuracy of 0.98, showing that it made correct classifications in nearly all cases. While its AUC (0.70) and PR-AUC (0.266) were lower than the logistic regression model, its improved precision and balance make it the more practical and effective choice.

Overall, the domain adaptation model outperformed logistic regression in terms of reliability and real-world usefulness. The logistic regression model was overly aggressive, predicting too many false positives despite high theoretical separability, while the domain adaptation model provided more precise, accurate, and actionable predictions.

TABLE 12. Strikeout Test Results

<table>
<tr><td>Test Metrics</td><td>Decision Tree</td><td>Domain Adaptation</td></tr>
<tr><td>Confusion Matrix</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>40</td><td>16</td></tr>
<tr><td>1</td><td>83</td><td>49</td></tr>
</table>
N = 188
</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>77</td><td>25</td></tr>
<tr><td>1</td><td>46</td><td>40</td></tr>
</table>
N = 188
</td></tr>
<tr><td>Precision</td><td>0.37</td><td>0.47</td></tr>
<tr><td>Recall</td><td>0.75</td><td>0.62</td></tr>
<tr><td>F1</td><td>0.50</td><td>0.53</td></tr>
<tr><td>Accuracy</td><td>0.47</td><td>0.62</td></tr>
<tr><td>AUC</td><td>0.62</td><td>0.62</td></tr>
<tr><td>PR-AUC</td><td> 

PR-AUC = 0.475</td><td> 

PR-AUC = 0.480</td></tr>
<tr><td>Gamma</td><td>NA</td><td>0.05</td></tr>
</table>

The comparison between the decision tree and domain adaptation models shows that both performed similarly, but the domain adaptation model achieved slightly better overall results. The decision tree model produced a recall of 0.75, meaning it successfully identified most of the true positive cases, but its lower precision of 0.37 indicates that it also generated a higher number of false positives. This imbalance led to an F1 score of 0.50, reflecting moderate predictive

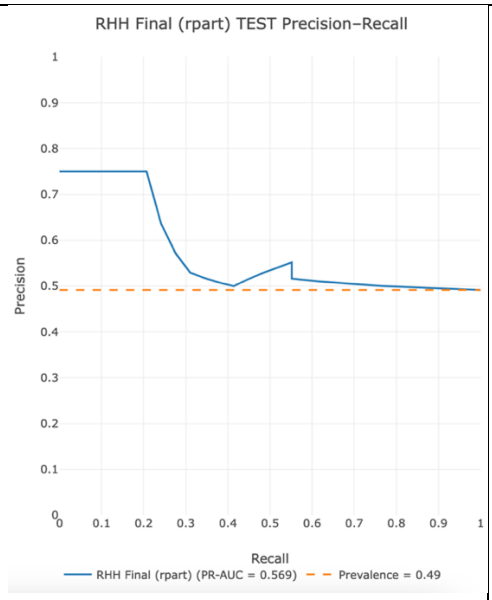
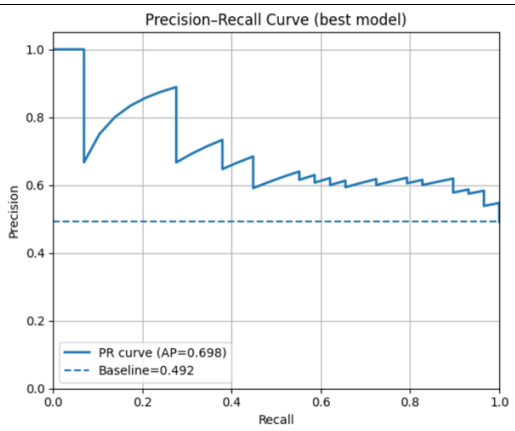
consistency. Its overall accuracy of 0.47 and AUC of 0.62 suggest that while the model captured many true cases, it struggled with correctly classifying negative ones.

The domain adaptation model demonstrated more balanced performance, with improvements in both precision (0.47) and accuracy (0.62). Although its recall dropped slightly to 0.62, it made up for it by producing fewer false positives, leading to a higher F1 score of 0.53. Both models achieved the same AUC of 0.62 and nearly identical PR-AUC values, indicating similar overall discrimination ability, but the domain adaptation model provided a better trade-off between capturing true positives and maintaining precision.

Overall, the domain adaptation model performed slightly better, offering more reliable and balanced predictions. While the decision tree was more aggressive in identifying true positives, the domain adaptation model's improved precision, accuracy, and F1 score make it the stronger and more stable option for this analysis.

TABLE 13. Walks Test Results

<table>
<tr><th>Test Metrics</th><th>Random Forest</th><th>Domain Adaptation</th></tr>
<tr><td>Confusion Matrix</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>23</td><td>14</td></tr>
<tr><td>1</td><td>7</td><td>15</td></tr>
</table>
N = 59
</td><td>
<table>
<tr><td></td><td colspan="3">Actual</td></tr>
<tr><td rowspan="3">Predicted</td><td></td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>14</td><td>3</td></tr>
<tr><td>1</td><td>16</td><td>26</td></tr>
</table>
N = 59
</td></tr>
<tr><td>Precision</td><td>0.68</td><td>0.62</td></tr>
<tr><td>Recall</td><td>0.52</td><td>0.90</td></tr>
<tr><td>F1</td><td>0.59</td><td>0.73</td></tr>
<tr><td>Accuracy</td><td>0.64</td><td>0.68</td></tr>
<tr><td>AUC</td><td>0.71</td><td>0.71</td></tr>
</table>

PR-AUC	 PR-AUC = 0.569	 PR-AUC = 0.698
Gamma	NA	0.82

The comparison between the random forest and domain adaptation models shows that both performed well, but the domain adaptation model achieved stronger overall results. The random forest model demonstrated a precision of 0.68, meaning most of its positive predictions were correct, but its recall of 0.52 indicates it missed nearly half of the true positive cases. This balance resulted in an F1 score of 0.59 and an overall accuracy of 0.64. While its AUC of 0.71 suggests good class separation, the model leaned toward being conservative, identifying fewer true positives.

The domain adaptation model, however, achieved a better balance between precision and recall. With a precision of 0.62 and a much higher recall of 0.90, it captured nearly all true positives while maintaining reasonable accuracy. This combination produced a stronger F1 score of 0.73 and a slightly higher overall accuracy of 0.68. Both models achieved the same AUC of 0.71, but the domain adaptation model's higher PR-AUC value (0.698 compared to 0.569) indicates that it performed better across varying decision thresholds.

Overall, the domain adaptation model outperformed the random forest by providing a better mix of accuracy, recall, and consistency. Its ability to identify more true positives while maintaining solid precision makes it the more effective and dependable model in this comparison.

7.1 Model Interpretation

In this section, I focus on analyzing the best-performing traditional models. While domain adaptation models rely more heavily on machine learning techniques and are primarily used for detection, the traditional models provide a clearer foundation for interpretation and understanding. By examining these models, we can identify the key factors that influence outcomes and validate that our results align with real-world softball trends and coaching knowledge.

7.1.1 Hits

The literature review emphasized the importance of minimizing hits, as limiting offensive contact is key to overall pitching success. Building on this, we will now explore the specific features and conditions that contribute to reducing hits, identifying which pitch types, locations, and situational factors are most effective in preventing solid contact and keeping runners off base. A decision tree was selected.

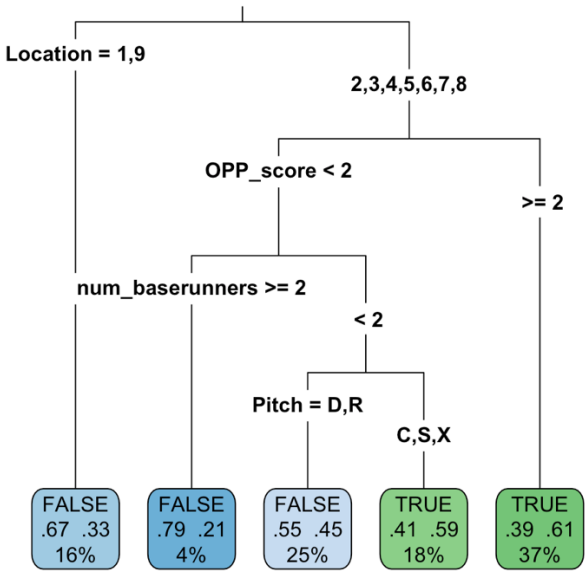


FIGURE 26. Hits decision tree model.

The decision tree illustrates the conditions under which hitters are more or less likely to record a hit, with "TRUE" indicating a hit and "FALSE" indicating not a hit. As a pitching staff, the objective is to minimize occurrences of "TRUE" outcomes. The model reveals that pitch location is the most influential factor in determining hitting success. When pitches are located in zones 1 or 9, hitters are less likely to get a hit, particularly when the opponent's score is low and there are multiple baserunners. In this scenario, the probability of a hit is only 21%, indicating strong effectiveness in limiting offensive production. Even when the opponent's score is higher or the bases are less occupied, these outside locations continue to yield lower hit rates (33–45%), reinforcing their strategic value. In contrast, pitches thrown to other locations (zones 2–8) are associated with a higher likelihood of hits, especially when the opponent's score is greater than or equal to two, where the hit probability increases to 61%. Additionally, when the opponent's score is low but the pitch type is classified as C, S, or X in these central zones, the chance of a hit remains elevated at 59%. Overall, this analysis suggests that targeting locations 1 and 9 is the

most effective strategy for reducing hits, while avoiding central zones—particularly when using certain pitch types or in higher-scoring situations—can further suppress offensive outcomes.

7.1.2 OBP

In this analysis, we focus on OBP because it captures all instances where a batter successfully reaches base, whether through a hit or walk. From a pitching perspective, any baserunner—regardless of how they reach—is a potential scoring threat, so minimizing OBP is essential to limiting offensive production. Unlike outcome-specific models that separate hits and walks, OBP provides a broader view of pitcher effectiveness in preventing runners altogether. By investigating OBP using a decision tree model, we can identify the key factors and conditions that most strongly influence whether a batter reaches base. This approach allows us to interpret the most impactful features that contribute to base runners and, ultimately, helps guide strategies to reduce overall baserunner frequency.

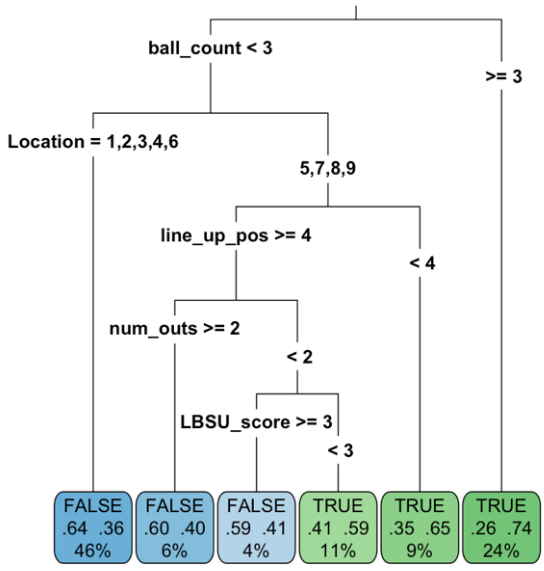


FIGURE 27. OBP decision tree model.

This decision tree shows how different pitch and game conditions relate to whether a batter reaches base (TRUE) or is retired (FALSE), meaning whether the pitcher successfully prevents a baserunner. Since the goal is to minimize on-base percentage, we want the FALSE outcomes, where no runner reaches.

The model's first and most important split is ball count less than three, indicating that keeping the count below three balls is the strongest predictor of preventing a baserunner. When pitchers stay ahead in the count, they control the at-bat and are much more likely to record an out. Within those lower-ball-count situations, location also matters: pitches thrown to zones 1, 2, 3, 4, or 6 most often result in outs, with these paths leading to FALSE outcomes about 46% of the time. This suggests that throwing strikes in those locations early in the count is the most effective combination for limiting on-base percentage.

When the ball count reaches three or more, the probability of a TRUE outcome (a batter reaching base) increases sharply—74% of these situations result in someone getting on base. This reinforces that deep counts are dangerous, as they often force pitchers to throw more predictable or hittable pitches. Additionally, when the ball count is low but pitches are located in zones 5, 7, 8, or 9, the outcome depends on other context: facing top-of-the-lineup hitters or pitching with fewer than two outs increases the likelihood of allowing a baserunner. Overall, this decision tree shows that the best way to prevent runners is to stay ahead in the count and work to the outer or lower zones (1–4 or 6), while deep counts and poor pitch location, especially against stronger hitters, increase the chance of someone reaching base.

This model, tested on 390 observations, shows moderate performance with an accuracy of 0.58 and an AUC of 0.68 in Table 10. It captures most instances where a batter reaches base, as shown by its high recall of 0.76, but its lower precision of 0.41 indicates it overpredicts those

situations. The balance between precision and recall (F1 = 0.53) suggests it's better at identifying risk than avoiding false alarms. Overall, the model is useful for spotting potential on-base threats but would benefit from improved precision to better distinguish true baserunner situations from false positives.

7.1.3 Homeruns

In this analysis, we focus on developing models to predict homeruns. Because homeruns occur far less frequently than other outcomes such as singles, walks, or hits, our dataset is inherently imbalanced. This imbalance presents unique challenges for model performance and evaluation, as traditional metrics may overstate accuracy by favoring the majority class. Our goal is to identify which pitch features are most strongly associated with the likelihood of a home run, while carefully addressing the effects of class imbalance to ensure reliable and interpretable results. In the EDA, it hints that pitches thrown in the middle of plate and getting behind in the count put a pitcher at risk for hitters hitting a homerun. Logistics regression domain adaptation is chosen for this investigation.

The home run models should be applied exclusively to power hitters, as these models are specifically designed to analyze and predict outcomes related to home run performance. Coaches can determine which players qualify as power hitters by reviewing their season statistics, particularly the total number of home runs they have hit. This ensures that the model is used appropriately and yields meaningful insights for players whose performance aligns with the intended scope of the analysis.

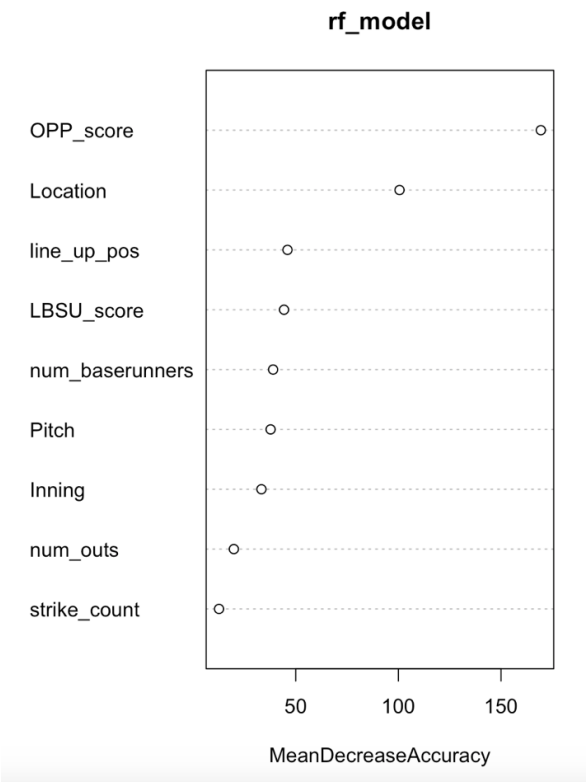


FIGURE 28. Home run random forest model.

The random forest model identifies the top three most important variables influencing whether a play results in a home run as OPP_score, Location, and line_up_pos. OPP_score has the greatest impact, indicating that the opponent's score plays a key role in shaping conditions where home runs are more likely, possibly reflecting the pressure or pitching decisions made when trailing or leading by certain margins. Location is the second most influential factor, suggesting that the setting of the game, such as specific ballparks or whether the team is at home or away, affects the frequency of home runs. The third most important variable, line_up_pos, implies that the batting order position contributes meaningfully to predicting home runs, as certain positions may have stronger power hitters or situational tendencies. Together, these variables highlight that game context and lineup structure are central to the model's understanding of when home runs are likely to occur.

7.1.4 Strikeouts

In the EDA, it was found that team strikeout totals were significantly associated with higher winning percentages**,** highlighting strikeouts as a key driver of team success. Given this relationship, it is important to take a deeper look into this outcome to identify which features are most influential in generating strikeouts. By understanding these factors, teams can better target controllable elements in both games and practice settings to improve strikeout performance collectively. This data was set to only include 2-strike counts.

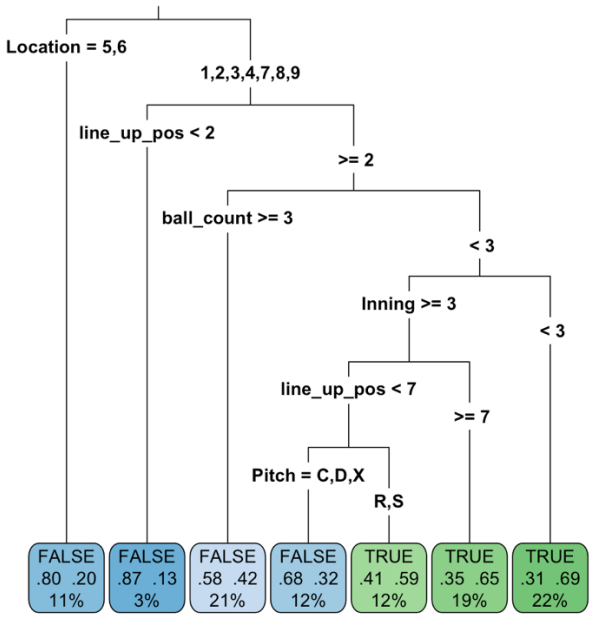


FIGURE 29. Strikeout decision tree model.

This decision tree represents the conditions that influence strikeout outcomes, where "TRUE" indicates a strikeout and "FALSE" indicates that the batter did not strike out. Since the goal from a pitching standpoint is to maximize strikeouts, the focus should be on the "TRUE" branches.

The first and most important split is based on pitch location. When pitches are located in zones 5 and 6, the probability of a strikeout is relatively low, with only 20% of outcomes being

true. In contrast, pitches located in zones 1, 2, 3, 4, 7, 8, and 9 create more opportunities for strikeouts, particularly under certain situational conditions. Within these locations, lineup position, ball count, and inning play key roles in determining strikeout likelihood.

For hitters in the top of the lineup (line_up_pos < 2), strikeouts are uncommon, with only 13% resulting in a strikeout, suggesting that these hitters have stronger plate discipline or contact skills. However, when facing batters in lower lineup positions (line_up_pos ≥ 2), the likelihood of a strikeout increases under specific circumstances. When the ball count is high (≥ 3), strikeout rates remain modest at 32%, but when pitchers work ahead in the count (ball_count < 3), the chances of a strikeout rise substantially depending on the inning and pitch type.

During later innings (inning ≥ 3), strikeouts are more likely among batters hitting lower in the lineup (line_up_pos ≥ 7), especially when using certain pitch types. When pitchers rely on pitches classified as R or S, the probability of a strikeout increases to 65–69%, marking the most favorable scenario for generating strikeouts. In contrast, pitches labeled as C, D, or X show a slightly lower strikeout probability of 59%, though they still represent effective strikeout options when executed under the right conditions.

Overall, the analysis indicates that maximizing strikeouts depends on both pitch location and situational control. Pitchers are most effective when working in locations outside the central zones (1–4 and 7–9), staying ahead in the count, and using specific pitch types such as R and S during later innings against the bottom half of the lineup. These results suggest that command early in the count, paired with effective pitch sequencing in later innings, leads to the highest strikeout success.

7.1.5 Walks

In the cluster analysis in EDA, it was found that team strikeout to BB ratio were related to teams in Cluster 2 with the highest winning percentage. Thus, it is important for pitchers to "work ahead in the count". This is a common phrase used by coaches to their pitchers, and it means to start the at bat throwing strikes and then slowly inch away from the strike zone as the at bat continues. The models shown below are from a subset of the data where the ball count is already at 3 balls. Pitchers generally do not want to get to 3 balls in an at bat, but it happens. The goal of these models is to see which pitches drive the most success in a count with 3 balls.

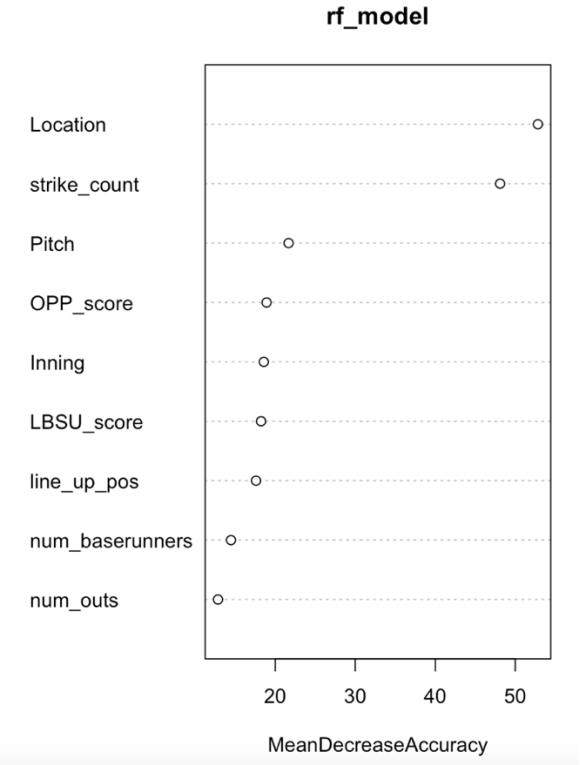


FIGURE 30. Walk random forest model.

This random forest model evaluates the factors influencing walks, where higher importance values (as measured by Mean Decrease Accuracy) indicate stronger predictors of whether a walk occurs. Since the goal from a pitching perspective is to minimize walks, attention should focus on controlling the variables that have the greatest effect on walk probability.

According to the model, Location and strike_count are the two most influential variables. Location shows the highest Mean Decrease Accuracy, meaning that pitch placement has the greatest impact on avoiding walks. Maintaining command of pitch location—especially by consistently hitting the strike zone and avoiding marginal areas—is therefore the most effective way to prevent free passes. Strike_count follows closely behind, suggesting that getting ahead early in the count and effectively finishing at-bats are critical to reducing walks.

Overall, the random forest analysis emphasizes that minimizing walks depends primarily on consistent pitch location and early-count control**.** By refining location accuracy and maintaining an aggressive approach within the strike zone, pitchers can significantly reduce the likelihood of issuing walks. Situational awareness and confident pitch selection further support this outcome, but mechanical consistency and strike-zone precision remain the most decisive factors in preventing walks.

CHAPTER 8

COACHES' ANALYSIS

The following coaches' analysis presents the model findings in a clear, concise, and practical format designed specifically for in-game and pre-game application. These summaries are intended to serve as a quick reference tool—a way for coaches and pitchers to easily interpret complex data at a glance. They are not meant to dictate decision-making or replace situational judgment, but rather to enhance it. By offering data-driven insights in an accessible form, these visuals and summaries can help coaches leverage probabilities, especially in moments of uncertainty or when deciding between two viable options. In some cases, these insights may also reveal patterns or tendencies that were previously overlooked, providing new strategic

angles to inform pitch calling and overall game planning. All models are summaries of the models in chapter 7.

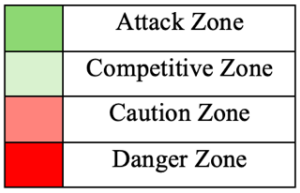


FIGURE 31. Legend for summaries.

Each summary includes a zone map that identifies where pitchers should attack with specific pitch types and where they should exercise caution. A color-coded legend is provided in Figure 31 to explain the zones and corresponding pitch recommendations.

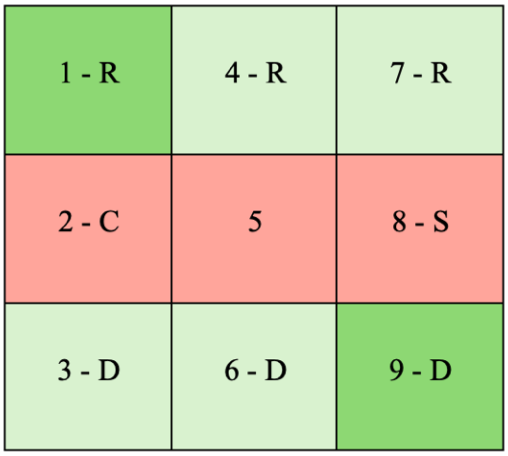


FIGURE 32. Recommended pitch locations and pitch types by zone to limit hits.

Coaches should emphasize attacking the outer edges of the strike zone, specifically zones 1 and 9, as these areas produce the lowest hit rates. Pitchers should be encouraged to work these locations consistently, particularly when the opponent's score is low or when multiple baserunners are on base, as these scenarios further reduce the likelihood of a hit. Conversely, coaches should advise pitchers to avoid central zones (2–8), especially when the opponent's score is higher or when using certain pitch types such as C, S, or X, since these situations result

in more frequent hits. Overall, the focus should be on commanding pitches to the outer edges and minimizing mistakes in the middle of the zone to limit offensive production.

1	4	7 (after 3 batter)
2	5	8 (after 3 batter)
3	6	9 (after 3 batter)

FIGURE 33. Recommended pitch locations and pitch types by zone to limit baserunners.

To prevent baserunners effectively, pitchers should focus on getting ahead early in the count and maintaining control of the at-bat. Keeping the ball count below three is the strongest factor in reducing on-base percentage, as it forces hitters to be more defensive and limits their ability to wait for ideal pitches. When working in these low-count situations, targeting zones 1, 2, 3, 4, and 6 has been shown to produce the most outs, making them the most effective areas to attack. In contrast, once the count reaches three balls or more, the likelihood of allowing a baserunner increases significantly, with nearly three-quarters of these situations resulting in someone reaching base. Coaches should emphasize the importance of avoiding deep counts and maintaining consistent command within the strike zone. Additionally, when facing top-of-the-lineup hitters or pitching with fewer than two outs, pitchers should be especially cautious, as these conditions increase the risk of giving up baserunners even with low ball counts. Overall, staying ahead in the count, locating pitches in the outer and lower zones, and maintaining command early in at-bats are key strategies for minimizing on-base percentage.

1	4	7
2 - X	5	8 - S
3 – D, X	6	9 – D, X

FIGURE 34. Recommended pitch locations and pitch types by zone to limit home runs.

The model shows that home runs are most influenced by the opponent's score, game location, and lineup position. When the opponent's score is high, pitchers may become predictable, increasing home run risk—so maintaining pitch variety and focus under pressure is key. Certain ballparks also appear more prone to home runs, suggesting the need for adjusted pitch strategies or defensive alignments. Lastly, specific lineup spots pose greater power threats, so matching pitchers carefully to those hitters can help limit damage. Overall, strategic adjustments based on game context, location, and lineup tendencies can reduce home run risk.

1 - R	4 - R	7 – R, S
2 – C, D, X	5	8 – R, S
3 – C, D, X	6	9 - S

FIGURE 35. Recommended pitch locations and pitch types by zone to increase strikeouts.

To get more strikeouts, pitchers should work ahead in the count and attack the outer and upper parts of the zone. They should avoid throwing down the middle and use their R and S pitches more often in later innings, especially against the bottom of the lineup. Staying aggressive early and mixing pitch types effectively will create more swing-and-miss opportunities.

To decrease walks, pitchers should focus on getting ahead early in the count and consistently attacking the strike zone. They need to trust their stuff, avoid nibbling around the edges, and challenge hitters instead of falling behind. Staying aggressive, repeating their mechanics, and maintaining control of pitch location will help prevent free passes. Coaches should emphasize strike-first mentality, command work in bullpens, and confidence in throwing all pitches for strikes early in the count.

CHAPTER 9

CONCLUSION

This thesis has opened the door for more advanced softball data analysis by demonstrating that the integration of baseball data and analytical techniques can meaningfully enhance model performance in softball. By successfully obtaining and structuring softball pitch-level data and aligning it with corresponding baseball variables, this research established a transferable analytical framework capable of leveraging mature baseball datasets to strengthen emerging softball models. The use of domain adaptation between the two sports proved particularly valuable, improving most models and confirming that cross-sport data transfer can compensate for the current scarcity of detailed softball data.

The results show that despite the mechanical and contextual differences between baseball and softball pitching, the effects of pitch type and location on batter outcomes are remarkably

consistent across both sports. This indicates that the pitch mapping strategy—aligning baseball and softball pitch equivalents based on movement and velocity—was effective. The improved performance of softball models trained with baseball-informed domain adaptation suggests that these shared patterns can be used to accelerate the analytical development of softball performance models as larger, more refined datasets become available in the future.

Furthermore, the binary logistic regression model predicting umpire-called balls and strikes outperformed the outcome-based models developed for softball. This difference is expected, as that model included a continuous pitch location variable and excluded hitter influence, focusing instead on a simpler and more consistent decision-making process. Its high accuracy highlights the impact of modeling precision and feature granularity—reinforcing that as softball data becomes more comprehensive, particularly with continuous features like pitch location, similar improvements in performance should follow.

In summary, this research establishes a foundation for future work in softball analytics by illustrating that data integration across related sports domains is not only feasible but beneficial. As softball data collection grows in detail and consistency, predictive models will continue to mature—ultimately supporting more informed coaching strategies, player development, and competitive success.

CHAPTER 10

FUTURE WORK

As this project lays a strong foundation for integrating advanced analytics into LBSU softball, several areas remain for future development. Continued data collection, system enhancements, and strategic partnerships will significantly expand the depth and accuracy of insights derived from softball performance analysis.

One critical area for growth is pitch-by-pitch data entry. If LBSU softball can dedicate a player, staff ,or student interns to input detailed pitch-level data for future seasons, the detail of analysis will drastically improve. Continuous zone location tracking requires advanced and often expensive technology, which may not be accessible in all training or game environments. However, increasing the number of zone quadrants and making them smaller can help improve the effectiveness of the feature inputs by providing finer spatial resolution. This approach also allows for the inclusion of zones located outside the strike zone, which can capture valuable information about pitch placement and hitter behavior, ultimately enhancing the accuracy and interpretability of the models. Consistent and efficient data tracking at this level enables the identification of patterns over time, improves predictive modeling, and supports more precise coaching decisions.

If future datasets are collected in the same manner as the 2025 data—capturing every pitch within an at-bat rather than only the final pitch or outcome—it will greatly expand the range of research questions and modeling opportunities available. This level of detail allows for in-depth analysis of pitch sequencing, including how different pitches interact when thrown in succession and which combinations are most effective at generating desired results. It also enables a deeper look at tendencies behind called strikes, called balls, and swings and misses, offering insight into both pitcher decision-making and hitter behavior. Beyond individual pitch outcomes, the richer dataset could also illuminate patterns in pitcher-hitter matchups, reveal context-dependent strategies (such as pitch selection in two-strike counts or late innings), and open the door for models that better capture the dynamic and sequential nature of softball at-bats.

In addition to pitch-level data, tracking baserunner locations throughout each play would further elevate situational analysis. Understanding not just the outcome of a play but the base

state before and after can provide insights into team tendencies, baserunning efficiency, and defensive vulnerabilities. This would require an extra step taken when compiling the play-by-play single game PDFs.

Another major area for expansion is the collection of hitting data for LBSU batters. This presents a unique challenge, as we do not have access to real-time information on opposing pitchers' pitch types and locations unless there is someone sitting behind home plate during each game. Manually reviewing Synergy footage after each game is time-intensive, so future work should explore strategies for efficient post-game data entry or potential collaborations with opponent coaching staffs to share pitch charts.

The standardization of variables moving forward is also essential. Now that a foundational system has been established, future seasons should begin with a consistent set of variables, formatting, and naming conventions. This will streamline the merging and comparison of datasets across years and reduce time spent cleaning data before analysis.

Another promising area is developing a time series model based on pitch count to predict when pitchers may begin to fatigue or when hitters start to adjust. Currently, we lack a formal method of tracking batter appearances in-game beyond lineup position. Adding a unique game ID and a batter appearance index would allow us to study performance degradation or improvements over time, enabling better in-game management of pitching rotations.

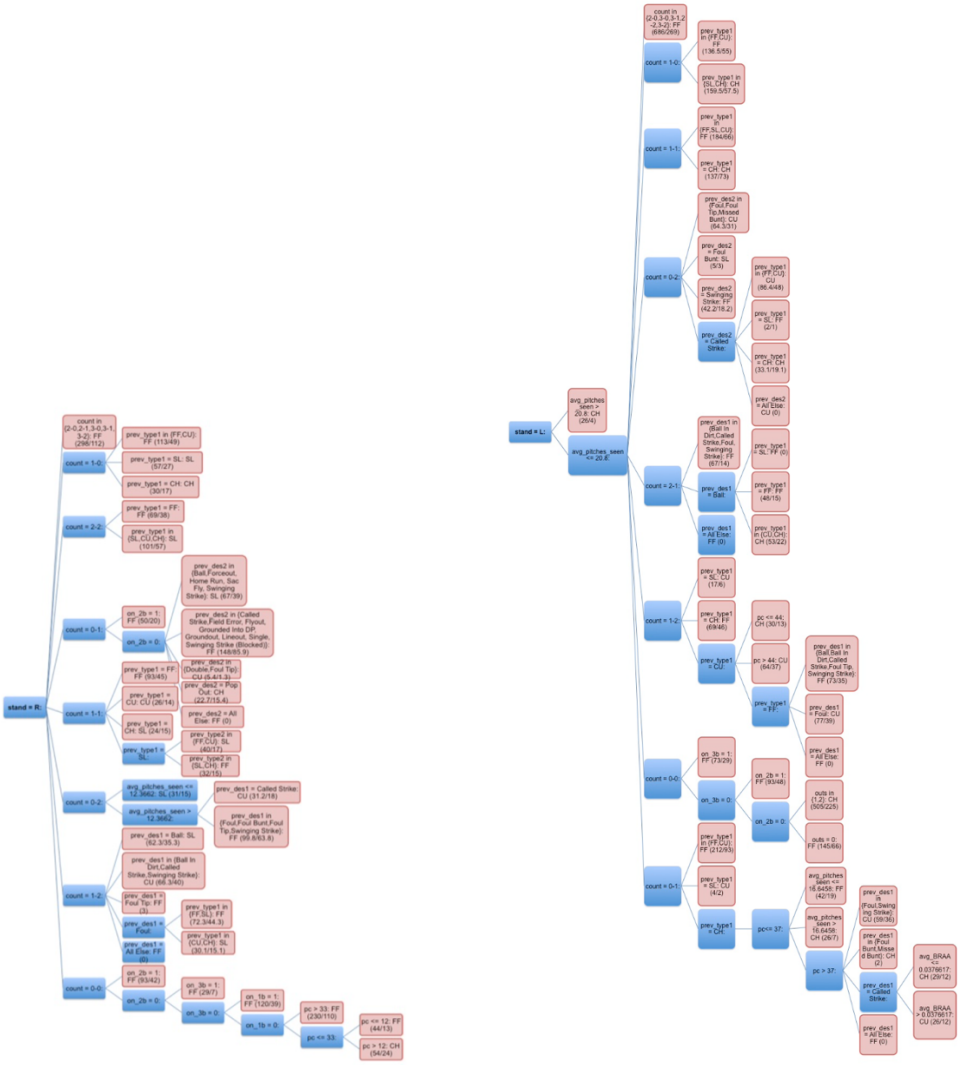
Finally, establishing a statistics internship within the athletic department could support all of the above initiatives. This role could focus on data collection, game charting, and report generation, while providing student interns with hands-on experience in sports analytics and contributing directly to the program's competitive advantage. This project could serve as the

foundation for a collaborative partnership between the athletic department and the statistics department, allowing both to share expertise and support each other's goals.

APPENDICES

APPENDIX A
DECISION TREE EXAMPLE

Justin Verlander's Pitch Selection Decision Tree (Woodward, 2014)



APPENDIX B
CHAPTER 3 R CODE

```
library(readxl)

library(dplyr)
library(MASS)
library(car)
library(nnet)
library(glmnet)
library(mgcv)
library(arm)
library(pROC)
library(ggplot2)
library(tidyr)
#robust regression
library(robustbase)
library(fmsb)

#  ========================= DATA ============================
df <- read_excel("/Users/teahthies/Desktop/CurrentLB_Stats/ERA_all_games.xlsx", sheet = 1)
nrow(df) #8
colnames(df)

#  ============ Strikeouts per appearance by pitcher bar graph ================
library(dplyr)
library(ggplot2)
library(forcats)
library(rlang)

# helper: convert baseball IP (e.g., 6.2) to true innings (6 + 2/3)
convert_ip <- function(x) {
  x_num <- suppressWarnings(as.numeric(x))
  whole <- floor(x_num)
  frac  <- x_num - whole
  out <- ifelse(
    is.na(x_num), NA_real_,
    whole + ifelse(abs(frac - 0.1) < 1e-9, 1/3,
                   ifelse(abs(frac - 0.2) < 1e-9, 2/3, 0))
  )
  out
}
```

```
rate_df <- df %>%
  mutate(
    ip_true   = convert_ip(.data[["ip"]]),
    so_per_ip = if_else(ip_true > 0, .data[["so"]] / ip_true, NA_real_)
  )

ggplot(
  rate_df %>% filter(!is.na(so_per_ip)),
  aes(x = fct_reorder(.data[["Pitcher"]], so_per_ip, .desc = TRUE), y = so_per_ip)
) +
  geom_col() +
  coord_flip() +
  labs(
    x = NULL,
    y = "Strikeouts per inning (SO/IP)",
    title = "Strikeouts per Inning by Pitcher"
  ) +
  theme_minimal(base_size = 12)

#  ============ dumbbell chart of avg vs RHH and LHH by pitcher ================
library(dplyr)
library(tidyr)
library(ggplot2)

# ----- prep (wide + long) -----
wide <- df %>%
  dplyr::select(Pitcher, LHH = avg_vs_left, RHH = avg_vs_right) %>%
  dplyr::mutate(diff = RHH - LHH) %>%                    # + means worse vs RHH
  dplyr::arrange(desc(abs(diff))) %>%
  dplyr::mutate(Pitcher = factor(Pitcher, levels = Pitcher))

long <- wide %>%
  tidyr::pivot_longer(c(LHH, RHH), names_to = "split", values_to = "avg") %>%
  dplyr::mutate(split = factor(split, levels = c("LHH","RHH")))

# overall reference (optional)
xbar <- mean(c(wide$LHH, wide$RHH), na.rm = TRUE)

# ----- plot -----
ggplot() +
```

```
# connector (use arrow to show direction LHH -> RHH)
geom_segment(
  data = wide,
  aes(x = LHH, xend = RHH, y = Pitcher, yend = Pitcher),
  linewidth = 0.9, color = "grey70",
  arrow = arrow(length = unit(4, "pt"), type = "closed")
) +
# points, colored by split
geom_point(
  data = long,
  aes(x = avg, y = Pitcher, color = split),
  size = 3
) +
# optional: annotate difference at the rightmost end
geom_text(
  data = wide,
  aes(x = pmax(LHH, RHH) + 0.005, y = Pitcher,
      label = sprintf("%+.3f", diff)),
  size = 3, hjust = 0
) +
# overall reference line
geom_vline(xintercept = xbar, linetype = "dashed", linewidth = 0.5, alpha = 0.4) +
scale_color_manual(
  values = c("LHH" = "#2C7FB8", "RHH" = "#D95F02"),
  labels = c("LHH (avg_vs_left)", "RHH (avg_vs_right)")
) +
scale_x_continuous(labels = scales::label_number(accuracy = 0.001)) +
labs(
  title = "Opponent AVG by Pitcher vs LHH and RHH",
  subtitle = "Connector shows LHH → RHH; label is (RHH − LHH)",
  x = "Opponent batting average (lower is better)",
  y = NULL, color = NULL
) +
coord_cartesian(clip = "off") +
theme_minimal(base_size = 12) +
theme(
  legend.position = "top",
  panel.grid.major.y = element_blank(),
  plot.margin = margin(10, 40, 10, 10)  # extra right room for labels
)
```

```
#  ============ Bar Graph of pitchers vs batter with runners on base. ============
library(dplyr)
library(ggplot2)
library(scales)

# 1) Prep: keep pitcher + runners-on split, drop non-pitcher summary rows if present
dat <- df %>%
  dplyr::select(Pitcher, avg_w_baserunner) %>%
  dplyr::filter(!Pitcher %in% c("Opp", "Opponents", "Total", "Totals")) %>%
  dplyr::mutate(
    highlight = dplyr::case_when(
      Pitcher == "Haddad"  ~ "Haddad",
      Pitcher == "Barnett" ~ "Barnett",
      Pitcher == "Nally"   ~ "Nally",
      TRUE                 ~ "Other"
    ),
    # order by performance (lowest = best) for plotting
    Pitcher = reorder(Pitcher, avg_w_baserunner)
  )

# 2) Colors: green = standout best (Haddad), blue = strong (Barnett), red = weakness (Nally),
grey = others
cols <- c("Haddad"="#1b9e77", "Barnett"="#386cb0", "Nally"="#e41a1c", "Other"="grey80")

# 3) Plot: horizontal bars with value labels
p <- ggplot(dat, aes(x = Pitcher, y = avg_w_baserunner, fill = highlight)) +
  geom_col(width = 0.6) +
  coord_flip() +
  geom_text(aes(label = number(avg_w_baserunner, accuracy = 0.001)),
            hjust = -0.1, size = 3) +
  scale_fill_manual(values = cols, guide = guide_legend(title = NULL)) +
  expand_limits(y = max(dat$avg_w_baserunner, na.rm = TRUE) + 0.02) +
  labs(title = "Opponent AVG with Runners On Base by Pitcher",
       subtitle = "Lower is better; Haddad and Barnett strong, Nally weaker",
       x = NULL, y = "Opponent AVG (runners on)") +
  theme_minimal(base_size = 12) +
  theme(legend.position = "top",
        panel.grid.major.y = element_blank())
```

```
print(p)

# (Optional) Print the exact values for the three pitchers mentioned
dat %>%
  filter(Pitcher %in% c("Barnett","Haddad","Nally")) %>%
  arrange(Pitcher) %>%
  mutate(avg_w_baserunner = number(avg_w_baserunner, accuracy = 0.001)) %>%
  print(row.names = FALSE)
```

APPENDIX C
CHAPTER 4 R CODE

######## 2023 and 2024 Play-By-Play Program

```
# This script will take the play by play pdf and turn them into an organized excel file.
# The only thing this pdf is missing that we will need to get from synergy is pitch type, and
# location, and speed.
#if note displays “EDIT”, the code varies based on what game is being complied.
#if not displays “RUN”, code can be ran as is and is automated

# EDIT - Upload Files
pdf_text <- pdf_text("/Users/teahthies/Desktop/BoxPDF2023/PDF/GAMEID.pdf") #edit

# RUN - play-by-play pdf format
# page to and on is play by play data
pdf_text_from_page_2 <- pdf_text[2:length(pdf_text)]
# get each line a row if its own
pdf_lines <- unlist(strsplit(pdf_text_from_page_2, "\n"))
# make tibble
clean_data <- pdf_lines %>%
  as_tibble()
# get name of player
clean_data <- clean_data %>%
  filter(grepl("^[A-Za-z]", value))

# EDIT - if a row in pdf goes to 2 lines, change it to 1
# Define the indices where merging should occur
merge_indices <- c(23)  # Indices where the next row should be merged
#                 -0, -1, -2, ....
# Loop through indices and merge rows
for (idx in merge_indices) {
  next_idx <- idx + 1
  # Check if the next row exists
  if (next_idx %in% rownames(clean_data)) {
    clean_data$value[rownames(clean_data) == idx] <-
paste(clean_data$value[rownames(clean_data) == idx],
                                          clean_data$value[rownames(clean_data) == next_idx])
    # Remove the next row after merging
    clean_data <- clean_data %>% filter(rownames(.) != next_idx)
  }
}
```

```
# EDIT - if we pitch or bat in top of the inning
table <- clean_data %>%
  mutate(
    Inning = case_when(
      grepl("Top", value) ~ (str_extract(value, "\\d+")),
      grepl("Bottom", value) ~ (str_extract(value, "\\d+")),
      TRUE ~ NA_character_
    ), # Bat or Pitch
    BatPitch = case_when(
      grepl("Top", value) ~ "Pitch",         #edit - LBSU is Pitching in the Top
      grepl("Bottom", value) ~ "Bat",    #edit - LBSU is Batting in the Bottom
      TRUE ~ NA_character_
    )
  ) %>%
  fill(Inning, BatPitch, .direction = "down")

# EDIT - normally not needed unless play by play gives initial
table <- table %>%
  mutate(Hitter = case_when(
# If BatPitch contains "Bat", take the name that matches lineup (first name value=1, last name
# value=2)
    str_detect(BatPitch, "Bat") ~ word(value, 1),
# If BatPitch contains "Pitch", take the name that matches lineup
    str_detect(BatPitch, "Pitch") ~ word(value, 1)
  ))

# EDIT - opposing team name
table <- table %>%
  filter(!(Hitter %in% c("Cal", "Long")))  # edit - opponent team name
## this removes columns that subs are made in the game
table <- table[-grep("for", table$value, ignore.case = TRUE), ]
## remove commas and periods and semicolons in Player column
table$Hitter <- gsub("[^A-Za-z]", "", table$Hitter)

# RUN - balls, strikes, pitch type, pitch location, speed
table$Pitch <- ""
table$Location <- ""
table$Catcher <- ""

# EDIT - get lineup
```

```
lineup <- read_excel("/Users/teahthies/Desktop/BoxPDF2023/lineup2023/Lineup.xlsx", sheet =
1)

# RUN - combine lineup
table <- table %>%
  left_join(lineup %>% select(player, line_up_pos),
            by = c("Hitter" = "player"))
table <- table %>%
  left_join(lineup %>% select(player, position),
            by = c("Hitter" = "player"))
table <- table %>%
  left_join(lineup %>% select(player, Bat_hand),
            by = c("Hitter" = "player"))

# STOP AND CHECK - check if each player name transferred from lineup (edit manually if not)
table <- table %>%
  filter(!is.na(Bat_hand))

# RUN - combine Event Table
event <- read_excel("/Users/teahthies/Desktop/BoxPDF2023/event.xlsx", sheet = 1)

table <- table %>%
  rowwise() %>%
  mutate(
    code = {
      # Check if the 'value' contains the 'event' from events_table
      matching_event <- event %>%
        filter(str_detect(value, event)) %>%
        pull(code) # Get the code for the matching event

      # If a match is found, use the code; otherwise, NA
      if (length(matching_event) > 0) matching_event[1] else NA
    }
  ) %>%
  ungroup()

# STOP AND CHECK - check for no NA CODES unless it is a sub (if not, edit manually)
table <- table %>% filter(!is.na(code))
table <- table %>%
  left_join(event %>%
```

```
            select(code, event_outs, bat_event_fl, bat_dest, swing),
          by = "code")
#remove duplicate rows
table <- table %>%
  distinct(value, .keep_all = TRUE)

# RUN - score columns
table <- table %>%
  mutate(
    score_increment = str_count(value, "\\bhomered\\b") + str_count(value, "\\bscored\\b") +
str_count(value, "\\bstole home\\b")
  )
# score columns
table <- table %>%
  mutate(
    LBSU_score = cumsum(ifelse(BatPitch == "Bat", score_increment, 0)),
    OPP_score = cumsum(ifelse(BatPitch == "Pitch", score_increment, 0))
  ) %>%
  select(-score_increment) # Remove the temporary column

# STOP AND CHECK - check if final score is correct (if not, edit manually here then continue)
# Manually adjust a specific row where a missing run needs to be added
#table$score_increment[18] <- table$score_increment[18] + 1

#EDIT - Outs Tracker: make sure bat/pitch is in correct order or it will count incorrect
current_outs <- 0
for(i in 1:nrow(table)) {
  # For the first batter of each half-inning, start with 0 outs
  if(i == 1 || (table$Inning[i] != table$Inning[i - 1])) {
    current_outs <- 0
  }
  # Assign the number of outs before the at-bat
  table$num_outs[i] <- current_outs
  # Update outs based on 'event_outs' column
  current_outs <- current_outs + table$event_outs[i]
  # Reset the counter at the end of each half-inning (when 'BatPitch' changes from "Pitch" to
"Bat")
  if(i < nrow(table) && table$BatPitch[i] == "Pitch" && table$BatPitch[i + 1] == "Bat") {
    current_outs <- 0
  }
```

```
}

#EDIT – Baserunner Count Tracker: make sure bat/pitch is in correct order or it will count
#incorrect
#Define keywords for tracking base runners
on_base_keywords <- c("walked", "singled", "doubled", "tripled",
                      "reached on an error", "hit by pitch",
                      "reached on a fielder's choice", "placed on second")
## add: caught stealing and pick-off
off_base_keywords <- c("reached on a fielder's choice", "caught stealing",
                       "scored", "picked off", "out at second",
                       "out at third", "out at home", "stole home")
# Create the 'num_baserunners' column and initialize to zero
table$num_baserunners <- 0
# Create a cumulative counter
current_baserunners <- 0
# run loop
for(i in 1:nrow(table)) {
  # For the first batter of each half inning, start with 0 runners
  if(i == 1 || (table$Inning[i] != table$Inning[i - 1])) {
    current_baserunners <- 0
  }
  # Assign the cumulative count to the current batter's 'num_baserunners'
  table$num_baserunners[i] <- current_baserunners
  # Now, update num_baserunners based on if the batter gets on or off base
  if(any(sapply(on_base_keywords, function(x) grepl(x, table$value[i])))) {
    current_baserunners <- current_baserunners + 1
  }
  if(any(sapply(off_base_keywords, function(x) grepl(x, table$value[i])))) {
    current_baserunners <- current_baserunners - 1
  }
  # Reset the counter at the end of each half inning (when 'BatPitch' is "Pitch")
  if(i < nrow(table) && table$BatPitch[i] == "Pitch" && table$BatPitch[i + 1] == "Bat") {
    current_baserunners <- 0
  }
}

#RUN - Tracking Balls and Strikes
table <- table %>%
  mutate(
```

```
    extracted = str_extract(value, "\\(\\d+-\\d+"),  # Extract the part that looks like "(X-Y"
    numbers = str_extract_all(extracted, "\\d+"),  # Extract numbers from "(X-Y"
    ball_count = as.numeric(sapply(numbers, function(x) if(length(x) >= 1) x[1] else NA)),  #
First number
    strike_count = as.numeric(sapply(numbers, function(x) if(length(x) >= 2) x[2] else NA))  #
Second number
  ) %>%
  mutate(
    ball_count = replace_na(ball_count, 0),  # Replace NA with 0
    strike_count = replace_na(strike_count, 0)  # Replace NA with 0
  ) %>%
  select(-extracted, -numbers)

# EDIT - add Pitchers and Catchers
#pitchers (last names for LBSU pitcher and handedness for opposing pitchers)
table <- table %>%
  mutate(Pitcher = NA) %>%
  # Assign the pitcher values to specific rows
  { .[1, "Pitcher"] <- "Haddad"; .[4, "Pitcher"] <- "R"; . } %>%
  # Group by BatPitch and fill Pitcher down within each group
  group_by(BatPitch) %>%
  fill(Pitcher, .direction = "down") %>%
  ungroup()  # Ungroup after filling

# catchers (last names for LBSU catchers, left empty for opposing catchers
table <- table %>%
  mutate(Catcher = NA) %>%
  # Assign the pitcher values to specific rows
  { .[1, "Catcher"] <- "Durazo"; .[4, "Catcher"] <- ""; . } %>%
  # Group by BatPitch and fill Pitcher down within each group
  group_by(BatPitch) %>%
  fill(Catcher, .direction = "down") %>%
  ungroup()  # Ungroup after filling

#EDIT - add general game info columns
table$game_location <- "home"           # home, away, neutral
table$game_type <- "conference"      # conference, non-conference
table$opponent <- "CP"
table$home_umpire <- "Eric Hawthorne"
table$time_of_day <- 1500              # military time
```

```
table$weather <- 69                  # degree F
table$outcome <- "Win"                 # Win, Loss
table$year <- "2023"                  # year
table$time_minutes <- 119               # game length in minutes
table$attendence <- 427
table$home_vis <- "H"                  # V, H
table$uniform <- "greypins"          # white_pins, black, black_Y_LB, greypins, LBC
                                        # blackpins, white

# add plate appearacne
table <- table %>%
  mutate(Plate_Appearance = cumsum(Hitter != lag(Hitter, default = first(Hitter))) + 1) %>%
  ungroup()

# add pitch result
table$Pitch_Result <- ifelse(table$bat_event_fl == TRUE, "BIP", NA)

table <- table %>%
  mutate(
    Strike = ifelse(code %in% c(4, 5, 11), 1, NA),
    Ball = ifelse(code %in% c(7, 12, 13, 18, 19), 1, NA)
  )

#EDIT - play by play to excel
# write_xlsx(table, "/Users/teahthies/Desktop/BoxPDF2023/excel/gameID.xlsx")
# manually go into Synergy Sport and input corresponding pitches and locations




###### NEW SCRIPT: 2025 Season Automated Play-By-Play and Pitches ######
# EDIT - Upload Files
# box score
pdf_text <- pdf_text("/Users/teahthies/Desktop/LBSU_2025/Box_Score/GameID.pdf") #edit
#lineup
lineup <- read_excel("/Users/teahthies/Desktop/LBSU_2025/lineups/Lineup.xlsx", sheet = 1)
# event code
event <- read_excel("/Users/teahthies/Desktop/BoxPDF2023/DONE/event.xlsx", sheet = 1)
#synergy pitches
```

```
pitches <-
read_excel("/Users/teahthies/Desktop/LBSU_2025/synergy_charts/GameSynnergy.xlsx", sheet =
1)

# RUN - play-by-play pdf format
# page to and on is play by play data
pdf_text_from_page_2 <- pdf_text[2:length(pdf_text)]
# get each line a row if its own
pdf_lines <- unlist(strsplit(pdf_text_from_page_2, "\n"))
# make tibble
clean_data <- pdf_lines %>%
  as_tibble()
# get name of player
clean_data <- clean_data %>%
  filter(grepl("^[A-Za-z]", value))

# EDIT - if a row in pdf goes to 2 lines, change it to 1
# Define the indices where merging should occur
merge_indices <- c(14,15,60)  # Indices where the next row should be merged
#.               -0, -1, -2, ....
# Loop through indices and merge rows
for (idx in merge_indices) {
  next_idx <- idx + 1
  # Check if the next row exists
  if (next_idx %in% rownames(clean_data)) {
    clean_data$value[rownames(clean_data) == idx] <-
paste(clean_data$value[rownames(clean_data) == idx],
                                          clean_data$value[rownames(clean_data) == next_idx])
    # Remove the next row after merging
    clean_data <- clean_data %>% filter(rownames(.) != next_idx)
  }
}

# EDIT - if we pitch or bat in top of the inning
table <- clean_data %>%
  mutate(
    Inning = case_when(
      grepl("Top", value) ~ (str_extract(value, "\\d+")),
      grepl("Bottom", value) ~ (str_extract(value, "\\d+")),
      TRUE ~ NA_character_
```

```
    ), # Bat or Pitch
    BatPitch = case_when(
      grepl("Top", value) ~ "Bat",         #edit - LBSU is Batting in the Top
      grepl("Bottom", value) ~ "Pitch",    #edit - LBSU is Pitching in the Bottom
      TRUE ~ NA_character_
    )
  ) %>%
  fill(Inning, BatPitch, .direction = "down")

# EDIT - normally not needed unless play by play gives initial
table <- table %>%
  mutate(Player = case_when(
    str_detect(BatPitch, "Pitch") ~ word(value, 1),  # If BatPitch contains "Bat", take the first word
    str_detect(BatPitch, "Bat") ~ word(value, 1) # If BatPitch contains "Pitch", take the second
word
  ))

# EDIT – team named
table <- table %>%
  filter(!(Player %in% c("Hawaii", "Long")))  # edit - opponent team name
## this removes columns that subs are made in the game
table <- table[-grep("for", table$value, ignore.case = TRUE), ]
## remove commas and periods and semicolons in Player column
table$Player <- gsub("[^A-Za-z]", "", table$Player)

# RUN - combine lineup
table <- table %>%
  left_join(lineup %>% select(player, line_up_pos),
            by = c("Player" = "player"))
table <- table %>%
  left_join(lineup %>% select(player, position),
            by = c("Player" = "player"))
table <- table %>%
  left_join(lineup %>% select(player, Bat_hand),
            by = c("Player" = "player"))

# STOP AND CHECK - check if each player name transferred from lineup
table <- table %>%
  filter(!is.na(Bat_hand))
```

```
# RUN - combine Event Table
table <- table %>%
  rowwise() %>%
  mutate(
    code = {
      # Check if the 'value' contains the 'event' from events_table
      matching_event <- event %>%
        filter(str_detect(value, event)) %>%
        pull(code) # Get the code for the matching event
      # If a match is found, use the code; otherwise, NA
      if (length(matching_event) > 0) matching_event[1] else NA
    }
  ) %>%
  ungroup()

# STOP AND CHECK - check for no NA CODES unless it is a sub
# remove substituitions
table <- table %>% filter(!is.na(code))
table <- table %>%
  left_join(event %>%
              select(code, event_outs, bat_event_fl, bat_dest, swing),
            by = "code")
# remove duplicate rows
table <- table %>%
  distinct(value, .keep_all = TRUE)

# RUN - score columns
table <- table %>%
  mutate(
    score_increment = str_count(value, "\\bhomered\\b") + str_count(value, "\\bscored\\b") +
str_count(value, "\\bstole home\\b")
  )

# STOP AND CHECK - check if final score is correct, then continue
# Manually adjust a specific row where a missing run needs to be added
table <- table %>%
  mutate(
    LBSU_score = cumsum(ifelse(BatPitch == "Bat", score_increment, 0)),
    OPP_score = cumsum(ifelse(BatPitch == "Pitch", score_increment, 0))
  ) %>%
```

```
  select(-score_increment) # Remove the temporary column

## EDIT - Outs Tracker: make sure bat/pitch is in correct order
current_outs <- 0
for(i in 1:nrow(table)) {
  # For the first batter of each half-inning, start with 0 outs
  if(i == 1 || (table$Inning[i] != table$Inning[i - 1])) {
    current_outs <- 0
  }
  # Assign the number of outs before the at-bat
  table$num_outs[i] <- current_outs
  # Update outs based on 'event_outs' column
  current_outs <- current_outs + table$event_outs[i]
  # Reset the counter at the end of each half-inning (when 'BatPitch' changes from "Pitch" to
"Bat")
  if(i < nrow(table) && table$BatPitch[i] == "Bat" && table$BatPitch[i + 1] == "Pitch") {
    current_outs <- 0
  }
}

#EDIT - Tracking Base Runners: make sure bat/pitch is in correct order
# Define keywords for tracking base runners
on_base_keywords <- c("walked", "singled", "doubled", "tripled",
                      "reached on an error", "hit by pitch",
                      "reached on a fielder's choice", "placed on second")
off_base_keywords <- c("reached on a fielder's choice", "caught stealing",
                       "scored", "picked off", "out at second",
                       "out at third", "out at home", "stole home")
# Create the 'num_baserunners' column and initialize to zero
table$num_baserunners <- 0
# Create a cumulative counter
current_baserunners <- 0
# run loop
for(i in 1:nrow(table)) {
  # For the first batter of each half inning, start with 0 runners
  if(i == 1 || (table$Inning[i] != table$Inning[i - 1])) {
    current_baserunners <- 0
  }
  # Assign the cumulative count to the current batter's 'num_baserunners'
  table$num_baserunners[i] <- current_baserunners
```

```
  # Now, update num_baserunners based on if the batter gets on or off base
  if(any(sapply(on_base_keywords, function(x) grepl(x, table$value[i])))) {
    current_baserunners <- current_baserunners + 1
  }
  if(any(sapply(off_base_keywords, function(x) grepl(x, table$value[i])))) {
    current_baserunners <- current_baserunners - 1
  }
  # Reset the counter at the end of each half inning (when 'BatPitch' is "Pitch")
  if(i < nrow(table) && table$BatPitch[i] == "Bat" && table$BatPitch[i + 1] == "Pitch") {
    current_baserunners <- 0
  }
}

##-------------------------------------------------
## EDIT - add general game info columns
##-------------------------------------------------
table$game_location <- "away"           # home, away, neutral
table$game_type <- "conference"      # conference, non-conference
table$opponent <- "UH"
table$home_umpire <- "Eric Jones"
table$time_of_day <- 1805              # military time
table$weather <- 70                   # degree F
table$outcome <- "Loss"                # Win, Loss
table$year <- 2025                   # year
table$time_minutes <- 120              # game length in minutes
table$attendence <- 375
table$home_vis <- "V"                  # V, H
table$uniform <- "black"          # white_pins, black, black_Y_LB, greypins, LBC
                                        # blackpins, white

# EDIT - add Catchers for LBSU
table <- table %>%
  mutate(Catcher = NA) %>%
  # Assign the pitcher values to specific rows
  { .[1, "Catcher"] <- ""; .[6, "Catcher"] <- "Durazo"; . } %>%
  # Group by BatPitch and fill Pitcher down within each group
  group_by(BatPitch) %>%
  fill(Catcher, .direction = "down") %>%
  ungroup()  # Ungroup after filling
```

```
# RUN – synergy (synergy charts were made in real time during the game)
#synergy pitches
pitches <- read_excel("/Users/teahthies/Desktop/LBSU_2025/synergy_charts/synergychart.xlsx",
sheet = 1)

# get only defense
table <- table %>%
  filter(BatPitch == "Pitch")
# add plate appearacne to table
table <- table %>%
  mutate(Plate_Appearance = cumsum(Player != lag(Player, default = first(Player))) + 1) %>%
  ungroup()

# fill down and with 0's
pitches <- pitches %>%
  fill(Pitcher, Player, Pitch, .direction = "down") %>%  # Fill down values
  mutate(across(c(Ball, Strike), ~replace_na(., 0)))
# number pitches in game
pitches <- pitches %>%
  mutate(pitch_count = row_number())

#Get hitter and player to match case
table <- table %>%
  mutate(Player = tolower(Player))  # Convert Player to lowercase
pitches <- pitches %>%
  mutate(Player = tolower(Player))

# add plate appearance column
pitches <- pitches %>%
  mutate(Plate_Appearance = cumsum(Player != lag(Player, default = first(Player)))+1)
table <- table %>%
  filter(bat_event_fl == TRUE)

# combine last pitches with table_pitch play by play
last_pitches <- pitches %>%
  group_by(Plate_Appearance, Player) %>%  # Group by batter and plate appearance
  slice_tail(n = 1)  # Select only the last pitch for each batter

# Merge with the play-by-play table_pitch
combine <- left_join(pitches, table, by = "Plate_Appearance")
```

```
# remove player
combine <- combine %>% select(-Player.x)

# change name to hitter to make consistent
combine <- combine %>%
  rename(Hitter = Player.y)

#fix columns
cols_to_keep <- c("code", "event_outs", "bat_event_fl", "bat_dest", "swing", "LBSU_score",
"OPP_score")

# Process the dataset
combine <- combine %>%
  group_by(Hitter, Plate_Appearance) %>%  # Group by Hitter and Plate_Appearance
  mutate(across(all_of(cols_to_keep), ~ ifelse(row_number() == n(), ., NA))) %>%  # Keep only
in last row
  ungroup()

#score
combine$LBSU_score[1] <- 0
combine$OPP_score[1] <- 0
combine <- combine %>%
  fill(LBSU_score, OPP_score, .direction = "down")

# event_outs, bat_dest
combine <- combine %>%
  group_by(Hitter, Plate_Appearance) %>%  # Group by Hitter and Plate_Appearance
  mutate(event_outs = ifelse(row_number() == n(), event_outs, 0)) %>%
  mutate(bat_dest = ifelse(row_number() == n(), bat_dest, 0)) %>%   # Keep only in last row,
else 0
  mutate(bat_event_fl = ifelse(row_number() == n(), bat_event_fl, FALSE)) %>%
  mutate(swing = ifelse(row_number() == n(), swing, FALSE)) %>%
  ungroup()

# swing
combine <- combine %>%
  mutate(swing = ifelse(Swing %in% c("1", "1f", "1c") & swing == FALSE, TRUE, swing))

# add foul and chase
```

```
combine <- combine %>%
  mutate(Pitch_Result = case_when(
    Swing == "1" & Strike == 1 ~ "miss",
    Swing == "1c" & Strike == 1 ~ "chase",
    Swing == "1f" & Strike %in% c(0, 1) ~ "foul",
    Swing == "1" & Strike == 0 ~ "BIP",
    is.na(Swing) & Strike == 1 ~ "StrikeTaken",
    is.na(Swing) & Strike == 0 ~ "ball",
    TRUE ~ NA_character_  # Assign NA if none of the conditions are met
  ))

## remove value column
combine <- combine %>% select(-Swing)

# make count column
# balls and strikes
combine <- combine %>%
  group_by(Hitter, Plate_Appearance) %>%
  mutate(
    strike_count = cumsum(lag(Strike, default = 0)),  # Cumulative sum of strikes
    ball_count = cumsum(lag(Ball, default = 0))       # Cumulative sum of balls
  )

## remove extra columns
colnames(combine)

# EDIT - create excel sheet
#write_xlsx(combine, "/Users/teahthies/Desktop/LBSU_2025/game_charts/UH1.xlsx")



##### NEW SCRIPT: Combining All Data
####combine 2023 data into 1 table
file_path <- "/Users/teahthies/Desktop/CleanData/2023"
files <- list.files(path = file_path, pattern = "\\.xlsx$", full.names = TRUE)
#fix "weather" & "attendance" to numeric
read_and_fix_columns <- function(file) {
  df <- read_excel(file)  # Read the file
  # Ensure "weather" column exists before converting
  if ("weather" %in% names(df)) {
```

```
  df$weather <- as.numeric(df$weather)  # Convert "weather" to numeric
 }
 # Ensure "attendance" column exists before converting
 if ("attendence" %in% names(df)) {
  df$attendence <- as.numeric(df$attendence)  # Convert "attendance" to numeric
 }
 return(df)
}

# Read all files while ensuring "weather" & "attendance" are numeric
data_list <- lapply(files, read_and_fix_columns)
# Combine all data frames
combined_data <- bind_rows(data_list)
# Save the combined data as an Excel file
write_xlsx(combined_data, "combined_game_data.xlsx")
# count rows
table <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2023_game_data.xlsx",
sheet=1)
nrow(table)  #3158 rows

####combine 2024 data
library(readxl)
library(dplyr)
library(writexl)
file_path <- "/Users/teahthies/Desktop/CleanData/2024"
files <- list.files(path = file_path, pattern = "\\.xlsx$", full.names = TRUE)
#fix "weather" & "attendance" to numeric
read_and_fix_columns <- function(file) {
 df <- read_excel(file)  # Read the file
 # Ensure "weather" column exists before converting
 if ("weather" %in% names(df)) {
  df$weather <- as.numeric(df$weather)  # Convert "weather" to numeric
 }
 # Ensure "attendance" column exists before converting
 if ("attendence" %in% names(df)) {
  df$attendence <- as.numeric(df$attendence)  # Convert "attendance" to numeric
 }
 return(df)
}
```

```
# Read all files while ensuring "weather" & "attendance" are numeric
data_list <- lapply(files, read_and_fix_columns)
# Combine all data frames
combined_data <- bind_rows(data_list)
# Save the combined data as an Excel file
write_xlsx(combined_data,
"/Users/teahthies/Desktop/CleanData/Combined/combined_2024_game_data.xlsx")
# count rows
table <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2024_game_data.xlsx",
sheet=1)
nrow(table)  #3460 rows

# 2025 combine
file_path <- "/Users/teahthies/Desktop/CleanData/2025"
files <- list.files(path = file_path, pattern = "\\.xlsx$", full.names = TRUE)

#fix "weather" & "attendance" to numeric
read_and_fix_columns <- function(file) {
  df <- read_excel(file)  # Read the file

  # Ensure "weather" column exists before converting
  if ("weather" %in% names(df)) {
    df$weather <- as.numeric(df$weather)  # Convert "weather" to numeric
  }

  # Ensure "attendance" column exists before converting
  if ("attendence" %in% names(df)) {
    df$attendence <- as.numeric(df$attendence)  # Convert "attendance" to numeric
  }

  # Location
  if ("Location" %in% names(df)) {
    df$Location <- as.character(df$Location)  # Convert "attendance" to numeric
  }

  # year
  if ("year" %in% names(df)) {
    df$year <- as.character(df$year)  # Convert "attendance" to numeric
```

```
  }
  return(df)
}

# Read all files while ensuring "weather" & "attendance" are numeric
data_list <- lapply(files, read_and_fix_columns)

# Combine all data frames
combined_data <- bind_rows(data_list)

# Save the combined data as an Excel file
write_xlsx(combined_data,
"/Users/teahthies/Desktop/CleanData/Combined/combined_2025_game_data.xlsx")

table <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2025_game_data.xlsx",
sheet=1)
nrow(table)  #3783 rows

###check all columns in 2023, 2024, and 2025 datasets
table23 <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2023_game_data.xlsx",
sheet=1)
table24 <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2024_game_data.xlsx",
sheet=1)
table25 <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2025_game_data.xlsx",
sheet=1)
setdiff(names(table24), names(table25))
setdiff(names(table25), names(table24))
write_xlsx(table23,
"/Users/teahthies/Desktop/CleanData/Combined/combined_2023_game_data.xlsx")
write_xlsx(table24,
"/Users/teahthies/Desktop/CleanData/Combined/combined_2024_game_data.xlsx")
write_xlsx(table25,
"/Users/teahthies/Desktop/CleanData/Combined/combined_2025_game_data.xlsx")
```

```
table23 <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2023_game_data.xlsx",
sheet=1)
table24 <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2024_game_data.xlsx",
sheet=1)
table25 <-
read_excel("/Users/teahthies/Desktop/CleanData/Combined/combined_2025_game_data.xlsx",
sheet=1)
table23$Location <- as.character(table23$Location)
table24$Location <- as.character(table24$Location)
table25$Location <- as.character(table25$Location)

#### write all combines data into an excel sheet
combined <- bind_rows(table23, table24, table25)
write_xlsx(combined, "/Users/teahthies/Desktop/CleanData/Combined/ALL_game_data.xlsx")
all <- read_excel("/Users/teahthies/Desktop/CleanData/Combined/ALL_game_data.xlsx",
sheet=1)
nrow(all)
nrow(table23)+nrow(table24)+nrow(table25)

#### create only LBSU pitcher dataset
pitch <- all %>%
  filter(BatPitch == "Pitch")

pitch$Pitch <- toupper(pitch$Pitch)
pitch$Location[pitch$Location %in% c("1f")] <- 1
# add advanced column
pitch$Advanced <- ifelse(
  is.na(pitch$value),
  NA,
  ifelse(grepl("advanced|scored", pitch$value, ignore.case = TRUE), 1, 0)
)
sum(is.na(pitch$Advanced))
write_xlsx(pitch, "/Users/teahthies/Desktop/CleanData/pitch2.xlsx")
```

NEW SCRIPT: 2024 NCAA Pitching Data

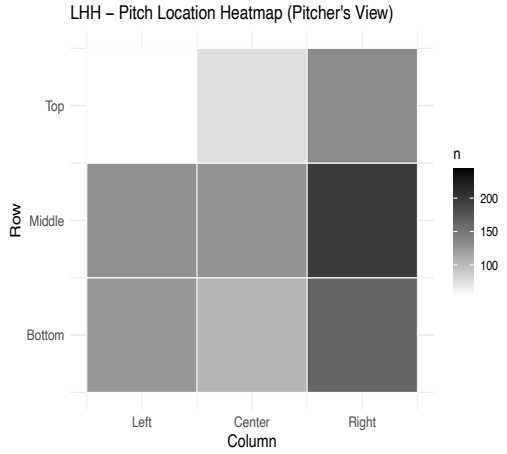
```
# NCAA files download
setwd("/Users/teahthies/Desktop/NCAA_STATS_2024/files")
files <- list.files(pattern = "^[^~].*\\.xlsx$")
library(readxl)
data_list <- lapply(files, read_excel)
file_names <- tools::file_path_sans_ext(files)
data_list <- setNames(lapply(files, read_excel), file_names)
data_list <- lapply(data_list, function(df) df[, !colnames(df) %in% "Rank"])
list2env(data_list, envir = .GlobalEnv)
names(data_list)

# Merge all dataframes by the "Team" column
pitch_data <- Reduce(function(x, y) merge(x, y, by = "Team", all = TRUE),
                     list(P_era, P_fieldperc, P_KtoBB, P_sbPgame))
nrow(pitch_data) # 296
pitch_data$Team <- gsub("\\(.*\\)", "", pitch_data$Team) # get rid of conference,
                                          # which is in ()

# Overall Rank and Win% based on 307 teams
OverallRank <- read_excel("/Users/teahthies/Desktop/NCAA_STATS_2024/Overall307.xlsx",
sheet = 1)
colnames(OverallRank)

# Find similar names between 2 dfs OverallRank and pitch_data
library(stringdist)
# Perform fuzzy matching and store the indices
match_indices <- amatch(pitch_data$Team, OverallRank$Team, method = "jw", maxDist = 0.2)
# Create a new column in df1 to store matched names
pitch_data$Matched_Team <- ifelse(!is.na(match_indices), OverallRank$Team[match_indices],
NA)
# Separate matching and non-matching teams
matched_df <- pitch_data[!is.na(pitch_data$Matched_Team), ]  # Rows with matches
unmatched_df <- pitch_data[is.na(pitch_data$Matched_Team), ]  # Rows without matches
# Print results
print("Matched Teams:")
print(matched_df)
print("Unmatched Teams:")
print(unmatched_df)
unmatched_hit <- unmatched_df %>% select(Team)
```

```
# Perform fuzzy matching and store the indices
match_indices <- amatch(OverallRank$Team, pitch_data$Team, method = "jw", maxDist = 0.2)
# Create a new column in df1 to store matched names
OverallRank$Matched_Team <- ifelse(!is.na(match_indices), pitch_data$Team[match_indices],
NA)
# Separate matching and non-matching teams
matched_df <- OverallRank[!is.na(OverallRank$Matched_Team), ]  # Rows with matches
unmatched_df <- OverallRank[is.na(OverallRank$Matched_Team), ]  # Rows without matches
# Print results
print("Matched Teams:")
print(matched_df)
print("Unmatched Teams:")
print(unmatched_df)
unmatched_rank <- unmatched_df %>% select(Team)

### match team in pitch_data
pitch_data[13, 16] <- "Army"
pitch_data[23, 16] <- "Boston University"
pitch_data[55, 16] <- "Cal State Northridge"
pitch_data[69, 16] <- "East Tennessee State"
pitch_data[72, 16] <- "Fairleigh Dickinson"
pitch_data[74, 16] <- "Florida International"
pitch_data[120, 16] <- "Lamar"
pitch_data[124, 16] <- "Long Island"
pitch_data[125, 16] <- "Loyola Marymount"
pitch_data[138, 16] <- "UMass"
pitch_data[247, 16] <- "Northern Illinois"
pitch_data[199, 16] <- "Fort Wayne"
pitch_data[221, 16] <- "Stephen F. Austin"
pitch_data[229, 16] <- "USF"
pitch_data[230, 16] <- "Southeast Missouri"
pitch_data[261, 16] <- "Connecticut"
pitch_data[262, 16] <- "Illinois-Chicago"
pitch_data[264, 16] <- "UL Monroe"
pitch_data[269, 16] <- "UNC Wilmington"
pitch_data[270, 16] <- "Northern Illinois"

# Change pitching data teams to match Overall Teams
pitch_data <- pitch_data %>% select(-Team)
colnames(pitch_data)[colnames(pitch_data) == "Matched_Team"] <- "Team"
```

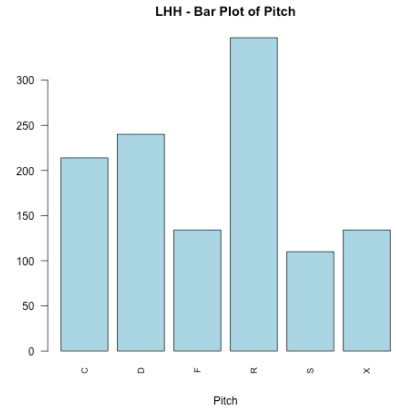
```
## missing about 10 teams from overall (smaller teams)
OverallRank <- merge(OverallRank, pitch_data, by = "Team", all = FALSE)
OverallRank <- OverallRank %>% select(-Matched_Team)
colnames(OverallRank)

## OVERALL RANK IS OUR DATASET
library(writexl)
#write_xlsx(OverallRank,
"/Users/teahthies/Desktop/NCAA_STATS_2024/unsupervised_learning/rank_VS_pitch.xlsx")
```

APPENDIX D
LHH EDA GRAPHS

Heat map of pitch location



Pitch type frequency bar graph



Count vs. outcome variables

count	batting_avg	SO_rate	HR_rate	BB_rate	on_base_rate	attempts
0-0	0.302	0.000	0.014	0.041	0.410	222
0-1	0.295	0.000	0.007	0.013	0.349	149
0-2	0.108	0.206	0.010	0.020	0.206	102
1-0	0.317	0.000	0.000	0.040	0.416	101
1-1	0.283	0.000	0.020	0.020	0.434	99
1-2	0.237	0.180	0.000	0.014	0.288	139
2-0	0.167	0.000	0.071	0.024	0.310	42
2-1	0.209	0.000	0.000	0.030	0.328	67
2-2	0.262	0.197	0.016	0.005	0.306	183
3-0	0.071	0.000	0.000	0.929	1.000	28
3-1	0.177	0.000	0.000	0.565	0.742	62
3-2	0.102	0.120	0.000	0.367	0.512	166

Location vs. outcome variables

Location	batting_avg	SO_rate	HR_rate	BB_rate	on_base_rate	attempts
1	0.160	0.040	0.032	0.184	0.368	125
2	0.277	0.031	0.008	0.054	0.415	130
3	0.164	0.036	0.018	0.255	0.527	55
4	0.257	0.133	0.000	0.105	0.390	105
5	0.398	0.008	0.031	0.000	0.477	128
6	0.273	0.065	0.013	0.078	0.416	77
7	0.178	0.147	0.006	0.166	0.387	163
8	0.258	0.076	0.005	0.045	0.333	198
9	0.178	0.111	0.000	0.178	0.407	135

Pitch vs. outcomes

Pitch	batting_avg	SO_rate	HR_rate	BB_rate	on_base_rate	attempts
C	0.290	0.061	0.014	0.075	0.439	214
D	0.200	0.050	0.004	0.167	0.429	240
F	0.239	0.037	0.022	0.112	0.433	134
R	0.213	0.110	0.012	0.110	0.352	347
S	0.282	0.055	0.018	0.127	0.445	110
X	0.254	0.119	0.000	0.037	0.358	134

APPENDIX E
DESCRIPTION OF ORIGINAL DATASET

Feature Description			
Feature	**Description**	**Variable Type**	**Input Type**
Value	Play-by-play description		Automated
Inning	Inning in current game	Numeric	Automated
BatPitch	LBSU on offense (Bat) or on defense (Pitch)	Factor	Automated
Hitter	Hitter's last name	Character	Automated
Pitch	Pitch type (R=rise, S=screw, C=curve, D-drop, X=change-up, F=fastball)	Factor	Manual (2023/2024), Automatic (2025)
Location	Location of where the pitch crossed the plate (from pitchers perspective…1=top left, 2=middle left, 3=lower left, 4=top middle, 5 middle middle, 6=lower middle, 7=top right, 8=middle right, 9=lower right)	Factor	Manual (2023/2024), Automatic (2025)
Catcher	Player who is in the catching position	Factor	Automated
line_up_pos	Where the batter is batting in the lineup (1-10 with 10 as the DP)	Factor	Manual
position	Where batter is playing on defense	Factor	Manual
Bat_hand	Right-handed ® or left-handed (L) batter	Factor	Manual
code	Result of at bat (See event code table)	Factor	Automated
event_outs	How many outs the result caused	Numeric	Automated
bat_event_fl	TRUE=Event ended an at bat, FALSE=Event did not end an at bat	Logical	Automated
bat_dest	Which base did the batter reach after the at bat (0-4)	Numeric	Automated
swing	TRUE=batter swung, FALSE=batter did not swing	Logical	Automated
LBSU_score	Score of LBSU team	Numeric	Automated
OPP_score	Score of opposing team	Numeric	Automated
num_outs	Current number of outs	Numeric	Automated
num_baserunners	Current number of baserunners on	Numeric	Automated
ball_count	Current ball count	Numeric	Automated
strike_count	Current strike count	Numeric	Automated
Pitcher	Name of LBSU pitcher or RHP/LHP for opposing team's pitcher	Factor	Automated
game_location	Home, Away, Neutral	Factor	Automated

game_type	Conference, non-conference, post-season	Factor	Automated
opponent	Opponent's School name	Character	Automated
home_umpire	Name of the home plate umpire	Character	Automated
time_of_day	Start time of game in military time	Numeric	Automated
weather	Temperature at the start of the game in degrees Fahrenheit	Numeric	Manual
outcome	Win or Loss for LBSU	Factor	Automated
year	Year of the season	Numeric	Automated
time_minutes	How long the games takes in minutes	Numeric	Automated
attendance	Number of people watching the game	Numeric	Automated
home_vis	H=LBSU is home team, V=LBSU is visiting team	Factor	Automated
uniform	LBSU's uniform combination	Character	Automated
Plate_Apearance	Plate appearance of the game	Numeric	Automated
Pitch_Result	BIP=Ball in play	Factor	Automated
Strike	1=Strike	Numeric	Automated
Ball	1=Ball	Numeric	Automated
hit_type	Ground, bunt, line, fly, bunt	Factor	Automated

Event Code Description					
Code	**Event**	**event_outs**	**bat_event_fl**	**bat_dest**	**swing**
0	placed on second	0	TRUE	2	FALSE
1	flied out	1	TRUE	0	TRUE
2	lined out	1	TRUE	0	TRUE
2	fouled out	1	TRUE	0	TRUE
2	infield fly	1	TRUE	0	TRUE
2	popped up	1	TRUE	0	TRUE
3	out at first	1	TRUE	0	TRUE
3	grounded out	1	TRUE	0	TRUE
4	out on batter's interference	1	TRUE	0	TRUE
4	struck out looking	1	TRUE	0	FALSE
5	struck out swinging	1	TRUE	0	TRUE
6	stole	0	FALSE	0	
7	passed ball	0	FALSE	0	
8	reached on an error	0	TRUE	1	TRUE
8	reached on a fielding error	0	TRUE	1	TRUE

8	reached on catcher's interference	0	TRUE	1	TRUE
8	reached on a throwing error	0	TRUE	1	TRUE
9	reached on a fielder's choice	1	TRUE	1	TRUE
10	picked off	1	FALSE	0	
12	wild pitch	0	FALSE	0	FALSE
16	flied into double play	2	TRUE	0	TRUE
16	lined into double play	2	TRUE	0	TRUE
16	grounded into double play	2	TRUE	0	TRUE
16	hit into double play	2	TRUE	0	TRUE
17	caught stealing	1	FALSE	0	
18	walked	0	TRUE	1	FALSE
19	hit by pitch	0	TRUE	1	FALSE
20	singled	0	TRUE	1	TRUE
21	doubled	0	TRUE	2	TRUE
22	tripled	0	TRUE	3	TRUE
23	homered	0	TRUE	4	TRUE

Feature Description in Domain Adaptation Models			
Feature	**Description**	**Variable Type**	**Input type**
inning	Inning in current game	Numeric	Automated
pitch_type	Pitch type (R=rise, S=screw, C=curve, D-drop, X=change-up, F=fastball)	Factor	Manual (2023/2024), Automatic (2025)
zone	Location of where the pitch crossed the plate (from pitchers perspective…1=top left, 2=middle left, 3=lower left, 4=top middle, 5 middle middle, 6=lower middle, 7=top right, 8=middle right, 9=lower right)	Factor	Manual (2023/2024), Automatic (2025)
stand	Right-handed (R) or left-handed (L) batter	Factor	Manual
events	Result of at bat (See event code table)	Factor	Automated
outs_when_up	Current number of outs	Numeric	Automated
balls	Current ball count	Numeric	Automated
strikes	Current strike count	Numeric	Automated
p_throws	Pitcher Handedness of LBSU pitcher	Factor	Automated
game_type	Win or Loss for LBSU	Factor	Automated

description	BIP=Ball in play	Factor	Automated
bb_type	Ground, bunt, line, fly, bunt	Factor	Automated

APPENDIX F
CHAPTER 5 CODE

```
#--------------------------------------------

# packages
#--------------------------------------------
library(readxl)
library(writexl)
library(dplyr)
library(stringr)
library(ggplot2)
library(caret)
library(dplyr)

#--------------------------------------------
# load play-by-play (pbp) dataset
#--------------------------------------------
pbp <- read_excel("/Users/teahthies/Desktop/Modeling/cleanpitch.xlsx", sheet=1)

# split to just training data for EDA, we will not look at testing data
n <- nrow(pbp)

# index cutoff for 80%
cutoff <- floor(0.80 * n)

# split: first 80% = train, last 20% = test
pbp <- pbp[1:cutoff, ]

pbp <- subset(pbp, bat_event_fl == TRUE)
# sanity check
nrow(pbp)   # 3675


#--------------------------------------------
# cols rows type
#--------------------------------------------
dim(pbp)

#--------------------------------------------
# missing data
#--------------------------------------------
# empty strings to NA
```

```
pbp[pbp == ""] <- NA

# how much is missing in each column
colSums(is.na(pbp))
threshold <- 0.3
missing_percent <- colSums(is.na(pbp)) / nrow(pbp)
names(missing_percent[missing_percent > threshold])
                # hit_type was only recorded for some games (keep and subset)
                # balls and strikes have lots of missing
                # still need to figure out what to do with value column
                # catchers were not recorded for opposing teams
                    # (this will need to be subsetted)
# look put for player, Strike, Ball, Catcher, hit_type



#---------------------------------------------
# variable type
#---------------------------------------------
# Convert to numeric
pbp$Inning <- as.numeric(pbp$Inning)
# Convert to factors (categorical)
pbp$Pitch <- as.factor(pbp$Pitch)
pbp$Bat_hand <- as.factor(pbp$Bat_hand)
pbp$code <- as.factor(pbp$code)
pbp$Pitch_Result <- as.factor(pbp$Pitch_Result)
pbp$home_vis <- as.factor(pbp$home_vis)
pbp$hit_type <- as.factor(pbp$hit_type)
pbp$Strike <- as.factor(pbp$Strike)
pbp$Ball <- as.factor(pbp$Ball)
pbp$pitch_count <- as.factor(pbp$pitch_count)
# Convert to logical (binary)
pbp$swing <- as.logical(pbp$swing)
pbp$outcome <- as.factor(pbp$outcome)

# str(pbp)
# sapply(pbp, class)

#---------------------------------------------
# correlation
```

```
#-------------------------------------------
numeric_data <- pbp[sapply(pbp, is.numeric)]
cor_matrix <- cor(numeric_data, use = "pairwise.complete.obs")

threshold <- 0.5  # you can adjust this
high_cor <- which(abs(cor_matrix) > threshold & abs(cor_matrix) < 1, arr.ind = TRUE)
high_cor_pairs <- unique(t(apply(high_cor, 1, sort)))
for (i in 1:nrow(high_cor_pairs)) {
  row <- high_cor_pairs[i, 1]
  col <- high_cor_pairs[i, 2]
  cat(rownames(cor_matrix)[row], " & ", colnames(cor_matrix)[col],
      ": ", cor_matrix[row, col], "\n")
}

# Inning  &  Plate_Appearance :  0.8093022
# OPP_score  &  Plate_Appearance :  0.5261127
# ball_count  &  pitch_count :  0.977381
# strike_count  &  pitch_count :  0.6014809

# take out pitch count and Plate Appearance
pbp <- pbp %>% select(-c(Hitter, Plate_Appearance, pitch_count, Ball, Strike))
colnames(pbp)

#-------------------------------------------
# Prepare Data - Split RHH & LHH
#-------------------------------------------
char_cols <- names(pbp)[sapply(pbp, is.character)]
setwd("/Users/teahthies/Desktop/EDA/eda_paper_FIGS")
r <- subset(pbp, Bat_hand == "R")
l <- subset(pbp, Bat_hand == "L")
nrow(r)
nrow(l)

#-------------------------------------------
# Univariate
#-------------------------------------------
### PITCH TYPE COUNT
# RHH
# can find these imgs in eda_pitch_FIGS folder in EDA
for (col in char_cols) {
```

```
  # Open a new PNG device to save the plot
  png(paste0(col, "_RIGHT.png"))

  # Create the bar plot
  barplot(table(r[[col]]),
          main = paste("RHH - Bar Plot of", col),
          xlab = col,
          col = "lightblue",
          las = 2,
          cex.names = 0.8)

  # Close the PNG device to save the file
  dev.off()
}

# LHH
# can find these imgs in eda_pitch_FIGS folder in EDA
for (col in char_cols) {
  # Open a new PNG device to save the plot
  png(paste0(col, "_LEFT.png"))

  # Create the bar plot
  barplot(table(l[[col]]),
          main = paste("LHH - Bar Plot of", col),
          xlab = col,
          col = "lightblue",
          las = 2,
          cex.names = 0.8)

  # Close the PNG device to save the file
  dev.off()
}
#-----------------------------------------------

### HEAT MAP LOCATION
# RHH
r$Location <- as.numeric(as.character(r$Location))
r2 <- r %>%
  mutate(
    row = case_when(
```

```
    Location %in% c(1, 4, 7) ~ 1,  # Top row
    Location %in% c(2, 5, 8) ~ 2,  # Middle row
    Location %in% c(3, 6, 9) ~ 3   # Bottom row
   ),
   col = case_when(
    Location %in% c(1, 2, 3) ~ 1,  # Left column
    Location %in% c(4, 5, 6) ~ 2,  # Middle column
    Location %in% c(7, 8, 9) ~ 3   # Right column
   )
  )

# Count the number of pitches in each (row, col) grid location
location_counts <- r2 %>%
  count(row, col)
# Plot the heatmap
ggplot(location_counts, aes(x = col, y = row)) +
  geom_tile(aes(fill = n), color = "white") +
  scale_fill_gradient(low = "white", high = "black") +  # Adjust the color scale as needed
  theme_minimal() +
  scale_x_continuous(breaks = 1:3, labels = c("Left", "Center", "Right")) +
  scale_y_continuous(breaks = 1:3, labels = c("Bottom", "Middle", "Top")) +
  labs(title = "RHH - Pitch Location Heatmap (Pitcher's View)", x = "Column", y = "Row") +
  theme(axis.title = element_text(size = 12), axis.text = element_text(size = 10))

# LHH
l$Location <- as.numeric(as.character(l$Location))
l2 <- l %>%
  mutate(
   row = case_when(
    Location %in% c(1, 4, 7) ~ 1,  # Top row
    Location %in% c(2, 5, 8) ~ 2,  # Middle row
    Location %in% c(3, 6, 9) ~ 3   # Bottom row
   ),
   col = case_when(
    Location %in% c(1, 2, 3) ~ 1,  # Left column
    Location %in% c(4, 5, 6) ~ 2,  # Middle column
    Location %in% c(7, 8, 9) ~ 3   # Right column
   )
  )
```

```
# Count the number of pitches in each (row, col) grid location
location_counts <- l2 %>%
  count(row, col)
# Plot the heatmap
ggplot(location_counts, aes(x = col, y = row)) +
  geom_tile(aes(fill = n), color = "white") +
  scale_fill_gradient(low = "white", high = "black") +  # Adjust the color scale as needed
  theme_minimal() +
  scale_x_continuous(breaks = 1:3, labels = c("Left", "Center", "Right")) +
  scale_y_continuous(breaks = 1:3, labels = c("Bottom", "Middle", "Top")) +
  labs(title = "LHH - Pitch Location Heatmap (Pitcher's View)", x = "Column", y = "Row") +
  theme(axis.title = element_text(size = 12), axis.text = element_text(size = 10))

#------------------------------------------
# Multivariate
#------------------------------------------
# pitch count vs outcomes
# RHH
r2 <- r %>%
  mutate(
    count = paste0(ball_count, "-", strike_count),
    is_hit = code %in% c(20, 21, 22, 23),
    is_SO = code %in% c(4),
    is_HR = code %in% c(23),
    is_BB = code %in% c(18),
    is_on_base = bat_dest > 0
  ) %>%
  # Remove NA counts or incomplete data
  filter(!is.na(ball_count), !is.na(strike_count)) %>%
  group_by(count) %>%
  summarize(
    batting_avg = round(mean(is_hit, na.rm = TRUE), 3),
    SO_rate = round(mean(is_SO, na.rm = TRUE), 3),
    HR_rate = round(mean(is_HR, na.rm = TRUE), 3),
    BB_rate = round(mean(is_BB, na.rm = TRUE), 3),
    on_base_rate = round(mean(is_on_base, na.rm = TRUE), 3),
    attempts = n()
  ) %>%
  arrange(count)
View(r2)
```

```
# LHH
l2 <- l %>%
  mutate(
    count = paste0(ball_count, "-", strike_count),
    is_hit = code %in% c(20, 21, 22, 23),
    is_SO = code %in% c(4),
    is_HR = code %in% c(23),
    is_BB = code %in% c(18),
    is_on_base = bat_dest > 0
  ) %>%
  # Remove NA counts or incomplete data
  filter(!is.na(ball_count), !is.na(strike_count)) %>%
  group_by(count) %>%
  summarize(
    batting_avg = round(mean(is_hit, na.rm = TRUE), 3),
    SO_rate = round(mean(is_SO, na.rm = TRUE), 3),
    HR_rate = round(mean(is_HR, na.rm = TRUE), 3),
    BB_rate = round(mean(is_BB, na.rm = TRUE), 3),
    on_base_rate = round(mean(is_on_base, na.rm = TRUE), 3),
    attempts = n()
  ) %>%
  arrange(count)
View(l2)

### location vs outcomes
# RHH
r2_location <- r %>%
  mutate(
    is_hit = code %in% c(20, 21, 22, 23),
    is_SO = code %in% c(4),
    is_HR = code %in% c(23),
    is_BB = code %in% c(18),
    is_on_base = bat_dest > 0
  ) %>%
  filter(!is.na(Location)) %>%
  group_by(Location) %>%
  summarize(
    batting_avg = round(mean(is_hit, na.rm = TRUE), 3),
    SO_rate = round(mean(is_SO, na.rm = TRUE), 3),
```

```
    HR_rate = round(mean(is_HR, na.rm = TRUE), 3),
    BB_rate = round(mean(is_BB, na.rm = TRUE), 3),
    on_base_rate = round(mean(is_on_base, na.rm = TRUE), 3),
    attempts = n()
  ) %>%
  arrange(Location)
View(r2_location)

# LHH
l2_location <- l %>%
  mutate(
    is_hit = code %in% c(20, 21, 22, 23),
    is_SO = code %in% c(4),
    is_HR = code %in% c(23),
    is_BB = code %in% c(18),
    is_on_base = bat_dest > 0
  ) %>%
  filter(!is.na(Location)) %>%
  group_by(Location) %>%
  summarize(
    batting_avg = round(mean(is_hit, na.rm = TRUE), 3),
    SO_rate = round(mean(is_SO, na.rm = TRUE), 3),
    HR_rate = round(mean(is_HR, na.rm = TRUE), 3),
    BB_rate = round(mean(is_BB, na.rm = TRUE), 3),
    on_base_rate = round(mean(is_on_base, na.rm = TRUE), 3),
    attempts = n()
  ) %>%
  arrange(Location)
View(l2_location)

### pitch vs outcomes
# RHH
r2_pitch<- r %>%
  mutate(
    is_hit = code %in% c(20, 21, 22, 23),
    is_SO = code %in% c(4),
    is_HR = code %in% c(23),
    is_BB = code %in% c(18),
    is_on_base = bat_dest > 0
  ) %>%
```

```
  filter(!is.na(Pitch)) %>%
  group_by(Pitch) %>%
  summarize(
    batting_avg = round(mean(is_hit, na.rm = TRUE), 3),
    SO_rate = round(mean(is_SO, na.rm = TRUE), 3),
    HR_rate = round(mean(is_HR, na.rm = TRUE), 3),
    BB_rate = round(mean(is_BB, na.rm = TRUE), 3),
    on_base_rate = round(mean(is_on_base, na.rm = TRUE), 3),
    attempts = n()
  ) %>%
  arrange(Pitch)
View(r2_pitch)
# LHH
l2_pitch <- l %>%
  mutate(
    is_hit = code %in% c(20, 21, 22, 23),
    is_SO = code %in% c(4),
    is_HR = code %in% c(23),
    is_BB = code %in% c(18),
    is_on_base = bat_dest > 0
  ) %>%
  filter(!is.na(Pitch)) %>%
  group_by(Pitch) %>%
  summarize(
    batting_avg = round(mean(is_hit, na.rm = TRUE), 3),
    SO_rate = round(mean(is_SO, na.rm = TRUE), 3),
    HR_rate = round(mean(is_HR, na.rm = TRUE), 3),
    BB_rate = round(mean(is_BB, na.rm = TRUE), 3),
    on_base_rate = round(mean(is_on_base, na.rm = TRUE), 3),
    attempts = n()
  ) %>%
  arrange(Pitch)
View(l2_pitch)

#---------------------------------------------
# correlation
#---------------------------------------------
pbp <- subset(pbp, select = -pitch_count)
numeric_data <- pbp[sapply(pbp, is.numeric)]
cor_matrix <- cor(numeric_data, use = "pairwise.complete.obs")
```

```
cor_matrix

threshold <- 0.5  # you can adjust this
high_cor <- which(abs(cor_matrix) > threshold & abs(cor_matrix) < 1, arr.ind = TRUE)
high_cor_pairs <- unique(t(apply(high_cor, 1, sort)))
for (i in 1:nrow(high_cor_pairs)) {
  row <- high_cor_pairs[i, 1]
  col <- high_cor_pairs[i, 2]
  cat(rownames(cor_matrix)[row], " & ", colnames(cor_matrix)[col],
      ": ", cor_matrix[row, col], "\n")
}

# heatmap
library(corrplot)

corrplot(cor_matrix,
         method = "color",      # colored tiles
         type   = "upper",      # upper triangle only
         order  = "hclust",     # cluster similar vars
         addCoef.col = NA,      # set to "black" to print r values on tiles
         tl.cex = 0.8,          # axis label size
         col = colorRampPalette(c("blue", "white", "red"))(200))
```

APPENDIX G
CHAPTER 7 R CODE

```
# ============================== PACKAGES

library(readxl)
library(dplyr)
library(MASS)
library(car)
library(nnet)
library(glmnet)
library(mgcv)
library(arm)
library(pROC)
# des tree
library(rpart)
library(rpart.plot)
# random forest
library(randomForest)
library(caret)

# Hits Decision Tree model

# ============================== LOAD DATA
==================================
df <- read_excel("/Users/teahthies/Desktop/Modeling/cleanpitch.xlsx", sheet = 1)
nrow(df) #6952
colnames(df) # 27 columns
#---------------------------------------------------------------------------
# Convert Columns
#---------------------------------------------------------------------------
# Convert to numeric
df$Inning <- as.numeric(df$Inning)
# Convert to factors (categorical)
df$Pitch <- as.factor(df$Pitch)
df$Bat_hand <- as.factor(df$Bat_hand)
df$code <- as.factor(df$code)
df$Pitch_Result <- as.factor(df$Pitch_Result)
df$home_vis <- as.factor(df$home_vis)
df$hit_type <- as.factor(df$hit_type)
df$Strike <- as.factor(df$Strike)
df$Ball <- as.factor(df$Ball)
df$pitch_count <- as.factor(df$pitch_count)
```

```
# Convert to logical (binary)
df$swing <- as.logical(df$swing)
df$outcome <- as.factor(df$outcome)


df$Location <- as.factor(df$Location)
df$Strike <- as.logical(df$Strike)
df$Ball <- as.logical(df$Ball)
df$pitch_count <- as.numeric(df$pitch_count)

#-------------------------------------------------------------------------
# check
#-------------------------------------------------------------------------
df <- subset(df, bat_event_fl == TRUE)
nrow(df) #4039

# ============================== Train/Test ==============================
library(dplyr)
set.seed(42)

#-------------------------------------------------------------------------
# last 20% into test set
#-------------------------------------------------------------------------
n <- nrow(df)

# index cutoff for 80%
cutoff <- floor(0.80 * n)

# split: first 80% = train, last 20% = test
train_df <- df[1:cutoff, ]
test_df <- df[(cutoff + 1):n, ]

#-------------------------------------------------------------------------
# check
#-------------------------------------------------------------------------
nrow(train_df)  # 3231
nrow(test_df)   #808

# ============================== OUTCOME VARIABLE
# make a binary outcome column
```

```
train_df <- train_df %>%
  mutate(sub_fl = ifelse(code %in% c(20, 21, 22, 23), 1, 0))
unique(train_df$sub_fl)
train_df <- train_df %>% filter(Pitch != "F")
train_df <- train_df %>% filter(Bat_hand == "R")
train_df$sub_fl <- as.logical(train_df$sub_fl)

test_df <- test_df %>%
  mutate(sub_fl = ifelse(code %in% c(20, 21, 22, 23), 1, 0))
unique(test_df$sub_fl)
test_df <- test_df %>% filter(Pitch != "F")
test_df <- test_df %>% filter(Bat_hand == "R")
test_df$sub_fl <- as.logical(test_df$sub_fl)

nrow(train_df)
nrow(test_df)   #404

#-------------------------------------------------------------------------
# Balance ONLY the training data (random oversampling of minority)
#-------------------------------------------------------------------------
df_0 <- train_df %>% filter(sub_fl == 0)
df_1 <- train_df %>% filter(sub_fl == 1)

if (nrow(df_0) >= nrow(df_1)) {
  df_1_os <- df_1 %>% sample_n(nrow(df_0), replace = TRUE)
  train_df <- bind_rows(df_0, df_1_os)
} else {
  df_0_os <- df_0 %>% sample_n(nrow(df_1), replace = TRUE)
  train_df <- bind_rows(df_0_os, df_1)
}

# shuffle rows
train_df <- train_df %>% slice_sample(prop = 1)

# sanity checks
nrow(train_df)
prop.table(table(train_df$sub_fl))

nrow(train_df)
nrow(test_df)   # 404
```

```
train_df <- subset(train_df, select = -Bat_hand)
test_df <- subset(test_df, select = -Bat_hand)

# ============================== RHH Model
# clean data for model
cols_used <- c("strike_count", "sub_fl", "Inning", "Pitch", "Location", "line_up_pos",
               "OPP_score", "num_baserunners", "ball_count", "p_throws", "LBSU_score",
               "num_outs", "home_vis")
sapply(train_df[cols_used], function(x) sum(is.na(x)))
sapply(test_df[cols_used], function(x) sum(is.na(x)))

# get rid of NA's in data we will use
df_clean_train <- train_df[ , cols_used]
df_clean_train <- na.omit(df_clean_train)
df_clean_test <- test_df[ , cols_used]
df_clean_test <- na.omit(df_clean_test)

nrow(df_clean_train)

base_model <- rpart(sub_fl ~ Pitch+Location+
                      line_up_pos+
                      LBSU_score+
                      OPP_score+
                      num_outs+
                      num_baserunners+
                      ball_count+
                      strike_count+
                      Inning+
                      home_vis+
                      p_throws,
                    data = df_clean_train,
                    method = "class",
                    control = rpart.control(cp = 0.02))

# Plot tree
rpart.plot(base_model, type = 3, extra = 104, fallen.leaves = TRUE)

# ----------------------------
# POSITIVE CLASS COLUMN: detect correctly
```

```
# ----------------------------
get_positive_col <- function(prob_mat, y_true) {
  # If logical labels, use "TRUE"
  if (is.logical(y_true)) {
    if ("TRUE" %in% colnames(prob_mat)) return("TRUE")
  }
  # If numeric 0/1, use "1"
  if (is.numeric(y_true)) {
    if ("1" %in% colnames(prob_mat)) return("1")
  }
  # If factor, try common positive names; else use last level as positive
  if (is.factor(y_true)) {
    common_pos <- c("1","TRUE","True","Yes","YES","Y","pos","Positive","Strike","true","T")
    hit <- intersect(colnames(prob_mat), common_pos)
    if (length(hit) > 0) return(hit[1])
    # fallback: last level is positive
    last_lev <- tail(levels(y_true), 1)
    # prefer matching column name if present
    if (last_lev %in% colnames(prob_mat)) return(last_lev)
  }
  # Ultimate fallback: take the last column
  tail(colnames(prob_mat), 1)
}

# ----------------------------
# Helper: metrics for a threshold
# ----------------------------
metrics_from_threshold <- function(y_true, p_hat, thr = 0.5) {
  # Convert to logical positive indicator
  if (is.factor(y_true)) {
    pos_level <- tail(levels(y_true), 1)  # assumes positive is last level if factor
    y_true <- y_true == pos_level
  }
  y_pred <- p_hat >= thr

  TP <- sum(y_pred &  y_true)
  TN <- sum(!y_pred & !y_true)
  FP <- sum(y_pred & !y_true)
  FN <- sum(!y_pred &  y_true)
```

```
  precision   <- ifelse((TP+FP)==0, 0, TP/(TP+FP))
  recall      <- ifelse((TP+FN)==0, 0, TP/(TP+FN))
  f1          <- ifelse((precision+recall)==0, 0, 2*precision*recall/(precision+recall))
  accuracy    <- (TP+TN)/length(y_true)
  specificity <- ifelse((TN+FP)==0, 0, TN/(TN+FP))

  c(accuracy=accuracy, precision=precision, recall=recall,
    f1=f1, specificity=specificity,
    TP=TP, TN=TN, FP=FP, FN=FN, prop_pos_pred=mean(y_pred))
}

# ----------------------------
# TRAIN predictions with correct positive column
# ----------------------------
train_probs_mat <- predict(base_model, newdata = df_clean_train, type = "prob")
pos_col <- get_positive_col(train_probs_mat, df_clean_train$sub_fl)
train_prob <- train_probs_mat[, pos_col]

# Quick sanity check: show column chosen and its range
cat(sprintf("Positive class column chosen: %s\n", pos_col))
cat(sprintf("Train prob summary (pos): min=%.3f median=%.3f max=%.3f\n",
            min(train_prob), median(train_prob), max(train_prob)))

# ----------------------------
# Threshold tuning to maximize F1 (prevent degenerate thresholds)
# ----------------------------
grid <- seq(0.10, 0.90, by = 0.01)

valid_thr <- sapply(grid, function(t) {
  pr <- train_prob >= t
  prop_TRUE <- mean(pr)
  prop_TRUE >= 0.05 && prop_TRUE <= 0.95   # keep both classes in predictions
})
grid2 <- grid[valid_thr]
if (length(grid2) == 0) {
  # If everything was filtered, relax bounds slightly
  valid_thr <- sapply(grid, function(t) {
    pr <- train_prob >= t
    prop_TRUE <- mean(pr)
    prop_TRUE >= 0.02 && prop_TRUE <= 0.98
```

```
  })
  grid2 <- grid[valid_thr]
  if (length(grid2) == 0) grid2 <- grid  # ultimate fallback
}

train_f1s <- sapply(grid2, function(t) {
  metrics_from_threshold(df_clean_train$sub_fl, train_prob, thr = t)["f1"]
})
best_idx <- which(train_f1s == max(train_f1s, na.rm = TRUE))
best_thr <- grid2[best_idx[which.min(abs(grid2[best_idx] - 0.50))]]  # tie-break toward 0.5

cat(sprintf("Chosen threshold to maximize F1 on TRAIN: %.2f\n", best_thr))

# TRAIN metrics at best_thr
m_train <- metrics_from_threshold(df_clean_train$sub_fl, train_prob, thr = best_thr)
print(round(m_train[c("accuracy","precision","recall","f1","specificity","prop_pos_pred")], 3))

# AUC (TRAIN) using the SAME positive column
roc_obj_train <- roc(df_clean_train$sub_fl, train_prob)
cat(sprintf("AUC (TRAIN): %.3f\n", auc(roc_obj_train)))

# ------------------------------
# TEST predictions with same column and threshold
# ------------------------------
test_probs_mat <- predict(base_model, newdata = df_clean_test, type = "prob")

# Use the SAME column name if present; otherwise, re-detect
pos_col_test <- if (pos_col %in% colnames(test_probs_mat)) pos_col else
get_positive_col(test_probs_mat, df_clean_test$sub_fl)
if (pos_col_test != pos_col) {
  cat(sprintf("Note: test prob column differs; using %s\n", pos_col_test))
}
test_prob <- test_probs_mat[, pos_col_test]

m_test <- metrics_from_threshold(df_clean_test$sub_fl, test_prob, thr = best_thr)

# ------------------------------
# Build and show confusion matrix
# ------------------------------
actual <- if (is.factor(df_clean_test$sub_fl)) {
```

```
  as.logical(df_clean_test$sub_fl == tail(levels(df_clean_test$sub_fl), 1))
} else if (is.numeric(df_clean_test$sub_fl)) {
  as.logical(df_clean_test$sub_fl == 1)
} else {
  as.logical(df_clean_test$sub_fl)
}

predicted <- test_prob >= best_thr

print(nrow(df_clean_test))
confusion_matrix <- table(Predicted = predicted, Actual = actual)
print(confusion_matrix)

print(round(m_test[c("accuracy","precision","recall","f1","specificity","prop_pos_pred")], 3))

roc_obj_test <- roc(df_clean_test$sub_fl, test_prob)
cat(sprintf("AUC (TEST): %.3f\n", auc(roc_obj_test)))

{
  cat("\nTEST size:\n")
  print(nrow(df_clean_test))
  cat("\nTEST confusion matrix:\n")
  print(confusion_matrix)
  cat("\nTEST metrics:\n")
  print(round(m_test[c("accuracy","precision","recall","f1","specificity","prop_pos_pred")], 2))
  cat(sprintf("AUC (TEST): %.3f\n", auc(roc_obj_test)))
}

#=========================================================
# PR curves (Plotly) + PR-AUC (PRROC) — robust & printed
#=========================================================
# install.packages(c("PRROC","plotly","htmlwidgets"))  # if needed
library(PRROC)
library(plotly)

make_pr_plot <- function(y_true, y_score, model_name = "Model", split = "TEST") {
  # Ensure score is numeric vector
  y_score <- as.numeric(y_score)

  # ---------- Align lengths ----------
```

```
if (!is.null(names(y_score)) && !is.null(names(y_true))) {
  common_ids <- intersect(names(y_true), names(y_score))
  if (length(common_ids) > 0L) {
    y_true  <- y_true [common_ids]
    y_score <- y_score[common_ids]
  } else {
    n <- min(length(y_true), length(y_score))
    if (length(y_true) != length(y_score)) {
      message(sprintf("[%s] No common names; trimmed to n=%d.", split, n))
    }
    y_true  <- y_true[seq_len(n)]
    y_score <- y_score[seq_len(n)]
  }
} else {
  n <- min(length(y_true), length(y_score))
  if (length(y_true) != length(y_score)) {
    message(sprintf("[%s] Trimmed to n=%d (y_true=%d, y_score=%d).",
                    split, n, length(y_true), length(y_score)))
  }
  y_true  <- y_true[seq_len(n)]
  y_score <- y_score[seq_len(n)]
}

# ---------- Normalize labels to logical (TRUE = positive) ----------
if (is.factor(y_true)) {
  y_true <- y_true == tail(levels(y_true), 1)
} else if (is.numeric(y_true)) {
  y_true <- y_true == 1
} else {
  y_true <- as.logical(y_true)
}

# ---------- Drop NA/Inf ----------
keep <- !is.na(y_true) & !is.na(y_score) & is.finite(y_score)
if (any(!keep)) message(sprintf("[%s] Dropped %d rows with NA/Inf.", split, sum(!keep)))
y_true  <- y_true[keep]
y_score <- y_score[keep]

base <- mean(y_true)
n_pos <- sum(y_true); n_neg <- sum(!y_true)
```

```
  # ---------- If single-class after filtering, draw baseline ----------
  if (length(y_true) == 0L || is.na(base) || n_pos == 0L || n_neg == 0L) {
    pr_df <- data.frame(recall = c(0, 1), precision = c(ifelse(is.na(base), 0, base),
ifelse(is.na(base), 0, base)))
    fig <- plot_ly(pr_df, x = ~recall, y = ~precision,
                   type = "scatter", mode = "lines",
                   name = paste0(model_name, " (PR-AUC = NA)")) |>
      layout(title = paste(model_name, split, "Precision–Recall"),
             xaxis = list(title = "Recall", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
             yaxis = list(title = "Precision", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
             legend = list(orientation = "h"),
             margin = list(l = 60, r = 20, t = 60, b = 60)) |>
      add_lines(x = c(0, 1), y = c(ifelse(is.na(base), 0, base), ifelse(is.na(base), 0, base)),
                name = sprintf("Prevalence = %.2f", ifelse(is.na(base), 0, base)),
                line = list(dash = "dash"))
    return(list(fig = fig, pr_auc = NA_real_))
  }

  # ---------- PR curve + AUC ----------
  sc_pos <- y_score[y_true]
  sc_neg <- y_score[!y_true]
  pr     <- PRROC::pr.curve(scores.class0 = sc_pos,
                            scores.class1 = sc_neg,
                            curve = TRUE)
  pr_auc <- pr$auc.integral

  pr_df <- data.frame(recall = pr$curve[, 1],
                      precision = pr$curve[, 2],
                      threshold = pr$curve[, 3])

  fig <- plot_ly(pr_df, x = ~recall, y = ~precision,
                 type = "scatter", mode = "lines",
                 name = paste0(model_name, sprintf(" (PR-AUC = %.3f)", pr_auc))) |>
    add_lines(x = c(0, 1), y = c(base, base),
              name = sprintf("Prevalence = %.2f", base),
              line = list(dash = "dash")) |>
    layout(title = paste(model_name, split, "Precision–Recall"),
           xaxis = list(title = "Recall", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
           yaxis = list(title = "Precision", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
```

```
        legend = list(orientation = "h"),
        margin = list(l = 60, r = 20, t = 60, b = 60))

  list(fig = fig, pr_auc = pr_auc)
}

# --- TRAIN PR curve & PR-AUC ---
pr_train <- make_pr_plot(df_clean_train$sub_fl, train_prob,
                         model_name = "RHH Final (rpart)", split = "TRAIN")
print(pr_train$fig)  # IMPORTANT when running via source()/Rterm
cat(sprintf("PR-AUC (TRAIN): %.3f\n", pr_train$pr_auc))

# --- TEST PR curve & PR-AUC ---
pr_test <- make_pr_plot(df_clean_test$sub_fl, test_prob,
                        model_name = "RHH Final (rpart)", split = "TEST")
print(pr_test$fig)   # IMPORTANT when running via source()/Rterm
cat(sprintf("PR-AUC (TEST): %.3f\n", pr_test$pr_auc))




### New Script: OBP Deceision Tree Model
# ============================== OUTCOME VARIABLE
================================
# make a binary outcome column
train_df <- train_df %>%
  mutate(sub_fl = ifelse(code %in% c(18, 20, 21, 22, 23), 1, 0))
unique(train_df$sub_fl)
train_df <- train_df %>% filter(Pitch != "F")
train_df <- train_df %>% filter(Bat_hand == "R")
train_df$sub_fl <- as.logical(train_df$sub_fl)

test_df <- test_df %>%
  mutate(sub_fl = ifelse(code %in% c(18, 20, 21, 22, 23), 1, 0))
unique(test_df$sub_fl)
test_df <- test_df %>% filter(Pitch != "F")
test_df <- test_df %>% filter(Bat_hand == "R")
test_df$sub_fl <- as.logical(test_df$sub_fl)

nrow(train_df)
```

```
nrow(test_df)   #404

#------------------------------------------------------------------------
# Balance ONLY the training data (random oversampling of minority)
#------------------------------------------------------------------------
df_0 <- train_df %>% filter(sub_fl == 0)
df_1 <- train_df %>% filter(sub_fl == 1)

if (nrow(df_0) >= nrow(df_1)) {
  df_1_os <- df_1 %>% sample_n(nrow(df_0), replace = TRUE)
  train_df <- bind_rows(df_0, df_1_os)
} else {
  df_0_os <- df_0 %>% sample_n(nrow(df_1), replace = TRUE)
  train_df <- bind_rows(df_0_os, df_1)
}

# shuffle rows
train_df <- train_df %>% slice_sample(prop = 1)

# sanity checks
nrow(train_df)
prop.table(table(train_df$sub_fl))

nrow(train_df)
nrow(test_df)   # 404
train_df <- subset(train_df, select = -Bat_hand)
test_df <- subset(test_df, select = -Bat_hand)

# ============================== RHH Model
# clean data for model
cols_used <- c("strike_count", "sub_fl", "Inning", "Pitch", "Location", "line_up_pos",
               "OPP_score", "num_baserunners", "ball_count", "p_throws", "LBSU_score",
               "num_outs", "home_vis")
sapply(train_df[cols_used], function(x) sum(is.na(x)))
sapply(test_df[cols_used], function(x) sum(is.na(x)))

# get rid of NA's in data we will use
df_clean_train <- train_df[ , cols_used]
df_clean_train <- na.omit(df_clean_train)
df_clean_test <- test_df[ , cols_used]
```

```
df_clean_test <- na.omit(df_clean_test)

nrow(df_clean_train)

base_model <- rpart(sub_fl ~ Pitch+Location+
                      line_up_pos+
                      LBSU_score+
                      OPP_score+
                      num_outs+
                      num_baserunners+
                      ball_count+
                      strike_count+
                      Inning+
                      home_vis+
                      p_throws,
                    data = df_clean_train,
                    method = "class",
                    control = rpart.control(cp = 0.012))

# Plot tree
rpart.plot(base_model, type = 3, extra = 104, fallen.leaves = TRUE)

# ------------------------------
# POSITIVE CLASS COLUMN: detect correctly
# ------------------------------
get_positive_col <- function(prob_mat, y_true) {
  # If logical labels, use "TRUE"
  if (is.logical(y_true)) {
    if ("TRUE" %in% colnames(prob_mat)) return("TRUE")
  }
  # If numeric 0/1, use "1"
  if (is.numeric(y_true)) {
    if ("1" %in% colnames(prob_mat)) return("1")
  }
  # If factor, try common positive names; else use last level as positive
  if (is.factor(y_true)) {
    common_pos <- c("1","TRUE","True","Yes","YES","Y","pos","Positive","Strike","true","T")
    hit <- intersect(colnames(prob_mat), common_pos)
    if (length(hit) > 0) return(hit[1])
    # fallback: last level is positive
```

```
  last_lev <- tail(levels(y_true), 1)
  # prefer matching column name if present
  if (last_lev %in% colnames(prob_mat)) return(last_lev)
 }
 # Ultimate fallback: take the last column
 tail(colnames(prob_mat), 1)
}

# ------------------------------
# Helper: metrics for a threshold
# ------------------------------
metrics_from_threshold <- function(y_true, p_hat, thr = 0.5) {
 # Convert to logical positive indicator
 if (is.factor(y_true)) {
  pos_level <- tail(levels(y_true), 1)  # assumes positive is last level if factor
  y_true <- y_true == pos_level
 }
 y_pred <- p_hat >= thr

 TP <- sum(y_pred &  y_true)
 TN <- sum(!y_pred & !y_true)
 FP <- sum(y_pred & !y_true)
 FN <- sum(!y_pred &  y_true)

 precision   <- ifelse((TP+FP)==0, 0, TP/(TP+FP))
 recall      <- ifelse((TP+FN)==0, 0, TP/(TP+FN))
 f1          <- ifelse((precision+recall)==0, 0, 2*precision*recall/(precision+recall))
 accuracy    <- (TP+TN)/length(y_true)
 specificity <- ifelse((TN+FP)==0, 0, TN/(TN+FP))

 c(accuracy=accuracy, precision=precision, recall=recall,
  f1=f1, specificity=specificity,
  TP=TP, TN=TN, FP=FP, FN=FN, prop_pos_pred=mean(y_pred))
}

# ------------------------------
# TRAIN predictions with correct positive column
# ------------------------------
train_probs_mat <- predict(base_model, newdata = df_clean_train, type = "prob")
pos_col <- get_positive_col(train_probs_mat, df_clean_train$sub_fl)
```

```
train_prob <- train_probs_mat[, pos_col]

# Quick sanity check: show column chosen and its range
cat(sprintf("Positive class column chosen: %s\n", pos_col))
cat(sprintf("Train prob summary (pos): min=%.3f median=%.3f max=%.3f\n",
            min(train_prob), median(train_prob), max(train_prob)))

# ----------------------------
# Threshold tuning to maximize F1 (prevent degenerate thresholds)
# ----------------------------
grid <- seq(0.10, 0.90, by = 0.01)

valid_thr <- sapply(grid, function(t) {
  pr <- train_prob >= t
  prop_TRUE <- mean(pr)
  prop_TRUE >= 0.05 && prop_TRUE <= 0.95   # keep both classes in predictions
})
grid2 <- grid[valid_thr]
if (length(grid2) == 0) {
  # If everything was filtered, relax bounds slightly
  valid_thr <- sapply(grid, function(t) {
    pr <- train_prob >= t
    prop_TRUE <- mean(pr)
    prop_TRUE >= 0.02 && prop_TRUE <= 0.98
  })
  grid2 <- grid[valid_thr]
  if (length(grid2) == 0) grid2 <- grid  # ultimate fallback
}

train_f1s <- sapply(grid2, function(t) {
  metrics_from_threshold(df_clean_train$sub_fl, train_prob, thr = t)["f1"]
})
best_idx <- which(train_f1s == max(train_f1s, na.rm = TRUE))
best_thr <- grid2[best_idx[which.min(abs(grid2[best_idx] - 0.50))]]  # tie-break toward 0.5

cat(sprintf("Chosen threshold to maximize F1 on TRAIN: %.2f\n", best_thr))

# TRAIN metrics at best_thr
m_train <- metrics_from_threshold(df_clean_train$sub_fl, train_prob, thr = best_thr)
print(round(m_train[c("accuracy","precision","recall","f1","specificity","prop_pos_pred")], 3))
```

```
# AUC (TRAIN) using the SAME positive column
roc_obj_train <- roc(df_clean_train$sub_fl, train_prob)
cat(sprintf("AUC (TRAIN): %.3f\n", auc(roc_obj_train)))

# ----------------------------
# TEST predictions with same column and threshold
# ----------------------------
test_probs_mat <- predict(base_model, newdata = df_clean_test, type = "prob")

# Use the SAME column name if present; otherwise, re-detect
pos_col_test <- if (pos_col %in% colnames(test_probs_mat)) pos_col else
get_positive_col(test_probs_mat, df_clean_test$sub_fl)
if (pos_col_test != pos_col) {
  cat(sprintf("Note: test prob column differs; using %s\n", pos_col_test))
}
test_prob <- test_probs_mat[, pos_col_test]

m_test <- metrics_from_threshold(df_clean_test$sub_fl, test_prob, thr = best_thr)

# ----------------------------
# Build and show confusion matrix
# ----------------------------
actual <- if (is.factor(df_clean_test$sub_fl)) {
  as.logical(df_clean_test$sub_fl == tail(levels(df_clean_test$sub_fl), 1))
} else if (is.numeric(df_clean_test$sub_fl)) {
  as.logical(df_clean_test$sub_fl == 1)
} else {
  as.logical(df_clean_test$sub_fl)
}

predicted <- test_prob >= best_thr
print(nrow(df_clean_test))
confusion_matrix <- table(Predicted = predicted, Actual = actual)
print(confusion_matrix)

print(round(m_test[c("accuracy","precision","recall","f1","specificity","prop_pos_pred")], 3))
roc_obj_test <- roc(df_clean_test$sub_fl, test_prob)
cat(sprintf("AUC (TEST): %.3f\n", auc(roc_obj_test)))
```

```
{
  cat("\nTEST size:\n")
  print(nrow(df_clean_test))
  cat("\nTEST confusion matrix:\n")
  print(confusion_matrix)
  cat("\nTEST metrics:\n")
  print(round(m_test[c("accuracy","precision","recall","f1","specificity","prop_pos_pred")], 2))
  cat(sprintf("AUC (TEST): %.3f\n", auc(roc_obj_test)))
}

#============================================================
# PR curves (Plotly) + PR-AUC (PRROC) — robust & printed
#============================================================
# install.packages(c("PRROC","plotly","htmlwidgets"))  # if needed
library(PRROC)
library(plotly)

make_pr_plot <- function(y_true, y_score, model_name = "Model", split = "TEST") {
  # Ensure score is numeric vector
  y_score <- as.numeric(y_score)

  # ---------- Align lengths ----------
  if (!is.null(names(y_score)) && !is.null(names(y_true))) {
    common_ids <- intersect(names(y_true), names(y_score))
    if (length(common_ids) > 0L) {
      y_true  <- y_true [common_ids]
      y_score <- y_score[common_ids]
    } else {
      n <- min(length(y_true), length(y_score))
      if (length(y_true) != length(y_score)) {
        message(sprintf("[%s] No common names; trimmed to n=%d.", split, n))
      }
      y_true  <- y_true[seq_len(n)]
      y_score <- y_score[seq_len(n)]
    }
  } else {
    n <- min(length(y_true), length(y_score))
    if (length(y_true) != length(y_score)) {
      message(sprintf("[%s] Trimmed to n=%d (y_true=%d, y_score=%d).",
                      split, n, length(y_true), length(y_score)))
```

```
  }
  y_true  <- y_true[seq_len(n)]
  y_score <- y_score[seq_len(n)]
 }

 # ---------- Normalize labels to logical (TRUE = positive) ----------
 if (is.factor(y_true)) {
  y_true <- y_true == tail(levels(y_true), 1)
 } else if (is.numeric(y_true)) {
  y_true <- y_true == 1
 } else {
  y_true <- as.logical(y_true)
 }

 # ---------- Drop NA/Inf ----------
 keep <- !is.na(y_true) & !is.na(y_score) & is.finite(y_score)
 if (any(!keep)) message(sprintf("[%s] Dropped %d rows with NA/Inf.", split, sum(!keep)))
 y_true  <- y_true[keep]
 y_score <- y_score[keep]

 base <- mean(y_true)
 n_pos <- sum(y_true); n_neg <- sum(!y_true)

 # ---------- If single-class after filtering, draw baseline ----------
 if (length(y_true) == 0L || is.na(base) || n_pos == 0L || n_neg == 0L) {
  pr_df <- data.frame(recall = c(0, 1), precision = c(ifelse(is.na(base), 0, base),
ifelse(is.na(base), 0, base)))
  fig <- plot_ly(pr_df, x = ~recall, y = ~precision,
                 type = "scatter", mode = "lines",
                 name = paste0(model_name, " (PR-AUC = NA)")) |>
   layout(title = paste(model_name, split, "Precision–Recall"),
          xaxis = list(title = "Recall", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
          yaxis = list(title = "Precision", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
          legend = list(orientation = "h"),
          margin = list(l = 60, r = 20, t = 60, b = 60)) |>
   add_lines(x = c(0, 1), y = c(ifelse(is.na(base), 0, base), ifelse(is.na(base), 0, base)),
             name = sprintf("Prevalence = %.2f", ifelse(is.na(base), 0, base)),
             line = list(dash = "dash"))
  return(list(fig = fig, pr_auc = NA_real_))
 }
```

```
  # ---------- PR curve + AUC ----------
  sc_pos <- y_score[y_true]
  sc_neg <- y_score[!y_true]
  pr     <- PRROC::pr.curve(scores.class0 = sc_pos,
                            scores.class1 = sc_neg,
                            curve = TRUE)
  pr_auc <- pr$auc.integral

  pr_df <- data.frame(recall = pr$curve[, 1],
                      precision = pr$curve[, 2],
                      threshold = pr$curve[, 3])

  fig <- plot_ly(pr_df, x = ~recall, y = ~precision,
                 type = "scatter", mode = "lines",
                 name = paste0(model_name, sprintf(" (PR-AUC = %.3f)", pr_auc))) |>
    add_lines(x = c(0, 1), y = c(base, base),
              name = sprintf("Prevalence = %.2f", base),
              line = list(dash = "dash")) |>
    layout(title = paste(model_name, split, "Precision–Recall"),
           xaxis = list(title = "Recall", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
           yaxis = list(title = "Precision", range = c(0, 1), tick0 = 0, dtick = 0.1, zeroline = FALSE),
           legend = list(orientation = "h"),
           margin = list(l = 60, r = 20, t = 60, b = 60))

  list(fig = fig, pr_auc = pr_auc)
}

# --- TRAIN PR curve & PR-AUC ---
pr_train <- make_pr_plot(df_clean_train$sub_fl, train_prob,
                         model_name = "RHH Final (rpart)", split = "TRAIN")
print(pr_train$fig)  # IMPORTANT when running via source()/Rterm
cat(sprintf("PR-AUC (TRAIN): %.3f\n", pr_train$pr_auc))

# --- TEST PR curve & PR-AUC ---
pr_test <- make_pr_plot(df_clean_test$sub_fl, test_prob,
                        model_name = "RHH Final (rpart)", split = "TEST")
print(pr_test$fig)   # IMPORTANT when running via source()/Rterm
cat(sprintf("PR-AUC (TEST): %.3f\n", pr_test$pr_auc))
```

APPENDIX H
CHAPTER 7 PYTHON CODE

```
# -*- coding: utf-8 -*-
"""DOMAIN.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1jlds8kwKBC_agV7g0z1kbHCGmEXMGu95

# **Load Required Packages and Designate "Hand-Event" Pair for the Notebook**
"""

import pandas as pd
import requests
from bs4 import BeautifulSoup
from io import StringIO
import plotly.express as px
import numpy as np
from IPython.display import Image   #Display images
import warnings                     #Ignore version warnings
warnings.simplefilter('ignore', FutureWarning)

"""Code book establishing event codes with their actual description.

Use this next cell lists the codes that corresponds to the different events:

*  Strikeouts: event_code = 4

*  Walks: event_code = 18

*  Hits: event_code = c(20, 21, 22, 23)

*  OBP: event_code = c(18, 20, 21, 22, 23)

*  HR: event_code = 23
"""

Strikeouts = [4,4.0,'4','4.0']

Walks = [18,18.0,'18','18.0']

Hits = [20, 20.0, '20', '20.0', 21, 21.0, '21', '21.0', 22, 22.0, '22', '22.0', 23, 23.0, '23', '23.0']

OBP = [18, 18.0, '18', '18.0', 20, 20.0, '20', '20.0', 21, 21.0, '21', '21.0', 22, 22.0, '22', '22.0', 23,
23.0, '23', '23.0']

HR = [23,23.0,'23','23.0']
```

"""In the next cell, let's choose the `Hand`: R or L and `event_code` that we'll use **throughout the notebook**

This will allow us to "set it and forget it", designating and running analysis for a given "Hand-Event" pair throughout the notebook.
"""

Hand = 'L'
event_code = Strikeouts
event_code_text = "Strikeouts"

"""# **Curate the Baseball Source Data**

Import the data
"""

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 100)

def read_google_sheet(sheet_id, sheet_name):
    url =
f"https://docs.google.com/spreadsheets/d/{sheet_id}/gviz/tq?tqx=out:csv&sheet={sheet_name}"
    return pd.read_csv(url)

baseball_sheet_id = '1H_g294-sN58ONfFGWTOKIYF3L13nsZem'
baseball_sheet_name = 'baseball'
baseball = read_google_sheet(baseball_sheet_id, baseball_sheet_name)
baseball

"""For machine learning analysis, we'll need to one-hot encode location category. Step one in that direction is mapping numeric categories to letters so we don't confuse the model."""

baseball['zone'] = baseball['zone'].astype('category')

# Create a mapping dictionary for zone to alphabetical categories
zone_mapping = {
    1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E',
    6: 'F', 7: 'G', 8: 'H', 9: 'I'
}

Create the new 'zone_cat' column using the mapping
baseball['zone_cat'] = baseball['zone'].map(zone_mapping)

baseball.rename(columns={'outcome':'events'},inplace=True)

#baseball['pt_flag'] = baseball['pitch_type'].isin(Hits)

"""## Customize Baseball Data Balance Based on Event Predicted

The baseball data was too massive for colab to handle, so we'll sample from it to cut down compute time. The next code block will create baseball dataframes (train, validate and test) of our desired size based on event.

* If `event = Strikeouts`, then only sample rows with `strikes == 2`. Then with the `prop_BB` parameter, we can dictate the proportion of strikes there are for each number of walks. This will strategically dictate class (im)balance in our target.
* If `event = Walks`, then only sample rows with `balls == 3`. Then with the prop3 parameter, determine the proportion of walks. This will strategically dictate class (im)balance in our target.
* If `event = Hits / OBP / HR` no restriction on rows selected. The `balanced` parameter will determine whether target is balanced or randomly determined.
"""

"""## Customize Data Balance by Strikes and Number of Balls"""

'''
# --- assume your DataFrame is named baseball_c ---
import math as math

1) Parse dates (safe even if already datetime)
#baseball_c = baseball.copy()
#baseball_c = baseball[((baseball['strikes']==2)&(baseball['stand']==Hand))]
#baseball_c = baseball[((baseball['balls']==3)&(baseball['stand']==Hand))]
baseball_c = baseball[baseball['stand']==Hand]

baseball_c['game_date'] = pd.to_datetime(baseball_c['game_date'], errors='coerce')

# 2) Split by month boundaries (Aug=8 through Oct=10) for test; strictly before Aug for train pool
is_test_n_val_window = baseball_c['game_date'].dt.month.between(8, 10, inclusive='both')
test_n_val_pool = baseball_c.loc[is_test_n_val_window].sort_values('game_date')
train_pool = baseball_c.loc[~is_test_n_val_window & baseball_c['game_date'].notna() & (baseball_c['game_date'].dt.month < 8)].sort_values('game_date')

# 3) Sample TEST set: any 4,000 rows from Aug–Oct (no balance requirement)
#   Use a fixed seed for reproducibility
val_n = 3000
val_n = min(val_n, len(test_n_val_pool)) # just in case

```
baseball_val = test_n_val_pool.sample(n=val_n, replace=False,
random_state=40).sort_values('game_date')

test_n = 3000
test_pool = test_n_val_pool.drop(index=baseball_val.index)          # remove val rows
test_n = min(test_n, len(test_pool))                                # just in case
baseball_test = test_pool.sample(n=test_n, replace=False,
random_state=42).sort_values('game_date')

# 4) Build a BALANCED TRAIN set (16,000 total; 8,000 with events==4.0 and 8,000 !=4.0)
BB = [5000,9000,8000,4000]
#BB = [0,0,0,20000]
prop_BB = [0.20, 0.4, 0.25, 0.15]

target_train_total = sum(BB)
half = target_train_total // 2

grp_4_0 = train_pool.loc[(train_pool['events'].isin(event_code)) & (train_pool['balls']==0)]
grp_4_1 = train_pool.loc[(train_pool['events'].isin(event_code)) & (train_pool['balls']==1)]
grp_4_2 = train_pool.loc[(train_pool['events'].isin(event_code)) & (train_pool['balls']==2)]
grp_4_3 = train_pool.loc[(train_pool['events'].isin(event_code)) & (train_pool['balls']==3)]

grp_not_0 = train_pool.loc[(~train_pool['events'].isin(event_code)) & (train_pool['balls']==0)]
grp_not_1 = train_pool.loc[(~train_pool['events'].isin(event_code)) & (train_pool['balls']==1)]
grp_not_2 = train_pool.loc[(~train_pool['events'].isin(event_code)) & (train_pool['balls']==2)]
grp_not_3 = train_pool.loc[(~train_pool['events'].isin(event_code)) & (train_pool['balls']==3)]

# If a group has fewer than needed, upsample with replacement to hit the target;
# otherwise downsample without replacement.
def sample_group(df, n, seed=42):
    if len(df) == 0:
        # create an empty frame with same columns if the group is empty
        return df.head(0)
    replace = len(df) < n
    return df.sample(n=n, replace=replace, random_state=seed)

train_not_0 = sample_group(grp_not_0, math.floor(BB[0]-prop_BB[0]*BB[0]), seed=41)
train_not_1 = sample_group(grp_not_1, math.floor(BB[1]-prop_BB[1]*BB[1]), seed=42)
train_not_2 = sample_group(grp_not_2, math.floor(BB[2]-prop_BB[2]*BB[2]), seed=43)
train_not_3 = sample_group(grp_not_3, math.floor(BB[3]-prop_BB[3]*BB[3]), seed=44)

train_4_0 = sample_group(grp_4_0, math.floor(prop_BB[0]*BB[0]), seed=4241)  # different
seed for variety
train_4_1 = sample_group(grp_4_1, math.floor(prop_BB[1]*BB[1]), seed=4242)
train_4_2 = sample_group(grp_4_2, math.floor(prop_BB[2]*BB[2]), seed=4243)
train_4_3 = sample_group(grp_4_3, math.floor(prop_BB[3]*BB[3]), seed=4244)
```

```
baseball_train = (
   pd.concat([train_4_0, train_4_1, train_4_2, train_4_3,train_not_0, train_not_1, train_not_2,
train_not_3], axis=0)
     .sort_values('game_date')
     .reset_index(drop=True)
)


# 5) (Optional) sanity checks
print("Train shape:", baseball_train.shape, "| % events==4:",
     (baseball_train['events'].isin(event_code).astype(int)).mean().round(3))
print("Val shape:", baseball_val.shape)
print("Test shape:", baseball_test.shape)
# If you later want to drop the date column after splitting, do:
# baseball_train = baseball_train.drop(columns=['game_date'])
# baseball_test  = baseball_test.drop(columns=['game_date'])
'''

import math
import numpy as np
import pandas as pd

def make_splits_and_sample(
   baseball: pd.DataFrame,
   event_code_text: str,
   Hand=None,                        # e.g. 'L' or 'R' to filter 'stand'
   val_n: int = 3000,
   test_n: int = 3000,
   random_state_val: int = 40,
   random_state_test: int = 42,

   # --- Strikeouts controls (per-balls sampling) ---
   BB = (5000, 9000, 8000, 4000),        # target rows for balls==0,1,2,3 (pre-Aug train only)
   prop_BB = (0.20, 0.40, 0.25, 0.15),    # positive share within each balls group

   # --- Walks controls (balls==3 only) ---
   B3: int = 12000,                    # total rows for balls==3 (pre-Aug train only)
   prop3: float = 0.35,                 # positive share within balls==3

   # --- Hits / OBP / HR controls (balanced option) ---
   balanced: bool = True,               # if True -> balanced 50/50 positives/negatives
   train_n: int | None = 16000,          # total train rows (if None, auto = 2 * min(pos,neg))
   oversample_ok: bool = True            # allow replacement when a bucket is small
):
```

```
    """
    Returns:
      baseball_train, baseball_val, baseball_test
    """

    # Map outcome names to event codes (robust to ints/floats/strings)
    EVENT_MAP = {
        "Strikeouts": [4, 4.0, "4", "4.0"],
        "Walks":      [18, 18.0, "18", "18.0"],
        "Hits":       [20, 20.0, "20", "20.0", 21, 21.0, "21", "21.0", 22, 22.0, "22", "22.0", 23, 23.0,
"23", "23.0"],
        "OBP":        [18, 18.0, "18", "18.0", 20, 20.0, "20", "20.0", 21, 21.0, "21", "21.0", 22, 22.0,
"22", "22.0", 23, 23.0, "23", "23.0"],
        "HR":         [23, 23.0, "23", "23.0"],
    }
    if event_code_text not in EVENT_MAP:
        raise ValueError(f"Unknown event_code_text: {event_code_text}")

    # Helper: robust event mask (match 'events' as string equality)
    ev_vals_str = set(str(x) for x in EVENT_MAP[event_code_text])
    def is_event(s: pd.Series) -> pd.Series:
        return s.astype(str).isin(ev_vals_str)

    # ---- 0) Base filtering & date parsing
    df = baseball.copy()
    if Hand is not None and 'stand' in df.columns:
        df = df.loc[df['stand'] == Hand].copy()

    df['game_date'] = pd.to_datetime(df['game_date'], errors='coerce')
    # optional coercions
    if 'balls' in df.columns:
        df['balls'] = pd.to_numeric(df['balls'], errors='coerce').astype('Int64')
    if 'strikes' in df.columns:
        df['strikes'] = pd.to_numeric(df['strikes'], errors='coerce').astype('Int64')

    # ---- 1) Event-specific row restriction (pre-split)
    if event_code_text == "Strikeouts" and 'strikes' in df.columns:
        df = df.loc[df['strikes'] == 2].copy()
    elif event_code_text == "Walks" and 'balls' in df.columns:
        df = df.loc[df['balls'] == 3].copy()
    # else: no additional restriction

    # ---- 2) Time split: Aug–Oct for val/test; pre-Aug for train pool
    is_valtest = df['game_date'].dt.month.between(8, 10, inclusive='both')
    test_n_val_pool = df.loc[is_valtest].sort_values('game_date')
    train_pool = df.loc[~is_valtest & df['game_date'].notna() &
```

```
                (df['game_date'].dt.month < 8)].sort_values('game_date')

    # ---- 3) Sample val, then test from the remainder (disjoint)
    val_n = min(val_n, len(test_n_val_pool))
    baseball_val = test_n_val_pool.sample(n=val_n, replace=False,
random_state=random_state_val).sort_values('game_date')
    test_pool = test_n_val_pool.drop(index=baseball_val.index)
    test_n = min(test_n, len(test_pool))
    baseball_test = test_pool.sample(n=test_n, replace=False,
random_state=random_state_test).sort_values('game_date')

    # ---- 4) Build TRAIN according to rules
    rng_seeds = [41, 42, 43, 44]  # per-balls seeds

    def sample_group(df_in: pd.DataFrame, n: int, seed: int) -> pd.DataFrame:
        if n <= 0:
            return df_in.head(0).copy()
        if len(df_in) == 0:
            return df_in.head(0).copy()
        replace = oversample_ok and (len(df_in) < n)
        return df_in.sample(n=n, replace=replace, random_state=seed)

    event_mask = is_event(train_pool['events']) if 'events' in train_pool.columns else
pd.Series(False, index=train_pool.index)

    if event_code_text == "Strikeouts":
        # per-balls buckets 0..3
        parts = []
        for b, (BBb, pp, seed) in enumerate(zip(BB, prop_BB, rng_seeds)):
            bucket = train_pool.loc[train_pool['balls'] == b]
            pos = bucket.loc[event_mask.loc[bucket.index]]
            neg = bucket.loc[~event_mask.loc[bucket.index]]
            n_pos = math.floor(pp * BBb)
            n_neg = BBb - n_pos
            parts.append(sample_group(pos, n_pos, seed))
            parts.append(sample_group(neg, n_neg, seed + 2000))
        baseball_train = (pd.concat(parts, axis=0)
                          .sort_values('game_date')
                          .reset_index(drop=True))

    elif event_code_text == "Walks":
        # train_pool already filtered to balls==3
        pos = train_pool.loc[event_mask]
        neg = train_pool.loc[~event_mask]
        n_pos = math.floor(prop3 * B3)
        n_neg = B3 - n_pos
```

```
        train_pos = sample_group(pos, n_pos, 4242)
        train_neg = sample_group(neg, n_neg, 4243)
        baseball_train = (pd.concat([train_pos, train_neg], axis=0)
                          .sort_values('game_date')
                          .reset_index(drop=True))

    else:  # Hits / OBP / HR
        pos = train_pool.loc[event_mask]
        neg = train_pool.loc[~event_mask]
        if balanced:
            if train_n is None:
                # auto size: as large as the smaller class allows (or oversample if allowed)
                n_each = min(len(pos), len(neg))
                if oversample_ok and n_each == 0:
                    n_each = max(len(pos), len(neg))
                n_pos, n_neg = n_each, n_each
            else:
                n_each = train_n // 2
                n_pos, n_neg = n_each, n_each
            train_pos = sample_group(pos, n_pos, 5251)
            train_neg = sample_group(neg, n_neg, 5252)
        else:
            # unbalanced: just take up to train_n from the natural mix (or all if None)
            if train_n is None:
                baseball_train = train_pool.copy().reset_index(drop=True)
            else:
                baseball_train = train_pool.sample(
                    n=min(train_n, len(train_pool)),
                    replace=False,
                    random_state=5250
                ).sort_values('game_date').reset_index(drop=True)
            # finalize outputs and sanity prints at the end
            print("Train shape:", baseball_train.shape,
                  "| % event:",
                  (is_event(baseball_train['events']).astype(int).mean() if 'events' in
baseball_train.columns else np.nan))
            print("Val shape:", baseball_val.shape)
            print("Test shape:", baseball_test.shape)
            return baseball_train, baseball_val, baseball_test

        baseball_train = (pd.concat([train_pos, train_neg], axis=0)
                          .sort_values('game_date')
                          .reset_index(drop=True))

    # ---- 5) Sanity prints
    pct_event = (is_event(baseball_train['events']).astype(int).mean().round(3)
```

```
            if 'events' in baseball_train.columns else np.nan)
    print("Train shape:", baseball_train.shape, "| % event:", pct_event)
    print("Val shape:", baseball_val.shape)
    print("Test shape:", baseball_test.shape)

    return baseball_train, baseball_val, baseball_test

# Run the function to create source data
baseball_train, baseball_val, baseball_test = make_splits_and_sample(
  baseball=baseball,
  event_code_text=event_code_text,
  Hand=Hand,
  balanced=True,
  train_n=20000)

"""Check it out to make sure it looks as it should:"""

baseball_train1 =
baseball_train[baseball_train['pitch_type'].notna()][baseball_train['zone'].notna()]
baseball_train1

baseball_train1['strikes'].value_counts()

"""Remove missing values"""

baseball_test1 = baseball_test[baseball_test['pitch_type'].notna()][baseball_test['zone'].notna()]
baseball_test1

"""Examine event code distribution"""

baseball_test1['events'].value_counts()

"""## Import Softball Data"""

softball_sheet_id = '1KsnPJJR_ecJ8YFYUeLIM0dIEvpK9xNii'
softball_sheet_name = 'softball'
softball_train = read_google_sheet(softball_sheet_id, softball_sheet_name)
softball_train

softball_sheet_idt = '1CsZG_pSBjAtsIOU2-N6fRU8yAzPRvqMX'
softball_sheet_namet = 'softball'
softball_test = read_google_sheet(softball_sheet_idt, softball_sheet_namet)
softball_test

softball_test1 = softball_test.drop(columns=['Unnamed: 17','Unnamed: 18','Unnamed:
19','Unnamed: 20','Unnamed: 21','Unnamed: 22','Unnamed: 23','Unnamed: 24','Unnamed: 25'])
```

```
softball_test1

pd.set_option('display.max_rows', 150)

"""Double check pitch_type distribution"""

pd.value_counts(softball_test1[((softball_test1['strikes']==2)&(softball_test1['stand']=='L'))]['pitc
h_type'])

"""# **Ensure Softball and Baseball have Same Features**

Automate feature pre-processing steps
"""

import pandas as pd
import numpy as np

def make_softball_c(softball: pd.DataFrame, zone_mapping: dict, event_code_text: str = None) -
> pd.DataFrame:
    """
    Build softball_c with optional event-specific row restrictions:

    - If event_code_text == 'Strikeouts' -> keep only rows with strikes == 2
    - If event_code_text == 'Walks'      -> keep only rows with balls   == 3
    - Otherwise                          -> no additional restriction

    Steps:
      1) Rename outcome -> events (if present)
      2) Cast zone to category and map to zone_cat via zone_mapping
      3) Apply conditional row restriction (see above)
      4) Drop rows with NA in pitch_type, zone, p_throws
      5) Remove pitch_type == 'F'
      6) Drop events in {6, 17} if present
      7) Create sub_fl = 1{events == 4} (robust to '4'/'4.0' strings)
      8) Return softball_c
    """
    # (0) Work on a copy
    df = softball.copy()
    df = df[df['stand']==Hand]
    # (1) Rename outcome -> events if needed
    if 'outcome' in df.columns and 'events' not in df.columns:
        df = df.rename(columns={'outcome': 'events'})
    elif 'outcome' in df.columns and 'events' in df.columns:
        df = df.drop(columns=['outcome'])

    # (2) Zone dtype + mapping
```

```
    if 'zone' in df.columns:
        df['zone'] = df['zone'].astype('category')
        df['zone_cat'] = df['zone'].map(zone_mapping)

    # (3) Conditional row restriction based on event_code
    if event_code_text == 'Strikeouts' and 'strikes' in df.columns:
        df1 = df.loc[df['strikes'] == 2].copy()
    elif event_code_text == 'Walks' and 'balls' in df.columns:
        df1 = df.loc[df['balls'] == 3].copy()
    else:
        df1 = df.copy()

    # (4) Drop NA in key columns
    required = ['pitch_type', 'zone', 'p_throws']
    mask = pd.Series(True, index=df1.index)
    for c in required:
        if c in df1.columns:
            mask &= df1[c].notna()
    softball_a = df1.loc[mask].copy()

    # (5) Remove pitch_type == 'F'
    if 'pitch_type' in softball_a.columns:
        softball_b = softball_a.loc[softball_a['pitch_type'] != 'F'].copy()
    else:
        softball_b = softball_a.copy()

    # (6) Drop events 6 and 17 (if events exists)
    if 'events' in softball_b.columns:
        ev_num = pd.to_numeric(softball_b['events'], errors='coerce')
        softball_b = softball_b.loc[~ev_num.isin([6, 17])].copy()

    # (7) sub_fl = 1 if events == 4 (handle strings like "4" or "4.0")
    if 'events' in softball_b.columns:
        events_num = pd.to_numeric(softball_b['events'], errors='coerce')
        softball_b['sub_fl'] = (events_num.isin(event_code)).astype(int)
    else:
        softball_b['sub_fl'] = 0

    # (8) Return
    return softball_b.reset_index(drop=True)

baseball_c_train = make_softball_c(baseball_train,{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E', 6: 'F', 7: 'G',
8: 'H', 9: 'I'},event_code_text)
baseball_c_val = make_softball_c(baseball_val,{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E', 6: 'F', 7: 'G', 8:
'H', 9: 'I'},event_code_text)
```

```
baseball_c_test = make_softball_c(baseball_test,{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E', 6: 'F', 7: 'G', 8: 'H', 9: 'I'},event_code_text)

softball_c_train = make_softball_c(softball_train,{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E', 6: 'F', 7: 'G', 8: 'H', 9: 'I'},event_code_text)
softball_c_test = make_softball_c(softball_test1,{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E', 6: 'F', 7: 'G', 8: 'H', 9: 'I'},event_code_text)

softball_c_train.isna().sum()

"""For all categorical variables, most notoriously `pitch_type`, softball and baseball must take only the same values, no mismatches."""

# prompt: what are the differences in the categories of pitch_type between the baseball and softball_c data frames

baseball_pitch_types =
set(np.concatenate([baseball_train['pitch_type'].unique(),baseball_val['pitch_type'].unique(),baseball_test['pitch_type'].unique()]))
softball_c_pitch_types =
set(np.concatenate([softball_c_train['pitch_type'].unique(),softball_c_test['pitch_type'].unique()])
)

print("Pitch types in baseball dataset:", baseball_pitch_types)
print("Pitch types in softball_c dataset:", softball_c_pitch_types)

# Find differences
difference_baseball_softball = set(baseball_pitch_types) - set(softball_c_pitch_types)
difference_softball_baseball = set(softball_c_pitch_types) - set(baseball_pitch_types)

print("\nPitch types present in baseball but not in softball_c:", difference_baseball_softball)
print("Pitch types present in softball_c but not in baseball:", difference_softball_baseball)

# prompt: remove the rows with pitch_type = 'SV' from baseball and pitch_type = 'F' from softball_c

baseball_train2 = baseball_c_train[(baseball_c_train['pitch_type'] != 'SV') &
(baseball_c_train['pitch_type'] != 'UN') & (baseball_c_train['pitch_type'] != 'CS') &
(baseball_c_train['pitch_type'] != 'FA')][baseball_c_train['pitch_type'].notna()]
baseball_val2 = baseball_c_val[(baseball_c_val['pitch_type'] != 'SV') &
(baseball_c_val['pitch_type'] != 'UN') & (baseball_c_val['pitch_type'] != 'CS') &
(baseball_c_val['pitch_type'] != 'FA')][baseball_c_val['pitch_type'].notna()]

baseball_test2 = baseball_c_test[(baseball_c_test['pitch_type'] != 'SV') &
(baseball_c_test['pitch_type'] != 'UN') & (baseball_c_test['pitch_type'] != 'CS') &
(baseball_c_test['pitch_type'] != 'FA')][baseball_c_test['pitch_type'].notna()]
```

```
print("Baseball dataframe after removing 'SV' pitch type:")
baseball_train2

pd.value_counts(softball_c_test['pitch_type'])

softball_c_test

n = int(len(softball_c_train) * 0.9)
softball_c_train1 = softball_c_train.iloc[:n].copy()
softball_c_val    = softball_c_train.iloc[n:].copy()

# prompt: separate the sub_fl as the target "y" for each of the train, test and validate data frames above. Then every other column becomes the features "X"

# Baseball
X_baseball_train =
baseball_train2.drop(['sub_fl','events','game_date','game_type','strikes','zone'], axis=1)
y_baseball_train = baseball_train2['sub_fl']

X_baseball_val = baseball_val2.drop(['sub_fl','events','game_date','game_type','strikes','zone'],
axis=1)
y_baseball_val = baseball_val2['sub_fl']

X_baseball_test = baseball_test2.drop(['sub_fl','events','game_date','game_type','strikes','zone'],
axis=1)
y_baseball_test = baseball_test2['sub_fl']

# Softball
X_softball_c_train =
softball_c_train.drop(['sub_fl','year','events','game_type2','strikes','zone','Pitcher'], axis=1)
y_softball_c_train = softball_c_train['sub_fl']

#X_softball_c_val =
softball_c_val.drop(['sub_fl','year','events','game_type2','strikes','zone','Pitcher'], axis=1)
#y_softball_c_val = softball_c_val['sub_fl']

X_softball_c_test = softball_c_test.drop(['sub_fl','year','events','strikes','zone'], axis=1)
y_softball_c_test = softball_c_test['sub_fl']

#X_softball_c_tv = pd.concat([X_softball_c_test,X_softball_c_val],axis=0)
#y_softball_c_tv = pd.concat([y_softball_c_test,y_softball_c_val],axis=0)

softball_c_test['sub_fl'].value_counts()

"""# **Preparing Data for Machine Learning**
```

```
We need to one-hot encode all categorical, and standardize continuous variables. Because
baseball counts are within a well defined, relatively similar range, I did not standardize some
variables.
"""

#Classes for transforming quantitative and qualitative variables, respectively in the
Preprocessing module
from sklearn.preprocessing import OneHotEncoder #, StandardScaler

#Module for create new transformed columns, needed for newly created indicators
from sklearn.compose import ColumnTransformer

#Identify categorical columns based on type, then create a list of (lists of categories for each
column)
categorical_columns=list(X_baseball_train.select_dtypes('object').columns)+list(X_baseball_trai
n.select_dtypes('category').columns)
all_categories = [X_baseball_train[col].unique().tolist() for col in categorical_columns]
#all_categories = [softball_c_pitch_types for col in categorical_columns]

X_baseball_train

#Identify categorical columns based on type, then create a list of (lists of categories for each
column)
categorical_columns_SB=list(X_softball_c_train.select_dtypes('object').columns)+list(X_softbal
l_c_train.select_dtypes('category').columns)
all_categories_SB = [X_softball_c_train[col].unique().tolist() for col in categorical_columns]

#categorical_columns_SB=list(X_baseball_train.select_dtypes('object').columns)+list(X_basebal
l_train.select_dtypes('category').columns)
#all_categories_SB = [X_baseball_train[col].unique().tolist() for col in categorical_columns]

#First identify and create a list of column names for each variable type in our training feature set
numeric_columns=list(X_baseball_train.select_dtypes('int64').columns)+list(X_baseball_train.se
lect_dtypes('float').columns)

numeric_columns

categorical_columns_SB

#Create a function that inputs our feature set and transforms variables (standardizes numerical,
and one hot encodes categorical) based on variable type
preprocessor=ColumnTransformer([
    ('num','passthrough',numeric_columns),
    ('cat',OneHotEncoder(handle_unknown='ignore',categories =
all_categories),categorical_columns)
```

```
])

preprocessor

pd.value_counts(X_baseball_test['pitch_type'])

# Transform the training dataframes. Note that the output is a numpy array
BB_Training_array = preprocessor.fit_transform(X_baseball_train)
BB_Validate_array = preprocessor.transform(X_baseball_val)
BB_Testing_array = preprocessor.transform(X_baseball_test)

SB_Training_array = preprocessor.fit_transform(X_softball_c_train)
#SB_Validate_array = preprocessor.transform(X_softball_c_val)
SB_Testing_array = preprocessor.transform(X_softball_c_test)
#SB_TV_array = preprocessor.transform(X_softball_c_tv)

# Get column names for one hot encoded columns
ohe_column_names = (preprocessor.named_transformers_['cat']
                    .get_feature_names_out(categorical_columns))

# Combine original numeric column names and one hot encoded column names
all_column_names = numeric_columns + ohe_column_names.tolist()

# Get column names for one hot encoded columns
ohe_column_names_SB = (preprocessor.named_transformers_['cat']
                    .get_feature_names_out(categorical_columns_SB))

# Combine original numeric column names and one hot encoded column names
all_column_names_SB = numeric_columns + ohe_column_names_SB.tolist()

numeric_columns

categorical_columns

all_column_names

all_column_names_SB

X_Train_p = pd.DataFrame(BB_Training_array, columns=all_column_names, index =
X_baseball_train.index)
X_Val_p = pd.DataFrame(BB_Validate_array, columns=all_column_names, index =
X_baseball_val.index)
X_Test_p = pd.DataFrame(BB_Testing_array, columns=all_column_names, index =
X_baseball_test.index)

#SB_Training_array = preprocessor.fit_transform(X_softball_c_train)
```

```
#SB_Validate_array = preprocessor.transform(X_softball_c_val)
#SB_Testing_array = preprocessor.transform(X_softball_c_test)

X_Train_p_W = pd.DataFrame(SB_Training_array, columns=all_column_names, index =
X_softball_c_train.index)
#X_Val_p_W = pd.DataFrame(SB_Validate_array, columns=all_column_names, index =
X_softball_c_val.index)
X_Test_p_W = pd.DataFrame(SB_Testing_array, columns=all_column_names, index =
X_softball_c_test.index)
#X_TV_p_W = pd.DataFrame(SB_TV_array, columns=all_column_names, index =
X_softball_c_tv.index)

# prompt: are there any differences between all_column_names and all_column_names_SB

print("all_column_names:", all_column_names)
print("all_column_names_SB:", all_column_names_SB)

if all_column_names == all_column_names_SB:
   print("all_column_names and all_column_names_SB are the same.")
else:
   print("all_column_names and all_column_names_SB are different.")

"""Optional: Add quadratic and interaction terms for better predictions. I did not include this for
fair comparison."""

from sklearn.preprocessing import PolynomialFeatures

def add_quadratic_and_interactions(df: pd.DataFrame, continuous=None):
   """
   Add all degree-2 polynomial terms (squares + interactions) for continuous variables,
   leaving 0/1 one-hot columns untouched. Returns an augmented copy of df.

   Parameters
   ----------
   df : pd.DataFrame
   continuous : list[str] or None
      Columns to treat as continuous. If None, inferred as numeric columns that are not 0/1
binaries.
   """
   out = df.copy()

   # Infer continuous cols if not provided: numeric & not strictly binary {0,1}
   if continuous is None:
      num = df.select_dtypes(include=[np.number])
      def is_binary_01(s: pd.Series) -> bool:
         vals = pd.unique(s.dropna())
```

```
            return len(vals) <= 2 and set(vals).issubset({0, 1})
        cont_cols = [c for c in num.columns if not is_binary_01(num[c])]
    else:
        cont_cols = list(continuous)

    if len(cont_cols) == 0:
        return out  # nothing to do

    pf = PolynomialFeatures(degree=2, include_bias=False)
    Xp = pf.fit_transform(out[cont_cols])
    names = pf.get_feature_names_out(cont_cols)  # e.g., ['x1', 'x2', 'x1^2', 'x1 x2']

    # Keep only NEW terms (exclude the original linear columns)
    keep_mask = [name not in cont_cols for name in names]
    new_cols = [n.replace(' ', ':') for n, k in zip(names, keep_mask) if k]  # 'x1 x2' -> 'x1:x2'
    new_vals = Xp[:, np.array(keep_mask)]

    new_df = pd.DataFrame(new_vals, columns=new_cols, index=out.index)
    return pd.concat([out, new_df], axis=1)

"""This code uses 2nd order terms, but..."""

X_Train_p2 = add_quadratic_and_interactions(X_Train_p)
X_Val_p2 = add_quadratic_and_interactions(X_Val_p)
X_Test_p2 = add_quadratic_and_interactions(X_Test_p)

X_Train_p_W2 = add_quadratic_and_interactions(X_Train_p_W)
#X_Val_p_W2 = add_quadratic_and_interactions(X_Val_p_W)
X_Test_p_W2 = add_quadratic_and_interactions(X_Test_p_W)
#X_TV_p_W2 = add_quadratic_and_interactions(X_TV_p_W)

"""...this code nullifies the 2nd order. Comment this block out if you want to use 2nd order."""

X_Train_p2 = X_Train_p
X_Val_p2 = X_Val_p
X_Test_p2 = X_Test_p

X_Train_p_W2 = X_Train_p_W
#X_Val_p_W2 = X_Val_p_W
X_Test_p_W2 = X_Test_p_W
#X_TV_p_W2 = X_TV_p_W

X_baseball_train.dtypes

"""# **Softball Models**
```

This section is for directly runing models on the softball data. There will be alot of commentinmg out, (i.e. ''' at beginning and end of cell) in this section because I chose to run only Domain Adaptation models for the sake of time.

Softball Logistic Regression Models

First balance the softball training set.
"""

'''
from imblearn.over_sampling import SMOTE

# Instantiate SMOTE
smote = SMOTE(random_state=42)

Apply SMOTE to the softball training data
X_Train_p_W_resampled, y_softball_c_train_resampled = smote.fit_resample(X_Train_p_W, y_softball_c_train)

print("Shape of resampled softball training features:", X_Train_p_W_resampled.shape)
print("Value counts of resampled softball training target:", y_softball_c_train_resampled.value_counts())
'''

'''
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score, f1_score

# Define the logistic regression model
log_reg_sb = LogisticRegression(random_state=42)

# Define the hyperparameters to tune
param_grid_sb = {
    'C': [0.00005, 0.005, 0.001, 0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'lbfgs']
}
'''

'''
# Define the k-fold cross-validation strategy
cv_strategy_sb = StratifiedKFold(n_splits=5, random_state=42, shuffle=True)
'''

"""## Perform grid search with cross-validation

Subtask:
Use GridSearchCV to find the best hyperparameters for the logistic regression model using cross-validation on the `softball_c` training data.

Reasoning:
Use GridSearchCV to find the best hyperparameters for the logistic regression model using cross-validation on the `softball_c` training data, and store the best parameters and best estimator.
"""

'''
# Instantiate GridSearchCV
grid_search_sb = GridSearchCV(estimator=log_reg_sb, param_grid=param_grid_sb, cv=cv_strategy_sb, scoring='precision')

# Fit GridSearchCV to the training data
grid_search_sb.fit(X_Train_p_W_resampled, y_softball_c_train_resampled)

# Store the best parameters and best estimator
best_params_sb = grid_search_sb.best_params_
best_estimator_sb = grid_search_sb.best_estimator_

print("Best Hyperparameters for Softball Data:", best_params_sb)
'''

"""## Train the model

Subtask:
Train the logistic regression model with the best hyperparameters on the entire `softball_c` training data.

Reasoning:
Train the logistic regression model with the best hyperparameters on the entire `softball_c` training data.
"""

'''
# Train the best estimator on the entire training data
best_estimator_sb.fit(X_Train_p_W_resampled, y_softball_c_train_resampled)
'''

"""## Calculate the positive class proportion threshold

Subtask:

Calculate the proportion of the positive class in the `softball_c` training data to be used as the threshold.

Reasoning:
Calculate the proportion of the positive class in the `softball_c` training data.
"""

```
threshold_sb = y_softball_c_train_resampled.mean()
print("Proportion of positive class in softball training data (threshold):", threshold_sb)
```

"""## Evaluate the model with threshold

Subtask:
Evaluate the trained model on the `softball_c` test data using the calculated threshold and a confusion matrix, along with other relevant metrics.

Reasoning:
Evaluate the trained model on the `softball_c` test data using the calculated threshold and a confusion matrix, along with other relevant metrics.
"""

```
# Predict probabilities on the test data
y_pred_proba_sb = best_estimator_sb.predict_proba(X_Test_p_W)[:, 1]

# Apply the calculated threshold to get predicted class labels
y_pred_thresholded_sb = (y_pred_proba_sb >= threshold_sb).astype(int)

# Calculate the confusion matrix
cm_sb = confusion_matrix(y_softball_c_test, y_pred_thresholded_sb)

# Calculate evaluation metrics
accuracy_sb_thresholded = accuracy_score(y_softball_c_test, y_pred_thresholded_sb)
precision_sb_thresholded = precision_score(y_softball_c_test, y_pred_thresholded_sb)
recall_sb_thresholded = recall_score(y_softball_c_test, y_pred_thresholded_sb)
f1_sb_thresholded = f1_score(y_softball_c_test, y_pred_thresholded_sb)

# Print the results
print("Confusion Matrix (Softball Data with Threshold):")
print(cm_sb)
print("\nAccuracy (Softball Data with Threshold):", accuracy_sb_thresholded)
print("Precision (Softball Data with Threshold):", precision_sb_thresholded)
print("Recall (Softball Data with Threshold):", recall_sb_thresholded)
print("F1-score (Softball Data with Threshold):", f1_sb_thresholded)
```

```
'''

"""## **Random Forest Models**

## Perform grid search with cross-validation for Softball

### Subtask:
Use GridSearchCV to find the best hyperparameters for the Random Forest model using cross-validation on the softball training data (`X_Train_p_W`).

**Reasoning**:
Use GridSearchCV to find the best hyperparameters for the Random Forest model using cross-validation on the softball training data (`X_Train_p_W`), and store the best parameters and best estimator.
"""

'''
from sklearn.ensemble import RandomForestClassifier

# Define the Random Forest model
rf_clf = RandomForestClassifier(random_state=42)

# Define the hyperparameters to tune
param_grid_rf = {
    'n_estimators': [100, 200, 500],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10]
}
'''

'''
from sklearn.model_selection import StratifiedKFold

cv_strategy_rf = StratifiedKFold(n_splits=5, random_state=42, shuffle=True)
'''

'''
# Instantiate GridSearchCV
grid_search_rf_sb = GridSearchCV(estimator=rf_clf, param_grid=param_grid_rf,
cv=cv_strategy_rf, scoring='f1')

# Fit GridSearchCV to the training data
grid_search_rf_sb.fit(X_Train_p_W_resampled, y_softball_c_train_resampled)

# Store the best parameters and best estimator
best_params_rf_sb = grid_search_rf_sb.best_params_
```

```
best_estimator_rf_sb = grid_search_rf_sb.best_estimator_

print("Best Hyperparameters for Softball Data (Random Forest):", best_params_rf_sb)
'''

"""## Train the Random Forest model for Softball

### Subtask:
Train the Random Forest model with the best hyperparameters on the entire softball training data (`X_Train_p_W`).

**Reasoning**:
Train the Random Forest model with the best hyperparameters on the entire softball training data (`X_Train_p_W`).
"""

'''# Train the best estimator on the entire training data
best_estimator_rf_sb.fit(X_Train_p_W_resampled, y_softball_c_train_resampled)
'''

"""## Calculate the positive class proportion threshold for Softball

### Subtask:
Calculate the proportion of the positive class in the softball training data (`y_softball_c_train`) to be used as the threshold.

**Reasoning**:
Calculate the proportion of the positive class in the softball training data.
"""

'''
threshold_rf_sb = y_softball_c_train_resampled.mean()
print("Proportion of positive class in softball training data (threshold):", threshold_rf_sb)
'''

"""## Evaluate the Random Forest model with threshold for Softball

### Subtask:
Evaluate the trained Random Forest model on the softball test data (`X_Test_p_W`) using the calculated threshold and a confusion matrix, along with other relevant metrics.

**Reasoning**:
Evaluate the trained Random Forest model on the softball test data (`X_Test_p_W`) using the calculated threshold and a confusion matrix, along with other relevant metrics.
"""
```

```
'''
# Predict probabilities on the test data
y_pred_proba_rf_sb = best_estimator_rf_sb.predict_proba(X_Test_p_W2)[:, 1]

# Apply the calculated threshold to get predicted class labels
y_pred_thresholded_rf_sb = (y_pred_proba_rf_sb >= threshold_rf_sb).astype(int)

# Calculate the confusion matrix
cm_rf_sb = confusion_matrix(y_softball_c_test, y_pred_thresholded_rf_sb)

# Calculate evaluation metrics
accuracy_rf_sb_thresholded = accuracy_score(y_softball_c_test, y_pred_thresholded_rf_sb)
precision_rf_sb_thresholded = precision_score(y_softball_c_test, y_pred_thresholded_rf_sb)
recall_rf_sb_thresholded = recall_score(y_softball_c_test, y_pred_thresholded_sb)
f1_rf_sb_thresholded = f1_score(y_softball_c_test, y_pred_thresholded_sb)

# Print the results
print("Confusion Matrix (Softball Data with Random Forest and Threshold):")
print(cm_rf_sb)
print("\nAccuracy (Softball Data with Random Forest and Threshold):",
accuracy_rf_sb_thresholded)
print("Precision (Softball Data with Random Forest and Threshold):",
precision_rf_sb_thresholded)
print("Recall (Softball Data with Random Forest and Threshold):", recall_rf_sb_thresholded)
print("F1-score (Softball Data with Random Forest and Threshold):", f1_rf_sb_thresholded)
'''

"""# **Domain Adaptation**

# Domain Adaptation EDA

Here we're trying to compare and contrast softball (Target Domainm = 1) and Baseball (Target
Domain = 0) in terms of descriptive statistics. Our steps:

First create a dataframe called both_Train that concatenates the X_Train_p_W dataframe with
the X_Train_p dataframe. Create a new variable 'Target_Domain' that is always equal to 1 for
the X_Train_p_W instances and equal to 0 for the X_Train_p instances
"""

# prompt: create a dataframe called both_Train that concatenates the X_Train_p_W dataframe
with the X_Train_p dataframe. Create a new variable 'Target_Domain' that is always equal to 1
for the X_Train_p_W instances and equal to 0 for the X_Train_p instances

import pandas as pd
# Concatenate the dataframes and add the 'Target_Domain' column
both_Train = pd.concat([
```

```
    X_Train_p_W.assign(Target_Domain=1),  # Assign 1 for X_Train_p_W instances
    X_Train_p.assign(Target_Domain=0)     # Assign 0 for X_Train_p instances
], ignore_index=True) # ignore_index=True resets the index of the combined DataFrame

print(both_Train.head())
print(both_Train['Target_Domain'].value_counts())

"""Next visually and numerically compare the Target_Domain = 0 versus Target_Domain = 1
groups for each variable in the combined training data.

"""

# prompt: visually and numerically compare the Target_Domain = 0 versus Target_Domain = 1
groups for each variable in both_Train_p

import plotly.express as px
import pandas as pd
# Separate data for Target_Domain = 0 and Target_Domain = 1
domain_0 = both_Train[both_Train['Target_Domain'] == 0].drop('Target_Domain', axis=1)
domain_1 = both_Train[both_Train['Target_Domain'] == 1].drop('Target_Domain', axis=1)

# Numerical Comparison
print("Numerical Comparison (Mean and Standard Deviation):")
for col in domain_0.columns:
  if domain_0[col].dtype in ['int64', 'float64']:
    print(f"\nVariable: {col}")
    print(f"  Target_Domain = 0: Mean = {domain_0[col].mean():.4f}, Std Dev =
{domain_0[col].std():.4f}")
    print(f"  Target_Domain = 1: Mean = {domain_1[col].mean():.4f}, Std Dev =
{domain_1[col].std():.4f}")
  else:
    print(f"\nVariable: {col} (Categorical)")
    print(f"  Target_Domain = 0:\n{domain_0[col].value_counts(normalize=True)}")
    print(f"  Target_Domain = 1:\n{domain_1[col].value_counts(normalize=True)}")

# Visual Comparison (for a few selected variables - adjust as needed)
# You might want to select a subset of variables or iterate through them
# For numerical variables, histograms or box plots are useful.
# For categorical variables, bar plots are useful.

print("\nVisual Comparison:")
# Example for a few numerical columns
numerical_cols_to_plot = ['stand', 'vx0', 'vy0', 'vz0', 'ax', 'ay', 'az']
for col in numerical_cols_to_plot:
  if col in both_Train.columns and both_Train[col].dtype in ['int64', 'float64']:
    fig = px.histogram(both_Train, x=col, color='Target_Domain', marginal="box",
```

```
                title=f'Distribution of {col} by Target Domain')
    fig.show()

# Example for a few categorical columns (from the OHE columns)
categorical_cols_to_plot = ['pitch_type_CH', 'pitch_type_CU', 'pitch_type_FF', 'pitch_type_SL']
for col in categorical_cols_to_plot:
    if col in both_Train.columns:
        # Need to sum the counts for 0 and 1 since they are binary
        domain_0_count = domain_0[col].sum()
        domain_1_count = domain_1[col].sum()

        total_domain_0 = len(domain_0)
        total_domain_1 = len(domain_1)

        data_for_plot = pd.DataFrame({
            'Category': [col] * 2,
            'Count': [domain_0_count, domain_1_count],
            'Proportion': [domain_0_count / total_domain_0 if total_domain_0 > 0 else 0,
                           domain_1_count / total_domain_1 if total_domain_1 > 0 else 0],
            'Target_Domain': [0, 1]
        })

        fig = px.bar(data_for_plot, x='Target_Domain', y='Proportion',
                     title=f'Proportion of {col} by Target Domain',
                     labels={'Target_Domain': 'Target Domain', 'Proportion': 'Proportion'})
        fig.show()

"""# Domain Adaptation

We tried the LR_best model as well as the RF_best models from above (both trained and tuned on the baseball data). The LR_best model does a poor job with precision, regardless how we tune domain adaptation hyperparameters. On the otherhand, the RF_best model did great on the baseball data and performed equally well in the domain adapt setting on the softball data. It outperformed the best random forest model tuned on the softball training data.

## Which Domain Adaptation Technique Should be Used?

To determine the appropriate technique for a given scenario, we will use the approach described by Spitzer et. al in [Towards a Problem-Oriented Domain Adaptation Framework for Machine Learning](https://arxiv.org/pdf/2501.04528). Let $x$ represent the features (explanatory variables, and $y$ the target (response). Then choice of method is based on the following:

**1. Causality**

Is the cause - effect relationship $x→y$ or $y\rightarrow x$?
```

In other words, do the explanatory variables determine whether there is a strike, or does the occurence of a strike lead to the observation of the explanatory variables?

2. What remains the same from Source to Target

A. Is it reasonable to assume that $P_S(y) = P_T(y)$? In other words, is the distribution of strikes the same in baseball and softball?

B. Is it reasonable to assume that $P_S(x) = P_T(x)$? In other words, is the distribution of explanatory variables the same in baseball and softball?

C. Is it reasonable to assume that $P_S(y|x) = P_T(y|x)$? In other words, from a distributional perspective, are strikes related to the explanatory variables in the same way in baseball and softball (e.g. if there is one runner on base with 2 balls and one strike, is the likelihood of a strike the same for both sports?

3. Nature of the shift

Based on answer to 2. Which one(s) of A, B or C is different? In other words identify the marginal or conditional distributions that are different between softball and baseball.
"""

```
!pip install adapt

from adapt.instance_based import NearestNeighborsWeighting
from adapt.instance_based import BalancedWeighting
```

"""# Cluster First

I wanted to use source baseball data that was in some sense "close" to the softball data. To visualize them together, I ran pca on the softball data so that I could reduce it to 2-dimensions (pca1 and pca2).
First, out of curiosity, I wanted to compare density plots for pca1:
"""

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from scipy.stats import gaussian_kde

# --- Replace with your actual data ---
# X_Train_p2: baseball training features (DataFrame)
# X_softball_c_train: softball training features (DataFrame)
# Both should have the same feature columns
```

```
# 1. Combine and standardize
#scaler = StandardScaler()
#X_all = pd.concat([X_Train_p2, X_softball_c_train], axis=0)
#X_all_scaled = scaler.fit_transform(X_all)

# --- STEP 2: PCA on baseball data only ---
pca = PCA(n_components=2)
X_softball_pca = pca.fit_transform(X_Train_p_W2)

# --- STEP 3: Project softball data using baseball PCA model ---
X_baseball_pca = pca.transform(X_Train_p2)


# 3. PCA on baseball
'''pca = PCA(n_components=2)
X_baseball_pca = pca.fit_transform(X_Train_p2)
X_softball_pca = pca.transform(X_Train_p_W2)'''

# 4. Extract PC1
baseball_pc1 = X_baseball_pca[:, 1]
softball_pc1 = X_softball_pca[:, 1]

# 5. Kernel Density Estimation and normalization
x_grid = np.linspace(-40, max(baseball_pc1.max(), softball_pc1.max()), 1000)

kde_baseball = gaussian_kde(baseball_pc1)
kde_softball = gaussian_kde(softball_pc1)

density_baseball = kde_baseball(x_grid)
density_softball = kde_softball(x_grid)

# Normalize to integrate to 1
density_baseball /= np.trapz(density_baseball, x_grid)
density_softball /= np.trapz(density_softball, x_grid)

# 6. Plot
plt.figure(figsize=(10, 6))
plt.plot(x_grid, density_baseball, label="Baseball", alpha=0.7)
plt.plot(x_grid, density_softball, label="Softball", alpha=0.7)
plt.title("Normalized KDE of PC1: Baseball vs. Softball")
plt.xlabel("First Principal Component (PC1)")
plt.ylabel("Density")
plt.xlim(left=-40)
plt.grid(True)
plt.legend()
```

```
plt.tight_layout()
plt.show()

"""Next I used those same pca thansformations on the baseball data and created a scatterplot of both on the same graph.

Finally, I selected the 70% of baseball observations that were closest to the "center" of the softball pca bivariate distribution. You can see that there is alot of "noise" in the baseball data, so this was my way of filtering only the most relevant baseball observations to include as the source domain data.
"""

import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from scipy.stats import gaussian_kde
import plotly.express as px
import plotly.graph_objects as go

# ------------------------------------------------------------
# Inputs:
# - X_Train_p2:  baseball training features (DataFrame, chronologically prepped)
# - X_Train_p_W2: softball training features (DataFrame), same columns/order as X_Train_p2
# - coverage: desired probability mass for the softball HDR (e.g., 0.90 for 90%)
# ------------------------------------------------------------
coverage = 0.7  # <- user-desired region of maximum probability

# 1) PCA on baseball only; then project softball
pca = PCA(n_components=2, random_state=42)
X_baseball_pca = pca.fit_transform(X_Train_p2)
X_softball_pca = pca.transform(X_Train_p_W2)

# 2) Build scatter DataFrame (for Plotly)
df_plot = pd.DataFrame({
    "PC1": np.concatenate([X_baseball_pca[:, 0], X_softball_pca[:, 0]]),
    "PC2": np.concatenate([X_baseball_pca[:, 1], X_softball_pca[:, 1]]),
    "Domain": (["Baseball"] * len(X_baseball_pca)) + (["Softball"] * len(X_softball_pca)),
})

# 3) Softball 2D KDE on (PC1, PC2)
xs, ys = X_softball_pca[:, 0], X_softball_pca[:, 1]
kde2d = gaussian_kde(np.vstack([xs, ys]))

# ---- Compute an HDR threshold 'lam' so that area where density >= lam contains 'coverage' mass
# Make a grid in the softball PCA space (clip to central quantiles for stability)
```

```
x_lo, x_hi = np.percentile(xs, [1, 99])
y_lo, y_hi = np.percentile(ys, [1, 99])
# pad a bit
pad_x = 0.1 * (x_hi - x_lo) if x_hi > x_lo else 1.0
pad_y = 0.1 * (y_hi - y_lo) if y_hi > y_lo else 1.0
x_grid = np.linspace(x_lo - pad_x, x_hi + pad_x, 200)
y_grid = np.linspace(y_lo - pad_y, y_hi + pad_y, 200)
Xg, Yg = np.meshgrid(x_grid, y_grid)
Z = kde2d(np.vstack([Xg.ravel(), Yg.ravel()])).reshape(Xg.shape)

dx = x_grid[1] - x_grid[0]
dy = y_grid[1] - y_grid[0]
Z_flat = Z.ravel()
order = np.argsort(Z_flat)[::-1]
cdf = np.cumsum(Z_flat[order]) * dx * dy
idx = np.searchsorted(cdf, coverage)
lam = Z_flat[order[idx]] if idx < len(Z_flat) else Z_flat[order[-1]]

# 4) Baseball points inside HDR (by definition: density >= lam under softball KDE)
xb, yb = X_baseball_pca[:, 0], X_baseball_pca[:, 1]
dens_baseball = kde2d(np.vstack([xb, yb]))
inside_hdr_mask = dens_baseball >= lam

# Produce DataFrame of the original baseball training rows inside HDR
if isinstance(X_Train_p2, pd.DataFrame):
    baseball_in_hdr = X_Train_p2.loc[inside_hdr_mask].copy()
else:
    # If X_Train_p2 is a NumPy array, wrap it in a DataFrame temporarily
    baseball_in_hdr = pd.DataFrame(X_Train_p2[inside_hdr_mask])

# Add helpful diagnostics (optional)
baseball_in_hdr["PC1"] = xb[inside_hdr_mask]
baseball_in_hdr["PC2"] = yb[inside_hdr_mask]
baseball_in_hdr["softball_kde"] = dens_baseball[inside_hdr_mask]

print(f"Baseball rows inside the {int(coverage*100)}% HDR: {inside_hdr_mask.sum()} /
{len(inside_hdr_mask)}")

# 5) Plotly scatter + overlay the softball density contours (filled from lam upward)
fig = px.scatter(
    df_plot, x="PC1", y="PC2", color="Domain",
    title=f"PCA Scatter with Softball KDE HDR (≥ {int(coverage*100)}% mass)",
    labels={"PC1": "Principal Component 1", "PC2": "Principal Component 2"},
    opacity=0.7, template="plotly_white"
)
```

```
# Add filled contour for softball density (region >= lam)
fig.add_trace(go.Contour(
    x=x_grid,
    y=y_grid,
    z=Z,                        # Plotly expects z[y, x]-like 2D array; our shape matches (len(y), len(x))
    contours=dict(
        start=lam, end=float(Z.max()), size=(float(Z.max()) - float(lam)) / 5,
        coloring="heatmap", showlabels=False
    ),
    showscale=False,
    opacity=0.45,
    name=f"Softball KDE ≥ HDR threshold"
))

# Optional: add a thin boundary line exactly at lam (the HDR boundary)
fig.add_trace(go.Contour(
    x=x_grid, y=y_grid, z=Z,
    contours=dict(start=float(lam), end=float(lam), size=1e-9, coloring="lines",
showlabels=False),
    line=dict(color="black", width=2),
    showscale=False,
    name=f"{int(coverage*100)}% HDR boundary"
))

fig.update_layout(legend=dict(itemsizing="constant"))
fig.show()

# 'baseball_in_hdr' now holds the baseball training rows that fall within the softball HDR.
# You can proceed to use 'baseball_in_hdr' downstream.

X_baseball_filtered = baseball_in_hdr.iloc[:,:-3]

y_baseball_filtered = y_baseball_train[baseball_in_hdr.index]

# --- Step 3: Filter based on PC1 < 0 and PC2 < 100 ---
#mask = (X_baseball_pca[:, 0] < 0) & (X_baseball_pca[:, 1] < 100)

# --- Step 4: Get the original rows that satisfy the mask ---
#X_baseball_filtered = X_Train_p2[mask]
#y_baseball_filtered = y_baseball_train[mask]
# --- Step 5: View or save result ---
#X_baseball_filtered  # or use .to_csv(...) to save

"""### Creating Baseball Models on the Reduced Data
```

Classifiers that are built on the newly filtered baseball train data are tested on teh baseball test data. A model that generalizes well with strong precision and recall is a good candidate for domain adaptation. Let's try a few.
"""

'''
```
from xgboost import XGBClassifier
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import (confusion_matrix, accuracy_score, precision_score,
                             recall_score, f1_score, ConfusionMatrixDisplay)

# Define the XGBoost model
xgb_clf = XGBClassifier(random_state=42, eval_metric='logloss')

# Define the hyperparameters to tune
param_grid_xgb = {
    'n_estimators': [100, 200, 300],
    'max_depth': [3, 5, 7],
    'learning_rate': [0.01, 0.1, 0.2],
    'subsample': [0.8, 1.0],
    'colsample_bytree': [0.8, 1.0]
}

# Define the k-fold cross-validation strategy
cv_strategy_xgb = StratifiedKFold(n_splits=5, random_state=42, shuffle=True)

# Instantiate GridSearchCV
grid_search_xgb = GridSearchCV(estimator=xgb_clf, param_grid=param_grid_xgb,
cv=cv_strategy_xgb, scoring='f1')

# Fit GridSearchCV to the training data
grid_search_xgb.fit(X_baseball_filtered, y_baseball_filtered)

# Store the best parameters and best estimator
best_params_xgb = grid_search_xgb.best_params_
best_estimator_xgb = grid_search_xgb.best_estimator_

print("Best Hyperparameters for Baseball Data (XGBoost):", best_params_xgb)

# Train the best estimator on the entire training data
best_estimator_xgb.fit(X_baseball_filtered, y_baseball_filtered)

# Predict on the train data
y_pred_xgb_train = best_estimator_xgb.predict(X_baseball_filtered)

# Calculate the confusion matrix
```

```
cm_xgb_train = confusion_matrix(y_baseball_filtered, y_pred_xgb_train, labels=None,
sample_weight=None, normalize=None)

# Calculate evaluation metrics
accuracy_xgb_train = accuracy_score(y_baseball_filtered, y_pred_xgb_train)
precision_xgb_train = precision_score(y_baseball_filtered, y_pred_xgb_train)
recall_xgb_train= recall_score(y_baseball_filtered, y_pred_xgb_train)
f1_xgb_train = f1_score(y_baseball_filtered, y_pred_xgb_train)

# Print the results
print("\nConfusion Matrix - Train (Baseball Data with XGBoost):")
disp = ConfusionMatrixDisplay(cm_xgb_train, display_labels=None)
disp.plot()

print("\nEvaluation Metrics (Baseball Data with XGBoost):")
print(f"Accuracy: {accuracy_xgb_train:.4f}")
print(f"Precision: {precision_xgb_train:.4f}")
print(f"Recall: {recall_xgb_train:.4f}")
print(f"F1-score: {f1_xgb_train:.4f}")

print("_____")

# Predict on the test data
y_pred_xgb = best_estimator_xgb.predict(X_Test_p2)

# Calculate the confusion matrix
cm_xgb = confusion_matrix(y_baseball_test, y_pred_xgb, labels=None, sample_weight=None,
normalize=None)

# Calculate evaluation metrics
accuracy_xgb = accuracy_score(y_baseball_test, y_pred_xgb)
precision_xgb = precision_score(y_baseball_test, y_pred_xgb)
recall_xgb = recall_score(y_baseball_test, y_pred_xgb)
f1_xgb = f1_score(y_baseball_test, y_pred_xgb)

# Print the results
print("\nConfusion Matrix - Test (Baseball Data with XGBoost):")
disp = ConfusionMatrixDisplay(cm_xgb, display_labels=None)
disp.plot()

print("\nEvaluation Metrics (Baseball Data with XGBoost):")
print(f"Accuracy: {accuracy_xgb:.4f}")
print(f"Precision: {precision_xgb:.4f}")
print(f"Recall: {recall_xgb:.4f}")
print(f"F1-score: {f1_xgb:.4f}")
'''
```

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import (confusion_matrix, accuracy_score, precision_score,
                             recall_score, f1_score, ConfusionMatrixDisplay)

# --- Step 1: Define the Logistic Regression model ---
logreg_clf = LogisticRegression(solver='liblinear', random_state=42, class_weight='balanced')

# --- Step 2: Define hyperparameter grid ---
param_grid_logreg = {
    'penalty': ['l1', 'l2'],
    'C': [0.01, 0.1, 1, 10, 100]  # Inverse regularization strength
}

# --- Step 3: Define cross-validation strategy ---
cv_strategy_logreg = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# --- Step 4: Instantiate GridSearchCV ---
grid_search_logreg = GridSearchCV(
    estimator=logreg_clf,
    param_grid=param_grid_logreg,
    cv=cv_strategy_logreg,
    scoring='f1',
    n_jobs=-1
)

# --- Step 5: Fit to training data ---
grid_search_logreg.fit(X_baseball_filtered, y_baseball_filtered)

# --- Step 6: Store best estimator and params ---
best_params_logreg = grid_search_logreg.best_params_
best_estimator_logreg = grid_search_logreg.best_estimator_

print("Best Hyperparameters for Baseball Data (Logistic Regression):", best_params_logreg)

# --- Step 7: Train best estimator on full training data ---
best_estimator_logreg.fit(X_baseball_filtered, y_baseball_filtered)

# --- Step 8: Evaluate on training data ---
y_pred_train = best_estimator_logreg.predict(X_baseball_filtered)
cm_train = confusion_matrix(y_baseball_filtered, y_pred_train)

accuracy_train = accuracy_score(y_baseball_filtered, y_pred_train)
```

```
precision_train = precision_score(y_baseball_filtered, y_pred_train)
recall_train = recall_score(y_baseball_filtered, y_pred_train)
f1_train = f1_score(y_baseball_filtered, y_pred_train)

print("\nConfusion Matrix - Train (Baseball Data with Logistic Regression):")
ConfusionMatrixDisplay(cm_train).plot()

print("\nEvaluation Metrics (Baseball Data with Logistic Regression):")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-score: {f1_train:.4f}")

print("_____")

# --- Step 9: Predict on test data ---
y_pred_test = best_estimator_logreg.predict(X_Test_p2)
cm_test = confusion_matrix(y_baseball_test, y_pred_test)

accuracy_test = accuracy_score(y_baseball_test, y_pred_test)
precision_test = precision_score(y_baseball_test, y_pred_test)
recall_test = recall_score(y_baseball_test, y_pred_test)
f1_test = f1_score(y_baseball_test, y_pred_test)

print("\nConfusion Matrix - Test (Baseball Data with Logistic Regression):")
ConfusionMatrixDisplay(cm_test).plot()

print("\nEvaluation Metrics (Baseball Data with Logistic Regression):")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-score: {f1_test:.4f}")

set(X_Train_p_W2.columns) - set(X_baseball_filtered.columns)

from sklearn.metrics import (confusion_matrix, accuracy_score, precision_score,
                             recall_score, f1_score, ConfusionMatrixDisplay)

# Define the Naive Bayes model
lr_clf = LogisticRegression()


#print("Best Hyperparameters for Baseball Data (Naive Bayes):", best_params_nb)

# Train the best estimator on the entire training data
lr_clf.fit(X_baseball_filtered, y_baseball_filtered)
```

```
# Predict on the train data
y_pred_nb_train = lr_clf.predict(X_baseball_filtered)

# Calculate the confusion matrix
cm_nb_train = confusion_matrix(y_baseball_filtered, y_pred_nb_train)

# Calculate evaluation metrics
accuracy_nb_train = accuracy_score(y_baseball_filtered, y_pred_nb_train)
precision_nb_train = precision_score(y_baseball_filtered, y_pred_nb_train)
recall_nb_train = recall_score(y_baseball_filtered, y_pred_nb_train)
f1_nb_train = f1_score(y_baseball_filtered, y_pred_nb_train)

# Print the results
print("\nConfusion Matrix - Train (Baseball Data with Naive Bayes):")
disp = ConfusionMatrixDisplay(cm_nb_train)
disp.plot()

print("\nEvaluation Metrics (Baseball Data with Naive Bayes):")
print(f"Accuracy: {accuracy_nb_train:.4f}")
print(f"Precision: {precision_nb_train:.4f}")
print(f"Recall: {recall_nb_train:.4f}")
print(f"F1-score: {f1_nb_train:.4f}")

print("_____")

# Predict on the test data
y_pred_nb = lr_clf.predict(X_Test_p2)

# Calculate the confusion matrix
cm_nb = confusion_matrix(y_baseball_test, y_pred_nb)

# Calculate evaluation metrics
accuracy_nb = accuracy_score(y_baseball_test, y_pred_nb)
precision_nb = precision_score(y_baseball_test, y_pred_nb)
recall_nb = recall_score(y_baseball_test, y_pred_nb)
f1_nb = f1_score(y_baseball_test, y_pred_nb)

# Print the results
print("\nConfusion Matrix - Test (Baseball Data with Naive Bayes):")
disp = ConfusionMatrixDisplay(cm_nb)
disp.plot()

print("\nEvaluation Metrics (Baseball Data with Naive Bayes):")
print(f"Accuracy: {accuracy_nb:.4f}")
print(f"Precision: {precision_nb:.4f}")
```

```
print(f"Recall: {recall_nb:.4f}")
print(f"F1-score: {f1_nb:.4f}")

"""## Logistic Baseball Model with Statistical Inference"""

X_baseball_filtered[X_baseball_filtered.columns] =
X_baseball_filtered[X_baseball_filtered.columns].apply(
   pd.to_numeric, errors='coerce'
).astype('float64')


X_Test_p22 = X_Test_p2[X_Test_p2.columns].apply(
   pd.to_numeric, errors='coerce'
).astype('float64')

X_baseball_filtered.dtypes

import statsmodels.api as sm

# Add a constant to get an intercept
X_build_sm = sm.add_constant(X_baseball_filtered)
X_test_sm = sm.add_constant(X_Test_p22)
y_train = pd.to_numeric(y_baseball_filtered)
# Fit the logistic regression using ‘GLM’
lr_build_full = sm.GLM(y_train, X_build_sm, family=sm.families.Binomial()).fit()
predictions_lr_train = lr_build_full.predict(X_build_sm)
predictions_lr_test = lr_build_full.predict(X_test_sm)

print(lr_build_full.summary())

# Predict on the train data
prob_pred_nb = lr_build_full.predict(X_build_sm)
y_pred_nb = list(map(round,prob_pred_nb))

# Calculate the confusion matrix
cm_nb = confusion_matrix(y_baseball_filtered, y_pred_nb)

# Calculate evaluation metrics
accuracy_nb = accuracy_score(y_baseball_filtered, y_pred_nb)
precision_nb = precision_score(y_baseball_filtered, y_pred_nb)
recall_nb = recall_score(y_baseball_filtered, y_pred_nb)
f1_nb = f1_score(y_baseball_filtered, y_pred_nb)

# Print the results
print("\nConfusion Matrix - Train (Baseball Data with Naive Bayes):")
disp = ConfusionMatrixDisplay(cm_nb)
```

```
disp.plot()

print("\nEvaluation Metrics (Baseball Data with Naive Bayes):")
print(f"Accuracy: {accuracy_nb:.4f}")
print(f"Precision: {precision_nb:.4f}")
print(f"Recall: {recall_nb:.4f}")
print(f"F1-score: {f1_nb:.4f}")


# Predict on the test data
prob_pred_nb = lr_build_full.predict(X_test_sm)
y_pred_nb = list(map(round,prob_pred_nb))

# Calculate the confusion matrix
cm_nb = confusion_matrix(y_baseball_test, y_pred_nb)

# Calculate evaluation metrics
accuracy_nb = accuracy_score(y_baseball_test, y_pred_nb)
precision_nb = precision_score(y_baseball_test, y_pred_nb)
recall_nb = recall_score(y_baseball_test, y_pred_nb)
f1_nb = f1_score(y_baseball_test, y_pred_nb)

# Print the results
print("\nConfusion Matrix - Test (Baseball Data with Naive Bayes):")
disp = ConfusionMatrixDisplay(cm_nb)
disp.plot()

print("\nEvaluation Metrics (Baseball Data with Naive Bayes):")
print(f"Accuracy: {accuracy_nb:.4f}")
print(f"Precision: {precision_nb:.4f}")
print(f"Recall: {recall_nb:.4f}")
print(f"F1-score: {f1_nb:.4f}")

"""Let's use the grid search process to identify the best value of the *gamma* parameter.

The following code creates a grid of *gamma* values and estimators and reports the best combination, along with all relevant metrics.

## **Domain Adaptation Results**
"""

import numpy as np
import pandas as pd
from sklearn.base import clone
from sklearn.metrics import (
    f1_score, precision_score, recall_score, accuracy_score,
```

```
    confusion_matrix, ConfusionMatrixDisplay, roc_auc_score,
    precision_recall_curve, average_precision_score
)
from adapt.instance_based import BalancedWeighting
import matplotlib.pyplot as plt
import warnings

def bw_grid_f1_table(
    est_list, gamma_list,
    Xs, ys,              # SOURCE (e.g., baseball filtered)
    Xt_weight, yt_weight, # TARGET used by BW to compute weights (e.g., softball train)
    Xt_eval, yt_eval,     # EVAL set for scoring (e.g., TV softball)
    random_state=41,
    verbose=0
):
    """
    Returns:
      table_f1  : DataFrame pivot (rows=estimator, cols=gamma) of F1 on Xt_eval
      results   : long-form DataFrame with metrics for each (estimator, gamma)
      best_info : dict with best config and metrics (adds 'roc_auc' and 'pr_auc')
      best_bw   : fitted BalancedWeighting model for the best config

    Side effects:
      - Prints confusion matrix + metrics for best-by-F1 model
      - Plots Precision–Recall curve and prints PR-AUC for the best model
    """
    rows = []
    fitted_models = {}  # (name, gamma) -> model (optional reuse)

    def est_name(e, idx):
        base = e.__class__.__name__
        return f"{base}#{idx}"

    # ---------- Grid sweep ----------
    for i, base_est in enumerate(est_list):
        name_i = est_name(base_est, i)
        for gamma in gamma_list:
            try:
                bw = BalancedWeighting(
                    estimator=clone(base_est),
                    Xt=Xt_weight, yt=yt_weight,
                    gamma=gamma,
                    verbose=verbose,
                    copy=True,
                    random_state=random_state
                )
```

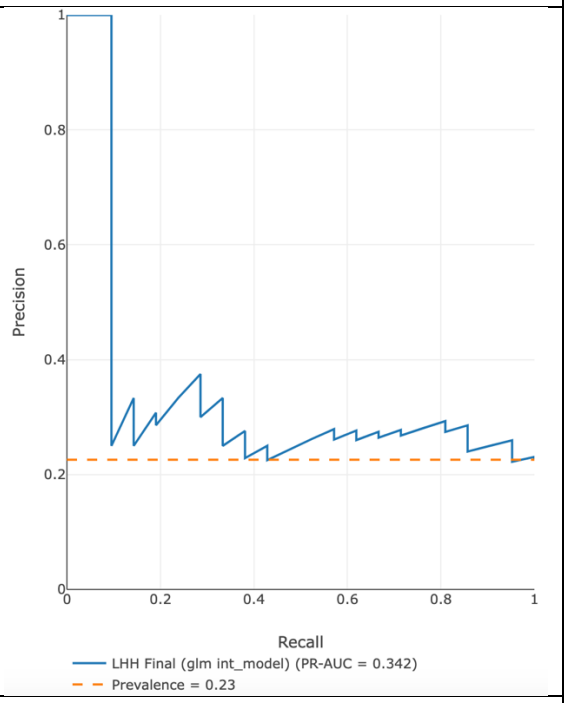
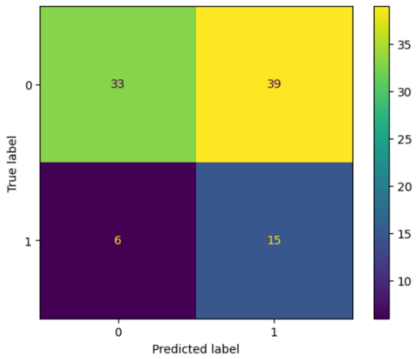
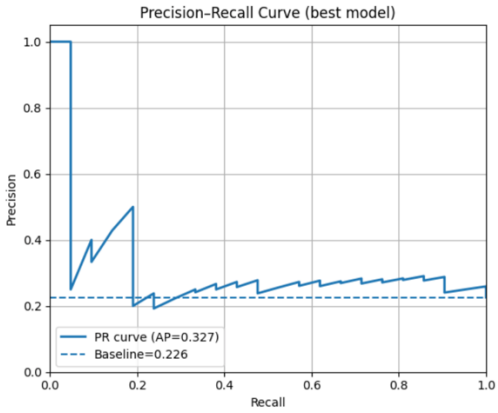
```
            with warnings.catch_warnings():
                warnings.simplefilter("ignore")
                bw.fit(Xs, ys)

            y_pred = bw.predict(Xt_eval)
            cm = confusion_matrix(yt_eval, y_pred)
            acc = accuracy_score(yt_eval, y_pred)
            pre = precision_score(yt_eval, y_pred, zero_division=0)
            rec = recall_score(yt_eval, y_pred, zero_division=0)
            f1  = f1_score(yt_eval, y_pred, zero_division=0)
            tn, fp, fn, tp = cm.ravel()
            spec = tn / (tn + fp) if (tn + fp) > 0 else 0.0

            rows.append({
                "estimator": name_i,
                "gamma": gamma,
                "f1": f1,
                "precision": pre,
                "recall": rec,
                "specificity": spec,
                "accuracy": acc
            })
            fitted_models[(name_i, gamma)] = bw

        except Exception as e:
            rows.append({
                "estimator": name_i, "gamma": gamma,
                "f1": np.nan, "precision": np.nan, "recall": np.nan,
                "specificity": np.nan, "accuracy": np.nan,
                "error": str(e)
            })

results = pd.DataFrame(rows)
table_f1 = results.pivot(index="estimator", columns="gamma", values="f1").sort_index()

# ---------- Pick best by F1 ----------
idx_best = results["f1"].idxmax()
best_row = results.loc[idx_best].to_dict()
best_name = best_row["estimator"]
best_gamma = best_row["gamma"]

bw_best = fitted_models.get((best_name, best_gamma))
if bw_best is None:
    base_est = est_list[int(best_name.split("#")[-1])]
    bw_best = BalancedWeighting(
        estimator=clone(base_est),
```

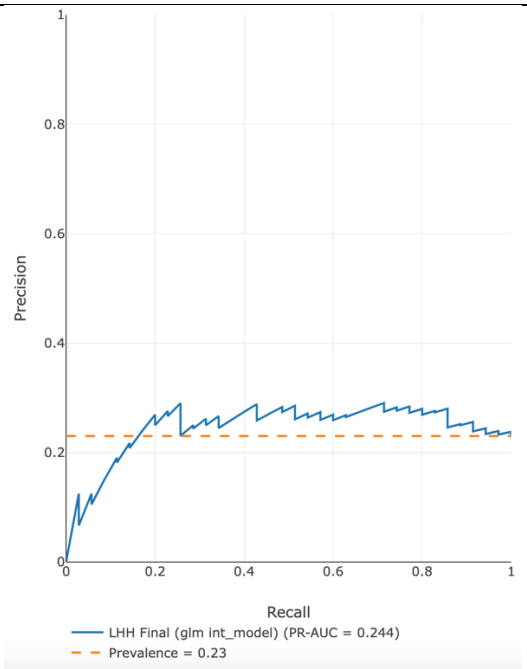
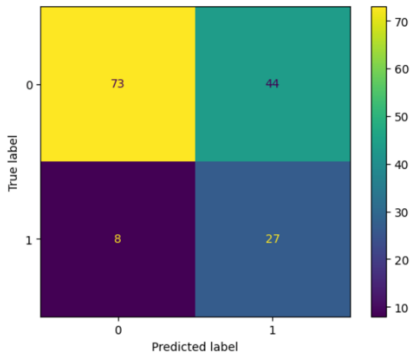
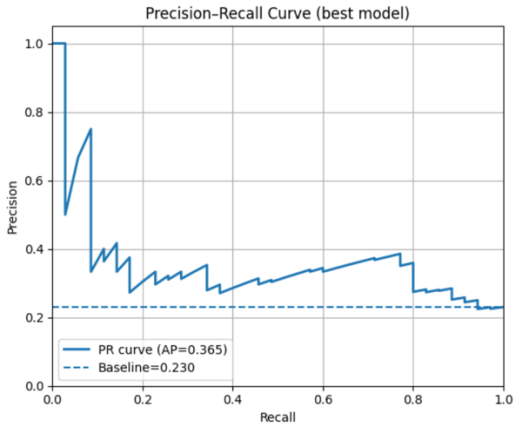
```
        Xt=Xt_weight, yt=yt_weight,
        gamma=best_gamma,
        verbose=verbose,
        copy=True,
        random_state=random_state
    )
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        bw_best.fit(Xs, ys)

# ---------- Final metrics on Xt_eval ----------
y_pred = bw_best.predict(Xt_eval)
cm = confusion_matrix(yt_eval, y_pred)
acc = accuracy_score(yt_eval, y_pred)
pre = precision_score(yt_eval, y_pred, zero_division=0)
rec = recall_score(yt_eval, y_pred, zero_division=0)
f1  = f1_score(yt_eval, y_pred, zero_division=0)
tn, fp, fn, tp = cm.ravel()
spec = tn / (tn + fp) if (tn + fp) > 0 else 0.0

print(f"\nBest config by F1: estimator={best_name}, gamma={best_gamma}")
print("\nConfusion Matrix (best):")
ConfusionMatrixDisplay(cm).plot()
print("\nEvaluation Metrics (best):")
print(f"Accuracy:  {acc:.4f}")
print(f"Precision: {pre:.4f}")
print(f"Recall:    {rec:.4f}")
print(f"Specificity: {spec:.4f}")
print(f"F1-score:  {f1:.4f}")

# ---------- Scores for ROC AUC + PR curve ----------
roc_auc = None
pr_auc = None
try:
    est_fitted = getattr(bw_best, "estimator_", None)
    if est_fitted is None:
        raise RuntimeError("BalancedWeighting not fitted.")
    if hasattr(est_fitted, "predict_proba"):
        scores = est_fitted.predict_proba(Xt_eval)[:, 1]
    elif hasattr(est_fitted, "decision_function"):
        scores = est_fitted.decision_function(Xt_eval)
    else:
        scores = None

    if scores is not None:
        # ROC AUC
```

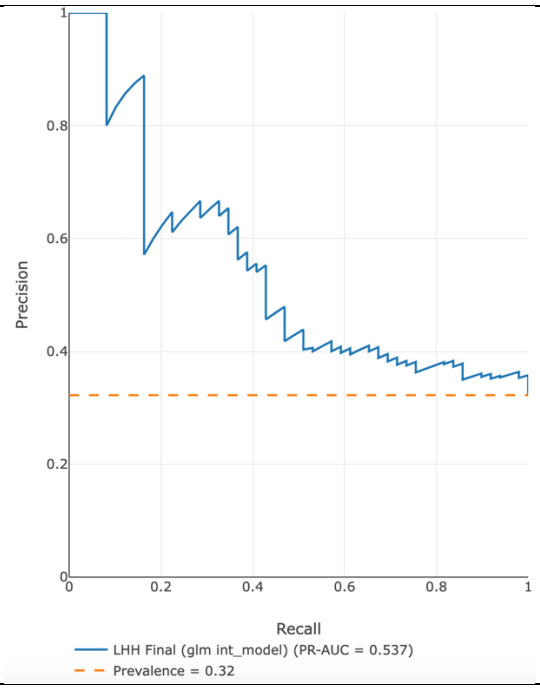
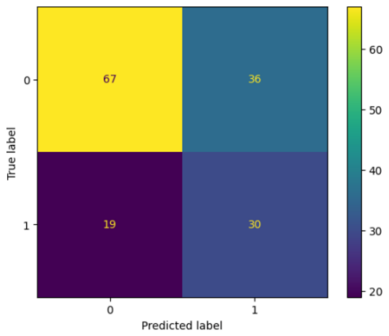
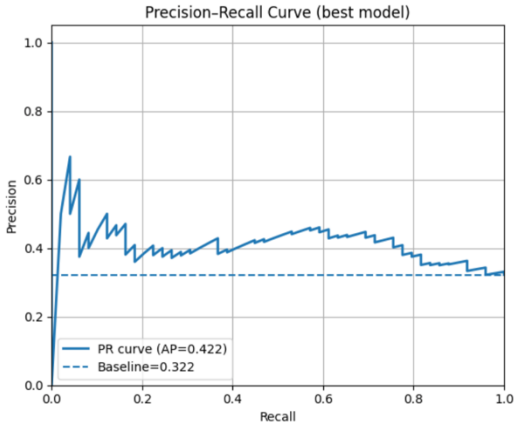
```
            roc_auc = roc_auc_score(yt_eval, scores)
            print(f"AUC:       {roc_auc:.4f}")

            # Precision–Recall curve + PR-AUC (Average Precision)
            precisions, recalls, _ = precision_recall_curve(yt_eval, scores)
            pr_auc = average_precision_score(yt_eval, scores)
            print(f"PR-AUC:    {pr_auc:.4f}")

            # Plot PR curve
            plt.figure(figsize=(6, 5))
            plt.plot(recalls, precisions, lw=2, label=f"PR curve (AP={pr_auc:.3f})")
            # baseline = positive prevalence
            base = np.mean(yt_eval)
            plt.hlines(base, 0, 1, linestyles="dashed", label=f"Baseline={base:.3f}")
            plt.xlim([0.0, 1.0])
            plt.ylim([0.0, 1.05])
            plt.xlabel("Recall")
            plt.ylabel("Precision")
            plt.title("Precision–Recall Curve (best model)")
            plt.legend(loc="lower left")
            plt.grid(True)
            plt.tight_layout()
            plt.show()
        else:
            print("AUC/PR: estimator has no predict_proba/decision_function.")

    except Exception as e:
        print(f"AUC/PR computation skipped due to error: {e}")

    best_info = {
        "estimator": best_name,
        "gamma": best_gamma,
        "metrics": {
            "accuracy": acc, "precision": pre, "recall": rec,
            "specificity": spec, "f1": f1,
            "roc_auc": roc_auc, "pr_auc": pr_auc
        },
        "confusion_matrix": cm
    }
    return table_f1, results, best_info, bw_best

# Your lists
from sklearn.ensemble import HistGradientBoostingClassifier, RandomForestClassifier
rf = RandomForestClassifier()
hg = HistGradientBoostingClassifier()
lr = LogisticRegression()
```

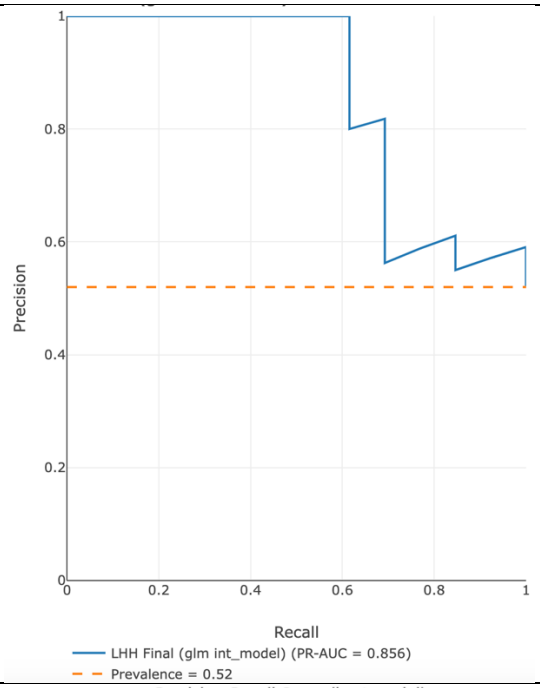
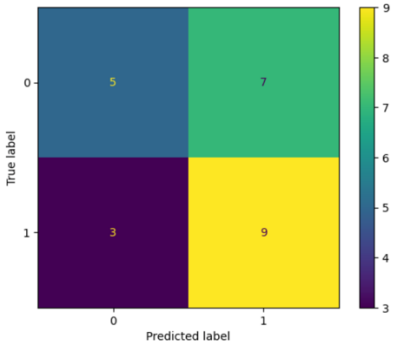
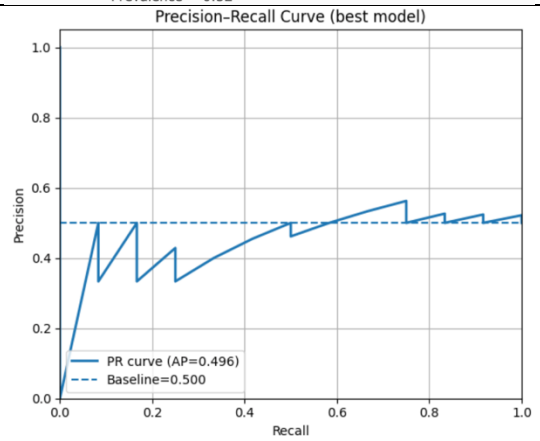
```
est = [lr, hg, logreg_clf, best_estimator_logreg, lr_clf,]
#g = [0.00005, 0.05, 0.2, 0.5, 0.9, 0.99]

#est = [logreg_clf]
g = np.arange(0.05,0.9995,0.07)

table_f1, results_long, best_info, best_model = bw_grid_f1_table(
    est_list=est,
    gamma_list=g,
    Xs=X_baseball_filtered, ys=y_baseball_filtered,   # source
    Xt_weight=X_Train_p_W2, yt_weight=y_softball_c_train,  # target for weighting
    Xt_eval=X_Test_p_W2, yt_eval=y_softball_c_test,       # TV softball set to evaluate
    random_state=41,
    verbose=0
)

print("\nF1 table (rows=estimator, cols=gamma):")
print(table_f1.round(4))

print("\nTop 5 rows of detailed results:")
print(results_long.sort_values('f1', ascending=False).head())
```

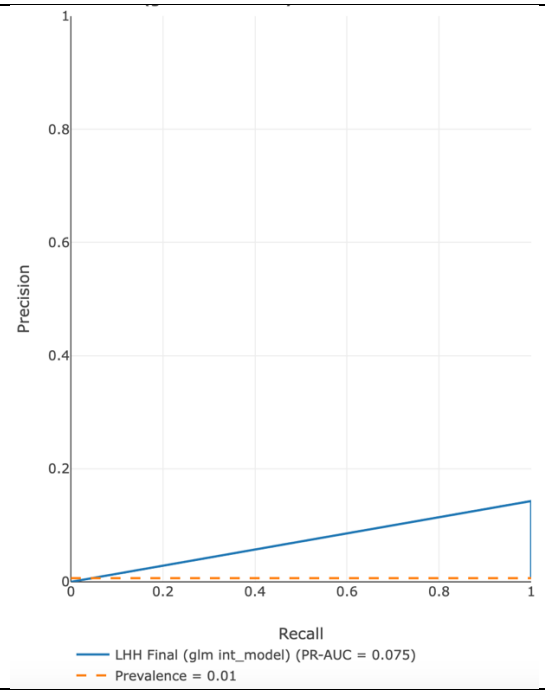
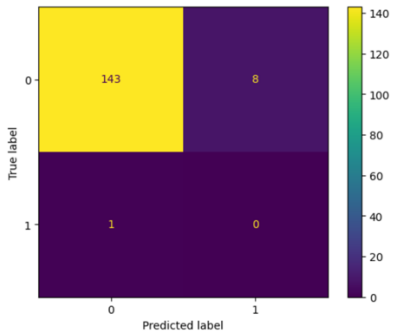
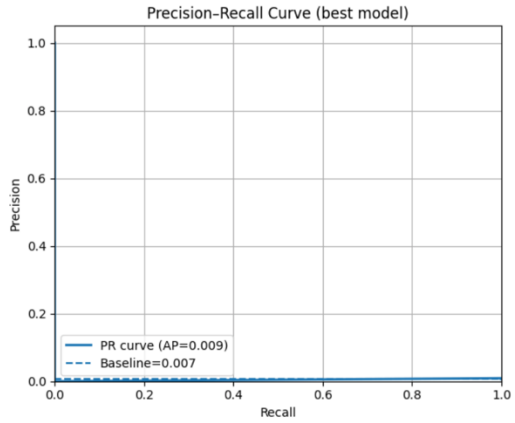
APPENDIX I
NON-SELECTED MODEL RESULTS

Left-Handed Hitters		
MODEL	**Test Metrics**	**PR-AUC**
Strikeout: Reduced Regression	**Actual** **Predicted Act 0 Act 1** **Pred 0 14 1** **Pred 1 58 20** N = 93 Accuracy = 0.37 Precision = 0.26 Recall = 0.95 F1 = 0.40 Specificity = 0.19 AUC = 0.64 PR-AUC = 0.34	
Strikeout: Domain Adaptation	 N = 93 Accuracy = 0.52 Precision = 0.28 Recall = 0.71 F1 = 0.40 Specificity = 0.46 AUC = 0.60 PR-AUC =0.33	

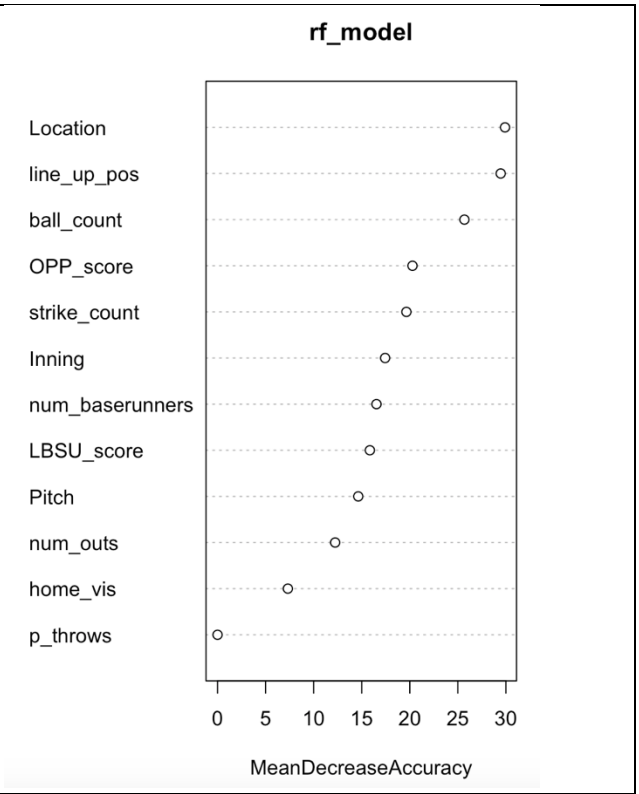
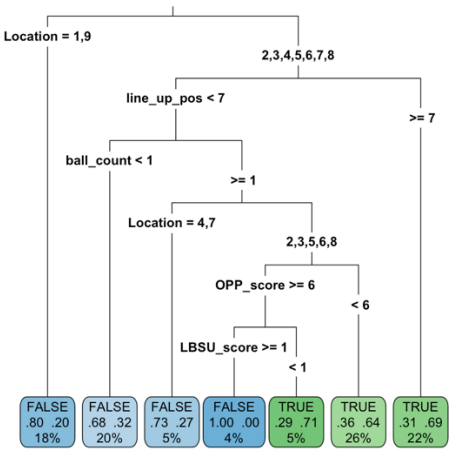
Hits: Regression	**Actual** **Predicted Act 0 Act 1** **Pred 0 34 5** **Pred 1 83 30** N = 160 Accuracy = 0.42 Precision = 0.27 Recall = 0.86 F1 = 0.41 Specificity = 0.29 AUC = 0.57 PR-AUC = 0.24	
Hits: Logistic Regression Domain Adaptation	 N = 152 Accuracy = 0.66 Precision = 0.38 Recall = 0.77 F1 = 0.51 Specificity = 0.62 AUC = 0.65 PR-AUC = 0.37	

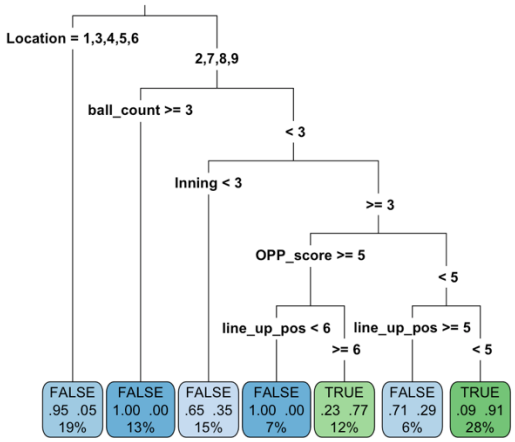
OBP: Reduced Logistic Regression	**Actual** **Predicted Act 0 Act 1** **Pred 0 37 8** **Pred 1 66 41** N = 160 Accuracy = 0.51 Precision = 0.38 Recall = 0.84 F1 = 0.53 Specificity = 0.36 AUC = 0.66 PR-AUC = 0.54	
OBP: Domain Adaptation	 N = 152 Accuracy = 0.64 Precision = 0.45 Recall = 0.61 F1 = 0.52 Specificity = 0.65 AUC = 0.63 PR-AUC = 0.42	

Walks: Reduced Logistic Regression	**Actual** **Predicted Act 0 Act 1** **Pred 0 10 4** **Pred 1 2 9** N = 25 Accuracy = 0.76 Precision = 0.82 Recall = 0.69 F1 = 0.75 Specificity = 0.83 AUC = 0.78 PR-AUC = 0.86	
Walks: Domain Adaptation	 N = 24 Accuracy = 0.58 Precision = 0.56 Recall = 0.75 F1 = 0.64 Specificity = 0.42 AUC = 0.47 PR-AUC = 0.50	

Home Runs: Reduced Logistic	**Actual** **Predicted Act 0 Act 1** **Pred 0 136 0** **Pred 1 15 1** N = 160 Accuracy = 0.90 Precision = 0.06 Recall = 1.00 F1 = 0.12 Specificity = 0.90 AUC = 0.96 PR-AUC = 0.075	
Home Run: Domain Adaptation	 N = 152 Accuracy = 0.94 Precision = 0 Recall = 0 F1 = 0 Specificity = 0.95 AUC = 0.27 PR-AUC = 0.01	

Individual Pitcher Models		
Model	**Test Metrics**	**Model**

Hits (RHH): Random Forrest	**Actual** **Predicted Act 0 Act 1** **Pred 0 40 13** **Pred 1 4 4** accuracy = 0.72 precision = 0.50 recall = 0.24 f1 = 0.32 specificity = 0.91 AUC = 0.60	
Hits (LHH): Decision Tree	**Actual** **Predicted FALSE TRUE** **FALSE 15 2** **TRUE 15 8** accuracy = 0.58 precision = 0.35 recall = 0.80 f1 = 0.48 specificity = 0.50 AUC = 0.65	
Strikeouts (RHH): Logistic Regression	**Actual** **Predicted Act 0 Act 1** **Pred 0 5 2** **Pred 1 22 7** accuracy = 0.33 precision = 0.24 recall = 0.78 f1 = 0.37 specificity = 0.19	**Coefficients:** **Estimate Std. Error z value Pr(>|z|)** **(Intercept) -0.16242 0.85683 -0.190 0.849650** **PitchD -1.29058 0.65011 -1.985 0.047125 *** **PitchR -0.12269 0.69214 -0.177 0.859308** **PitchS 1.20678 0.72164 1.672 0.094472 .** **PitchX -18.64102 1636.42468 -0.011 0.990911** **line_up_pos 0.41315 0.12526 3.298 0.000973 *** **num_outs 0.36986 0.33709 1.097 0.272558** **num_baserunners -0.76817 0.38916 -1.974 0.048391 *** **Plate_Appearance -0.05602 0.01655 -3.385 0.000711 *** **---** **Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1**

	AUC = 0.	
Strikeouts (LHH): Decision Tree	**Actual** **Predicted FALSE TRUE** **FALSE 3 1** **TRUE 11 4** accuracy = 0.37 precision = 0.27 recall = 0.80 f1 = 0.40 specificity = 0.21 AUC = 0.53	

REFERENCES

REFERENCES

Aggarwal, Charu C. 2015. *Data Mining: The Textbook.* Springer. https://doi.org/10.1007/978-3-319-14142-8.

Alamar, Benjamin C. 2013. "Data and Information." In *Sports Analytics: A Guide for Coaches, Managers, and Other Decision Makers*, 35–43. New York: Columbia University Press. http://www.jstor.org/stable/10.7312/alam16292.7.

Arnold, Andrew, Ramesh Nallapati, and William W. Cohen. 2007. "A Comparative Study of Methods for Transductive Transfer Learning." In *Proceedings of the 7th IEEE International Conference on Data Mining Workshops*, 77–82. Los Alamitos, CA: IEEE Computer Society. https://doi.org/10.1109/ICDMW.2007.4424142.

Baker, Lori L. 2021. "Women's History Month: Great Time to Reflect on Softball's Rich History." *National Fastpitch Coaches Association*, March 15, 2021. https://nfca.org/easyblog/womens-history-month-great-time-to-reflect-on-softballs-rich-history.

Boehmke, Brad, and Brandon M. Greenwell. 2020. *Hands-On Machine Learning with R.* Boca Raton, FL: CRC Press, Taylor & Francis Group.

Dormann, Carsten F., Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gianfranco Carré, et al. 2013. "Collinearity: A Review of Methods to Deal with It and a Simulation Study Evaluating Their Performance." *Ecography*36 (1): 27–46. https://doi.org/10.1111/j.1600-0587.2012.07348.x.

Feltey, Daniel, Sam Florence, and Sung-Hoon You. 2017. "Predicting Baseball Pitching Outcome." EECS 349 Machine Learning Project Report, Northwestern University.

Ganin, Yaroslav, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. 2016. "Domain-Adversarial Training of Neural Networks." *Journal of Machine Learning Research* 17 (59): 2096–2030.

Ganeshapillai, G., and John Guttag. 2012. "Predicting the Next Pitch." In *Proceedings of the MIT Sloan Sports Analytics Conference.*

G., Gilmer, Albert Lin, Michael Shannon, Asher B. Mirvish, Nicholas Aloi, Forrest Shooster, Justin J. Greiner. 2023. "Quantitative and Qualitative Disparities Exist Between Baseball and Softball Peer-Reviewed Pitching-Related Literature: A Systematic Review from 1990 to 2020." *JSES Reviews, Reports, and Techniques* 3: 499–505. https://doi.org/10.1016/j.xrrt.2023.07.003.

Hoffman, Judy, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei A. Efros, and Trevor Darrell. 2018. "CyCADA: Cycle-Consistent Adversarial Domain

Adaptation." In *Proceedings of the 35th International Conference on Machine Learning*, 1989–1998. Cambridge, MA: PMLR.

Huang, Jiayuan, Arthur Gretton, Karsten M. Borgwardt, Bernhard Schölkopf, and Alex J. Smola. 2006. "Correcting Sample Selection Bias by Unlabeled Data." *Journal of Machine Learning Research* 1 (1): 1–30.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning: With Applications in R.* 6th corrected printing. Springer. https://doi.org/10.1007/978-1-4614-7138-7.

Lee, Jae. 2022. "Prediction of Pitch Type and Location in Baseball Using an Ensemble of Deep Neural Networks." *Journal of Sports Analytics* 8 (4): 289–303. https://doi.org/10.3233/JSA-210645.

Lopez, Michael J., Gregory J. Matthews, and Ben S. Baumer. 2018. "How Often Does the Best Team Win? A Unified Approach to Understanding Randomness in North American Sport." *The Annals of Applied Statistics* 12 (4): 2483–2516. https://www.jstor.org/stable/26666161.

Long Beach State Softball. n.d. "Softball Statistics." Long Beach State University. https://longbeachstate.com/sports/softball/schedule.

Major League Baseball. *Statcast Data.* Baseball Savant. Accessed July, 2025. https://baseballsavant.mlb.com/.

Marcou, Casey. 2020. "Investigating Major League Baseball Pitchers and Quality of Contact through Cluster Analysis." Honors thesis, Grand Valley State University. ScholarWorks@GVSU. https://scholarworks.gvsu.edu/honorsprojects/805.

Murphy, Kevin P. 2022. *Probabilistic Machine Learning: An Introduction.* Cambridge, MA: MIT Press. https://lccn.loc.gov/2021027430.

NCAA. 2024. "NCAA Softball: Team Statistics." *NCAA.com.* https://www.ncaa.com/stats/softball/d1/current/team/282.

Pan, Sinno Jialin, and Qiang Yang. 2010. "A Survey on Transfer Learning." *IEEE Transactions on Knowledge and Data Engineering* 22 (10): 1345–1359. https://doi.org/10.1109/TKDE.2009.191.

Penn State. 2018. "12.1 – Logistic Regression." In *STAT 462: Applied Regression Analysis.* Pennsylvania State University. https://online.stat.psu.edu/stat462/node/207/.

Saerens, Marco, Patrice Latinne, and Christine Decaestecker. 2002. "Adjusting the Outputs of a Classifier to New a Priori Probabilities: A Simple Procedure." *Neural Computation* 14 (1): 21–41. https://doi.org/10.1162/089976602753284446.

Sidle, Graham, and Henry Tran. 2017. “Using Multi-Class Classification Methods to Predict Baseball Pitch Types.” *Journal of Sports Analytics* 3 (4): 271–283. https://doi.org/10.3233/JSA-170171.

Spitzer, Philipp, Daniel Martin, Lukas Eichberger, and Niklas Kühl. 2025. “Towards a Problem-Oriented Domain Adaptation Framework for Machine Learning.” *arXiv*, January 8, 2025. https://arxiv.org/abs/2501.04528.

Synergy Sports. n.d. “Synergy Sports Platform.” https://www.synergysportstech.com/.

West, James, Ryan Deitsch, and Jake Bardahl. 2024. “WCWS Championship Games Average 2 Million Viewers for Most-Watched Finals.” *The New York Times*, June 7, 2024. https://www.nytimes.com/athletic/5549789/2024/06/07/wcws-championship-finals-viewers-ratings/.

Woodward, Nathan. 2014. “A Decision Tree Approach to Pitch Prediction.” *The Hardball Times*, March 17, 2014. https://tht.fangraphs.com/a-decision-tree-approach-to-pitch-prediction/.