UmpAI: Classification of Home Plate Umpires' Weaknesses Through A Feedforward Neural Network

Aidan Eichman[1]

***Abstract*— As technology advances, baseball, also known as "the game of inches," frequently faces the debate between utilizing fallible human umpires and artificial automated strike systems. Nowadays, the majority of Machine Learning (ML) applications into sports resides in the realm of player analysis, betting strategy, and finding novelties to replace fallible aspects of the game. As such, I create UmpAI in order to open doors for ML applications that would mitigate umpires' human error. UmpAI is a feedfoward neural network (FNN) that logs every pitch for a specific umpire in the past five seasons and trains the model to predict, in a binary fashion, if the umpire will make the correct call. Thus, at the end of the model's training, I use SHAP values to determine which features—pitch speed, pitch location, pitch type, or inning number—most heavily influence individual umpire's incorrect call rate. This information can be further utilized to provide umpires with feedback and training regiments. This paper gives a brief overview of UmpAI, its ideation, implementation, evaluation, and future work.**

I. Introduction

For nearly two centuries, baseball has remained a preeminent competitive and recreational sport across the world. The game's sheer focus on data (e.g., probabilistic predictions and quantitative edges) lends itself nicely to applications with the technological progress of the past decades. One such development is known as sabermetrics, derived from the acronym SABR, which simply put is "the search for objective knowledge about baseball—how best to succeed in an in-game situation, determine a player's value, etc" [1]. As to be expected, initial reactions to sabermetrics were negative insofar that baseball traditionalists argued that long-standing scouts were more suitable to make these decisions. It was not until the Oakland A's experimented with and broke the seal on sabermetrics that the latter's strengths became indisputably apparent.

Umpires are another critical point of analysis after the introduction of mass data in baseball. It goes without saying that home plate umpires (those who call balls and strikes) are fallible. Whether it be because of a pitcher's deceptive arm slot, the pitch type, the weather, or otherwise, mistakes are inevitable. With an increase in incorrect umpire call rates over the past decade, the MLB has recently implemented the Automated Balls and Strikes (ABS) system in all 30 Triple A (AAA) ballparks for the 2023 season [2]. At first glance, this seems like a logical decision in that, with the capabilities of accurate spatiotemporal technology, detrimental incorrect calls on behalf of home plate umpires would become trivial. However, one must also acknowledge the drawbacks of such a solution to incorrect calls. First off, the implementation of these ABS systems is expensive, and there exists a latency issue that would intuitively contradict the recently-enacted pitch-clock rule to speed up the game. Further, although possibly leaning into the notion of traditionalism, the replacement of human umpires with an ABS system does pose a significant threat to fans' passion for the sport, as it dismantles zealous confrontation about umpiring decisions.[2]

Recognizing the possible shortcomings of using an ABS system, I set out to approach this debate from a less-frequently approached angle: with the hope of using a neural network (henceforth referred to as NN) to expose home plate umpires' weaknesses in terms of incorrect calls in order to provide them with off-season, personalized training regimens. This approach is novel in that the vast recent technological advancements have influenced most researchers to replace the fallible aspects of sports rather than use the newer technology to enhance the traditional facets of the sports (e.g., umpires).[3] I focus on the development of a feedforward neural network (FNN)—a type of acyclic artificial neural network (ANN)—in order to analyze how significantly certain in-game features impact an umpire's incorrect call rate. Figure 1 below outlines the high-level design for this project.

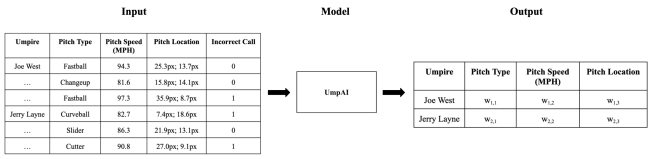

Fig. 1. Simplified example of the proposed problem in this paper.

In the following sections, I explore related works, explain the relevant technical background needed to understand UmpAI, delineate the model's high-level design and iterations, evaluate its effectiveness, explain its decisions with SHapley Additive exPlanations (SHAP values), and pave the path for its future steps and deployment. Through refinement and expansion, UmpAI could theoretically provide baseball leagues with a viable option to maintain the umpiring tradition of the sport while significantly lowering the rate of home plate umpires' incorrect calls. As a reference, the repository for UmpAI can be found at https://github.com.

[1]Aidan Eichman is a second-year at Columbia University studying Computer Science and Applied Mathematics. ane2123@columbia.edu

[2]That does not go to say that incorrect calls should exist in abundance, but rather that one must holistically consider the implications of using an ABS system.

[3]Note that this replacement/novelty often pertains to entities surrounding rules or the enforcers of such rules.

II. Related Work

Due to the fact that there is a paucity of umpire training and NN (more generally ML) previous work, one can turn one's attention to other realms of baseball which have seen an integration of ML techniques. Such applications predominantly reside in plate appearance (PA) outcome predictions, player performance predictions, and betting strategies. In 2021, Silver and Huffman [3] created an NN-based AI model called Singlearity-PA as an early PA-outcome prediction tool, a derivative of Haechrel's [4] log5 head-to-head manually probabilistic prediction model. The researchers inputted 87 floating point values, namely the a priori statistics on a batter versus pitcher during a PA, and trained their model to predict the likelihood of each of the 21 possible PA outcomes. With statistically significant results in terms of the accuracy of their model's predictions, this application of NNs for PA outcomes elicits questions regarding possible extensions of such approaches to not only other realms of baseball but also other data-driven sports.

Shortly thereafter, Sun et al. expanded the usage of NNs, specifically recurrent neural networks (RNNs), to create their Long Short-Term Memory as a predictor of home runs—a common power index for players [5]. The specific utilization of LSTM structures proved to be an accurate and effective simulation of players' performance and has shown an initial path to optimistic strategies for lineup adjustments. On the contrary, Yang and Luo [6] developed a backpropagation neural network (BPNN)—a multilayer feedforward neural network (FNN)—to test the relationship between sports performance and body weight, systolic and diastolic blood pressure, and oxygen saturation. For the purposes of this paper, Yang and Luo's statistically significant findings about the correlation between performance and biological processes are crucial. The BPNN sports prediction model had high accuracy in predicting sports performance in a more abstract and nontraditional manner, so the impetus was created to explore the use of FNNs as the foundation for UmpAI, a problem encompassing a unique approach angle.

All of these approaches have been shown to be effective applications of NNs to situational and player-performance predictions, so it becomes quite intuitive to redirect the focus from players to umpires. In doing so, researchers could, just as they do with players, establish prediction models for the accuracy of umpires' calls.

III. Background

A. Machine Learning (ML) & Deep Learning (DL)

ML is a general term for a vast array of algorithms that perform intelligent predictions derived from a data set. In essence, ML is the capability of machines to mimic human behavior and intelligence. It can be used to solve problems associated with the analytical model building. ML algorithms can be split into two main subsets: supervised and unsupervised learning [7]. Supervised learning is an approach that uses labeled data sets to train the ML algorithm and then measures the accuracy of the model using labeled expected outputs. On the other hand, unsupervised learning algorithms simply analyze and cluster unlabeled sets of data in an attempt to uncover patterns without supervision. Thus, the main difference between supervised and unsupervised learning exists in the fact that supervised learning wields labeled input and output data sets while unsupervised learning does not. For more information on supervised and unsupervised learning, please see [8], [9].

DL is a specific type of ML based on ANNs, a way in which, through nodes (neurons) and connected edges, machines simulate the behavior of a human brain. Through abstract hidden layers between the input and output layers, backpropagation, and gradient descent, DL models can refine their predictive accuracy. Although there is a multitude of ANN types, for the purposes of brevity, this background section will only focus on FNNs, the specific employment of ANNs in UmpAI. For more on DL and the kinds of ANNs, please see [10], [11], [12].

B. Feedforward Neural Networks (FNNs)

An FNN is an acyclic ANN, meaning that the information travels across node layers in one direction (rather in both directions like an RNN).

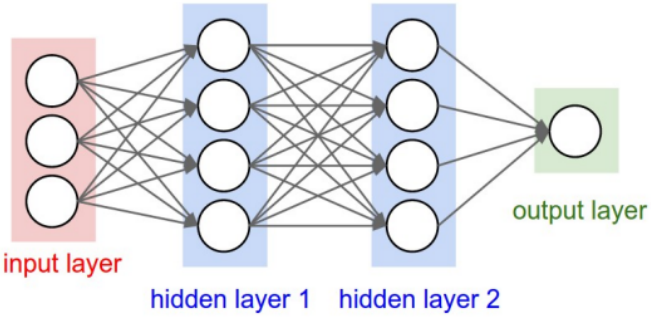

Fig. 2. High-level depiction of an FNN with two hidden layers [13].

Since FNNs can contain one or more hidden layers, there exists a distinction between single-layer and multi-layer perceptrons. FNNs that exist as single-layer perceptrons are used to classify linearly separable patterns [14], as a single hidden node layer is sufficient for the easily bound decision. With such an FNN, one can employ the Delta Rule [15] to compare the outputs of the FNNs nodes with the intended values for weight adjustment with a single hidden layer. FNNs are implemented as multi-layer perceptrons (MLPs) when a classification of non-linearly separable patterns is needed. The output of each neuron is computed as a non-linear function of the weighted sum of the inputs. Thus, FNNs lend themselves to classification problems where multiple FNNs could be run independently and then combined in the end. Ultimately, one could then analyze the absolute weight values of the input variables of an FNN to determine which input variables have the most influence on the probability of a specified output value. The salient problem in this paper is a complex one that should be approached using an MLP to capture the non-linear relationships. For further information about FNNs, please see [16], [17].

C. Data/Web Scraping

Data scraping (which I will use as nearly synonymous with web scraping, as this paper's application utilizes the World Wide Web in its data scraping process) is an automation technique to fetch data from websites. See Figure 3 below for a depiction of a generalized flowchart of data scraping. The acquisition of necessary data becomes burdensome if done manually without some form of data scraping. Based on the user's requirements, a web scraper (primarily implemented in Python) can be used to send requests and receive mass amounts of information/data which can be parsed and then, for instance, be used as inputs and expected outputs in machine learning models. Examples of automated extraction techniques range from HTML and DOM parsing to predefined web scraping software. Data scraping allows for the efficient retrieval and organization of data into a file or database.

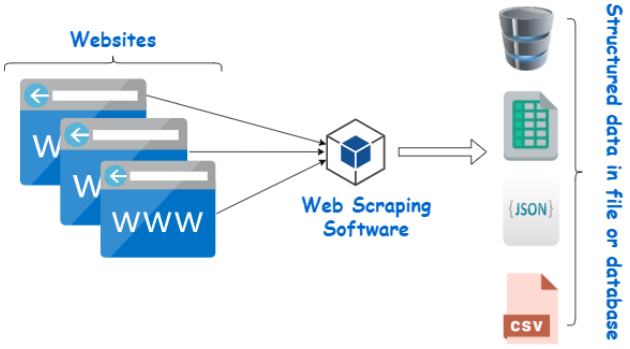

Fig. 3. High-level depiction of the web scraping process [18].

It is important to note, that with the power of data scraping and the sensitivity of certain data, one must consider the legal implications of scraping data from a website before doing so. For more information about data scraping, please refer to [19], [20].

D. SHapley Additive exPlanations (SHAP values)

ML models frequently fall into a "black box" phenomenon where one must trust the model's prediction without having a clear understanding of the sequential process which resulted in the model's decision. Therefore, it becomes important to generate a tool through which ML models can become more easily explainable. Silver and Huffman [3] were some of the pioneers of applying SHAP values to baseball and ML. Prior to this novelty, SHAP values' ordinary applications were in economic and game theory. As the researchers state, SHAP values "were later applied to complex AI models to make them explainable" and outline "a mechanism to provide a set of simple metrics and visualizations to show how each input feature contributed to the model's prediction" [3]. In essence, the SHAP value of a certain feature/input is the difference between the model's output when the feature is set to a specific value and the model's expected output. Thus, with a repetitive calculation of SHAP values with all feature subsets, one can gather general knowledge about each feature's contribution to the model's prediction and its subsequent accuracy. For further information on the technicalities of SHAP values, please refer to [21], [22].

IV. Design

In this section, I will describe the methodology for and implementations of data scraping and the UmpAI model itself.

A. Data Scraping

Prior to utilizing any scraping tools, it is crucial in an ML project such as this one to strictly define what data the model needs. Intuitively, a project about umpires' incorrect pitch calls necessitates umpire names and some indicator of whether or not the call was correct for each pitch thrown. Therefore, we can now consider which input variables would be informative features in our model. There is a myriad of such features including, but not limited to, pitch type (e.g., fastball, curveball, changeup, splitter, etc.), pitch speed (in MPH), pitch location (its breakdown will be described later), inning number (for possible umpire fatigue), weather (due to the visual impact of a glare, rain, etc.), and the batter's height (which could change the umpire's eye level). With a preliminary set of variables, we should analyze the public accessibility of such data.

Although baseball is one of, if not the, most data-driven sport, there is no truly efficient storage of mass pitch-by-pitch MLB data accessible to the public. It would have been helpful to have a database where big data could simply be extracted rather than manually gathered. MLB.com, Baseball-Reference.com, and Retrosheet.org all contain big data on each active and past player, but that data is limited to, at its smallest, a game-by-game scope. Hence, I turn to ESPN.com's play-by-play which includes pitch-by-pitch data; however, each aforementioned input feature is scattered across the ESPN page as shown in Figure 4. This exact problem lends itself to data/web scraping.

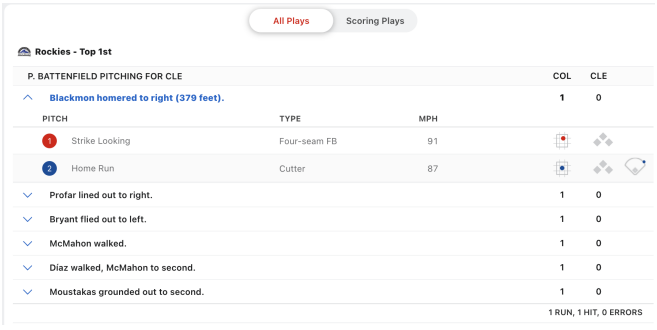

Fig. 4. An example ESPN play-by-play of the visitor's half of a Rockies baseball game. As shown, each PA contains all the pitches in a certain inning as well as their umpire's calls, their type, their speed, and their location (as designated by the rectangular hitzone icon).

ESPN's play-by-play information is aptly suited for our task of gathering data on the aforementioned features. The only features that are not included on this are the weather at

the time of the game and the batter's height. Theoretically, I could use the time, date, and location of the game on the ESPN page to web scrape for the weather and location of the sun in relation to the home plate umpire's direction and also web scrape for the batter's height. However, for the purposes of initial model creation, I determined that the four variables (type, speed, location, and inning number) were sufficient and that future work could expand the features to develop a more comprehensive model. I wanted to gather this play-by-play information for each game within the last five seasons. In the next sections, I will describe my initial, secondary, and final iterations of the scraping as well as where the iteration underperformed.

1) Iteration 1 — Game IDs: My initial scraping tactic entailed first gathering all of the necessary game IDs for the past five seasons and then once I have the list of those game IDs dynamically requesting the necessary data. I could not simply iterate through game IDs as I could not discern a pattern that I could use to directly access each play-by-play. Thus, collecting game IDs seemed like the correct approach. Such an algorithm's pseudocode can be seen in **Algorithm 1**.

Algorithm 1 Game ID Collection

```
Initialize a Chrome WebDriver instance, curr (← start) and end datetime variables, and an empty set of IDs
while curr ≤ end do
    url ← standard ESPN URL without the date at the end.
    date ← convert curr to YYYYMMDD format
    Append date to url; instruct driver to navigate to url
    Retrieve and parse HTML; extract ID from href
    Add the IDs to the set of IDs
    Increment curr value with timedelta
Save the set of IDs to a file path
```

Although this algorithm worked in the sense that it collected the game IDs in a reliable manner and saved them to a file, it was simply too slow for the purposes of collecting five times the number of games in a season. Then, if you put the optimization needed in conjunction with the fact that all of this was done to obtain game IDs that would then have to be re-utilized to scrape more off of the ESPN site, I turned away from this idea. it was time to look for a way to directly navigate through the ESPN site to gather the data.

2) Iteration 2 — Direct Selenium Approach: For the second iteration, I turned to Selenium's unique functionality. Selenium is an automated web scraping tool that is particularly useful when the navigation of a website requires user interaction (e.g., clicking buttons and navigating multiple pages). Because Selenium allows for the navigation of buttons, my next idea was to simply first request the driver to navigate to the URL with the schedule of games associated with the start date (five seasons ago). Then, click on each game and its play-by-play. For each PA drop-down in the game, we could then click and obtain all of the necessary pitch information for each pitch in that PA. Under the assumption that for all pitches in the game, the home plate umpire was the same, this seemed like a great way to retrieve the input data necessary for the model. Once the data from one game was retrieved, we should click the back button of the browser and navigate to the next game of the day (if there is no such next game, we click to the next day on the schedule bar at the top of the screen). Although what to look for in the HTML is unique to the ESPN site, I attempted to keep the pseudocode as general as possible to focus on the scraping ideation itself. Such an algorithm's pseudocode can be seen in **Algorithm 2**.

Algorithm 2 Selenium Usage for Every MLB Pitch in the Last Five Seasons

```
Initialize a Chrome WebDriver instance and curr (← start) and end datetime variables
Initialize and navigate to the ESPN MLB Scoreboard page for the start date (five seasons ago: April 2, 2017)
while curr ≤ end do
    for all games on the schedule do
        Click on the PLAY-BY-PLAY button
        for all PAs in the game (AtBatAccordion class) do
            Click on the PA drop-down to view its pitches
            for all pitches in the PA do
                Parse sequential entries in TableTD classes for data
                Send data to file path
        Use window.history.go(-1) to go back on browser
        if ∄ a next "Scoreboard" section class then
            Click to the next day on the DatePicker
    Increment curr value with timedelta
```

Upon testing this algorithm on a set of 20 MLB gamedays (April 2-11) I found that it both navigated the ESPN page well and sent the correct data to the output file. However, because there is a single computational thread, the extension of the algorithm scraping data for the entire five seasons was estimated at around 44 days. This is because, with sequential requests, we have to send them, wait for the response, and then continue only once a response has been received. Thus, this process necessitated heavy optimization in the next iteration.

3) Iteration 3 — Optimization of Iteration 2: There are two main limitations that can be addressed from the approach above. First of all, data is needed for in-season games, but the DatePicker contains every date out of the year. Thus, the first change I made for this iteration was to focus only on dates between April 1 and November 6 of each year. Therefore, even though our scraper would quickly move on from a blank game schedule on one of the dates outside of the season time frame, we will reduce a significant amount of clicking and navigating the ESPN site. This should speed up our scraping a fair amount. The second, more substantial optimization is the introduction of multithreading into the algorithm using a ThreadPoolExecutor. Put simply, multithreading is a way to multitask in that we can create various threads (which have their own stack-allocated local variables) and submit them to the ThreadPoolExecutor, which will perform asynchronous—as opposed to sequential—execution of the threads. In effect, we now have a pool of threads that can be executed at the same time, which will speed up our run time quite significantly. Note that the upper limit of the number of threads in our thread pool is contingent upon a device's RAM, so it is best to pre-define some number of threads that can be trivially handled. Further optimization can be done

where instead of creating monthly threads, we create weekly threads, but for the purposes of this project, monthly threads were sufficient. This optimization took the run time down to approximately 90 minutes. It is important to note that in this iterationm I decided to also scrape what the pitch was called by the umpire (which will allow us to see if the umpire made the correct call). Such an algorithm's pseudocode can be seen in Algorithm 3.

Algorithm 3 Multithreaded Selenium Usage for Every MLB Pitch in the Last Five Seasons (with ThreadPoolExecutor)

```
Initialize a Chrome WebDriver instance and initial (← start), curr, and end datetime variables
Define ThreadPoolExecutor with max number of threads that can be handled
for all years from initial.year() to end.year() do
    Submit a new thread for each year to the thread pool
    #Note that we can do this same thing below with weeks instead
    Find pairs of month start and end dates for the current year
    Disregard the pairs of dates for Jan., Feb., Mar., Dec.
    for all pairs of month start and end dates do
        Submit a new thread to the thread pool
        Set curr date to month start date
        while month curr date ≤ end date do
            Navigate to the schedule for the month's curr date
            for all games on schedule for the month's curr date do
                Click on the PLAY-BY-PLAY button
                for all PAs in the game (AtBatAccordion class) do
                    Click on the PA drop-down to view its pitches
                    for all pitches in the PA do
                        Parse entries in TableTD classes for data
                Use window.history.go(-1) to go back on browser
                if ∄ a next "Scoreboard" section class then
                    Click to the next day on the DatePicker
            Increment month curr date with timedelta
Wait for all threads to finish executing
```

Through this scraping process, I obtained all of the pitches, their locations, speeds, types, innings, calls, and corresponding umpires for the past five MLB seasons. The result was a csv file that I had appended to every time I retrieved new data. I could now move forward with data cleaning and model implementation. See Figure 5 for an example of what the csv file looked like after the scraping process.

Num	Pitch	Type	MPH	play-hitzone	Inning	umpire
1	Ball	Fastball	90	top: 14px; right: 13px;	Top 1st	Jerry Layne
2	Ground Out	Fastball	90	top: 10.37px; right: 26.39px;	Top 1st	Jerry Layne
1	Ground Out	Fastball	90	top: 12.97px; right: 22.31px;	Top 1st	Jerry Layne
1	Strike Looking	Fastball	90	top: 9.5px; right: 28.83px;	Top 1st	Jerry Layne
2	Ball	Slider	83	top: 22px; right: 23.13px;	Top 1st	Jerry Layne
3	Ball	Changeup	84	top: 10.59px; right: 30.87px;	Top 1st	Jerry Layne
4	Ball	Fastball	90	top: 19.47px; right: 30.87px;	Top 1st	Jerry Layne
5	Foul Ball	Fastball	91	top: 15.57px; right: 25.98px;	Top 1st	Jerry Layne
6	Strike Looking	Fastball	92	top: 16.01px; right: 17.02px;	Top 1st	Jerry Layne

Fig. 5. Example csv file after scraping the ESPN data.

4) Reflection on Data Scraping: As I reflect upon the scraping, there are two main suggestions I would give to anyone approaching a similar project. First off, although quite trivial, it is deeply important to deconstruct each part of the website from which you scrape data in order to know exactly where the data is within the scripting of the website and how you can strategically search for it. I fell into the trap of assuming where the data would be within the PA drop downs and it spent me quite a bit of time that could have been focused elsewhere. Secondly, find a smaller yet comprehensive subset of your desired data to test your scraper on. With a process as time-intensive as scraping, you will not want to debug by running the scraper in its entirety. Thus, by analyzing your necessary data and establishing a subset that covers all cases that the scraper could run into, you can tackle accuracy in a timely manner. Both of these make the scraping process a bit more bearable considering the retrieval of data is often the most tedious part of a project like this one.

B. FNN Model

Now that the data is in a condensed format, we can move on to implement the model. However, we must first consider the possibility that some data scraped from the ESPN site is missing or faulty. In addition, we must consider the ease or difficulty with which the model would interact with our data in its current format. After that, we can begin to create the model. In the ensuing sections, I will describe my data-cleaning process as well as a few iterations of my UmpAI model.

1) Data Cleaning: Data cleaning is simply the process used to remove or modify existing data in order to prepare that data for analysis. In order to tackle the missing or faulty data issue described above, I scripted a quick Python code to iterate through each row of the data csv file and check if all columns contained non-empty entries and that the entries were of the correct format. Luckily there was only one row that had an issue, because ESPN's format of play-by-play information is very consistent, making it easy for the scraper. I am not exactly how the row did not have a pitch location entry, but I manually went to the game in which the entry corresponded and entered the location data into the csv. In addition to this, I wrote another quick Python program to iterate through all of the pitches and analyze the umpire's call in relation to the pitch location. I skipped over all of the pitches which resulted in any event other than a strike looking or ball as those are irrelevant in our analysis. Then, I went over to ESPN to look into what the coordinates of the edges of the hit zone (strike zone) icon were. With this information, I could find if the umpire made an incorrect call on the pitch and then added that as another column to that which is in Figure 5. The row contained a 1 in the incorrect call column if the umpire made an incorrect call on that pitch and a 0 otherwise.

The other aspect of this cleaning process that was important to address was the format of the data within the csv file. Without a doubt, the umpire name, pitch speed, and inning for each pitch were in an acceptable format (string, integer, and integer, respectively). However, the format of the pitch type and location raised questions. I figured that pitch type could be label encoded for the model's input, so that would not be a prominent issue. On the contrary, the x, and y coordinate pixel location of each pitch is not

as informative as a model's input. Therefore, to make the data more effectively stored, I divided the hit zone and its surrounding space into 25 different zones. See Figure 6 for this division. Then, each of the zones could be treated as a feature for our model. This, too, was done with a short Python program that just checked the coordinates and classified that pitch into one of the 25 zones.

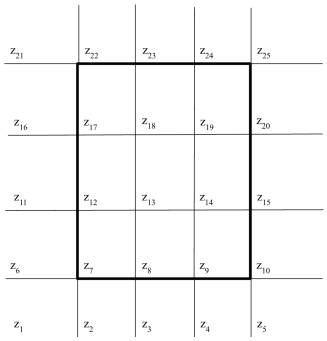

Fig. 6. Division of the strike zone for data simplification purposes.

With this final change, I finally had a cleaned csv file with data that I thought would be adequate for the model. I now will turn to the model iterations for the next sections of this paper.

2) Iteration 1: Generally speaking, I initially thought that **Algorithm 4** below would work. I wanted to work with the first 50 umpires to make sure that the model would run in a reasonable time (even though it would then not be as accurate as intended). At first, the model seemed to be testing well on the test data, which I was excited about. It was a week later when I returned to my initial implementation that I realized some major flaws in this version. First off, I was generating one overall model for all of the umpires. This would be an effective approach if I hoped to learn more about trends among MLB umpires as a whole. However, given that my task is to create regiments for each umpire to expose what areas they need to focus on in training, this model is rendered useless. I was pleasantly surprised to find out that my initial encoding and scaling worked for transforming the csv file's data into usable data for the model. The other issue that I realized was that my csv file still contained all of the pitches for both incorrect and correct calls. Correct calls are much more common than incorrect calls, so the model has access to much more data on the correct call events than incorrect call events. This could have led to the model simply guessing that a correct call was made (or equivalently that an incorrect call was not made), resulting in a high prediction accuracy. Hence, I wanted to move forward with another iteration of the model with these two issues in mind. Other than that, the structuring and control flow of the model seemed to be quite effective.

Algorithm 4 General UmpAI FNN Model

1: Import pandas as pd, numpy as np, OneHotEncoder, StandardScaler, train test split, tensorflow as tf, Sequential, Dense, and Dropout
2: Load input data into dataframe *data* from pitches.csv
3: Extract a list of the first 50 unique umpire names from *data* into *umpire list*
4: Filter rows from *data* where the umpire name is in *umpire list*
5: Split *data* into training, validation, and test sets using *train test split* with a test size of 0.2
6: Further split *train data* into new training and test sets using *train test split* with a test size of 0.25
7: Instantiate a StandardScaler object *scaler*
8: Instantiate a OneHotEncoder object *encoder* with the parameter *handle unknown* set to 'ignore'
9: Use *scaler* to transform the 'pitch speed' and 'inning' columns of *train data*, *val data*, and *test data*
10: Use *encoder* to fit and transform the 'pitch location', 'pitch type', and 'umpire name' columns of *train data*
11: Transform the 'pitch location', 'pitch type', and 'umpire name' columns of *val data* and *test data* using the trained *encoder*
12: Concatenate the scaled 'pitch speed' and 'inning' columns and the encoded 'pitch location', 'pitch type', and 'umpire name' columns of *train data*, *val data*, and *test data* into new *X train*, *X val*, and *X test* dataframes
13: Store the 'incorrect call' column of *train data*, *val data*, and *test data* into new *Y train*, *Y val*, and *Y test* arrays
14: Instantiate a Sequential object *model*
15: Add a Dense layer with 64 units and ReLU activation to *model* with input shape equal to the number of columns of *X train*
16: Add a Dropout layer with a rate of 0.5 to *model*
17: Add a Dense layer with 32 units and ReLU activation to *model*
18: Add a Dropout layer with a rate of 0.5 to *model*
19: Add a Dense layer with 1 unit and sigmoid activation to *model*
20: Compile *model* with binary cross-entropy loss function, Adam optimizer, and accuracy metric
21: Train *model* on *X train* and *Y train* for 50 epochs with a batch size of 32 and validation data *(X val, Y val)*
22: Evaluate the trained *model* on the test data *(X test, Y test)* and print the test accuracy

3) Iteration 2: This algorithm below mimics the structure of my current version of UmpAI (as of now). In it, I addressed both of the aforementioned issues from **Algorithm 4**. First off, now the algorithm trains a distinct model for each of the umpires in the csv file as opposed to one generalized model. This allows me to analyze the impacts of each feature on each umpire's incorrect call rate and thus possibly create regiments. Moreover, to address the issue with too many correct call data points, I ended up gathering all of the incorrect call data points and then randomly selecting the same number of correct call data points (as $\#_{incorrect} << \#_{correct}$ for all umpires) to use for the model training. All in all, with these changes, the algorithm trained models for each umpire and were quite accurate. I will further describe the results of this iteration in the next section.

Algorithm 5 UmpAI FNN Model for Specific Umpires

Import all necessary modules
vl, va, vls, vas, tl, ta, tls, tas ← []
data ← load data from file csv file
umpires ← data['umpire'].unique()
for all umpire_name in umpires **do**
    correct_calls ← data[data['umpire'] == umpire_name]
                    [data['incorrect_call'] == 0]
    incorrect_calls ← data[data['umpire'] == umpire_name]
                    [data['incorrect_call'] == 1]
    Normalize the amount of incorrect and correct call data so that our overall data it is split in half between the two call types (randomly select correct call data points to match the number of total incorrect call data points)
    Split each of the correct_call and incorrect_call data sets into training, validation, and testing sets (60, 20, 20)
    Concatenate the train, val, and test data sets so that they contain both the incorrect and correct calls and initialize those concatentations to our general train, val, and test data sets
    Instantiate a StandardScaler object *scaler*
    Instantiate a OneHotEncoder object *encoder* with the parameter *handle unknown* set to 'ignore'
    Use *scaler* to transform the 'pitch speed' and 'inning' columns of *train data*, *val data*, and *test data*
    Use *encoder* to fit and transform the 'pitch location', 'pitch type', and 'umpire name' columns of *train data*
    Transform the 'pitch location', 'pitch type', and 'umpire name' columns of *val data* and *test data* using the trained *encoder*
    Concatenate the scaled 'pitch speed' and 'inning' columns and the encoded 'pitch location', 'pitch type', and 'umpire name' columns of *train data*, *val data*, and *test data* into new *X train*, *X val*, and *X test* data frames
    Store the 'incorrect call' column of *train data*, *val data*, and *test data* into new *Y train*, *Y val*, and *Y test* arrays
    Instantiate a Sequential object *model*
    Add a Dense layer with 64 units and ReLU activation to *model* with input shape equal to the number of columns of *X train*
    Add a Dropout layer with a rate of 0.5 to *model*
    Add a Dense layer with 32 units and ReLU activation to *model*
    Add a Dropout layer with a rate of 0.5 to *model*
    Add a Dense layer with 1 unit and sigmoid activation to *model*
    Compile *model* with binary cross-entropy loss function, Adam optimizer, and accuracy metric
    Train *model* on *X train* and *Y train* for 50 epochs with a batch size of 32 and validation data *(X val, Y val)*
    Evaluate the trained *model* on the test data *(X test, Y test)* and print the test accuracy
    Print those values and then append them to the overall list of means
Calculate and output the overall means and standard deviations

V. Results

The main results that correspond with an UmpAI model are its prediction accuracy and loss. First off, I turn to the accuracy of models. I created a model for each of the umpires and then appended those accuracy values to a general accuracy average. I did this 10 different times to confirm the results and show the general accuracy of the unique models across all umpires. The results are shown in Figure 7. The accuracies seem to hover over the mid-80s which is a great place for FNN model accuracies to be. The results suggest that the model is neither underfitting nor overfitting at first glance. It will be exponentially helpful to gather more data for each umpire in order to further train, validate, and test each umpire's model. Let us now turn to an analysis of each model's loss. I decided to use Binary Cross Entropy as a loss function because the binary—correct or incorrect call—output lends itself nicely to such usage. The

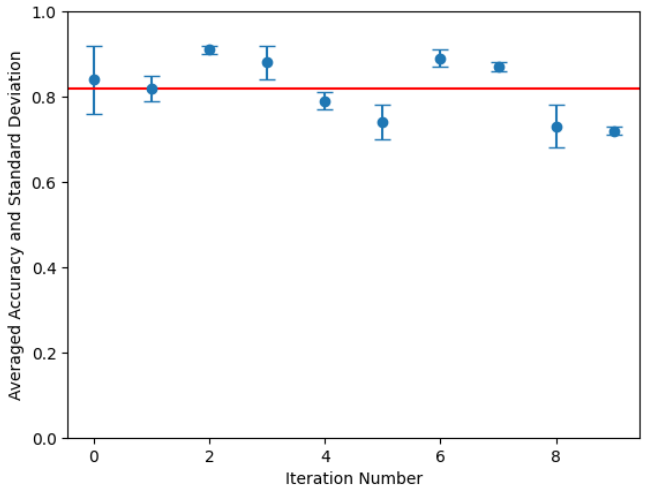

Fig. 7. Averaged accuracies for each iteration of creating each umpire's model.

loss is calculated with this formula:

$$-\frac{1}{N}\Sigma_{i=1}^{N}(y_i \cdot log(p_i) + (1-y_i) \cdot log(1-p_i)) \qquad (1)$$

. In general, you want your loss to be as low as possible for each model, because loss represents the summation of errors in your model. In Figure 8 below is a plot of the loss as obtained in a similar fashion as the accuracy averages above. Unfortunately, these loss values are quite high. I would like

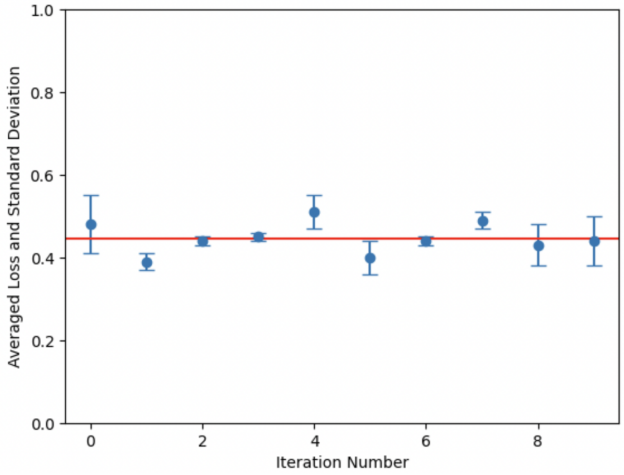

Fig. 8. Averaged losses for each iteration of creating each umpire's model.

them to realistically be around 0.2. I think that some of this issue is that my current implementation lacks the diversity of features that I want in the future. Once I gather more data and broaden the scope of the features, the model should be trained to better understand this relationship and make fewer "bad" predictions. Now let us look into how the accuracy and loss trend during the training and validation states in one umpire's model. Note that we hope to see the training and validation accuracy increase and eventually plateau near the top of the accuracy axis. There should not be too much

of a difference between the two of them, or overfitting might be in play. We hope to see the exact opposite occur with the training and validation loss values. The ideal model would be such that the loss would steadily decrease and then plateau near the bottom of the loss axis. Below, in Figure 9, is a depiction of the accuracy and loss values in the training and validation sets for Jerry Layne's UmpAI model.

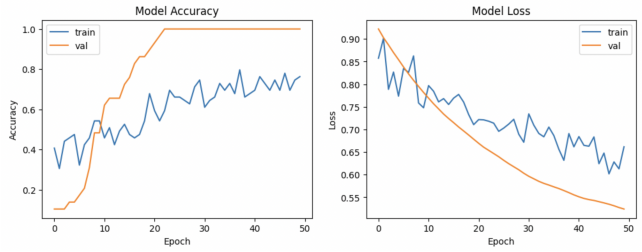

Fig. 9. Accuracy and loss trends in Jerry Layne's UmpAI model.

Overall, the asymptotic trends are in the correct direction, but there seems to be a bit of overfitting suggested by the medium-sized difference in the accuracies between the two sets as well as the step-like movement of the validation accuracy. In the near future, I plan to work on mitigating this overfitting and then trial-and-error testing hyperparameters to possibly lower the loss values. In general, though, the accuracy and loss trends seem to exist in loose region that I hope they would.

Lastly, below is an early implementation of using SHAP values to explain the models decision (see Figure 10). I have printed the top 3 most important features for Jerry Layne's model. This information, along with the fact that Jerry Layne's model is quite accurate, could give us preliminary insight into which factors Layne could work on during the off season in order to become a more accurate home plate umpire. The features which impact Layne's model will be those which have the greatest absolute weight in the FNN. Granted, the results would be further substantiated once the model gained a bit more accuracy and its loss values decreased significantly. In the future, I hope to figure out a way to use a violin plot to show these SHAP values as well as display the step-by-step decision process a model goes through using a SHAP explainer. Currently, I am running into a roadblock with the label encoder but hope to get past that issue soon.

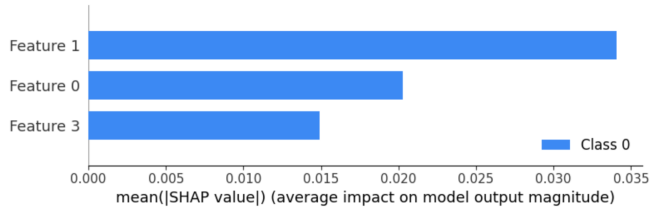

Fig. 10. Jerry Layne's weakest areas in pitch calling. Feature 1, 0, and 3 correspond to zone (pitch location), pitch speed, and pitch type, respectively

With these results, I think that I can further refine UmpAI to solidify a tool that could be effectively used for its purpose. The results show that most models are quite accurate but do make bigger errors sometimes; however, these issues are quite solvable with expansion of feature utilization and available data.

VI. Future Work

In this section, I will outline where I plan to take this project in the future. AI is an ever-changing field, which means that there will always be optimizations and new approaches for each project. As such, I will guide you through what I perceive my next step to be in regard to UmpAI. There are plenty of aspects of this project that could benefit from continuous further work.

A. Refinement

The first area of business to attend to is refining UmpAI. There are three main areas that need to be refined:

1) Work on mitigating overfitting
2) Expand the features included in the model
3) Have SHAP values and plots for each umpire's model

I want to make sure that the model is as robust as possible, which means that I should find ways to mitigate overfitting. One way this can be done is by simply increasing the amount of data we have for each umpire's model to work with. Then, we could split the data up and have a larger amount of data at each stage (training, validation, and testing). I have also looked into regularization and data augmentation as possibilities. In the near future, I will be exploring these three options and detecting which could be most helpful in addressing this concern. Secondly, with only four features as of now, the models are restricted in their ability. As explained in Section IV, there is a multitude of other variables that could influence an umpire's rate of incorrect calls. The next steps would be to configure ways to gather this data for the model, as it is more qualitative (e.g., the weather and glare) than our current features. However, this is a necessary refinement in order to increase the effective functionality of UmpAI. Lastly, SHAP values will be crucial for producing the regiments for the umpires. The main setback as of now is how computationally expensive these visualizations are. Because the explainable AI plots work by removing certain aspects of features and then testing how much difference that creates between the actual and expected predictions, if there are an abundance of features, SHAP value enactment takes—for a lack of better terms—ages. Therefore, it might be helpful to gather the complete SHAP values for two or three umpires in the near future in order to present what such a visualization would look like. I could not get the complete SHAP analysis to be produced quickly enough, even with my computer running as long as it realistically could at the time. Furthermore, I hope to implement the SHAP analysis such that I can figure out which type of pitch, which speed, etc. specifically is a weak combination for each umpire (but I need much more data for this). Each of these areas of refinement would further UmpAI's effectiveness and scope, creating a better product overall.

B. Dynamic database usage

During this process, I had the fortunate opportunity to speak to managers within the Los Angeles Dodgers Organization. Not only were they helpful in guiding the direction of my project and pointing out areas of improvement, but they also suggested that I use their database for data. They had recognized the same issue with the limited consistent storage of pitch-by-pitch data and have begun creating a database that effectively stores this information. Therefore, I hope to modify my model so that instead of being restricted to the data that has been scraped previously, I can dynamically pull data from the aforementioned database. With this, the model can continuously improve, which would give a more comprehensive investigation of incorrect umpire calls and thus allows for the possibility of monthly or more frequent regiment creation for umpires. Overall, access to this database would dramatically increase the application scope for UmpAI, and I am thankful for the opportunity to access this data through an organized database.

C. Formal documentation of regiments

Furthermore, one of the next steps is to establish a methodology for translating UmpAI's analysis into a structured regiment for umpires. I have not yet decided how the information should be presented in order to make the off-season training process smooth. It might come down to simply relaying the SHAP values for each model for each umpire to whoever will create the training as an overseer. Then, after walking them through the explainable AI predictions, they would be able to alter existing umpire training processes or maybe even tailor new ones. Regardless, I will be sitting down with my friend's father, who is an ex-MLB umpire, in order to discuss how he thinks I should translate this information into usable regiments from an umpire's point of view.

D. Extension into other sports

There are quite a few applications of ML to sports in general, so it often takes looking at the sport or problem from a different angle to decipher a novel and informative project. I think that once one does so, one will discover that there is much, much more to uncover in the realm of ML and sports. As most researchers turn to betting techniques and player performance analytics, there are undoubtedly other aspects of the game that could benefit from a model and its interpretation. Hence, I hope to use what I have learned about an ML model's application to umpires to springboard into analyzing referees from another sport. A plethora of the ambiguity and issues around bad referees could possibly be explained or even mitigated using ML models. I will take what I have learned from, struggled through, and grappled with to hopefully make the next extension of this project more efficient time and iteration-wise.

VII. Conclusion

In this work, I introduce UmpAI, a FNN which is used to help classify the weak points in a home plate umpire's calls. In the FNN, I develop a strategy to web scrape necessary data and predict the binary expected output—correct or incorrect call—with a high accuracy and medium loss. With these models, I use SHAP values to convert the ML into explainable AI that can be later structured into training regiments. As my access to pitch-by-pitch data increases and I am able to utilize more in-game situational features, UmpAI will become more powerful in its quest to be a cohesive analysis of home plate umpires' incorrect calls. This methodology provides a road map for exploring NN applications in other sports, specifically for referee and umpire analysis.

VIII. Acknowledgments

I would like to thank Dr. Brian Plancher for his unwavering support throughout this project. He provided informative insights into the structure of the project and guided its trajectory with feedback.

References

[1] J. Mizels, B. Erickson, and P. Chalmers, "Current state of data and analytics research in baseball," *Curr Rev Musculoskelet Med.*, vol. 15, no. 4, 2022.
[2] B. Olney, "All aaa parks to use electronic strike zones in 2023," *ESPN*, 2023.
[3] J. Silver and T. Huffman, "Baseball predictions and strategies using explainable ai." MIT Sloan Sports Analytics Conference and ESPN, 2021.
[4] M. Haechral, "Matchup probabilities in major league baseball," *Sabr. Org*, 2014.
[5] H.-C. Sun, T.-Y. Lin, and Y.-L. Tsai, "Performance prediction in major league baseball by long short-term memory networks," *International Journal of Data Science and Analytics*, vol. 15, no. 1, 2022.
[6] H. Yang and C. Luo, "Accuracy analysis of sports performance prediction based on bp neural network intelligent algorithm," *Security and Communication Networks*, pp. 1–11, 2022.
[7] J. Delua. Supervised vs. unsupervised learning: What's the difference? IBM.
[8] C. Donalek, "Supervised and unsupervised learning," California Institute of Technology.
[9] R. Sathya and A. Abraham, "Comparison of supervised and unsupervised learning algorithms for pattern classification," *International Journal of Advanced Research in Artificial Intelligence*, vol. 2, no. 2, 2013.
[10] V. Kumar and M. L., "Deep learning as a frontier of machine learning: A review," *International Journal of Computer Applications*, vol. 182, no. 1, pp. 22–30, 2018.
[11] D. Lee, S. Derrible, and F. C. Pereira, "Comparison of four types of artificial neural network and a multinomial logit model for travel mode choice modeling," *Transportation Research Record: Journal of the Transportation Research Board*, vol. 2672, no. 49, pp. 101–112, 2018.
[12] R. Dastres and M. Soori, "Artificial neural network systemsw," *International Journal of Imaging and Robotics*, vol. 21, no. 2, 2021.
[13] J. McGonagle, J. Garcia, and S. Mollick, "Feedforward neural networks," https://brilliant.org/wiki/feedforward-neural-networks/, [Online].
[14] "Feed forward neural network," DeepAI.
[15] Y. Ioannou, "Backpropogation derivation - delta rule," University of Cambridge, 2018.
[16] M. Sazli, "A brief review of feed-forward neural networks," *Communications, Faculty Of Science, University of Ankara*, pp. 11–17, 2006.
[17] Z. Zhang, F. Feng, and T. Huang, "Fnns: An effective feedforward neural network scheme with random weights for processing large-scale datasets," *Applied Sciences*, vol. 12, no. 23, 2022.
[18] WebHarvy, "What is web scraping?" https://www.webharvy.com/articles/what-is-web-scraping.html, [Online].
[19] B. Zhao, "Web scraping," *Encyclopedia of Big Data*, pp. 951–953, 2022.

[20] M. Khder, "Web scraping or web crawling: State of art, techniques, approaches and application," *International Journal of Advances in Soft Computing and its Applications*, pp. 145–168, 2021.

[21] S. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," 2017.

[22] D. Fryer, I. Strumke, and H. Nguyen, "Shapley values for feature selection: The good, the bad, and the axioms," *IEEE Access*, 2021.