**Machine Learning Applied to Baseball Umpire Calls**

**MEEN 423**

A logo on a black background

Description automatically generated

**By: Rohan Singh, Angela Bauer, Zachary Foster**

**Fall 2023**

## **Introduction**

This report provides a focused examination of the baseball officiating domain, specifically homing in on the discernment of pitches by umpires. In the pursuit of objective insights, meticulous analysis of spatial dynamics within the strike zone has been conducted, utilizing empirical data sourced from live game scenarios.

The primary objective is to identify and elucidate patterns, tendencies, and potential variations in umpire decision-making. Through a stringent examination of umpire-specific behaviors and strike zone dimensions, this report endeavors to offer a concise yet comprehensive understanding of the consistency—or lack thereof—in pitch calls made by different umpires.

This study contributes to the discourse on the subjective nature of baseball officiating, presenting visualizations and analyses that transcend anecdotal discussions. The findings herein aim to serve as a valuable resource for stakeholder’s keen on comprehending the nuanced intricacies of baseball umpiring, elevating the discourse around this critical facet of the sport.

## **Application**

This Python script is designed to analyze and predict umpire calls in a baseball game. It uses a variety of machine learning and data analysis techniques to achieve this. Here's a detailed breakdown of what the code does:

1. **Data Loading and Preprocessing:** The script begins by importing necessary libraries and loading the data from a CSV file. The data includes pitch location (**pX**, **pZ**), the umpire's call (**pCall**), the top and bottom of the strike zone (**sZ\_top**, **sZ\_bot**), and the home plate umpire's name (**gameHP**). The script then normalizes the pitch's Z position (**pz**) based on the size of the strike zone for each pitch.
2. **Data Segregation:** The script segregates the data based on the home plate umpire. For each umpire, it creates separate lists for the X and Z positions of the pitches and the umpire's call. The calls are converted to binary format (1 for 'Strike', 0 for 'Ball').
3. **Data Visualization:** The script plots all the pitches for each umpire, with different colors for balls and strikes. The strike zone is also plotted as a rectangle.
4. **Machine Learning Model Training and Prediction:** The script uses the **RandomForestClassifier** from the **sklearn** library to predict the umpire's call based on the X and Z positions of the pitch. The data is split into training and test sets, and the model is trained on the training set. The model's accuracy is then evaluated on the test set.
5. **Decision Boundary Visualization:** The script visualizes the decision boundary of the trained model by creating a contour plot. The test pitches are also plotted on this contour plot.

The machine learning techniques used in this script include:

* **Data Normalization:** This is a preprocessing step that transforms the values of numeric variables in the dataset to a common scale, without distorting differences in the ranges of values or losing information.
* **Train-Test Split:** This is a technique for evaluating the performance of a machine learning algorithm. It involves splitting the dataset into two subsets: one for training the model and the other for testing the model's predictions.
* **Random Forest Classifier:** This is a type of ensemble machine learning algorithm that operates by constructing multiple decision trees during training and outputting the class that is the mode of the classes output by individual trees.
* **Accuracy Score:** This is a metric for evaluating classification models. Informally, accuracy is the fraction of predictions our model got right.

## **Possible Optimizations:**

The current code is already quite optimized and uses a robust machine learning model, the Random Forest Classifier, which is known for its high accuracy and ability to prevent overfitting. However, there are a few improvements and additional techniques that could be applied:

1. **Cross-Validation:** Instead of a simple train-test split, you could use cross-validation to get a more robust measure of your model's performance. This involves splitting your data into 'k' subsets and training/testing your model 'k' times, each time with a different subset reserved as the test set.
2. **Hyperparameter Tuning:** The parameters of the Random Forest Classifier (like **n\_estimators**, **max\_depth**, **min\_samples\_split**, and **min\_samples\_leaf**) are currently hardcoded. We could use **GridSearchCV** or **RandomizedSearchCV** from **sklearn** to find the optimal parameters for your model.
3. **Feature Importance:** Random Forests allow you to measure the importance of each feature in making predictions. You could add a section to your code that outputs the feature importance, which could give you insights into which features are most influential in an umpire's call.

## **Hyperparameter Tuning:**

Hyperparameter tuning is the process of finding the optimal set of hyperparameters (parameters that are not learned from the data) for a machine learning model. In the case of a Random Forest Classifier, these hyperparameters include the number of trees in the forest (**n\_estimators**), the maximum depth of the trees (**max\_depth**), the minimum number of samples required to split an internal node (**min\_samples\_split**), and the minimum number of samples required to be at a leaf node (**min\_samples\_leaf**).

The **ai\_umpire\_short\_hyperparameter\_tuning.py** script performs hyperparameter tuning using a technique called grid search. Grid search is a brute-force method that involves specifying a list of possible values for each hyperparameter and then training and evaluating a model for every possible combination of hyperparameters. The combination that results in the best model performance is considered the optimal set of hyperparameters.

Here's how the **ai\_umpire\_short\_hyperparameter\_tuning.py** script performs hyperparameter tuning:

1. It defines the parameter grid to search (lines 71-76). The parameter grid is a dictionary where the keys are the hyperparameter names and the values are lists of possible hyperparameter values.
2. It creates a RandomForestClassifier (line 79) and a GridSearchCV object (line 82). The GridSearchCV object is a utility provided by scikit-learn that automates the process of performing grid search.
3. It performs the grid search by calling grid\_search.fit(X\_train, y\_train) (line 85). This trains a RandomForestClassifier for every combination of hyperparameters and evaluates its performance using 5-fold cross-validation.
4. It prints the best hyperparameters for each umpire (line 88) and uses the best model to predict the test set (lines 94-97).

The hyperparameter tuning process helps the ai\_umpire\_short.py script by providing a more optimized model. The ai\_umpire\_short.py script uses a RandomForestClassifier with hard-coded hyperparameters, which may not be optimal. By tuning the hyperparameters, the ai\_umpire\_short\_hyperparameter\_tuning.py script can potentially achieve better model performance.

**Best Hyperparameters for Angel Hernandez:** {'max\_depth': 10, 'min\_samples\_leaf': 1, 'min\_samples\_split': 10, 'n\_estimators': 200}

**Best Hyperparameters for Erich Bacchus:** {'max\_depth': 10, 'min\_samples\_leaf': 1, 'min\_samples\_split': 5, 'n\_estimators': 50}

**Best Hyperparameters for Junior Valentine:** {'max\_depth': 20, 'min\_samples\_leaf': 5, 'min\_samples\_split': 10, 'n\_estimators': 100}

**Best Hyperparameters for Malachi Moore:** {'max\_depth': 10, 'min\_samples\_leaf': 5, 'min\_samples\_split': 2, 'n\_estimators': 100}

**Best Hyperparameters for Pat Hoberg: {**'max\_depth': 10, 'min\_samples\_leaf': 5, 'min\_samples\_split': 5, 'n\_estimators': 100}

**Best Hyperparameters for Quinn Wolcott:** {'max\_depth': 5, 'min\_samples\_leaf': 1, 'min\_samples\_split': 10, 'n\_estimators': 50}

## **Conclusion:**