MLB ‘Fielding Independent Pitching’ Prediction

Mathew Katz

December 12, 2023

Abstract:

Major League Baseball is a thriving North American professional baseball league with an impressive 11-billion-dollar valuation that translates into an average team value of $344 million U.S. dollars. The league not only creates employment opportunities, but also stimulates local economies through tourism and business activities associated with games, making it a significant contributor to the larger sporting industry.

Recognizing its profound historical, traditional, economic, and social significance, I sought a meaningful way to contribute to the success of Major League Baseball teams. With this goal in mind, the purpose of this project was to predict 'Fielding Independent Pitching' (FIP) using pitcher’s statistics from the previous year. FIP is a metric that focuses exclusively on aspects of a pitcher's performance where they exert the most influence—namely, strikeouts, walks, hit-by-pitches, and home runs. It intentionally excludes outcomes related to balls hit into the field of play.

After considering the option of utilizing Earned Run Average (ERA), a measure that signifies the average number of earned runs a pitcher concedes per nine innings pitched and is widely employed as a standard pitching performance metric, I ultimately decided that FIP would be a more precise gauge of a pitcher's effectiveness.

To address this inquiry, I employed a five-step approach, encompassing Data Gathering, Data Preprocessing, Data Exploration, Regression Modeling, and Model Evaluation. In my investigation, I explored baseball statistics with the strongest correlation to the following year's FIP, selecting optimal features for the model. Several regression models were tested, with Linear Regression emerging as the top performer, surpassing a baseline model created for comparative purposes.

Notably, the model achieved a Mean Squared Error of 0.73, demonstrating superior predictive accuracy compared to the baseline model, which simply predicted the target variable's outcome based on the mean for all samples.

Key Words: *Fielding Independent Pitching, Sabermetrics, Regression, Baseball, Analytics*

Introduction:

The Problem

Predicting a pitcher's performance stands as a critical endeavor within the realm of baseball analytics, carrying profound implications for strategic, financial, and performance-related facets of the sport.

- Understanding a pitcher's ability empowers teams to optimize overall performance. Informed decisions about pitching rotation, bullpen utilization, and defensive strategies can be made based on predicted individual performance.
- When making player selections during drafts or when making decisions regarding recruitment, teams can assess a pitcher's potential, aligning talent with the team's needs and overarching strategies.
- Coaches leverage predictive models to formulate game strategies, determining optimal moments for pitcher substitutions, matchup strategies against specific hitters, and strategic substitutions to maximize the team's chances of winning.
- Given the substantial resources invested in players, predicting a pitcher's future performance aids teams in making sound financial decisions. It enables effective budget allocation, ensuring a successful financial return for money spent on player’s contracts.
- Predictive analytics not only benefit teams but also enhance the fan experience. Fans gain insights into their favorite team's pitcher performances, fostering discussions and engagement within the fan community.
- Predictive models are instrumental in player development, allowing teams to assess individual pitcher's strengths and weaknesses. This information guides coaching staff in tailoring training programs to enhance skills and address areas needing improvement.
- In a highly competitive sports environment, accurate predictions regarding a pitcher's performance provide a significant advantage. Teams leveraging data science models gain an edge over competitors when making data-based decisions.
- Predictive models play a pivotal role in fantasy baseball and sports betting. Fans and analysts rely on these predictions to make informed decisions in fantasy leagues or when placing bets on game outcomes.

Predicting a pitcher's pitching ability through data science models is not merely a statistical exercise; it is a transformative tool shaping the strategic, financial, and experiential landscape of baseball. The question arises: Can FIP be forecasted based on a player's statistics from previous years?

Literature Review:

This literature review critically examines recent research on the application of regression models to forecast FIP based on a player's statistics from previous years. By closely analyzing a range of studies, their methodologies, and findings published within the last decade, this review aims to provide nuanced insights into the challenges faced, the advancements made, and potential avenues for enhancing the accuracy of FIP predictions.

Introduction:

FIP provides a more accurate representation of a pitcher's abilities by focusing on factors that the pitcher can directly control. The primary significance of FIP lies in its attempt to isolate a pitcher's performance from the influence of defensive and other team factors, offering a more precise evaluation of their individual pitching skills. FIP is designed to evaluate a pitcher based solely on the events that involve the pitcher directly (home runs, strikeouts, walks, and hit-by-pitches.) By excluding the impact of fielding and team defense, FIP provides a clearer picture of the pitcher's contribution to the game. FIP has been shown to have predictive value in forecasting a pitcher's future performance. Pitchers with consistently low or high FIP values are likely to continue performing well or struggle, respectively, even if other traditional statistics may suggest otherwise. Accurate forecasting models for FIP provide a foundation for better decision-making across various aspects of baseball operations. From player evaluation to strategic planning, these models enhance the ability to make informed, data-driven choices that align with the goals and objectives of baseball organizations and fantasy baseball enthusiasts.

Methodologies in FIP Forecasting:

Integration of Various Kinematic Factors:

- All three studies (Howenstein, Martin, Whiteside et al.) incorporate a range of kinematic factors related to pitching, including ball speed, release consistency, pitch selection, ball movement (horizontal and lateral), and variations in these parameters.
- The consideration of multiple kinematic aspects reflects the complexity of pitching mechanics and the need for a comprehensive analysis to predict pitching success.

Regression and Machine Learning Models:

- Two of the studies (Whiteside et al., and Howenstein) employ regression models and machine learning algorithms to analyze the relationships between pitching metrics and FIP.
- Whiteside et al. use a forward stepwise multiple regression model to assess the impact of various factors on FIP, while Howenstein applies a machine learning algorithm to categorize pitchers into elite, average, or poor categories based on performance characteristics.

Prediction and Forecasting:

- The primary goal in all three studies is to predict or forecast pitching success, with FIP being a central metric. FIP is used as an indicator of a pitcher's effectiveness while trying to eliminate the impact of fielding on the pitcher's performance.

Model Limitations and Challenges:

- The studies acknowledge limitations and challenges associated with their respective models. Howenstein notes the difficulty in predicting elite or poor pitchers, and the impact of variability between trials due to randomization.

- The sample sizes in the studies (e.g., 190 pitchers in Whiteside et al. and 44 pitchers in Howenstein) are recognized as potential limitations, and suggestions are made to expand datasets for more robust predictions.

Variable Importance:

- Each study identifies specific variables that are deemed important in predicting FIP. For example, Whiteside et al. highlight pitch speed, release location variability, and variation in pitch speed as significant predictors, while Martin emphasizes factors such as maximum velocity, strike rate, and variations in vertical movement.

Practical Recommendations for Coaches and Players:

- Practical implications for coaching and player improvement are discussed in all studies. Whiteside et al. recommend optimizing ball speed, establishing consistent release locations, and incorporating varied pitch speeds. Martin suggests that pitchers looking to enhance their strikeout percentage should prioritize maximizing specific pitch movements.

Call for Further Research:

- Each study concludes with a call for additional research to uncover more contributors to success in pitching. This highlights the evolving nature of the field and the recognition that current models may not capture all relevant factors.

Conclusion:

The literature on regression models for FIP forecasting reflects a growing interest in leveraging advanced statistical techniques to enhance performance predictions in baseball. Key findings highlight the significance of incorporating multiple factors, such as pitcher skill metrics and defense-independent statistics, to create more accurate forecasts of a pitcher's FIP. Challenges identified include the complexity of capturing nuanced player performance and the need for high-quality data. Researchers emphasize the importance of refining existing models to account for contextual factors, team dynamics, and the evolving nature of player skills. Incorporating machine learning approaches and advanced analytics techniques appears to be a promising avenue for future research, aiming to improve the precision and robustness of FIP forecasting models. Additionally, the literature acknowledges the ongoing need for real-time data integration and the incorporation of novel metrics to enhance the predictive power of regression models. As baseball analytics continue to evolve, the exploration of innovative features and methodologies in FIP forecasting models is likely to be a focal point for researchers in the coming years.

Methodology:

As outlined in the **introduction**, my research aimed to address the following two inquiries:

1. Can FIP be forecasted based on a player's statistics from previous years?
2. Is it possible to develop a statistical model that outperforms the 'DummyRegressor' Baseline model across the four key model performance metrics: R-Squared (-0.002),

Mean Squared Error (0.866), Root Mean Squared Error (0.930), and Mean Absolute Error (0.736)?

Five-step approach:

1. *Data Gathering*

   Baseball statistical data spanning the years 2015 to 2023 was acquired in CSV format from two distinct online repositories, Baseball Savant and Fangraphs. Notably, the data for the year 2020 was omitted from the dataset, as the standard 162-game season was abridged to 60 games in response to the delayed commencement prompted by the Covid-19 pandemic.

2. *Data Preprocessing*

   Consolidating the information was a critical step in preparing the baseball statistical data for analysis. Notably, a pivotal step involved appending a new column for each data row, representing the FIP of the subsequent year for every player. The data refinement process also addressed missing values in specific columns and involved systematically removed non-numeric columns.

3. *Data Exploration*

   Ensuring accuracy when appending the new column to the dataset was of utmost importance. The precision of preprocessing tasks played a pivotal role in the data exploration efforts. To examine the correlation between the features and the newly added 'NextYrFIP' column, comprehensive graphs were generated. This step set the stage for the impending modeling process.

4. *Regression Modeling*

   An initial model was established as a baseline for comparative analysis with the newly developed models, incorporating all numeric data columns as features for analysis. Subsequently, specific columns demonstrating a substantial correlation with the target variable were selected to construct diverse regression models. The ensemble included Linear, K Neighbors, Decision Tree, Bagging, Random Forest, Ada Boost, Support Vector, Ridge, Gradient Boosting, Lasso, and Elastic Net Regressions.
   Following comprehensive evaluation, Linear, Support Vector, Ridge, Random Forest, and Ada Boost emerged as top performers, leading to their incorporation into refined pipelines to optimize hyperparameters. Notably, Linear Regression outperformed the others, marking a pivotal point for further assessment.

5. *Model Evaluation*

   The Linear Regression model exhibited notable performance across the four key metrics: R-Squared (0.212), Mean Squared Error (0.731), Root Mean Squared Error (0.855), and

Mean Absolute Error (0.658). In comparison to the baseline model, the Linear Regression model outperformed on all four metrics. Projections for the 2024 MLB season were generated and will be further analyzed in subsequent sections of the research paper.

Experimentation and Results:

Data Gathering and Preprocessing:

Baseball statistical data spanning from 2015 to 2023, with the exclusion of the year 2020 due to the limited game count amid the Covid-19 Pandemic, was merged from two reputable sources—Baseball Savant and Fangraphs. As mentioned earlier, the decision to omit data from 2020 was motivated by the intention to avoid drawing conclusions from a small sample size of only sixty games.

The decision to initiate the analysis from 2015 was intentional, marking the commencement of baseball's full integration with 'Statcast.' In 2015, Statcast, a cutting-edge, high-speed, and high-accuracy automated tool, was introduced across all thirty MLB stadiums, signaling the beginning of the Statcast era. This revolutionary technology employs doppler radar and high-definition video to meticulously measure player speed, acceleration, and various other aspects of play on the field. It was in 2015 that official records of the innovative statistics generated by Statcast were consistently maintained, establishing these novel metrics as essential components that demanded thorough analysis and consideration.

last_name, first_name	player_id	year	player_age	p_game	p_formatted_ip	pa	ab	hit	single	...	n_offspeed_formatted	offspeed_avg_speed	offspeed_avg_spin	offspeed_avg_break_x
Colon, Bartolo	112526	2015	42	33	194.2	815	771	217	149	...	7.4	82.6	1727.0	-13.2
Burnett, A.J.	150359	2015	38	26	164.0	699	633	174	134	...	8.8	86.3	1678.0	-10.8
Hudson, Tim	218596	2015	39	24	123.2	525	476	134	103	...	11.2	80.9	1369.0	-9.1
Benoit, Joaquin	276542	2015	37	67	65.1	254	226	36	19	...	34.4	84.7	1434.0	-12.4
Buehrle, Mark	279824	2015	36	32	198.2	827	768	214	140	...	21.1	78.7	1625.0	14.1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Kelly, Kevin	687330	2023	25	57	67.0	276	251	53	33	...	NaN	NaN	NaN	NaN
Perez, Eury	691587	2023	20	19	91.1	374	337	72	33	...	9.8	89.7	1766.0	-14.1
Woo, Bryan	693433	2023	23	18	87.2	371	331	75	48	...	3.6	89.7	1700.0	-14.3
Elder, Bryce	693821	2023	24	31	174.2	732	654	160	106	...	13.7	85.0	2070.0	-11.2
Pfaadt, Brandon	694297	2023	24	19	96.0	421	386	109	59	...	12.1	86.7	1954.0	-15.8

The next phase involved developing a function tailored to handle a dataset with rows corresponding to individual pitcher's years (e.g., Matt Harvey in 2015). This function iterates through the rows, creating a new 'NextYrFIP' column by referencing the FIP value of the following year. In instances where there is no available data for the subsequent year, the function sets 'NextYrFIP' to NaN.

An illustration of Noah Syndergaard's statistics post-function implementation:

last_name, first_name	player_id	year	FIP	NextYrFIP
Syndergaard, Noah	592789	2015	3.246933	2.286317
Syndergaard, Noah	592789	2016	2.286317	NaN
Syndergaard, Noah	592789	2018	2.804058	3.598495
Syndergaard, Noah	592789	2019	3.598495	NaN
Syndergaard, Noah	592789	2022	3.832727	6.198651
Syndergaard, Noah	592789	2023	6.198651	NaN

A couple of important points to note: The 'NextYrFIP' is missing for every player's 2019 season due to the intentional exclusion of the 2020 season. Additionally, for each player's 2023 season, there is no 'NextYrFIP' as the 2024 season has yet to occur.

Managing multicollinearity was crucial in the analysis. Columns exhibiting high correlations (>99%) with each other were consolidated into a single column. For example, it was deemed unnecessary to retain both the total pitches and total strikes thrown by a pitcher in the dataframe.

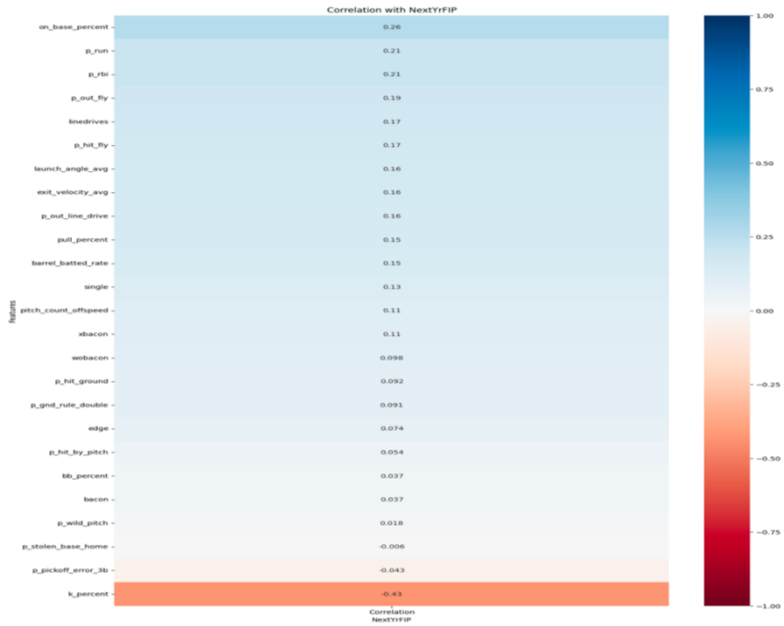
Next to be addressed was missing data. My initial step involved the removal of columns with more than 10% of missing data. This set of data encompasses statistics related to a pitcher's cutter, sinker, splitter, curveball, slider, and knuckleball. Given that not every pitcher utilizes these pitches, opting for imputation in such cases would entail fabricating substantial information.

Moving forward, attention turned to fourteen columns with less than ten percent of missing data, implementing mean imputation to handle these gaps. Mean imputation, a straightforward and widely adopted technique, involves replacing missing values with the mean of observed (non-missing) values for a specific variable or column. This approach serves as an effective strategy for addressing missing data in a simplified manner.

In the concluding phases of preparation, the data underwent scaling, and MinMaxScaler was utilized to normalize the numerical features within the dataset, constraining them to a range between 0 and 1. This step proved essential because certain machine learning algorithms exhibit sensitivity to the scale of input features. Notably, distance-based algorithms like k-nearest neighbors and support vector machines can be significantly impacted by the magnitude of feature values. By aligning the features to a unified range, the performance of these algorithms is notably enhanced.

Data Exploration:

A heatmap was generated to explore the correlations between various features and the target variable, 'NextYrFIP.' The code encompassed more than 170 features, each with its corresponding correlation coefficients with the target variable. Here, I offer a concise visualization featuring 25 randomly selected features to convey the essence of the analysis.



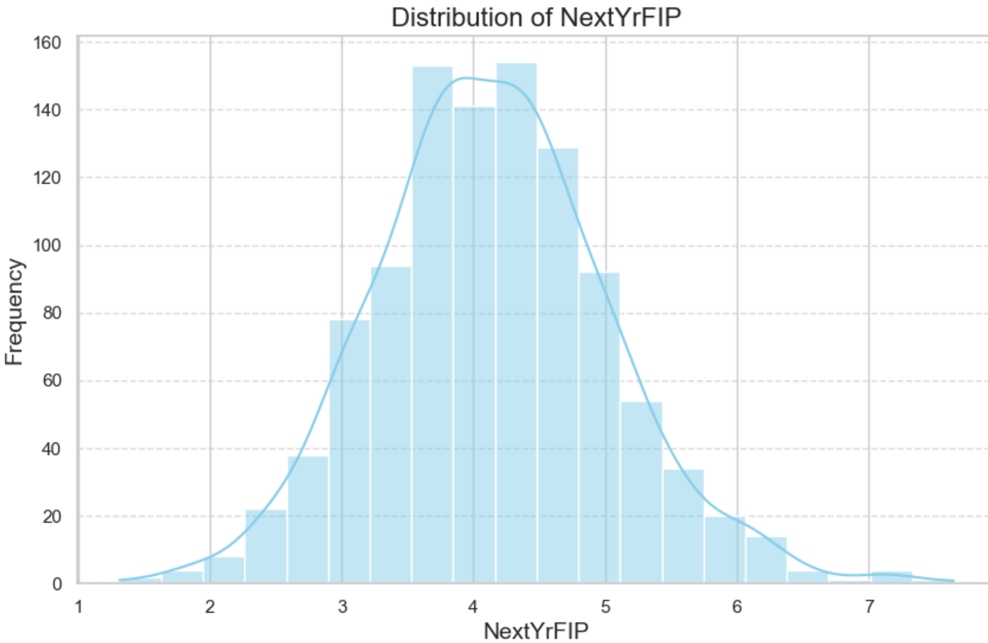
Examining the distribution of the target variable, 'NextYrFIP,' is crucial in regression modeling. Regression models typically assume that the target variable adheres to a specific distribution, and a thorough understanding of this distribution aids in selecting appropriate regression models. Some models are better suited for specific distribution types; for instance, those designed for normally distributed data may not perform optimally when the target variable exhibits skewness.

The presence of anomalies or outliers within the target variable's distribution can significantly impact model performance. Outliers have the potential to distort predictions and influence the coefficients within the regression equation. Identifying and managing outliers carefully is crucial for constructing robust and accurate models.

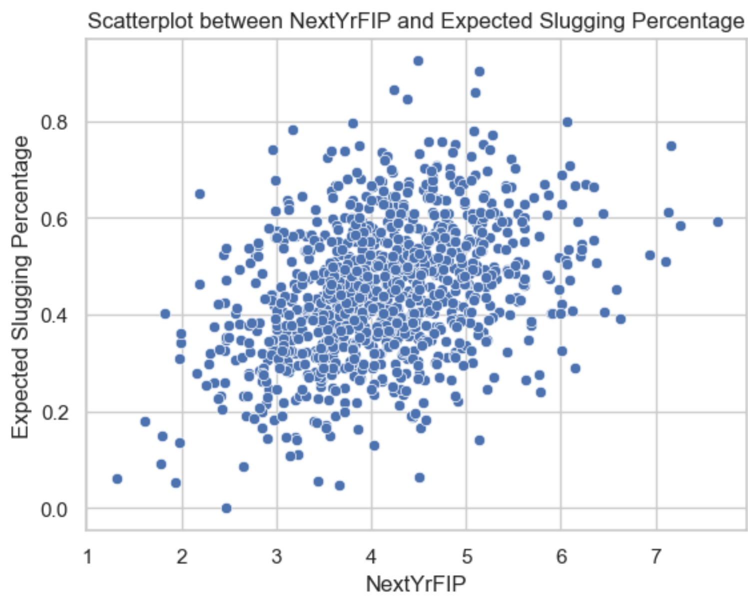
Moreover, understanding the distribution of the target variable, guides decisions related to feature engineering. If the target variable shows a distinct non-linear relationship with predictors, including polynomial features or interaction terms may be beneficial. Additionally, the distribution plays a pivotal role in determining the interpretability of model results. For cases where the target variable has a wide range, standardizing or normalizing the data becomes advantageous for easy comparability of coefficient values.

The risk of introducing bias into the model is associated with skewed or imbalanced distributions of the target variable. Such scenarios may make the model more sensitive to specific data patterns, potentially leading to biased predictions. Hence, a comprehensive understanding of the target variable distribution is fundamental for making informed decisions throughout the regression modeling process.

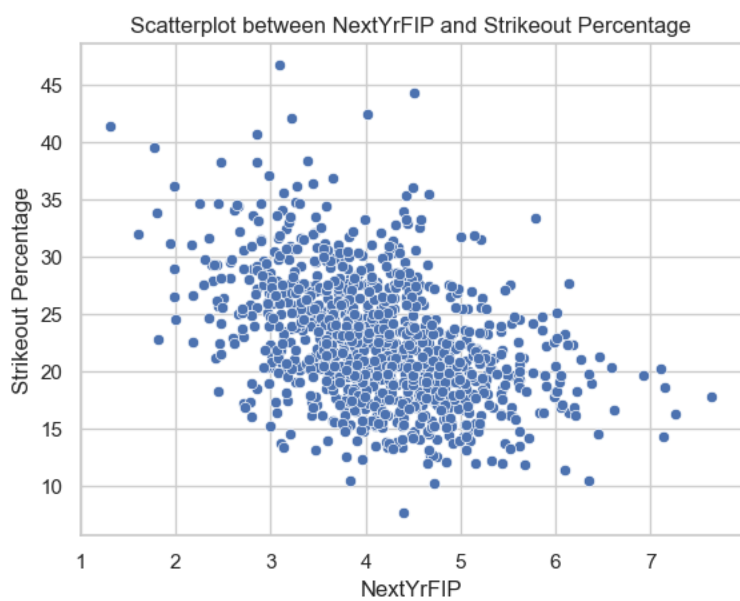
Fortunately, as depicted below, the distribution of the target variable appears to follow an approximate normal pattern, with no notable outliers present. The average value of the target variable hovers around 4.13, and the proximity of the median to the mean implies a roughly symmetric distribution. With a standard deviation of 0.887, the values exhibit a moderate dispersion around the mean. Additionally, the skewness, measuring at 0.25, indicates a subtle rightward skew, but it is close to zero, further signifying a relatively balanced distribution.



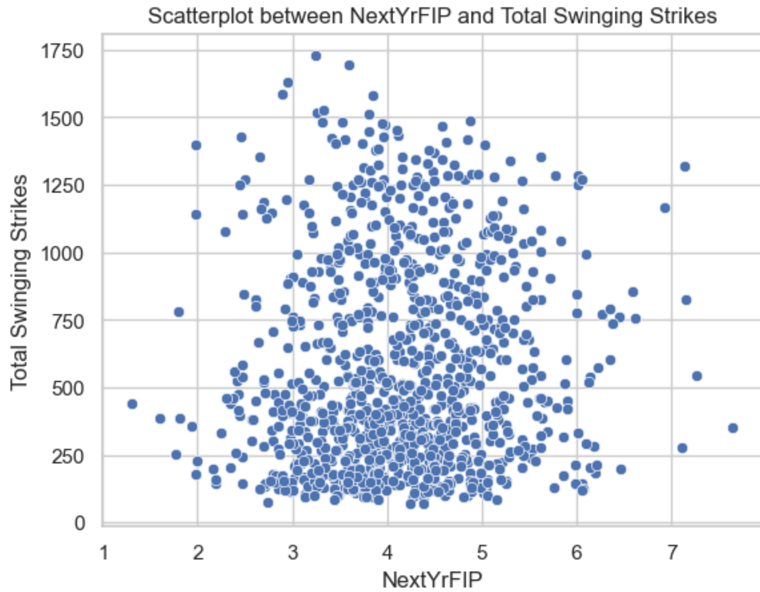
While not a perfect correlation, one of the features utilized in the modeling process is 'Expected Slugging Percentage.' This metric is computed based on factors such as exit velocity, launch angle, and, in the case of specific types of batted balls, Sprint Speed. Below, there is an observable positive correlation between xwOBA and 'NextYrFIP.'



Moving on to the statistical measure known as Strikeout Percentage, which, while not perfect, also exhibits a correlation with 'NextYrFIP.' Strikeout Percentage expresses the percentage of batters faced by the pitcher that result in strikeouts. The distinguishing factor this time is the presence of a noticeable negative correlation.



Having observed the impact of both positive and negative correlations with 'NextYrFIP,' let's complete the picture by delving into a statistic omitted from our models. This exclusion is due to the absence of any correlation between 'NextYrFIP' and the statistic. Total Swinging Strikes, representing the pitches resulting in a swing and a miss by a pitcher, though potentially valuable for pitcher assessment, is not deemed crucial for our modeling endeavors.



Regression Modeling:

Before proceeding with the modeling phase, the initial step involved creating a quick and easy 'Baseline Model' for comparison with the new, more robust models. All numeric features were selected as the features to predict 'NextYrFIP.' Using 'DummyRegressor' from scikit-learn, with a strategy of predicting the mean of 'NextYrFIP,' a model was built, and the following four model performance scores were exhibited: R-Squared: -0.002, Mean Squared Error: 0.866, Root Mean Squared Error: 0.930, and Mean Absolute Error: 0.736. Now that we had the four scores from the baseline model, our goal was to surpass these metrics with our new, more robust models.

The next step involved determining the features to be utilized in predicting the target variable, 'NextYrFIP.' I systematically identified all numeric features exhibiting an absolute correlation with the target variable ('NextYrFIP') surpassing the predefined threshold of 0.2. This process was crucial for selecting relevant features with meaningful associations to the target variable. Once more, to mitigate multicollinearity within the chosen features, those exhibiting excessive correlation with each other were consolidated into a single feature. Here is the list of the selected features:

Feature	***Correlation to 'NextYrFIP'***	***Feature Explanation***
Expected Slugging Percentage	0.38	Estimate of a player's slugging percentage based on the quality of contact (exit velocity and launch angle) rather than the actual outcomes

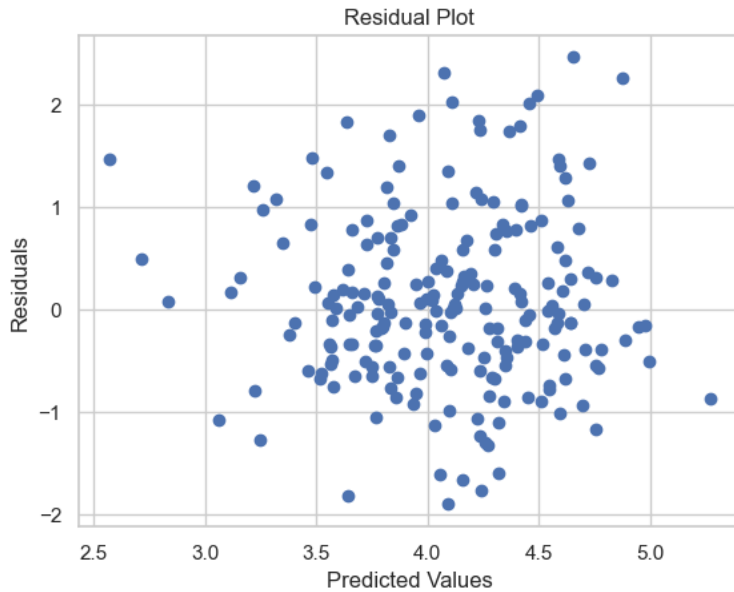
Fielding Independent Pitching	0.37	Evaluation of a pitcher's performance while focusing on the events a pitcher can control
Expected Batting Average	0.37	Estimate of a player's batting average based on the quality of their batted ball contact
Expected On Base Percentage	0.33	Estimate of a player's ability to get on base
Slugging Percentage	0.31	Measure of a player's power and ability to hit for extra bases
Batting Average	0.28	Measure of a player's success in getting hits during their plate appearances
On Base Percentage	0.26	Measure of a player's ability to reach base safely (various ways a player can reach base, not just hits)
Isolated Power	0.25	Measure of a hitter's raw power by quantifying the extra bases a player achieves per at-bat
Barrels Allowed	0.22	Batted balls with the perfect combination of exit velocity and launch angle allowed
Earned Runs Allowed	0.21	Runs that a pitcher allows that are scored without the aid of defensive errors or other defensive miscues
Popups Allowed	0.2	Number of times a pitcher has allowed batters to hit a batted ball that goes very high in the air but doesn't travel a significant distance horizontally
Games Finished	-0.2	Times one is last pitcher to pitch for his team in a given game
Fastball Average Spin Rate	-0.21	The average of how quickly a baseball rotates in revolutions per minute (RPM) when a fastball is thrown
Offspeed Average Speed	-0.21	Average velocity of a pitcher's offspeed pitches (Offspeed pitches include a variety of slower pitches that are intended to deceive hitters by disrupting their timing such as changeups, curveballs, sliders, etc.)
Games Pitched	-0.22	Number of games in which a pitcher has participated by throwing at least one pitch
Fastball Average Speed	-0.28	Average velocity of a pitcher's fastball
Inside-Zone Swing Miss Percentage	-0.3	Percentage of pitches swung at inside the strikezone
Outside-Zone Swing Miss Percentage	-0.33	Percentage of pitches swung at outside the strikezone
Whiff Rate Percentage	-0.36	Percentage of swings by batters that result in a swinging strike
Strikeout Percentage	-0.43	Percentage of plate appearances by the hitter that result in a strikeout

To effectively model and assess performance, a train-test split was conducted, dividing the data into an 80% segment for training the model and a remaining 20% for evaluating its effectiveness. Subsequently, a comprehensive evaluation function was developed to assess the performance of any regression model. This function computes two vital metrics, namely Root Mean Squared Error and R-squared, offering a robust assessment. It accepts a regression model along with sets of training and testing input features and corresponding target variable values. The following outcomes are derived from the chosen regressors, including Linear, K Neighbors, Decision Tree, Bagging, Random Forest, Ada Boost, Support Vector, Ridge, Gradient Boosting, Lasso, and Elastic Net Regressions.

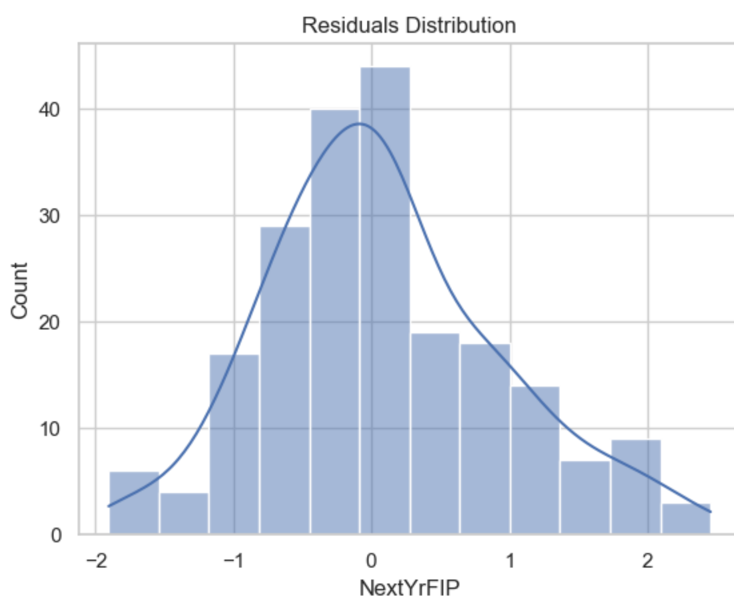
Regressor	Train_RMSE	Test_RMSE	Train_R2	Test_R2
LinearRegression	0.738186	0.854740	0.273785	0.212212
SVR	0.670078	0.857677	0.401609	0.206789
Ridge	0.740747	0.865601	0.268736	0.192064
RandomForestRegressor	0.289885	0.873577	0.888008	0.177105
AdaBoostRegressor	0.672323	0.887904	0.397594	0.149892
GradientBoostingRegressor	0.512784	0.888981	0.649569	0.147829
BaggingRegressor	0.320800	0.909306	0.862848	0.108417
KNeighborsRegressor	0.682953	0.919035	0.378393	0.089237
Lasso	0.866230	0.963616	0.000000	-0.001267
ElasticNet	0.866230	0.963616	0.000000	-0.001267
DecisionTreeRegressor	0.000000	1.149203	1.000000	-0.424084

As Linear Regression, SVR, Ridge, Random Forest, and Ada Boost demonstrated superior performance in terms of test RMSE and test R-squared, individualized pipelines were constructed for each regressor to optimize hyperparameters for each model. Following this comprehensive effort, the Linear Regression model emerged as the top performer, showcasing notable excellence across the four key metrics: R-Squared (0.212), Mean Squared Error (0.731), Root Mean Squared Error (0.855), and Mean Absolute Error (0.658).

Model Evaluation:



The scatterplot reveals residuals that exhibit a notably random pattern, demonstrating a consistent distribution around zero. The residual spread remains approximately constant across various predicted values, indicating desirable homoscedasticity. While a few outliers are present, they do not raise significant concerns. Overall, the graph suggests that the model performed admirably overall.



The histogram substantiates our observations from the scatterplot. The distribution of residuals is predominantly symmetrical, exhibiting a bell-shaped curve centered around zero. This characteristic alignment implies a strong concordance between the model's predictions and the true values. The visual coherence between the scatterplot and the histogram underscores the model's reliability in capturing the underlying patterns in the data.

2024 MLB Season FIP Predictions:

Employing the proficiently trained Linear Regression model, predictions were generated based on the statistical data for MLB pitchers' seasons in 2023. The outcomes are as follows:

Top 25:

Rank	Pitcher Name	Projected_FIP
1	Duran, Jhoan	2.92
2	Scott, Tanner	2.94
3	Strider, Spencer	3.11
4	Skubal, Tarik	3.11
5	Hicks, Jordan	3.21
6	Kimbrel, Craig	3.23
7	Glasnow, Tyler	3.23
8	Minter, A.J.	3.31
9	Bummer, Aaron	3.37
10	Pressly, Ryan	3.37
11	Woodruff, Brandon	3.40
12	Holmes, Clay	3.41
13	Soto, Gregory	3.42
14	Greene, Hunter	3.43
15	Fried, Max	3.46
16	Doval, Camilo	3.47
17	Jax, Griffin	3.49
18	Lopez, Pablo	3.49
19	Strahm, Matt	3.52
20	Clase, Emmanuel	3.53
21	Alzolay, Adbert	3.53
22	King, Michael	3.57
23	Richards, Trevor	3.58
24	Pivetta, Nick	3.58
25	Keller, Mitch	3.61

Bottom 25:

Rank	Pitcher Name	Projected_FIP
175	Syndergaard, Noah	4.63
176	Perez, Martin	4.63
177	Urquidy, Jose	4.64
178	Flexen, Chris	4.67
179	Corbin, Patrick	4.67
180	Mikolas, Miles	4.67
181	Manoah, Alek	4.67
182	Chirinos, Yonny	4.67
183	Javier, Cristian	4.69
184	Hill, Rich	4.72
185	Freeland, Kyle	4.74
186	Anderson, Tyler	4.75
187	Hendricks, Kyle	4.78
188	Miley, Wade	4.80
189	Gonsolin, Tony	4.81
190	Gray, Josiah	4.81
191	Williams, Trevor	4.82
192	Quantrill, Cal	4.84
193	Gomber, Austin	4.86
194	Sears, JP	4.87
195	Lyles, Jordan	4.89
196	Manning, Matt	4.90
197	Hudson, Dakota	4.90
198	Wainwright, Adam	5.00
199	Kluber, Corey	5.04

Conclusions and Next Steps:

What was Learned:

Our initial objective revolved around addressing two fundamental questions:

1. Can FIP be forecasted based on a player's statistics from previous years?

2. Is it possible to develop a statistical model that outperforms the 'DummyRegressor' Baseline model across the four key model performance metrics: R-Squared (-0.002), Mean Squared Error (0.866), Root Mean Squared Error (0.930), and Mean Absolute Error (0.736)?

I effectively predicted FIP by analyzing players' statistics from previous years. The model I constructed demonstrated superior performance across all four key metrics when compared to the baseline model.

Model	R-Squared	Mean Squared Error	Root Mean Squared Error	Mean Absolute Error
Dummy Regressor (Baseline)	-0.002	0.866	0.930	0.736
Linear Regression	0.212	0.731	0.855	0.658

R-Squared gauges the extent to which the model accounts for the variability in the response variable. A higher R-Squared signifies increased efficacy in capturing variability within the Linear Regression model. Meanwhile, Mean Squared Error computes the average squared disparity between predicted and actual values. A diminished Mean Squared Error underscores the Linear Regression model's propensity for closer average proximity to actual values. Root Mean Squared Error, as the square root of Mean Squared Error, mirrors the standard deviation of residuals. A lower Root Mean Squared Error reinforces that the Linear Regression model's predictions, on average, are in closer agreement with actual values. Lastly, Mean Absolute Error quantifies the average absolute difference between predicted and actual values, with a lower value indicating superior accuracy in the Linear Regression model's predictions.

Highlighting the inherent difficulty in forecasting pitching performance from one year to the next is crucial. This awareness guided my decision to predict FIP rather than the more commonly used metric, Earned Run Average (ERA). FIP provides an estimate of pitching performance independent of defensive performance and luck. In contrast, ERA quantifies the average number of runs a pitcher concedes per nine innings. FIP provides a more focused view of a pitcher's individual performance by considering factors directly within their control. This makes FIP more consistent and less susceptible to external influences compared to ERA, which considers a broader range of team-related factors. Given that the fielding-independent metrics incorporated into FIP's formula demonstrate greater year-to-year stability compared to ERA, FIP exhibits a more consistent trend over time. Consequently, owing to its reduced variability, FIP emerges as a superior estimator for forecasting future pitching performance.
In general, predicting pitching performance in Major League Baseball from one year to the next is challenging due to several factors. Injuries are common in baseball, and a pitcher's health can change from season to season. Injuries can affect a pitcher's mechanics, velocity, and overall effectiveness. Issues like Tommy John surgery, rotator cuff injuries, and elbow strains can occur due to the repetitive and high-impact nature of pitching putting significant stress on a pitcher’s dominant arm. Pitchers are constantly tinkering with their repertoire, or approach from one season to the next. Player development and adjustments to the game can impact performance. One of the most unpredictable aspects of pitching lies in the fact that a pitcher can typically execute a pitch—or even a series of pitches—with precision and expertise, yet success is not guaranteed. The inherent randomness in baseball becomes evident when, despite a pitcher's skillful delivery, a hitter prevails in a given at-bat. In these instances, the dynamics between the

pitcher and the hitter, influenced by split-second decisions, timing, and strategic choices, play a pivotal role in determining the outcome. The element of unpredictability underscores the complexity of the pitcher-hitter interaction, where even optimal execution by the pitcher does not always ensure victory in the ongoing battle between the two players. Forecasting pitching performance proves to be a challenging endeavor.

The R-squared value of 0.212 serves as an indicator of the extent to which the model, built upon a pitcher's prior year statistics, explains approximately 21% of the variability in predicting FIP. This numeric representation unveils that while the model provides some insight into FIP, a substantial portion of the variability remains unaccounted for. The implication is that additional factors, not encompassed by the pitcher's previous year's statistics, contribute significantly to the observed fluctuations in FIP. This realization is pivotal, as it underscores the complex nature of the relationship between a pitcher's performance and FIP, hinting at the presence of hidden variables or nuanced dynamics that extend beyond the scope of the current model. While the achieved R-squared value signifies progress compared to a baseline model with an R-squared of -0.002, denoting a positive advancement, the relatively modest value prompts further exploration. The need for refinement becomes apparent, and avenues for improvement may involve delving into supplementary features that capture previously unconsidered aspects of a pitcher's performance or adopting more sophisticated modeling techniques. Acknowledging the limitations of the current model is crucial for developing a more comprehensive understanding of the multifaceted factors influencing FIP, ultimately paving the way for a more robust and accurate predictive framework.

Areas for Future Research:

1. Feature Engineering:

- Investigate the evolution of each pitcher's pitch repertoire to gain insights into performance.
- Analyze the spatial dimension of pitching, considering pitch locations and effectiveness in specific zones within the strike zone.
- Develop metrics incorporating a pitcher's performance in various game situations, such as high-leverage scenarios or late-game situations.
- Explore the impact of pressure situations on a pitcher's performance.
- Study pitch sequencing and how pitchers adapt based on the count or batter.
- Develop metrics for evaluating a pitcher's consistency over time, using rolling averages to smooth short-term fluctuations.

2. Temporal Analysis:

- Investigate seasonal trends, career trajectories, and year-to-year variability in a pitcher's performance.
- Examine distinctions between early and late-season performances.
- Break down the season into smaller segments to identify evolving player conditions and strategies.
- Analyze the impact of injuries, trades, or team changes on FIP dynamics.
- Assess pitcher aging curves, sequential performance patterns, and historical comparisons for nuanced forecasting models.

3. Injury Data:
- Incorporate injury data into the analysis to understand its impact on a player's performance.
- Examine the timing, duration, and nature of injuries.
- Analyze the performance upon returning from injury to uncover insights into the rehabilitation process.
- Consider injury-related variables in forecasting FIP to provide a comprehensive view of a player's resilience.

4. Pitching Mechanics and Biomechanics:
- Delve into pitching mechanics and biomechanics to understand their contribution to performance outcomes.
- Examine a pitcher's throwing motion, release point, and body mechanics.
- Incorporate biomechanical data for a nuanced evaluation of mechanical factors influencing FIP.

5. Psychological Factors:
- Include psychological factors such as mental resilience, confidence, and concentration levels in the analysis.
- Explore how pitchers respond to high-pressure situations and adapt to changes in team dynamics.
- Consider the role of mental fortitude in the game to understand its impact on consistency and performance outcomes.

6. Dynamic Models:
- Employ dynamic modeling techniques, such as recurrent neural networks (RNNs) or comparable approaches, to construct predictive frameworks capable of adapting and evolving over time. Unlike static models, dynamic models consider the fluid nature of a player's career and performance trajectory.
- Capture the unfolding patterns in player statistics, enabling the FIP forecasting model to promptly respond to shifts in a player's form and strategic approaches.

7. Validation and External Testing:
- Conduct rigorous validation on different datasets or seasons not used during training to assess model generalizability.
- Perform external testing on entirely independent datasets or different baseball leagues to evaluate the model's performance in varied contexts.
- Ensure the model's reliability and credibility for real-world applications through thorough validation and external testing.

Bibliography:

1. MLB Glossary - Fielding Independent Pitching (FIP). [Link](https://www.mlb.com/glossary/advanced-stats/fielding-independent-pitching)

2. "What to Know About Fielding Independent Pitching (FIP) Metric in MLB." Beacon Journal. [Link](https://www.beaconjournal.com/story/sports/mlb/cleveland-guardians/2022/08/14/what-to-know-about-fielding-independent-pitching-fip-metric-in-mlb/65401149007/)

3. Fangraphs Library - Pitching: Fielding Independent Pitching (FIP). [Link](https://library.fangraphs.com/pitching/fip/)

4. ESPN - MLB Stat Definition: What is FIP? [Link](https://www.espn.com/blog/statsinfo/post/_/id/62051/mlb-stat-definition-what-is-fip)

5. Baseball Savant - MLB Stat Leaderboard (Custom). [Link](https://baseballsavant.mlb.com/leaderboard/custom)

6. "Statistically Significant: Understanding FIP." Athletics Nation. [Link](https://www.athleticsnation.com/2010/4/27/1446531/statistically-significant-fip)

7. scikit-learn Documentation - Linear Regression. [Link](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

8. Real Python - Linear Regression in Python. [Link](https://realpython.com/linear-regression-in-python/)

9. "Simple Linear Regression Analysis Using Python." Medium - Geek Culture. [Link](https://medium.com/geekculture/simple-linear-regression-analysis-using-python-c5b2f637942)

10. GeeksforGeeks - Linear Regression Python Implementation. [Link](https://www.geeksforgeeks.org/linear-regression-python-implementation/)

11. Whiteside, David1,2,3; Martini, Douglas N.1; Zernicke, Ronald F.1,4,5; Goulet, Grant C.1. Ball Speed and Release Consistency Predict Pitching Success in Major League Baseball. Journal of Strength and Conditioning Research 30(7):p 1787-1795, July 2016. | DOI: 10.1519/JSC.0000000000001296

12. Engineered Athletics. (October 10, 2017). "Can We Predict the Success of a Starting Pitcher with Machine Learning Using Statcast Pitching Data?" Engineered Athletics. [Link] (https://engineeredathletics.com/2017/10/10/can-we-predict-the-success-of-a-starting-pitcher-with-machine-learning-using-statcast-pitching-data/)

13. Eric P. Martin (2019). Predicting Major League Baseball Strikeout Rates Update. Retrieved from [Link] (https://assets-global.website-files.com/5f1af76ed86d6771ad48324b/5f6d38971aa75c2f6af77911_Predicting-Major-League-Baseball-Strikeout-Rates-Update.pdf)

Appendix With Code:

MLBProject

December 11, 2023

1 MLB 'Fielding Independent Pitching' Prediction

1.1 Mathew Katz

1.1.1 December 12, 2023

[ ]:
```
# Importing necessary libraries
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import Ridge, Lasso, LinearRegression, ElasticNet
from sklearn.ensemble import BaggingRegressor, RandomForestRegressor,␣
↪AdaBoostRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.dummy import DummyRegressor

# Set working directory
os.chdir('/Users/mathewkatz/Desktop/CunySPS/DATA698')

# Set display options
pd.set_option('display.max_rows', None)
pd.reset_option('display.max_columns')
```

[ ]:
```
# Read CSV files
stats = pd.read_csv('but2020.csv')
fip = pd.read_csv('FIPs.csv')

# Create a new column 'FIPID' by combining columns
fip['FIPID'] = fip['Season'].astype(str) + fip['MLBAMID'].astype(str)
stats['FIPID'] = stats['year'].astype(str) + stats['player_id'].astype(str)
```

```python
# Merge DataFrames on 'FIPID' column
stats = pd.merge(stats, fip[['FIPID', 'FIP']], on='FIPID', how='left')

stats.head()
```

[ ]:
```
  last_name, first_name  player_id  year  player_age  p_game  p_formatted_ip  \
0        Colon, Bartolo     112526  2015          42      33           194.2
1         Burnett, A.J.     150359  2015          38      26           164.0
2           Hudson, Tim     218596  2015          39      24           123.2
3       Benoit, Joaquin     276542  2015          37      67            65.1
4         Buehrle, Mark     279824  2015          36      32           198.2

    pa   ab  hit  single  …  n_offspeed_formatted  offspeed_avg_speed  \
0  815  771  217     149  …                   7.4                82.6
1  699  633  174     134  …                   8.8                86.3
2  525  476  134     103  …                  11.2                80.9
3  254  226   36      19  …                  34.4                84.7
4  827  768  214     140  …                  21.1                78.7

   offspeed_avg_spin  offspeed_avg_break_x  offspeed_avg_break_z  \
0             1727.0                 -13.2                   NaN
1             1678.0                 -10.8                   NaN
2             1369.0                  -9.1                   NaN
3             1434.0                 -12.4                   NaN
4             1625.0                  14.1                   NaN

   offspeed_avg_break  offspeed_range_speed  Unnamed: 249       FIPID  \
0                16.4                   2.4           NaN  2015112526
1                12.0                   1.8           NaN  2015150359
2                10.3                   1.4           NaN  2015218596
3                13.1                   1.4           NaN  2015276542
4                16.2                   1.7           NaN  2015279824

        FIP
0  3.837366
1  3.359210
2  4.532526
3  3.745845
4  4.261115

[5 rows x 252 columns]
```

[ ]:
```python
def add_next_year_fip(df):
    # Check if 'player_id' column exists in the DataFrame
    if 'player_id' in df.columns:
        # Count occurrences of each player_id
```

```
        player_counts = df['player_id'].value_counts()

        # Create a DataFrame with only rows where player_id occurs more than␣
↪once
        filtered_df = df[df['player_id'].isin(player_counts[player_counts > 1].
↪index)]

        # Check if there are any duplicates
        if not filtered_df.empty:
            # Sort DataFrame by 'player_id' and 'year'
            filtered_df = filtered_df.sort_values(by=['player_id', 'year'])

            # Create a new column 'NextYrFIP' and initialize with NaN
            filtered_df['NextYrFIP'] = float('nan')

            # Set the index to 'player_id' and 'year' for efficient iteration
            filtered_df.set_index(['player_id', 'year'], inplace=True)

            # Iterate through rows to fill 'NextYrFIP' based on the next year's␣
↪FIP
            for index, row in filtered_df.iterrows():
                player_id, current_year = index
                if (player_id, current_year + 1) in filtered_df.index:
                    next_year_fip = filtered_df.loc[(player_id, current_year +␣
↪1), 'FIP']
                    filtered_df.at[index, 'NextYrFIP'] = next_year_fip
                else:
                    # If no next year data, set 'NextYrFIP' to NaN
                    filtered_df.at[index, 'NextYrFIP'] = float('nan')

            # Reset the index to the default integer index
            filtered_df.reset_index(inplace=True)

            # Return the DataFrame with 'NextYrFIP' column added
            return filtered_df
        else:
            # If no duplicates found, print a message and return the original␣
↪DataFrame
            print("No duplicate player_id values in the DataFrame.")
            return df
    else:
        # If 'player_id' column is not present, print a message and return None
        print("Invalid DataFrame")
        return None
```

[ ]:
```
stats = add_next_year_fip(stats)
stats.head()
```

[ ]:
```
   player_id  year last_name, first_name  player_age  p_game  p_formatted_ip  \
0     112526  2015        Colon, Bartolo          42      33           194.2
1     112526  2016        Colon, Bartolo          43      34           191.2
2     112526  2017        Colon, Bartolo          44      28           143.0
3     112526  2018        Colon, Bartolo          45      28           146.1
4     282332  2015          Sabathia, CC          34      29           167.1

    pa   ab  hit  single  …  offspeed_avg_speed  offspeed_avg_spin  \
0  815  771  217     149  …                82.6             1727.0
1  791  745  200     134  …                81.4             1683.0
2  648  603  192     110  …                81.3             1678.0
3  628  591  172      95  …                80.8             1657.0
4  726  659  188     134  …                83.9             1950.0

   offspeed_avg_break_x  offspeed_avg_break_z  offspeed_avg_break  \
0                 -13.2                   NaN                16.4
1                 -13.2                 -32.0                16.2
2                 -14.3                 -33.6                18.1
3                 -13.8                 -36.1                16.1
4                  11.1                   NaN                15.0

   offspeed_range_speed  Unnamed: 249       FIPID       FIP  NextYrFIP
0                   2.4           NaN  2015112526  3.837366   3.986571
1                   1.5           NaN  2016112526  3.986571   5.213584
2                   1.3           NaN  2017112526  5.213584   5.470231
3                   1.4           NaN  2018112526  5.470231        NaN
4                   1.2           NaN  2015282332  4.675436   4.282004

[5 rows x 253 columns]
```

Checking to see if function worked as intended:

[ ]:
```python
# Selecting rows where the player's last name and first name match␣
↪'Syndergaard, Noah'
Syndergaard = stats[stats['last_name, first_name'] == 'Syndergaard, Noah']

# Sorting the selected player's data based on the 'year' column
Syndergaard = Syndergaard.sort_values(by='year')

# Defining a list of columns to be selected from the player's DataFrame
selected_columns = ['last_name, first_name', 'player_id', 'year', 'FIP',␣
↪'NextYrFIP']

# Displaying only the specified columns for the player
Syndergaard[selected_columns]
```

[ ]:

	last_name, first_name	player_id	year	FIP	NextYrFIP
1074	Syndergaard, Noah	592789	2015	3.246933	2.286317
1075	Syndergaard, Noah	592789	2016	2.286317	NaN
1076	Syndergaard, Noah	592789	2018	2.804058	3.598495
1077	Syndergaard, Noah	592789	2019	3.598495	NaN
1078	Syndergaard, Noah	592789	2022	3.832727	6.198651
1079	Syndergaard, Noah	592789	2023	6.198651	NaN

[ ]:

```
# Selecting rows where the player's last name and first name match 'Greinke,␣
↪Zack'
Greinke = stats[stats['last_name, first_name'] == 'Greinke, Zack']

# Sorting the selected player's data based on the 'year' column
Greinke = Greinke.sort_values(by='year')

# Displaying only the specified columns for the player
Greinke[selected_columns]
```

[ ]:

	last_name, first_name	player_id	year	FIP	NextYrFIP
32	Greinke, Zack	425844	2015	2.760846	4.117156
33	Greinke, Zack	425844	2016	4.117156	3.305911
34	Greinke, Zack	425844	2017	3.305911	3.704571
35	Greinke, Zack	425844	2018	3.704571	3.218802
36	Greinke, Zack	425844	2019	3.218802	NaN
37	Greinke, Zack	425844	2021	4.713887	4.032138
38	Greinke, Zack	425844	2022	4.032138	4.744502
39	Greinke, Zack	425844	2023	4.744502	NaN

[ ]:

```
# Selecting rows where the player's last name and first name match 'deGrom,␣
↪Jacob'
deGrom = stats[stats['last_name, first_name'] == 'deGrom, Jacob']

# Sorting the selected player's data based on the 'year' column
deGrom = deGrom.sort_values(by='year')

# Displaying only the specified columns for the player
deGrom[selected_columns]
```

[ ]:

	last_name, first_name	player_id	year	FIP	NextYrFIP
1139	deGrom, Jacob	594798	2015	2.704281	3.322246
1140	deGrom, Jacob	594798	2016	3.322246	3.500356
1141	deGrom, Jacob	594798	2017	3.500356	1.985315
1142	deGrom, Jacob	594798	2018	1.985315	2.674794
1143	deGrom, Jacob	594798	2019	2.674794	NaN
1144	deGrom, Jacob	594798	2021	1.235248	NaN

```
[ ]: # Selecting rows where the player's last name and first name match 'Corbin,␣
     ↪Patrick'
     Corbin = stats[stats['last_name, first_name'] == 'Corbin, Patrick']

     # Sorting the selected player's data based on the 'year' column
     Corbin = Corbin.sort_values(by='year')

     # Displaying only the specified columns for the player
     Corbin[selected_columns]
```

```
[ ]:    last_name, first_name  player_id  year       FIP  NextYrFIP
     896        Corbin, Patrick     571578  2015  3.345365   4.836081
     897        Corbin, Patrick     571578  2016  4.836081   4.075039
     898        Corbin, Patrick     571578  2017  4.075039   2.470430
     899        Corbin, Patrick     571578  2018  2.470430   3.486287
     900        Corbin, Patrick     571578  2019  3.486287        NaN
     901        Corbin, Patrick     571578  2021  5.406919   4.835133
     902        Corbin, Patrick     571578  2022  4.835133   5.277262
     903        Corbin, Patrick     571578  2023  5.277262        NaN
```

```
[ ]: # Selecting rows where the player's last name and first name match 'Colon,␣
     ↪Bartolo'
     Colon = stats[stats['last_name, first_name'] == 'Colon, Bartolo']

     # Sorting the selected player's data based on the 'year' column
     Colon = Colon.sort_values(by='year')

     # Displaying only the specified columns for the player
     Colon[selected_columns]
```

```
[ ]:  last_name, first_name  player_id  year       FIP  NextYrFIP
     0        Colon, Bartolo     112526  2015  3.837366   3.986571
     1        Colon, Bartolo     112526  2016  3.986571   5.213584
     2        Colon, Bartolo     112526  2017  5.213584   5.470231
     3        Colon, Bartolo     112526  2018  5.470231        NaN
```

```
[ ]: # Extracting columns with data type 'object' from the DataFrame 'stats'
     # 'stats.dtypes' returns a Series with data types of each column in 'stats'
     # 'stats.dtypes == "object"' creates a boolean Series with True for columns␣
     ↪having data type 'object' and False otherwise
     stats.dtypes[stats.dtypes == 'object']
```

```
[ ]: last_name, first_name    object
     pitch_hand               object
     FIPID                    object
     dtype: object
```

[ ]:
```
# Deleting the 'pitch_hand' column from the DataFrame 'stats'
# Deleting the 'FIPID' column from the DataFrame 'stats'
del stats['pitch_hand']
del stats['FIPID']
```

[ ]:
```
# Select only numeric columns
numeric_columns = stats.select_dtypes(include='number')

# Calculate the correlation matrix
correlation_matrix = numeric_columns.corr().abs()

# Create a mask for selecting the upper triangle of the correlation matrix
upper_triangle_mask = correlation_matrix.where(np.triu(np.
 ↪ones(correlation_matrix.shape), k=1).astype(bool))

# Find columns with correlation greater than a threshold (e.g., 0.99),␣
 ↪excluding 'NextYrFIP'
highly_correlated_columns = [column for column in upper_triangle_mask.columns␣
 ↪if column != 'NextYrFIP' and any(upper_triangle_mask[column] > 0.99)]

# Drop the highly correlated columns from the DataFrame
stats = stats.drop(columns=highly_correlated_columns)
```

[ ]:
```
# Calculate the number of NaNs in each column
nan_count = stats.isnull().sum()

# Filter out columns with zero NaNs
nan_count = nan_count[nan_count > 0]

# Calculate the percentage of NaNs in each column
nan_percentage = (nan_count / len(stats)) * 100

# Create a new DataFrame to store NaN information
nan_info = pd.DataFrame({
    'Column Name': nan_count.index,
    'NaN Count': nan_count.values,
    'NaN Percentage': nan_percentage.values
})

# Display the new DataFrame, sorted by NaN Percentage in descending order
nan_info.sort_values(by='NaN Percentage', ascending=False)
```

[ ]:

	Column Name	NaN Count	NaN Percentage
0	p_opp_batting_avg	1886	100.000000
60	Unnamed: 249	1886	100.000000
1	p_opp_on_base_avg	1886	100.000000
48	fs_avg_break_z	1692	89.713680

50	fs_range_speed	1676	88.865323
46	fs_avg_spin	1671	88.600212
44	n_fs_formatted	1671	88.600212
45	fs_avg_speed	1671	88.600212
47	fs_avg_break_x	1671	88.600212
49	fs_avg_break	1671	88.600212
41	fc_avg_break_z	1232	65.323436
43	fc_range_speed	1151	61.028632
39	fc_avg_spin	1144	60.657476
42	fc_avg_break	1143	60.604454
40	fc_avg_break_x	1143	60.604454
38	fc_avg_speed	1143	60.604454
37	n_fc_formatted	1143	60.604454
61	NextYrFIP	839	44.485684
13	sl_avg_break_z	641	33.987275
27	cu_avg_break_z	627	33.244963
34	si_avg_break_z	578	30.646872
15	sl_range_speed	490	25.980912
11	sl_avg_spin	486	25.768823
9	n_sl_formatted	481	25.503712
10	sl_avg_speed	481	25.503712
12	sl_avg_break_x	481	25.503712
14	sl_avg_break	481	25.503712
20	ch_avg_break_z	466	24.708378
29	cu_range_speed	463	24.549311
25	cu_avg_spin	452	23.966066
23	n_cukc_formatted	450	23.860021
24	cu_avg_speed	450	23.860021
28	cu_avg_break	450	23.860021
26	cu_avg_break_x	450	23.860021
36	si_range_speed	405	21.474019
31	si_avg_speed	393	20.837752
35	si_avg_break	393	20.837752
33	si_avg_break_x	393	20.837752
32	si_avg_spin	393	20.837752
30	n_sift_formatted	393	20.837752
57	offspeed_avg_break_z	328	17.391304
22	ch_range_speed	303	16.065748
6	ff_avg_break_z	280	14.846235
16	n_ch_formatted	276	14.634146
18	ch_avg_spin	276	14.634146
21	ch_avg_break	276	14.634146
17	ch_avg_speed	276	14.634146
19	ch_avg_break_x	276	14.634146
59	offspeed_range_speed	147	7.794274
55	offspeed_avg_speed	126	6.680806
58	offspeed_avg_break	126	6.680806

```
56     offspeed_avg_spin        126        6.680806
54  n_offspeed_formatted        126        6.680806
8         ff_range_speed         88        4.665960
2          n_ff_formatted         76        4.029692
4             ff_avg_spin         76        4.029692
7            ff_avg_break         76        4.029692
3            ff_avg_speed         76        4.029692
5          ff_avg_break_x         76        4.029692
52     breaking_avg_spin         12        0.636267
51  n_breaking_formatted         12        0.636267
53  breaking_avg_break_x         12        0.636267
```

[ ]:
```
# Filter out columns with NaN percentage >= 10%, excluding 'NextYrFIP'
subset_nan_info = nan_info[(nan_info['NaN Percentage'] > 10) &␣
 ↪(nan_info['Column Name'] != 'NextYrFIP')]

# Get the column names to be removed
columns_to_remove = subset_nan_info['Column Name'].tolist()

# Remove columns from the original DataFrame
stats = stats.drop(columns=columns_to_remove)
```

[ ]:
```
# Calculate the number of NaNs in each column
nan_count = stats.isnull().sum()

# Filter out columns with zero NaNs
nan_count = nan_count[nan_count > 0]

# Calculate the percentage of NaNs in each column
nan_percentage = (nan_count / len(stats)) * 100

# Create a new DataFrame
nan_info = pd.DataFrame({
    'Column Name': nan_count.index,
    'NaN Count': nan_count.values,
    'NaN Percentage': nan_percentage.values
})

# Display the new DataFrame
nan_info_sorted = nan_info.sort_values(by='NaN Percentage', ascending=False)
nan_info_sorted
```

[ ]:
```
             Column Name  NaN Count  NaN Percentage
14             NextYrFIP        839       44.485684
13   offspeed_range_speed       147        7.794274
9    n_offspeed_formatted       126        6.680806
10     offspeed_avg_speed       126        6.680806
```

```
11     offspeed_avg_spin        126        6.680806
12    offspeed_avg_break        126        6.680806
5         ff_range_speed         88        4.665960
0         n_ff_formatted         76        4.029692
1           ff_avg_speed         76        4.029692
2            ff_avg_spin         76        4.029692
3         ff_avg_break_x         76        4.029692
4           ff_avg_break         76        4.029692
6   n_breaking_formatted         12        0.636267
7      breaking_avg_spin         12        0.636267
8   breaking_avg_break_x         12        0.636267
```

[ ]:

```
# Function to perform mean imputation for specified columns
def mean_imputation(df, columns):
    """
    Perform mean imputation for specified columns in the DataFrame.

    Parameters:
    - df (DataFrame): The DataFrame containing the columns to be imputed.
    - columns (list): List of column names for mean imputation.

    Returns:
    None
    """
    for column in columns:
        if column != 'NextYrFIP':  # Skip 'NextYrFIP' column
            mean_value = df[column].mean()
            df[column].fillna(mean_value, inplace=True)

# Extract columns with missing values from nan_info
columns_with_nan = nan_info['Column Name'].tolist()
columns_with_nan = [column for column in columns_with_nan if column !=␣
 ↪'NextYrFIP']

# Apply mean imputation to specified columns
mean_imputation(stats, columns_with_nan)
```

[ ]:

```
# List of columns to be removed from the DataFrame 'stats'
removed_columns = ["NextYrFIP", "player_id", "year", "last_name, first_name"]

# Selecting columns that are not in the 'removed_columns' list
# 'stats.columns.isin(removed_columns)' returns a boolean Series, and '~' is␣
 ↪used to negate it
# 'stats.columns[~stats.columns.isin(removed_columns)]' filters out the columns␣
 ↪specified in 'removed_columns'
selected_columns = stats.columns[~stats.columns.isin(removed_columns)]
```

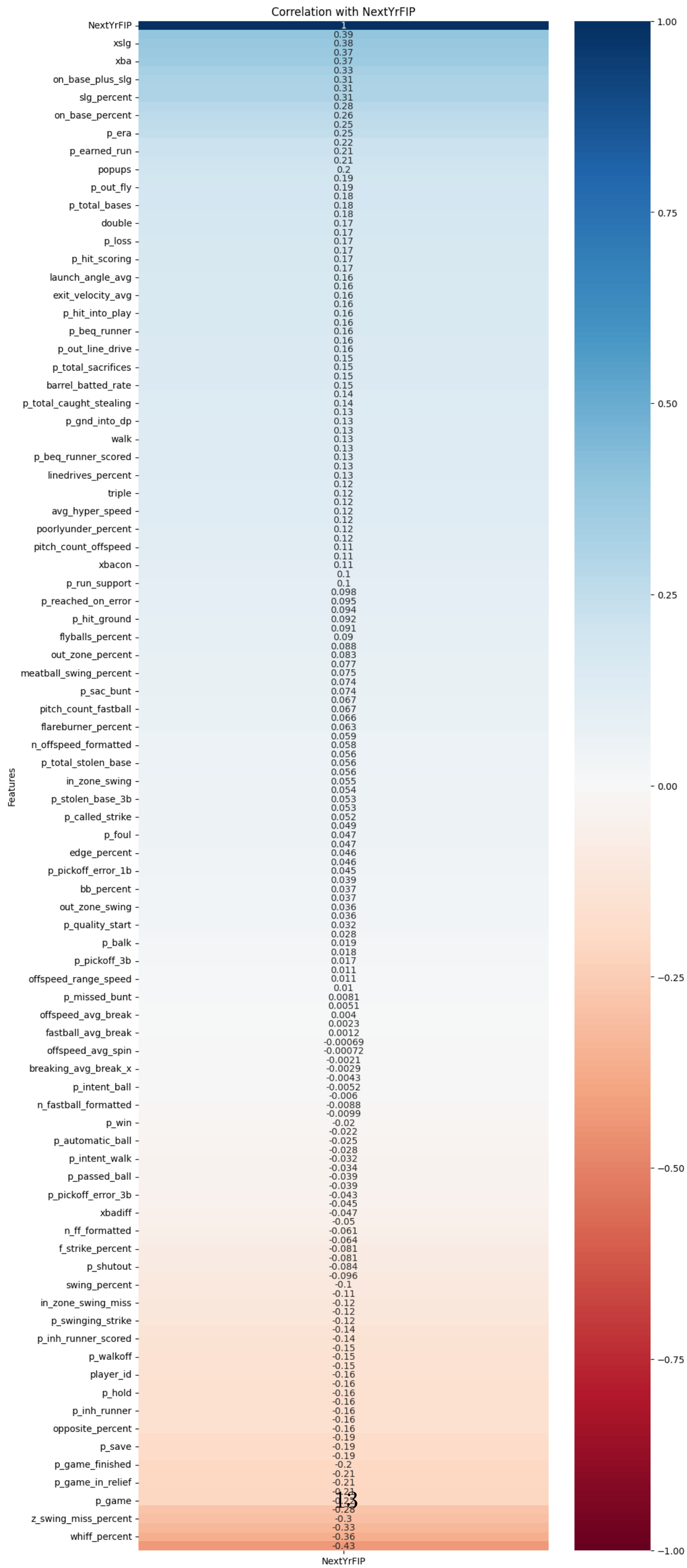
```
# Creating a MinMaxScaler instance
scaler = MinMaxScaler()

# Scaling and transforming the selected columns in the DataFrame 'stats'
# 'stats.loc[:, selected_columns]' selects the specified columns, and 'scaler.
↪fit_transform()' scales and transforms the data
stats.loc[:, selected_columns] = scaler.fit_transform(stats[selected_columns])
stats.head()
```

```
[ ]:    player_id  year last_name, first_name  player_age    p_game  \
0     112526  2015        Colon, Bartolo    0.884615  0.315068
1     112526  2016        Colon, Bartolo    0.923077  0.328767
2     112526  2017        Colon, Bartolo    0.961538  0.246575
3     112526  2018        Colon, Bartolo    1.000000  0.246575
4     282332  2015          Sabathia, CC    0.576923  0.260274

   p_formatted_ip       hit    single    double    triple  …  \
0        0.787709  0.891509  0.923611  0.745098  0.166667  …
1        0.770950  0.811321  0.819444  0.647059  0.500000  …
2        0.501676  0.773585  0.652778  0.921569  0.333333  …
3        0.518994  0.679245  0.548611  0.627451  0.833333  …
4        0.636313  0.754717  0.819444  0.372549  0.333333  …

   n_breaking_formatted  breaking_avg_spin  breaking_avg_break_x  \
0              0.110615           0.458215              0.567839
1              0.069274           0.525298              0.580402
2              0.080447           0.554861              0.572864
3              0.081564           0.585560              0.610553
4              0.250279           0.264355              0.221106

   n_offspeed_formatted  offspeed_avg_speed  offspeed_avg_spin  \
0              0.113323            0.453287           0.469154
1              0.065850            0.411765           0.447264
2              0.148545            0.408304           0.444776
3              0.169985            0.391003           0.434328
4              0.214395            0.498270           0.580100

   offspeed_avg_break  offspeed_range_speed       FIP  NextYrFIP
0            0.590476              0.277778  0.405804   3.986571
1            0.580952              0.111111  0.429073   5.213584
2            0.671429              0.074074  0.620427   5.470231
3            0.576190              0.092593  0.660451        NaN
4            0.523810              0.055556  0.536502   4.282004

[5 rows x 171 columns]
```

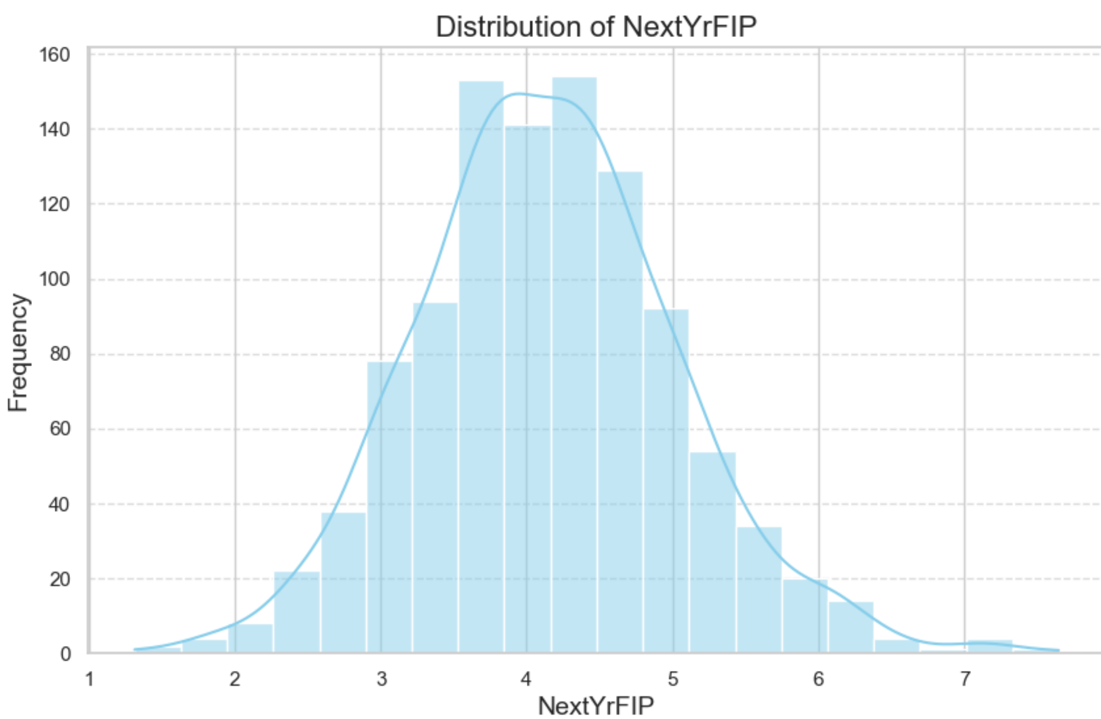
[ ]:
```
# Creating a copy of the DataFrame 'stats' and assigning it to the variable␣
↪'forlater'
# This creates a new DataFrame with the same data as 'stats', allowing␣
↪modifications to be made without affecting the original DataFrame
forlater = stats.copy()
```

[ ]:
```
# Remove all rows with any NaN values (at this point would only be 'NextYrFIP')
stats = stats.dropna()
```

[ ]:
```
# Identify columns with numeric data types in the 'stats' DataFrame
numeric_feats = stats.dtypes[stats.dtypes != "object"].index

# Set the size of the plotting figure
plt.figure(figsize=(10, 30))

# Create a correlation matrix for numeric features with respect to the target␣
↪column 'NextYrFIP'
correlation_matrix = stats[numeric_feats].corr()[['NextYrFIP']].
↪sort_values('NextYrFIP', ascending=False)

# Generate a heatmap to visualize the correlation matrix, with annotations and␣
↪a color map ('RdBu') representing correlation strength
sns.heatmap(correlation_matrix, annot=True, cmap='RdBu', vmin=-1, vmax=1)

# Set plot title, x-axis label, and y-axis label
plt.title('Correlation with NextYrFIP')
plt.xlabel('NextYrFIP')
plt.ylabel('Features')

# Display the heatmap
plt.show()
```


Correlation with NextYrFIP
Features
NextYrFIP
xslg
xba
on_base_plus_slg
slg_percent
on_base_percent
p_era
p_earned_run
popups
p_out_fly
p_total_bases
double
p_loss
p_hit_scoring
launch_angle_avg
exit_velocity_avg
p_hit_into_play
p_beq_runner
p_out_line_drive
p_total_sacrifices
barrel_batted_rate
p_total_caught_stealing
p_gnd_into_dp
walk
p_beq_runner_scored
linedrives_percent
triple
avg_hyper_speed
poorlyunder_percent
pitch_count_offspeed
xbacon
p_run_support
p_reached_on_error
p_hit_ground
flyballs_percent
out_zone_percent
meatball_swing_percent
p_sac_bunt
pitch_count_fastball
flareburner_percent
n_offspeed_formatted
p_total_stolen_base
in_zone_swing
p_stolen_base_3b
p_called_strike
p_foul
edge_percent
p_pickoff_error_1b
bb_percent
out_zone_swing
p_quality_start
p_balk
p_pickoff_3b
offspeed_range_speed
p_missed_bunt
offspeed_avg_break
fastball_avg_break
offspeed_avg_spin
breaking_avg_break_x
p_intent_ball
n_fastball_formatted
p_win
p_automatic_ball
p_intent_walk
p_passed_ball
p_pickoff_error_3b
xbadiff
n_ff_formatted
f_strike_percent
p_shutout
swing_percent
in_zone_swing_miss
p_swinging_strike
p_inh_runner_scored
p_walkoff
player_id
p_hold
p_inh_runner
opposite_percent
p_save
p_game_finished
p_game_in_relief
p_game
z_swing_miss_percent
whiff_percent
NextYrFIP
NextYrFIP
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

[ ]:
```python
# Set the default aesthetic style and color palette for the Seaborn plot
sns.set(style="whitegrid")

# Create a histogram plot for the 'NextYrFIP' column (our target variable)
plt.figure(figsize=(10, 6))
sns.histplot(stats['NextYrFIP'], kde=True, color="skyblue", bins=20)

# Add title and labels to the plot
plt.title('Distribution of NextYrFIP', fontsize=16)
plt.xlabel('NextYrFIP', fontsize=14)
plt.ylabel('Frequency', fontsize=14)

# Display the plot with grid lines
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

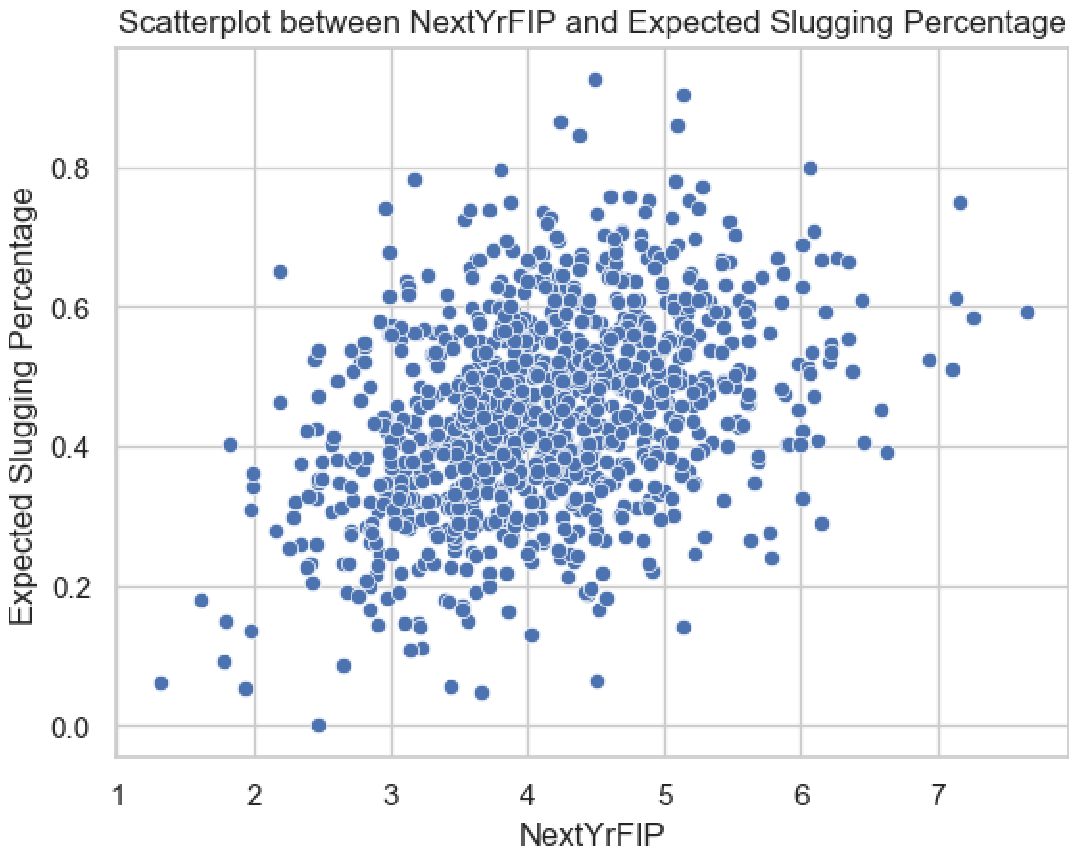


[ ]:
```python
# Calculate the mean of the 'NextYrFIP' column in the 'stats' dataframe
mean_value = stats['NextYrFIP'].mean()

# Calculate the median of the 'NextYrFIP' column in the 'stats' dataframe
median_value = stats['NextYrFIP'].median()
```

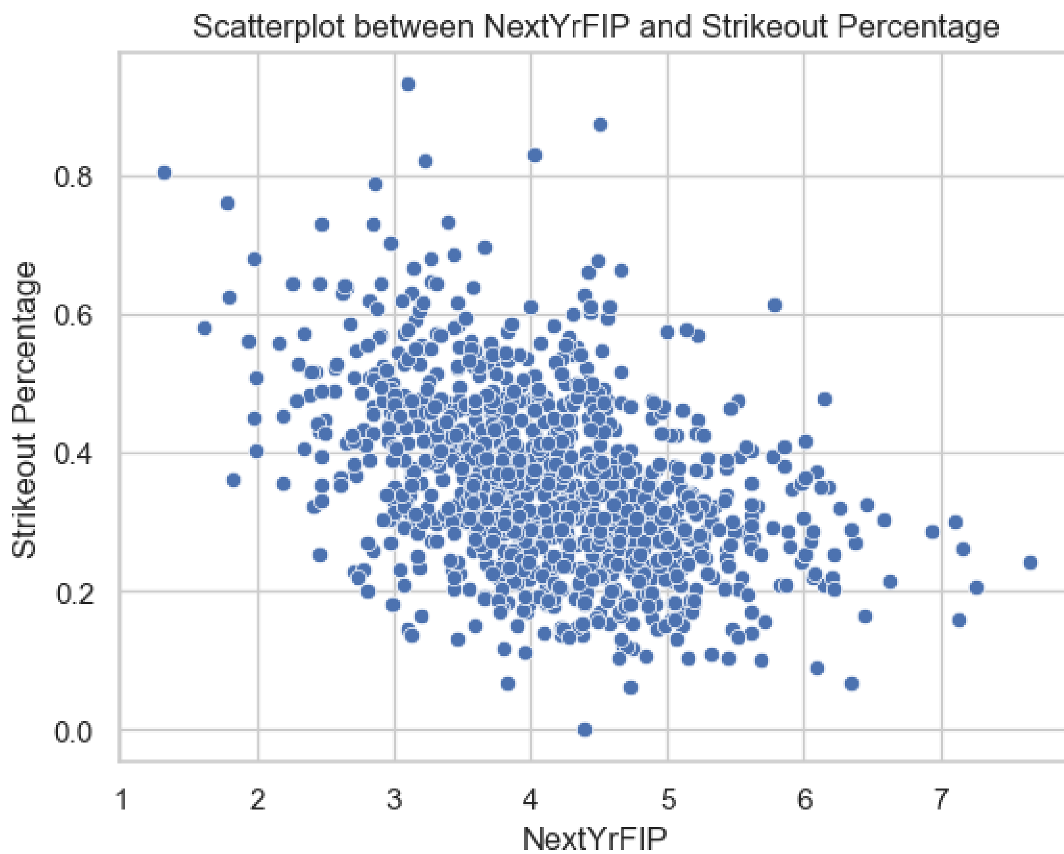
```
# Calculate the standard deviation of the 'NextYrFIP' column in the 'stats'␣
↪dataframe
std_dev = stats['NextYrFIP'].std()

# Calculate the skewness of the 'NextYrFIP' column in the 'stats' dataframe
skewness = stats['NextYrFIP'].skew()

# Print the mean value
print(f"Mean: {mean_value}")

# Print the median value
print(f"Median: {median_value}")

# Print the standard deviation value
print(f"Standard Deviation: {std_dev}")

# Print the skewness value
print(f"Skewness: {skewness}")
```

```
Mean: 4.139005035100555
Median: 4.125005681258069
Standard Deviation: 0.8870177454893141
Skewness: 0.25359302013630003
```

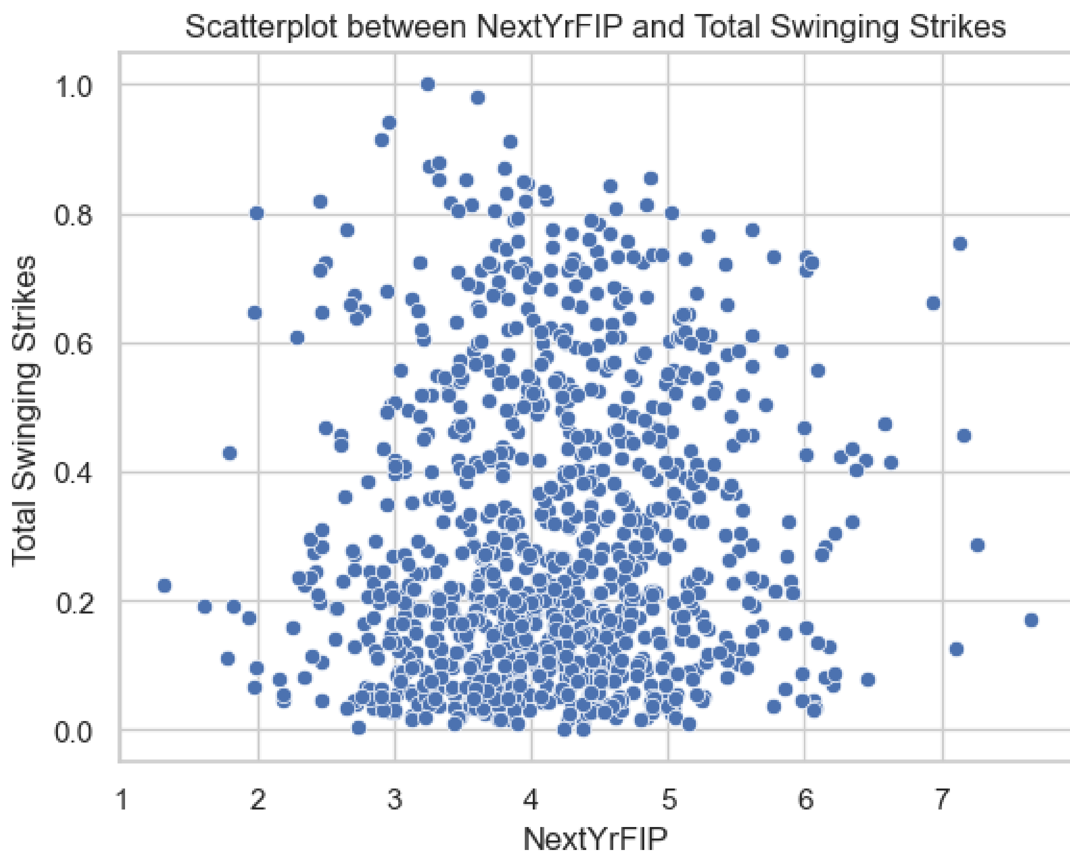
[ ]:
```
# Create a scatter plot for 'NextYrFIP' and 'xwoba' columns
sns.scatterplot(x=stats['NextYrFIP'], y=stats['xslg'])

# Add a title to the plot
plt.title('Scatterplot between NextYrFIP and Expected Slugging Percentage')

# Add labels to the x and y axes
plt.xlabel('NextYrFIP')
plt.ylabel('Expected Slugging Percentage')

# Display the plot
plt.show()
```



[ ]:
```
# Create a scatter plot for 'NextYrFIP' and 'k_percent' columns
sns.scatterplot(x=stats['NextYrFIP'], y=stats['k_percent'])

# Add a title to the plot
plt.title('Scatterplot between NextYrFIP and Strikeout Percentage')

# Add labels to the x and y axes
plt.xlabel('NextYrFIP')
plt.ylabel('Strikeout Percentage')

# Display the plot
plt.show()
```



[ ]:
```python
# Create a scatter plot for 'NextYrFIP' and 'p_total_swinging_strike' columns
sns.scatterplot(x=stats['NextYrFIP'], y=stats['p_total_swinging_strike'])

# Add a title to the plot
plt.title('Scatterplot between NextYrFIP and Total Swinging Strikes')

# Add labels to the x and y axes
plt.xlabel('NextYrFIP')
plt.ylabel('Total Swinging Strikes')

# Display the plot
plt.show()
```



Creating a quick easy 'Baseline Model' to compare my model to:

```
[ ]: # Selecting all numeric features for input (X)
X = stats[numeric_feats[numeric_feats != 'NextYrFIP']]

# Selecting the target variable (y)
y = stats['NextYrFIP']
```

```
[ ]: # Splitting the data into training and testing sets using train_test_split from␣
  ↪scikit-learn
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

```
[ ]: # Creating a baseline model using DummyRegressor with a strategy of predicting␣
  ↪the mean of the target variable
base_model = DummyRegressor(strategy='mean')

# Fitting the baseline model on the training data
base_model = base_model.fit(X_train, y_train)

# Predicting the target variable values on the testing set
```

```
base_y_pred = base_model.predict(X_test)

# True values of the target variable
y_true = y_test

# Calculating R-squared as a measure of model performance
r_squared = r2_score(y_true, base_y_pred)
print("R-Squared:", r_squared)

# Calculating Mean Squared Error (MSE) as a measure of model performance
mse = mean_squared_error(y_true, base_y_pred)

# Calculating Root Mean Squared Error (RMSE) from MSE
rmse = np.sqrt(mse)
print("Mean Squared Error:", mse)
print("Root Mean Squared Error:", rmse)

# Calculating Mean Absolute Error (MAE) as another measure of model performance
mae = mean_absolute_error(y_true, base_y_pred)
print("Mean Absolute Error:", mae)
```

```
R-Squared: -0.001955072420542159
Mean Squared Error: 0.865674032478461
Root Mean Squared Error: 0.930416053429035
Mean Absolute Error: 0.7363107248683622
```

Now we have the four scores we are looking to beat in our model.

[ ]:
```
# Creating a correlation matrix for numeric features in the 'stats' DataFrame
correlation_matrix = stats[numeric_feats].corr()

# Extracting the absolute correlation values of the target variable␣
↪('NextYrFIP')
target_correlation = correlation_matrix['NextYrFIP'].abs()

# Setting a correlation threshold
threshold = 0.2

# Selecting features with correlation values above the threshold
selected_features = target_correlation[target_correlation > threshold].index

# Converting the selected features to a list
index_list = selected_features.tolist()

# Displaying the list of selected features
index_list
```

[ ]:
```
['p_game',
 'k_percent',
 'batting_avg',
 'slg_percent',
 'on_base_percent',
 'on_base_plus_slg',
 'isolated_power',
 'p_earned_run',
 'p_era',
 'p_rbi',
 'p_game_finished',
 'p_game_in_relief',
 'xba',
 'xslg',
 'xwoba',
 'xobp',
 'xiso',
 'barrel',
 'z_swing_miss_percent',
 'oz_swing_miss_percent',
 'whiff_percent',
 'popups',
 'ff_avg_speed',
 'ff_avg_spin',
 'offspeed_avg_speed',
 'FIP',
 'NextYrFIP']
```

[ ]:
```
# Creating a subset DataFrame with the selected features
selected_features_subset = stats[index_list]

# Calculating the correlation matrix for the selected features
selected_features_correlation = selected_features_subset.corr()

# Finding highly correlated features and removing them
removed_features = set()
for i in range(len(selected_features_correlation.columns)):
    for j in range(i):
        if abs(selected_features_correlation.iloc[i, j]) > 0.9:
            colname_i = selected_features_correlation.columns[i]
            colname_j = selected_features_correlation.columns[j]
            removed_features.add(colname_i)
            print(f"Removed: {colname_i} (correlated with {colname_j})")

# Updating the index_list after removing highly correlated features
index_list = [feature for feature in index_list if feature not in␣
 ↪removed_features]
```

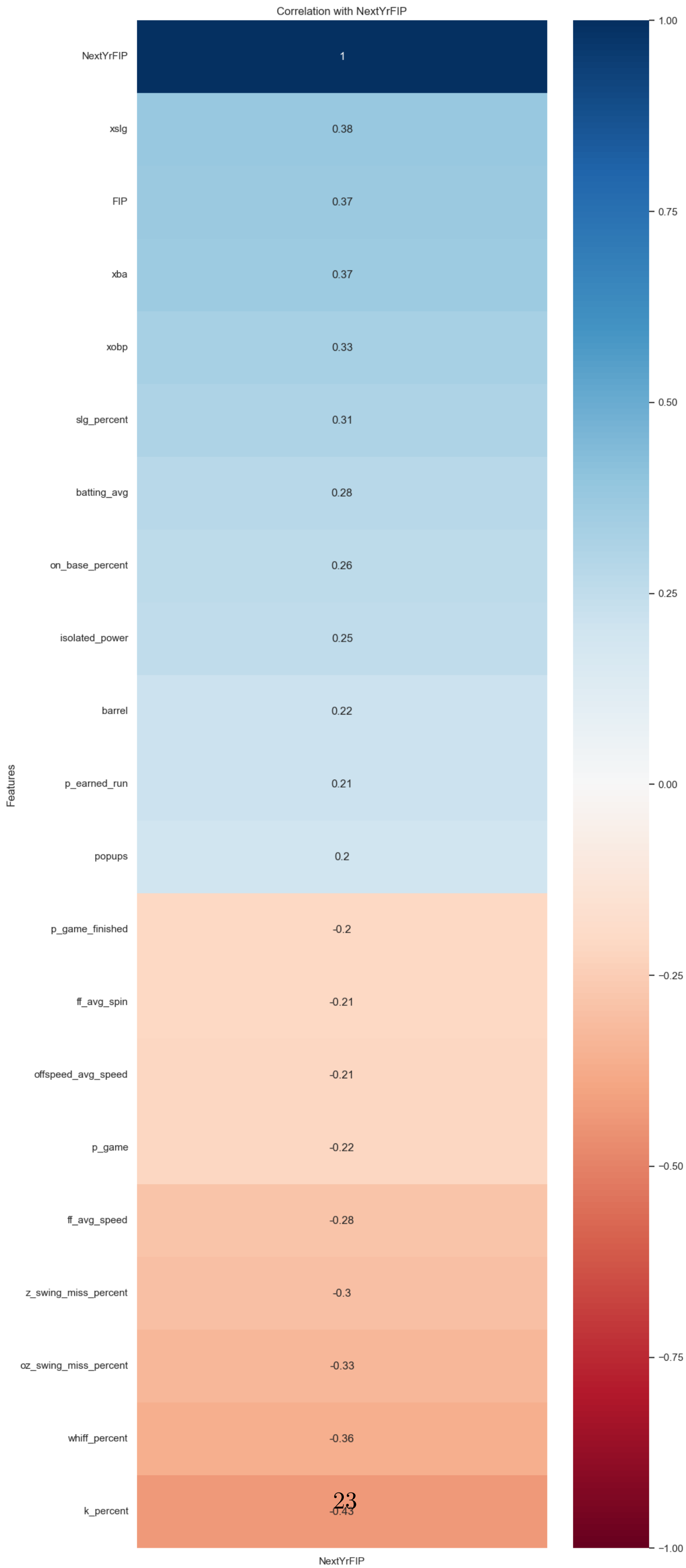
```
Removed: on_base_plus_slg (correlated with slg_percent)
Removed: p_era (correlated with on_base_plus_slg)
Removed: p_rbi (correlated with p_earned_run)
Removed: p_game_in_relief (correlated with p_game)
Removed: xwoba (correlated with xslg)
Removed: xiso (correlated with xslg)
```

[ ]:
```
index_list
```

[ ]:
```
['p_game',
 'k_percent',
 'batting_avg',
 'slg_percent',
 'on_base_percent',
 'isolated_power',
 'p_earned_run',
 'p_game_finished',
 'xba',
 'xslg',
 'xobp',
 'barrel',
 'z_swing_miss_percent',
 'oz_swing_miss_percent',
 'whiff_percent',
 'popups',
 'ff_avg_speed',
 'ff_avg_spin',
 'offspeed_avg_speed',
 'FIP',
 'NextYrFIP']
```

[ ]:
```
# Set the size of the plotting figure
plt.figure(figsize=(10, 30))

# Create a correlation matrix for numeric features with respect to the target␣
↪column 'NextYrFIP'
correlation_matrix = stats[index_list].corr()[['NextYrFIP']].
↪sort_values('NextYrFIP', ascending=False)

# Generate a heatmap to visualize the correlation matrix, with annotations and␣
↪a color map ('RdBu') representing correlation strength
sns.heatmap(correlation_matrix, annot=True, cmap='RdBu', vmin=-1, vmax=1)

# Set plot title, x-axis label, and y-axis label
plt.title('Correlation with NextYrFIP')
plt.xlabel('NextYrFIP')
plt.ylabel('Features')
```

```python
# Display the heatmap
plt.show()
```


Correlation with NextYrFIP
Features
NextYrFIP 1
xslg 0.38
FIP 0.37
xba 0.37
xobp 0.33
slg_percent 0.31
batting_avg 0.28
on_base_percent 0.26
isolated_power 0.25
barrel 0.22
p_earned_run 0.21
popups 0.2
p_game_finished -0.2
ff_avg_spin -0.21
offspeed_avg_speed -0.21
p_game -0.22
ff_avg_speed -0.28
z_swing_miss_percent -0.3
oz_swing_miss_percent -0.33
whiff_percent -0.36
k_percent -0.43
NextYrFIP
NextYrFIP
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

```
[ ]: # Removing 'NextYrFIP' from the list of selected features as it is our target
index_list.remove('NextYrFIP')
```

```
[ ]: # Selecting features based on the updated 'index_list' for input (X)
X = stats[index_list]

# Selecting the target variable (y)
y = stats['NextYrFIP']
```

```
[ ]: # Splitting the updated data into training and testing sets using␣
 ↪train_test_split
# Using a random seed (random_state=42) for reproducibility
# Splitting the data into 80% training and 20% testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .2,␣
 ↪random_state=42)
```

```
[ ]: def rmse_and_r2(regressor, X_train, y_train, X_test, y_test):
    """
    Calculate Root Mean Squared Error (RMSE) and R-squared for both training␣
 ↪and testing sets.

    Parameters:
    - regressor: An instance of a regression model (e.g., LinearRegression())
    - X_train: Training set of input features
    - y_train: Training set of target variable values
    - X_test: Testing set of input features
    - y_test: Testing set of target variable values

    Returns:
    - train_rmse: Root Mean Squared Error on the training set
    - test_rmse: Root Mean Squared Error on the testing set
    - train_r2: R-squared on the training set
    - test_r2: R-squared on the testing set
    """

    # Creating a pipeline with the provided regressor
    steps = [('regressor', regressor())]
    pipe = Pipeline(steps=steps)

    # Fitting the model on the training data
    model = pipe.fit(X_train, y_train)

    # Predictions on both training and testing sets
    train_pred = model.predict(X_train)
```

```
    test_pred = model.predict(X_test)

    # Calculating Root Mean Squared Error (RMSE)
    train_rmse = np.sqrt(mean_squared_error(y_train, train_pred))
    test_rmse = np.sqrt(mean_squared_error(y_test, test_pred))

    # Calculating R-squared
    train_r2 = r2_score(y_train, train_pred)
    test_r2 = r2_score(y_test, test_pred)

    return train_rmse, test_rmse, train_r2, test_r2
```

[ ]:
```
# List of regressors to evaluate
regressors = [LinearRegression, KNeighborsRegressor, DecisionTreeRegressor,␣
 ↪BaggingRegressor, RandomForestRegressor, AdaBoostRegressor, SVR, Ridge,␣
 ↪GradientBoostingRegressor, Lasso, ElasticNet]

# Creating an empty DataFrame to store results
results_df = pd.DataFrame(columns=['Regressor', 'Train_RMSE', 'Test_RMSE',␣
 ↪'Train_R2', 'Test_R2'])

# Looping through each regressor and evaluating its performance
for regressor in regressors:
    regressor_name = regressor.__name__  # Getting the name of the regressor
    # Calling the rmse_and_r2 function to obtain evaluation metrics
    train_rmse, test_rmse, train_r2, test_r2 = rmse_and_r2(regressor, X_train,␣
 ↪y_train, X_test, y_test)

    # Appending the results to the DataFrame
    results_df = pd.concat([results_df, pd.DataFrame([[regressor_name,␣
 ↪train_rmse, test_rmse, train_r2, test_r2]],
                                                       columns=['Regressor',␣
 ↪'Train_RMSE', 'Test_RMSE', 'Train_R2', 'Test_R2'])],
                            ignore_index=True)

# Sorting the DataFrame by Test_R2 in descending order to find the␣
 ↪best-performing regressor
results_df.sort_values(by='Test_R2', ascending=False)
```

[ ]:

	Regressor	Train_RMSE	Test_RMSE	Train_R2	Test_R2
0	LinearRegression	0.738186	0.854740	0.273785	0.212212
6	SVR	0.670078	0.857677	0.401609	0.206789
7	Ridge	0.740747	0.865601	0.268736	0.192064
4	RandomForestRegressor	0.288495	0.877040	0.889080	0.170569
5	AdaBoostRegressor	0.680502	0.880953	0.382847	0.163152
8	GradientBoostingRegressor	0.512784	0.890741	0.649569	0.144452
3	BaggingRegressor	0.333479	0.900199	0.851792	0.126186

```
1          KNeighborsRegressor    0.682953   0.919035  0.378393  0.089237
9                        Lasso    0.866230   0.963616  0.000000 -0.001267
10                  ElasticNet    0.866230   0.963616  0.000000 -0.001267
2        DecisionTreeRegressor    0.000000   1.192217  1.000000 -0.532683
```

[ ]:
```
# Creating a pipeline for Linear Regression
lr_pipeline = Pipeline([
    ("lr", LinearRegression(n_jobs=-1))
])

# Defining hyperparameters for grid search
lr_parameters = {
    'lr__fit_intercept': [True, False],
    'lr__positive': [True, False],
    'lr__copy_X': [True, False],
}

# Creating a GridSearchCV object with negative root mean squared error as the␣
↪scoring metric
lr_gs = GridSearchCV(lr_pipeline, param_grid=lr_parameters, n_jobs=-1,␣
↪scoring='neg_root_mean_squared_error')

# Fitting the GridSearchCV object to the training data
lr_gs.fit(X_train, y_train)
```

[ ]:
```
GridSearchCV(estimator=Pipeline(steps=[('lr', LinearRegression(n_jobs=-1))]),
             n_jobs=-1,
             param_grid={'lr__copy_X': [True, False],
                         'lr__fit_intercept': [True, False],
                         'lr__positive': [True, False]},
             scoring='neg_root_mean_squared_error')
```

[ ]:
```
# Creating a pipeline for Support Vector Regression (SVR)
svr_pipeline = Pipeline([
    ("svr", SVR())
])

# Defining hyperparameters for grid search
svr_parameters = {
    'svr__kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'svr__C': [0.1, 1, 10],
    'svr__epsilon': [0.1, 0.2, 0.5],
}

# Creating a GridSearchCV object with negative mean squared error as the␣
↪scoring metric
```

```
svr_gs = GridSearchCV(svr_pipeline, param_grid=svr_parameters, n_jobs=-1,␣
↪scoring='neg_mean_squared_error')

# Fitting the GridSearchCV object to the training data
svr_gs.fit(X_train, y_train)
```

[ ]:

```
GridSearchCV(estimator=Pipeline(steps=[('svr', SVR())]), n_jobs=-1,
             param_grid={'svr__C': [0.1, 1, 10],
                         'svr__epsilon': [0.1, 0.2, 0.5],
                         'svr__kernel': ['linear', 'poly', 'rbf', 'sigmoid']},
             scoring='neg_mean_squared_error')
```

[ ]:

```
# Creating a pipeline for Ridge regression
ridge_pipeline = Pipeline([
    ("ridge", Ridge())
])

# Defining hyperparameters for grid search
ridge_parameters = {
    'ridge__alpha': [0.1, 1, 10],
    'ridge__fit_intercept': [True, False],
    'ridge__copy_X': [True, False],
    'ridge__solver': ['auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg']
}

# Creating a GridSearchCV object with negative root mean squared error as the␣
↪scoring metric
ridge_gs = GridSearchCV(ridge_pipeline, param_grid=ridge_parameters, n_jobs=-1,␣
↪scoring='neg_root_mean_squared_error')

# Fitting the GridSearchCV object to the training data
ridge_gs.fit(X_train, y_train)
```

[ ]:

```
GridSearchCV(estimator=Pipeline(steps=[('ridge', Ridge())]), n_jobs=-1,
             param_grid={'ridge__alpha': [0.1, 1, 10],
                         'ridge__copy_X': [True, False],
                         'ridge__fit_intercept': [True, False],
                         'ridge__solver': ['auto', 'svd', 'cholesky', 'lsqr',
                                           'sparse_cg']},
             scoring='neg_root_mean_squared_error')
```

[ ]:

```
# Creating a pipeline for Random Forest regression
random_forest_pipeline = Pipeline([
    ("random_forest", RandomForestRegressor())
])

# Defining hyperparameters for grid search
```

```
random_forest_parameters = {
    'random_forest__n_estimators': [50, 100, 200],
    'random_forest__max_depth': [None, 10, 20],
    'random_forest__min_samples_split': [2, 5, 10],
    'random_forest__min_samples_leaf': [1, 2, 4],
    'random_forest__bootstrap': [True, False]
}

# Creating a GridSearchCV object with negative root mean squared error as the␣
↪scoring metric
random_forest_gs = GridSearchCV(random_forest_pipeline,␣
↪param_grid=random_forest_parameters, n_jobs=-1,␣
↪scoring='neg_root_mean_squared_error')

# Fitting the GridSearchCV object to the training data
random_forest_gs.fit(X_train, y_train)
```

[ ]:
```
GridSearchCV(estimator=Pipeline(steps=[('random_forest',
                                        RandomForestRegressor())]),
             n_jobs=-1,
             param_grid={'random_forest__bootstrap': [True, False],
                         'random_forest__max_depth': [None, 10, 20],
                         'random_forest__min_samples_leaf': [1, 2, 4],
                         'random_forest__min_samples_split': [2, 5, 10],
                         'random_forest__n_estimators': [50, 100, 200]},
             scoring='neg_root_mean_squared_error')
```

[ ]:
```
# Creating a pipeline for AdaBoostRegressor
adaboost_pipeline = Pipeline([
    ("adaboost", AdaBoostRegressor())
])

# Defining hyperparameters for grid search
adaboost_parameters = {
    'adaboost__n_estimators': [50, 100, 200],
    'adaboost__learning_rate': [0.01, 0.1, 0.5, 1.0],
    'adaboost__loss': ['linear', 'square', 'exponential']
}

# Creating a GridSearchCV object with negative root mean squared error as the␣
↪scoring metric
adaboost_gs = GridSearchCV(adaboost_pipeline, param_grid=adaboost_parameters,␣
↪n_jobs=-1, scoring='neg_root_mean_squared_error')

# Fitting the GridSearchCV object to the training data
adaboost_gs.fit(X_train, y_train)
```

[ ]: GridSearchCV(estimator=Pipeline(steps=[('adaboost', AdaBoostRegressor())]),
             n_jobs=-1,
             param_grid={'adaboost__learning_rate': [0.01, 0.1, 0.5, 1.0],
                         'adaboost__loss': ['linear', 'square', 'exponential'],
                         'adaboost__n_estimators': [50, 100, 200]},
             scoring='neg_root_mean_squared_error')

[ ]:
```
# Creating dictionaries to store results for each model
lr_results = {
    'Model': 'Linear Regression',
    'Best Score': -lr_gs.best_score_,
    'Best Parameters': lr_gs.best_params_
}

ridge_results = {
    'Model': 'Ridge Regression',
    'Best Score': -ridge_gs.best_score_,
    'Best Parameters': ridge_gs.best_params_
}

svr_results = {
    'Model': 'SVR Regression',
    'Best Score': -svr_gs.best_score_,
    'Best Parameters': svr_gs.best_params_
}

rf_results = {
    'Model': 'Random Forest Regression',
    'Best Score': -random_forest_gs.best_score_,
    'Best Parameters': random_forest_gs.best_params_
}

adaboost_results = {
    'Model': 'Adaboost Regression',
    'Best Score': -adaboost_gs.best_score_,
    'Best Parameters': adaboost_gs.best_params_
}

# Creating a DataFrame to display the results
results_df = pd.DataFrame([lr_results, ridge_results, svr_results, rf_results,
 ↪adaboost_results])

# Sorting the DataFrame by the 'Best Score' column in descending order
results_df = results_df.sort_values(by='Best Score', ascending=False)

results_df
```

[ ]:

```
                          Model  Best Score  \
3  Random Forest Regression    0.768716
4       Adaboost Regression    0.765113
0         Linear Regression    0.759808
1          Ridge Regression    0.755675
2            SVR Regression    0.567919

                                     Best Parameters
3  {'random_forest__bootstrap': True, 'random_for…
4  {'adaboost__learning_rate': 0.1, 'adaboost__lo…
0  {'lr__copy_X': True, 'lr__fit_intercept': Fals…
1  {'ridge__alpha': 1, 'ridge__copy_X': True, 'ri…
2  {'svr__C': 1, 'svr__epsilon': 0.5, 'svr__kerne…
```

[ ]:

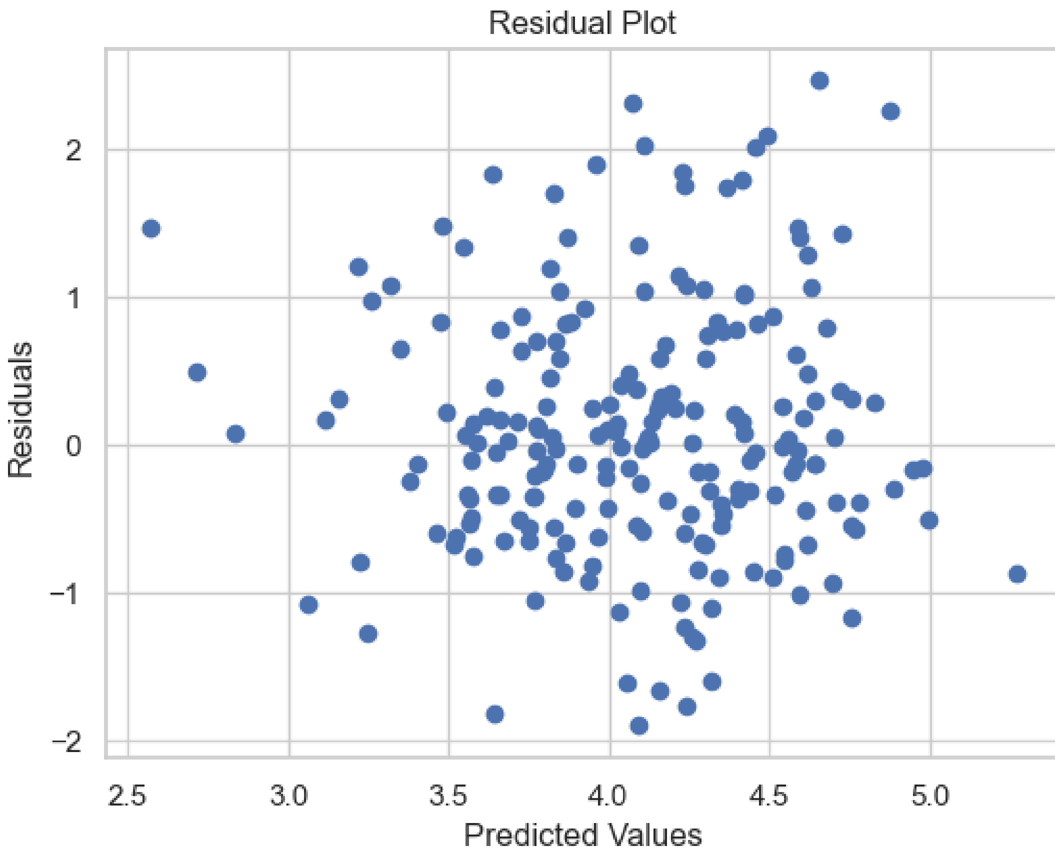
```
# Predict the target values using the trained linear regression model (lr_gs)
y_pred = lr_gs.predict(X_test)

# Assign the true target values from the test set to y_true
y_true = y_test

# Calculate and print R-Squared score, a measure of the goodness of fit of the␣
↪model
r_squared = r2_score(y_true, y_pred)
print("R-Squared:", r_squared)

# Calculate and print Mean Squared Error (MSE), a measure of the average␣
↪squared difference between predicted and true values
mse = mean_squared_error(y_true, y_pred)
print("Mean Squared Error:", mse)

# Calculate and print Root Mean Squared Error (RMSE), the square root of the␣
↪MSE, providing a more interpretable scale
rmse = np.sqrt(mse)
print("Root Mean Squared Error:", rmse)

# Calculate and print Mean Absolute Error (MAE), a measure of the average␣
↪absolute difference between predicted and true values
mae = mean_absolute_error(y_true, y_pred)
print("Mean Absolute Error:", mae)
```

```
R-Squared: 0.21221161341331019
Mean Squared Error: 0.7305801405101947
Root Mean Squared Error: 0.8547398086612058
Mean Absolute Error: 0.6576109097394935
```

[ ]:

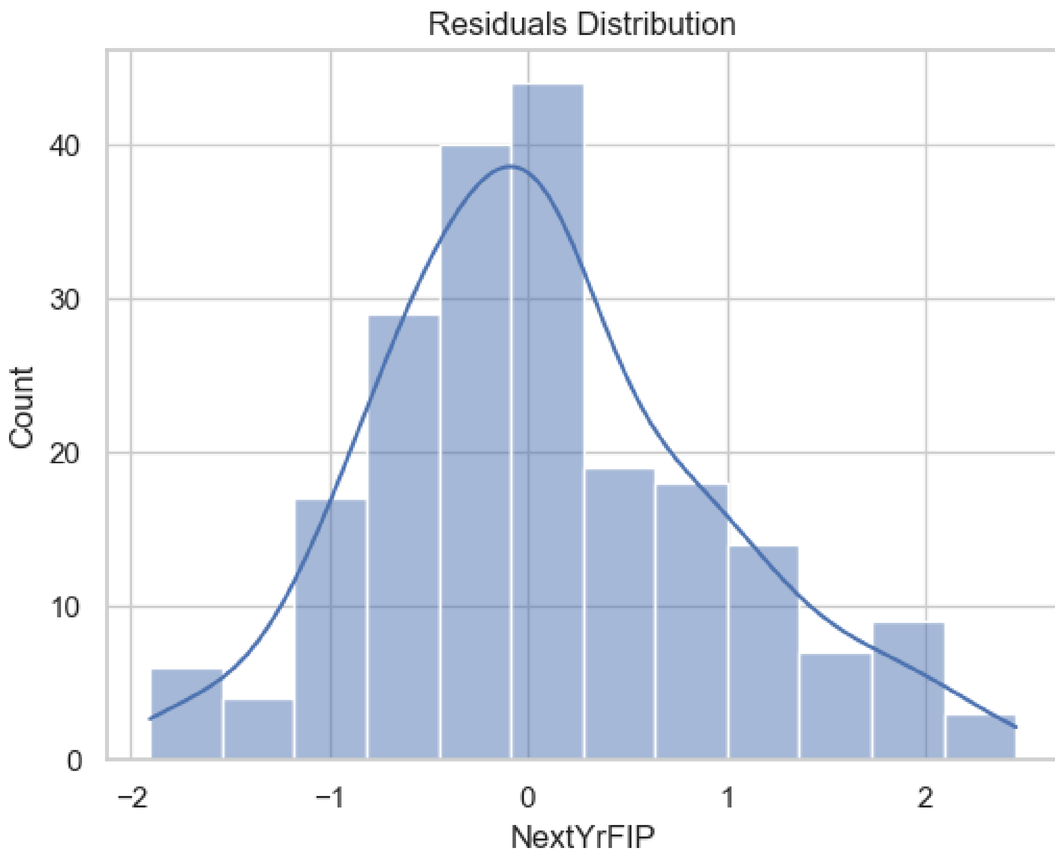
```
# Calculate residuals by subtracting predicted values from true values
residuals = y_true - y_pred
```

```python
# Scatter plot to visualize residuals against predicted values
plt.scatter(y_pred, residuals)

# Set plot title
plt.title("Residual Plot")

# Label for x-axis (Predicted Values)
plt.xlabel("Predicted Values")

# Label for y-axis (Residuals)
plt.ylabel("Residuals")

# Display the plot
plt.show()
```



[ ]:
```python
# Create a histogram of residuals using seaborn
# Set kde=True to display the kernel density estimate along with the histogram
sns.histplot(residuals, kde=True)
```

```
# Set plot title
plt.title("Residuals Distribution")

# Display the plot
plt.show()
```



Predictions for 2024 MLB Season:

```
[ ]: # Create a subset DataFrame containing only rows where the 'year' column is␣
      ↪equal to 2023
     twentythree = forlater[forlater['year'] == 2023].copy()

     # Set 'last_name, first_name' as the index for the DataFrame
     twentythree.set_index('last_name, first_name', inplace=True)

     # Display the resulting DataFrame for the year 2023
     twentythree.head()
```

```
[ ]:                     player_id  year  player_age    p_game  p_formatted_ip  \
last_name, first_name
Wainwright, Adam            425794  2023    0.846154  0.150685        0.267039
Greinke, Zack               425844  2023    0.769231  0.273973        0.496648
Verlander, Justin           434378  2023    0.807692  0.232877        0.608380
Kluber, Corey               446372  2023    0.692308  0.068493        0.010056
Hill, Rich                  448179  2023    0.923077  0.301370        0.518994

                            hit    single    double    triple  home_run  …  \
last_name, first_name                                                    …
Wainwright, Adam       0.580189  0.534722  0.647059  0.166667  0.441860  …
Greinke, Zack          0.613208  0.625000  0.431373  0.166667  0.558140  …
Verlander, Justin      0.523585  0.506944  0.529412  0.166667  0.395349  …
Kluber, Corey          0.193396  0.125000  0.254902  0.166667  0.372093  …
Hill, Rich             0.646226  0.541667  0.803922  0.333333  0.511628  …

                       n_breaking_formatted  breaking_avg_spin  \
last_name, first_name
Wainwright, Adam                   0.345251           0.732234
Greinke, Zack                      0.379888           0.616259
Verlander, Justin                  0.505028           0.707220
Kluber, Corey                      0.313966           0.681637
Hill, Rich                         0.496089           0.702672

                       breaking_avg_break_x  n_offspeed_formatted  \
last_name, first_name
Wainwright, Adam                   0.917085              0.078101
Greinke, Zack                      0.821608              0.254211
Verlander, Justin                  0.640704              0.071975
Kluber, Corey                      0.894472              0.243492
Hill, Rich                         0.067839              0.033691

                       offspeed_avg_speed  offspeed_avg_spin  \
last_name, first_name
Wainwright, Adam                 0.415225           0.437313
Greinke, Zack                    0.588235           0.484577
Verlander, Justin                0.543253           0.517413
Kluber, Corey                    0.435986           0.466667
Hill, Rich                       0.467128           0.419900

                       offspeed_avg_break  offspeed_range_speed       FIP  \
last_name, first_name
Wainwright, Adam                 0.614286              0.185185  0.741154
Greinke, Zack                    0.485714              0.111111  0.547273
Verlander, Justin                0.666667              0.092593  0.407215
Kluber, Corey                    0.538095              0.074074  0.916111
Hill, Rich                       0.471429              0.148148  0.566501
```

```
                          NextYrFIP
last_name, first_name
Wainwright, Adam                NaN
Greinke, Zack                   NaN
Verlander, Justin               NaN
Kluber, Corey                   NaN
Hill, Rich                      NaN

[5 rows x 170 columns]
```

[ ]:
```
# Extract a subset of columns from the DataFrame 'twentythree' using the␣
↪specified 'index_list' to be our features
X = twentythree[index_list]
```

[ ]:
```
# Use the trained linear regression model (lr_gs) to make predictions on the␣
↪input data (X)
predictions = lr_gs.predict(X)
```

[ ]:
```
# Round each number to the nearest hundredth
predictions = [round(num, 2) for num in predictions]
```

[ ]:
```
# Create a DataFrame 'preddf' from the predictions with the same index as␣
↪'twentythree'
preddf = pd.DataFrame(predictions)
preddf.index = twentythree.index

# Rename the columns in 'preddf' for clarity
preddf.columns = ['Projected_FIP']

# Set the index name of 'preddf' for better presentation
preddf.index.name = 'Pitcher Name'

# Sort 'preddf' by the 'Projected_FIP' column in ascending order
preddf_sorted = preddf.sort_values(by='Projected_FIP', ascending=True)

# Add a new column 'rank' to the DataFrame and assign ranks to each row␣
↪starting from 1
preddf_sorted['Rank'] = range(1, len(preddf_sorted) + 1)
```

[ ]:
```
# Display top 25 projected FIPs in MLB in 2024
preddf_sorted.reset_index(inplace=True)
preddf_top = preddf_sorted[['Rank', 'Pitcher Name', 'Projected_FIP']].head(25)
preddf_top
```

[ ]:

	Rank	Pitcher Name	Projected_FIP
0	1	Duran, Jhoan	2.92

```
1      2         Scott, Tanner          2.94
2      3       Strider, Spencer         3.11
3      4         Skubal, Tarik          3.11
4      5         Hicks, Jordan          3.21
5      6        Kimbrel, Craig          3.23
6      7        Glasnow, Tyler          3.23
7      8          Minter, A.J.          3.31
8      9         Bummer, Aaron          3.37
9      10        Pressly, Ryan          3.37
10     11    Woodruff, Brandon          3.40
11     12          Holmes, Clay         3.41
12     13        Soto, Gregory          3.42
13     14        Greene, Hunter         3.43
14     15            Fried, Max         3.46
15     16         Doval, Camilo         3.47
16     17          Jax, Griffin         3.49
17     18          Lopez, Pablo         3.49
18     19          Strahm, Matt         3.52
19     20      Clase, Emmanuel          3.53
20     21      Alzolay, Adbert          3.53
21     22        King, Michael          3.57
22     23      Richards, Trevor         3.58
23     24         Pivetta, Nick         3.58
24     25         Keller, Mitch         3.61
```

[ ]:
```python
# Display worst 25 projected FIPs in MLB in 2024

preddf_bot = preddf_sorted[['Rank', 'Pitcher Name', 'Projected_FIP']].tail(25)
preddf_bot
```

[ ]:
```
     Rank        Pitcher Name  Projected_FIP
174   175   Syndergaard, Noah           4.63
175   176       Perez, Martin           4.63
176   177       Urquidy, Jose           4.64
177   178       Flexen, Chris           4.67
178   179     Corbin, Patrick           4.67
179   180       Mikolas, Miles          4.67
180   181         Manoah, Alek          4.67
181   182     Chirinos, Yonny           4.67
182   183    Javier, Cristian           4.69
183   184           Hill, Rich          4.72
184   185       Freeland, Kyle          4.74
185   186      Anderson, Tyler          4.75
186   187     Hendricks, Kyle           4.78
187   188          Miley, Wade          4.80
188   189       Gonsolin, Tony          4.81
189   190         Gray, Josiah          4.81
```

```
190   191   Williams, Trevor           4.82
191   192     Quantrill, Cal           4.84
192   193     Gomber, Austin           4.86
193   194          Sears, JP           4.87
194   195       Lyles, Jordan          4.89
195   196       Manning, Matt          4.90
196   197     Hudson, Dakota           4.90
197   198   Wainwright, Adam           5.00
198   199       Kluber, Corey          5.04
```

Final look at models:

```
[ ]: data = {
         'Model': ['Dummy Regressor (Baseline)', 'Linear Regression'],
         'R-Squared': [-0.002, 0.212],
         'Mean Squared Error': [0.866, 0.731],
         'Root Mean Squared Error': [0.930, 0.855],
         'Mean Absolute Error': [0.736, 0.658]
     }

     df = pd.DataFrame(data)

     df
```

```
[ ]:                         Model  R-Squared  Mean Squared Error  \
     0  Dummy Regressor (Baseline)     -0.002               0.866
     1           Linear Regression      0.212               0.731

        Root Mean Squared Error  Mean Absolute Error
     0                    0.930                0.736
     1                    0.855                0.658
```