

MLB's Run Support Issue

Solomon O. Stevens

Northwest Missouri State University, Maryville MO 64468, USA
S568272@nwmissouri.edu

Abstract. When, in baseball, a pitcher performs well but his/her offense doesn't score many runs, it's commonly referred to as a lack of run support. Such a pitcher may feel helpless in resolving this issue. But are they completely helpless? Through various manipulation tactics of Major League Baseball (MLB) statistics retrieved from Baseball Savant, this project's game plan is to determine any underlying correlations between different pitching statistics and having a lack of run support as well as develop an appropriate machine learning model to statistically predict any chance of low run support in the future. The Random Forest model derived is able to predict such cases with an 80% accuracy rate. And through the strongest correlations (home runs allowed, and percent of pitches that hit the barrel of the bat) it can be determined that, by actually allowing the other team to score via the home run, one may receive more support from his/her offense.

Keywords: baseball · pitch · hit · run · score · support

1 Introduction

What makes one pitcher more valuable than another? Why are teams willing to pay some pitchers more than others? One major part of an organization's job is to ensure quality of play for their fans; and fans of Major League Baseball (MLB) are responsive to the success of a major league team [2]. So raw run-prevention statistics that organizations often use to gauge the value of pitchers, such as earned run average (ERA), may not properly assign credit for their performances [1]. Rather, wins are what matter the most; and this is true for any sport franchise, not just baseball.

The goal of a baseball team is to score more runs than their opposition. But even if their pitcher lets up only one run, there is a chance that team won't be able to score at all. This would result in a loss due to a lack of run support, and it would mean a pitcher has a lesser value.

1.1 Goals of this Research

This project was designed to use MLB pitching data to discover what pitching factors can consistently attribute to having little run support. Avoiding these attributes can increase a team's chance of winning and thus increase the quality and value of a pitcher.

1.2 Steps of this Project

Upon the acquisition of data, appropriate steps to clean it must be taken. Null values, outliers, and factors with very little relation to very little run support must be addressed (either get rid of them, or explain why they stay). Exploratory Data Analysis (EDA) is then performed to discover underlying relations and remove multicollinearity. Once that data is normalized, a machine learning model can be created and trained to determine if a pitcher is receiving less run support than he should be. With these predictions at the ready, it can finally be determined which factors lead to lower run support.

1.3 Implementation

There are three annual statistical pieces to observe in detecting poor run support: wins, losses, and ERA. If a pitcher has a low ERA but does not have many wins in relation to games lost, that pitcher did his job, but didn't receive much help from his offense.

Using these statistics, a label can be placed on each pitcher as to whether he suffers from lack of run support. Predictions are then made using machine learning to attempt to identify possible targets of low run support by observing several factors other than the three original.

1.4 Resources

Most of the coding was performed in Python. Through use of its library Pandas, the data sets were read into the coding environment, and the refined data sets were sent back to CSV format. Graphs were created using Tableau, Pyplot, and Seaborn. The machine learning aspect of this project was done using Sk-Learn.

1.5 Limitations

This project looks only at Major League Baseball statistics from the years 2015 to 2023. At the time of this project, the MLB is in the middle of their 2024 season, so the statistical measures are incomplete.

Because this project is working with only MLB games, other organizations like the KBO, Northwoods League, etc. are not included. Similar sports like softball, wiffle ball, blitz ball, etc. are also not included.

While searching each and every game for each and every pitcher would be more ideal as it would return much more accurate results, the time frame of this project does not permit such a process to take place.

Other statistical information such as the time elapsed per inning pitched is not kept by Major League Baseball at the time of this project. These factors could lead to better results as well, and they should be considered for future study.

1.6 Links

- [Baseball Savant](#) (where the data comes from)
- [Overleaf LaTeX Report](#)
- [GitHub Repository](#)

2 Data Collection

All of the data collected for this project originates from Baseball Savant which is a website dedicated to recording and storing a plethora of Major League Baseball statistics covering a wide range of topics. These bits of data are taken from countless sensors and high-speed cameras planted throughout every MLB game. What makes Baseball Savant amazing is that it allows a user to select what statistics to include as well as what time frame to focus on. Once the user has an appropriate data set, Baseball Savant allows that user to download the set as a CSV file.

2.1 Data Set Generation

Using the process described in the previous section, a structured data set is created consisting of 21 possible contributions to low run support as columns. This data set has 999 instances spanning the years 2015-2023. In 2015, the MLB launched its Statcast system [4] which contains more advanced statistics used in this project. As stated before, this project is being constructed during the 2024 MLB season, so the data for this year is not yet complete. Hence, 2023 is the latest recorded year. This data set is then ready to be sent into a pandas database for further processing.

Below is a data attribute dictionary describing all factors included with the original data set.

Table 1: All Original Attributes

Column Name	Description	Data Type	Example
last_name, first_name	The name of the pitcher (last, first)	object	Colon, Bartolo
player_id	The identification number of the pitcher	int64	112526
year	What year is being measured	int64	2105
p_game	The number of games the pitcher appeared in	int64	33
p_formatted_ip	The number of innings the pitcher pitched	int64	194.2
pa	The number of batters the pitcher faced	int64	815
hit	The number of hits allowed	int64	217
home_run	The number of home runs allowed	int64	25
strikeout	The total number of strikeouts	int64	136
walk	The total number of walks allowed	int64	24
k_percent	The percentage of batters faced that struckout	float	16.7
bb_percent	The percentage of batters faced that walked	float	2.9
p_earned_run	The number of earned runs allowed	int64	90
p_run	The number of unearned runs allowed	int64	94
p_win	The total number of wins the pitcher got	int64	14
p_loss	The total number of losses the pitcher got	string	13
p_era	The Earned Run Average	float	4.16
p_foul	The number of foul balls allowed	int64	459
barrel_batted_rate	The percentage of pitches that hit the barrel of the opponents' bat	float	5.4
hard_hit_percent	The percentage of pitches that were hit over 95 miles per hour	float	34.7
n	The number of pitches thrown	int64	2683

2.2 Problem Detection

Though the data itself is rather clean, there is one potential glaring issue. Due to the COVID19 Pandemic, the MLB season was shortened from the usual 162 regular season games to just 60 [3] in 2020. So all the counting statistics could be heavily skewed due to a smaller number of games played. This must be taken into account when measuring for a lack of run support.

3 Data Manipulation

Baseball Savant gives users the option of what factors to include in a data set. Once the appropriate data set is created, downloaded, and put into a data frame, the cleaning process begins.

3.1 Data Cleaning

Before one works with data, one must ensure the cleanliness of that data. This involves checking for empty inputs, outliers, or if the data points need to be normalized.

Empty Values Using the following commands:

```
print(df.isna().sum())
```

and / or

```
print(df.isnull().sum())
```

Output:

last_name, first_name	0
player_id	0
year	0
p_game	0
p_formatted_ip	0
pa	0
hit	0
home_run	0
strikeout	0
walk	0
k_percent	0
bb_percent	0
p_earned_run	0
p_run	0
p_win	0
p_loss	0
p_era	0
p_foul	0
barrel_batted_rate	0
hard_hit_percent	0
n	0

one can note that there are no empty inputs in the data set.

Outliers Even though there are outliers in this data set, box plots below [6](#) show that no outlier is extreme or the result of an incorrect input. So because all the outliers are a natural part of the "population" in this data set, they are permitted to stay.

Shortened 2020 Season As previously mentioned, one other detail that must be inspected is how the 60-game 2020 season affects certain factors... mainly a pitcher's win-loss percentage and Earned Run Average (ERA) as these are needed to create this project's dependent variable.

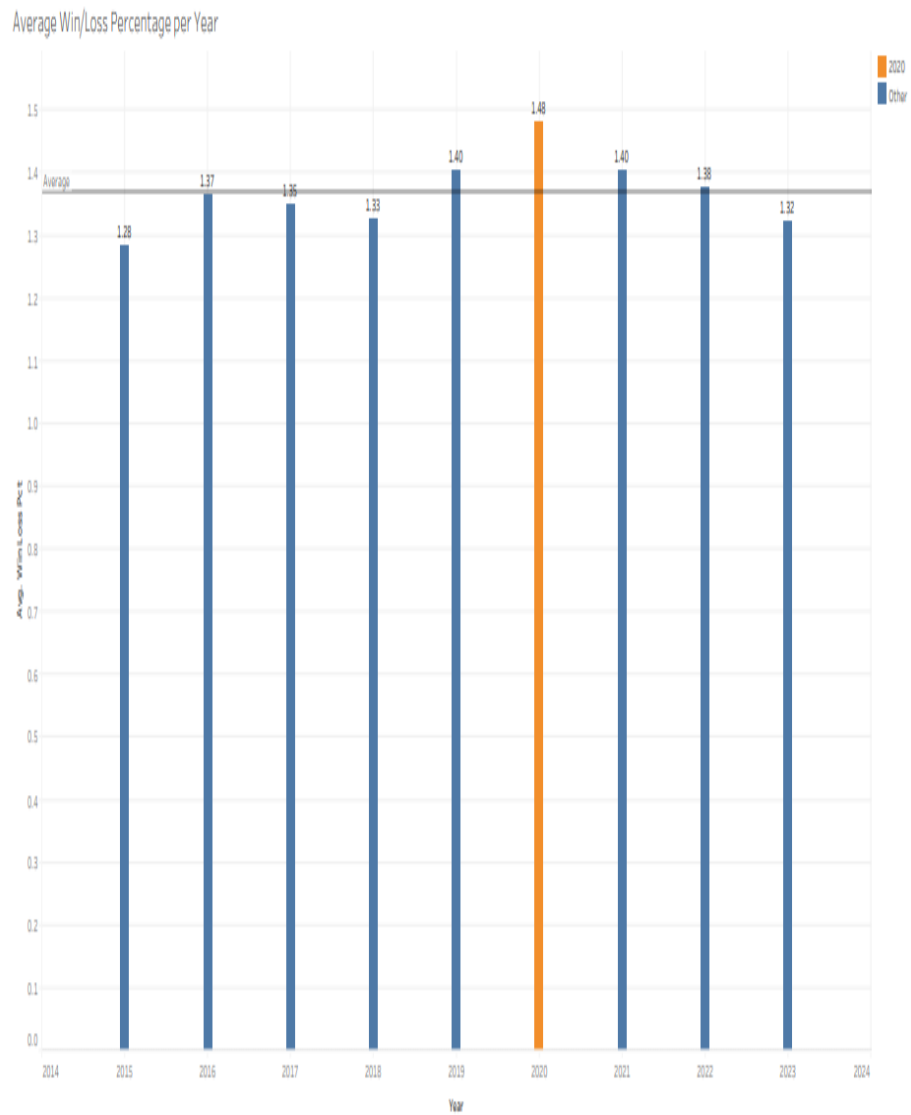


Fig. 1: A pitcher's win-loss percentage by year shows the year 2020 had the highest winning percentage. But it wasn't by much.

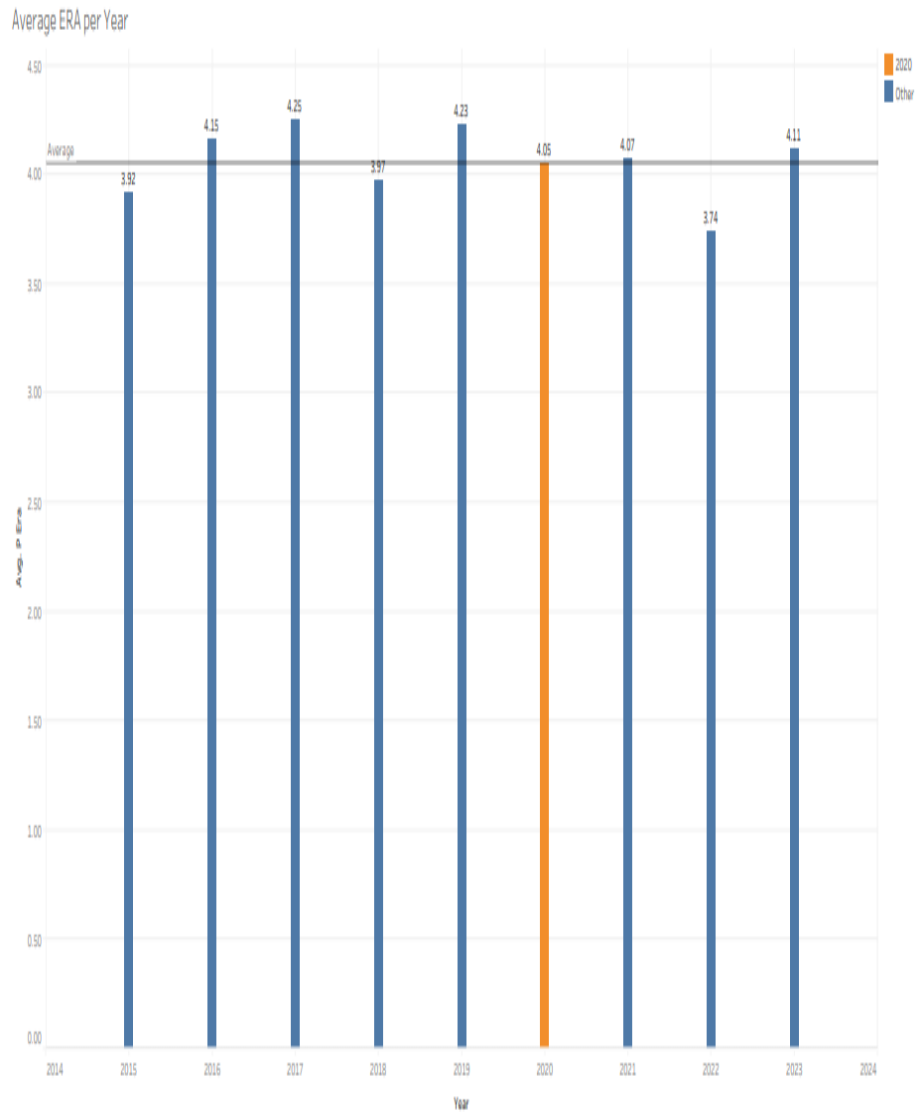


Fig. 2: A pitcher's ERA by year: the year in question, 2020, actually has an average ERA exactly.

While 2020's win-loss percentage was the highest, it isn't drastically different from the win-loss percentage of other years [1](#). It's value of 1.48 still falls within

the max value of

$$\text{max} = Q3 + (1.5 * IQR) = 1.5125 \quad (1)$$

In addition, 2020's ERA is exactly the average ERA of all the years in question 2. So no adjustments will have to be made regarding the shortened 2020 season.

3.2 Generate Potential Supporting Factors

By combining existing variables collected, new factors can be created that may have an impact on whether or not a pitcher receives ample run support.

First up is to create an actual variable for the win-loss percentage of each pitcher which was generated using the following equation:

$$\text{winlosspercentage} = \text{numberofwins} / \text{numberoflosses} \quad (2)$$

If a pitcher has zero recorded losses, the win-loss percentage is assigned to be just the number of wins.

Using what was just calculated, the next step is to determine a few averages: one for the win-loss percentage (which turned out to be approximately 1.37) 1 and another for the ERA (about 4.06) 2.

3.3 Create an Outcome Boolean

Using these calculated averages a variable **can be created** describing if a pitcher was subject to poor run support or not.

The lower a pitcher's ERA is, the better. However, if a pitcher is subject to low run support, his win-loss percentage won't be very great. So to determine whether or not a pitcher is a candidate for low run support, his ERA should be lower than average. But his win-loss percentage would also be lower than average.

Creating a Python lambda function for each condition and combining them with boolean logic gives us our dependant variable.

3.4 Eliminate Irrelevant Factors

Some statics seem to contribute to low run support, but may end up having little or nothing to do with it. Observing the correlation between these statistics and the lack-of-run-support boolean indicates which variables can be removed.

Due to the nature of all these pitching statistics having seemingly little to do with what a pitcher's offense does (which makes the pitcher feel helpless if his/her offense fails to score), extreme correlation values (positive or negative) aren't expected. So to remove any factors that have virtually nothing to do with low-run support, a correlation threshold of 5% is created. Any factor with a correlation between -5% and 5% is then removed from the data frame.


```

# Create lack-of-run-support column
#-> Better (lower) ERA than average, but also a lower winning percentage than average
low_win_pct = df.win_loss_pct.apply(lambda x: True if x < avg_win_loss_pct else False)
low_era = df.p_era.apply(lambda x: True if x < avg_ERA else False)

df['lack_of_run_support'] = low_win_pct & low_era

# Calculate the correlation between each numeric column and the lack-of-run-support column
#-> Remove columns with hardly any correlation (positive or negative)
for i in range((len(df.columns)-2), 1, -1):
    # Calculate the correlation between the statistic and run-support column
    correlation = df.lack_of_run_support.corr(df.iloc[:,i])

    #print(i, df.columns[i], correlation)

    # If the correlation is closer to zero than our given number, get rid of the column
    if (correlation < CORRELATION_THRESHOLD and correlation > -CORRELATION_THRESHOLD):
        df = df.drop(columns=df.columns[i])

```

Fig. 3: The code creates Lack-of-Run-Support variable and drops all factors with a too low correlation to it

Also, a column containing player id numbers came with the initial data set. So in the spirit of deleting unwanted factors, now would be the perfect time to drop this one as well.

3.5 Cleaned Data Set

The columns with a high enough correlation remain for further testing.

Table 2: Remaining Factors

Column Name	Description
last_name, first_name	The name of the pitcher (last, first)
year	What year is being measured
home_run	The number of home runs allowed
k_percent	The percentage of batters faced that struckout
p_earned_run	The number of earned runs allowed
p_run	The number of unearned runs allowed
p_win	The total number of wins the pitcher got
p_loss	The total number of losses the pitcher got
p_era	The Earned Run Average
barrel_batted_rate	The percentage of pitches that hit the barrel of the opponents' bat
hard_hit_percent	The percentage of pitches that were hit over 95 miles per hour
win_loss_pct	The number of wins / the number of losses
lack_of_run_support	Whether or not a pitcher is subject to lesser run support

The name of a player is used as an identifier, the year is to differentiate between multiple instances of the same pitcher, and the `lack_of_run_support` variable is a boolean serving as our dependant variable. The rest are treated as input factors.

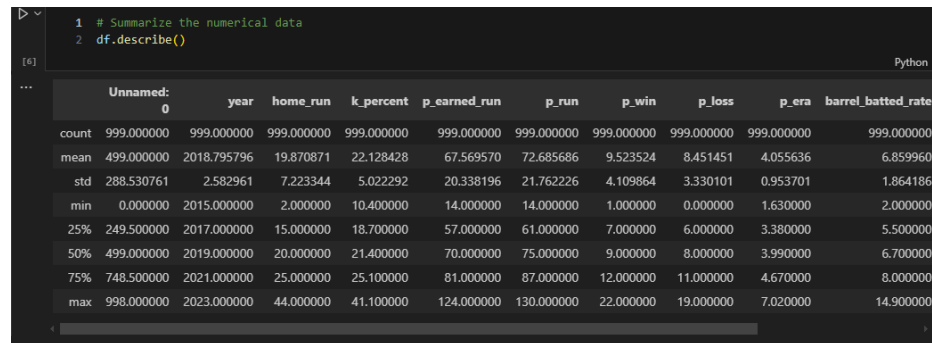
4 Exploratory Data Analysis (EDA)

Now that the data is much cleaner, the analyzing process begins. All the code for this part can be found [here](#). The best place to begin is via a couple of python commands to allow access to basic statistics of each factor.

'`df.shape`' tells how many rows and columns the data set has.

'`df.columns`' gives the name of each column.

'`df.describe()`' provides basic statistics (mean, median, quartiles, etc.) for each numerical column.

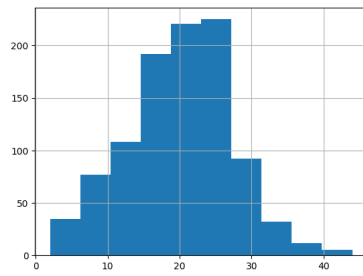


	Unnamed: 0	year	home_run	k_percent	p_earned_run	p_run	p_win	p_loss	p_era	barrel_batted_rate
count	999.000000	999.000000	999.000000	999.000000	999.000000	999.000000	999.000000	999.000000	999.000000	999.000000
mean	499.000000	2018.795796	19.870871	22.128428	67.569570	72.685686	9.523524	8.451451	4.055636	6.859960
std	288.530761	2.582961	7.223344	5.022292	20.338196	21.762226	4.109864	3.330101	0.953701	1.864186
min	0.000000	2015.000000	2.000000	10.400000	14.000000	14.000000	1.000000	0.000000	1.630000	2.000000
25%	249.500000	2017.000000	15.000000	18.700000	57.000000	61.000000	7.000000	6.000000	3.380000	5.500000
50%	499.000000	2019.000000	20.000000	21.400000	70.000000	75.000000	9.000000	8.000000	3.990000	6.700000
75%	748.500000	2021.000000	25.000000	25.100000	81.000000	87.000000	12.000000	11.000000	4.670000	8.000000
max	998.000000	2023.000000	44.000000	41.100000	124.000000	130.000000	22.000000	19.000000	7.020000	14.900000

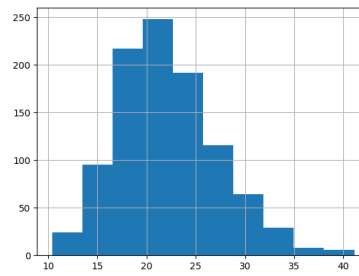
Fig. 4: Output of '`df.describe()`' shows a statistical description of each numeric factor included in the current data set.

4.1 Potential for Further Cleaning

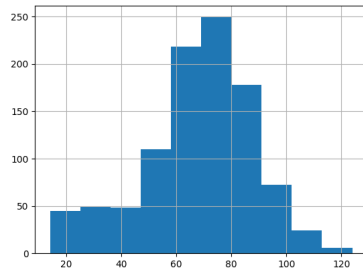
Our data was found earlier to have no null values. The same applies with the data set in its current state. From here, the next step was to create histograms of each numerical variable to check if it is normally distributed as that would be helpful down the road.



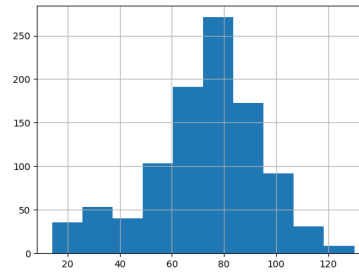
(a) Home Runs Allowed (normal)



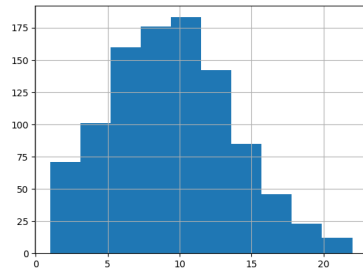
(b) Strikeout Percentage (normal)



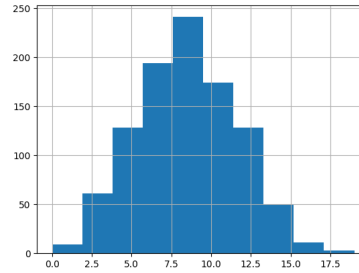
(c) Total Earned Runs Allowed (normal)



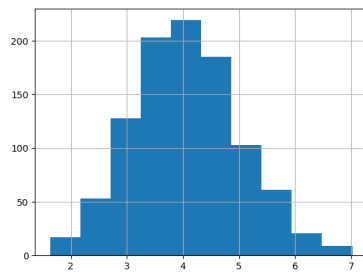
(d) Total Runs Allowed (normal)



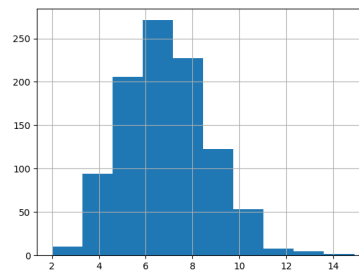
(e) Total Wins (normal)



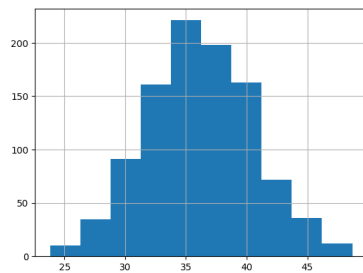
(f) Total Losses (normal)



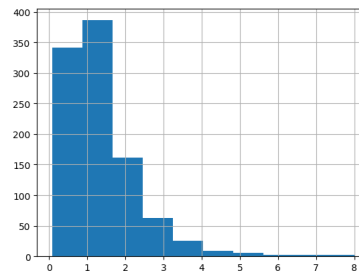
(g) Earned Run Average (ERA) (normal)



(h) Barreled Hits Percentage (normal)



(i) Hard Hit Percentage (normal)

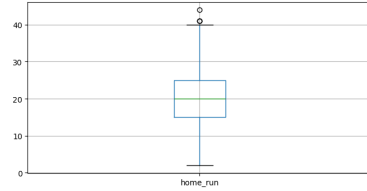


(j) Win/Loss Percentage (not normal)

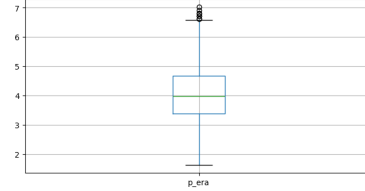
Fig. 5: Through the use of histograms, variables are checked to see if they are normally distributed. In this case, only the last one isn't.

Out of all the variables taken into account, only the win-loss percentage has a distribution that is nowhere near normal.

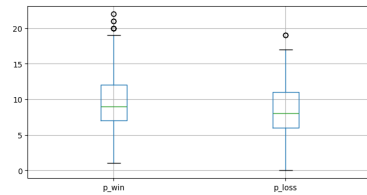
Next was to create box plots in order to check for any major outliers.



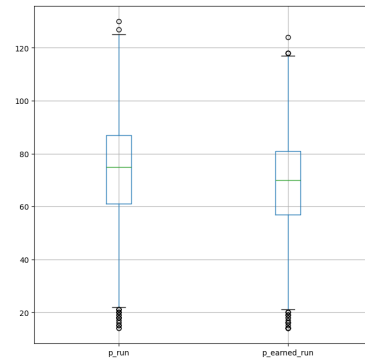
(a) Home Runs Allowed



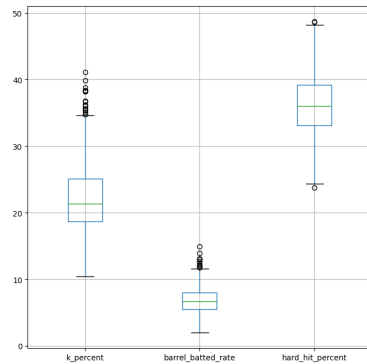
(b) Earned Run Average (ERA)



(c) Wins and Losses



(d) Runs and Earned Runs



(e) Strikeout, Barreled Ball, and Hard Hit Percentage

Fig. 6: By using box plots, outliers can be easily detected. Each variable has some outliers, but none of them are ridiculously removed from the rest of the data points as they all exist due to player performance being better/worse than the rest.

While there are several outliers, none of them show signs of being due to an incorrect input. Some pitchers can have fabulous seasons, while others can fall prey to miserable years. Since all these outliers are explainable by the data, and since they are a valuable part of the information retrieved from the data, they are not to be removed.

Finally, a heat map was created to check for [strong correlations](#) between independent variables.

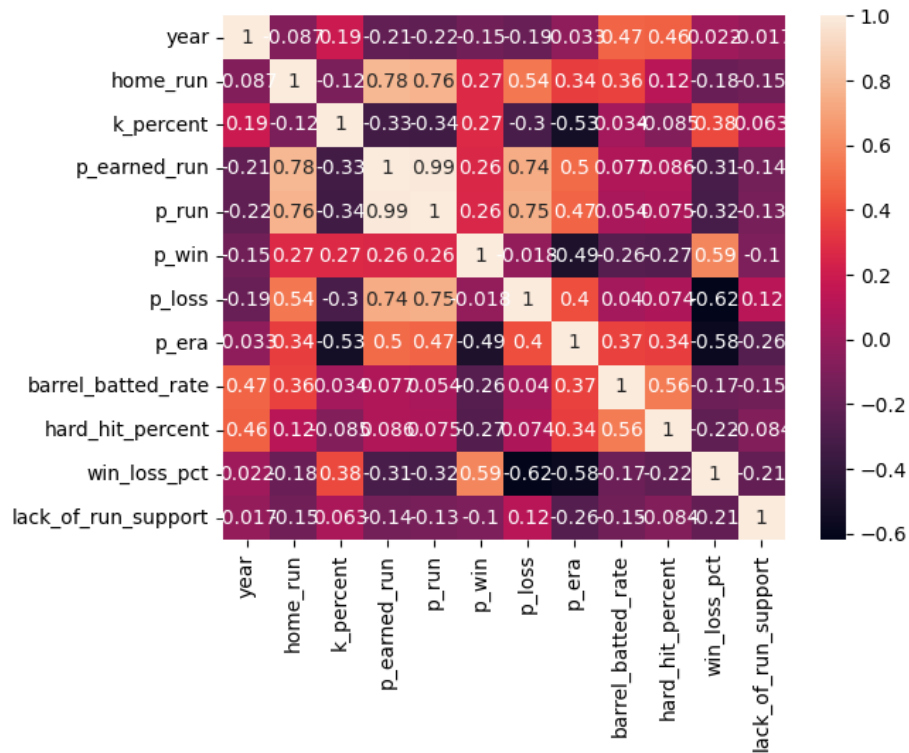


Fig. 7: A heat map shows the correlation between all variables in the data set. Multiple independent variables with a strong correlation to each other suggest they might be related. Thus they aren't independent. So at least one should be removed from the data set.

The heat map shows an extremely strong correlation between the number of runs allowed and the number of earned runs allowed. This is to be expected, but at least one of these variables needs to go.

There is also a strong correlation between each of these two factors and the number of home runs allowed which is also to be expected. Further investigation must be taken to determine how to deal with these correlations.

4.2 Deeper Analysis

Through the creation of pivot tables, a deeper understanding of the data can be achieved. First up was to check how many instances showed potential for a lack of run support.

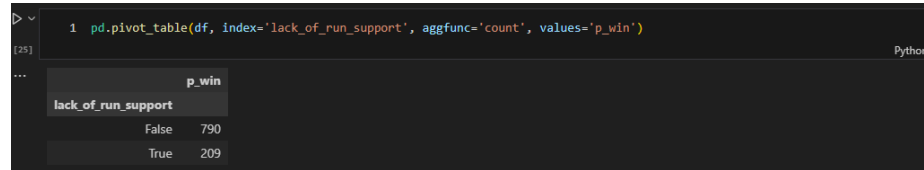


Fig. 8: Pivot table shows the number of pitcher who do and don't receive enough run support.

The next pivot table shows the median for all numeric variables grouped by whether that instance shows a lack of run support.

```
pd.pivot_table(df,
                index='lack_of_run_support',
                aggfunc='median',
                values=['home_run',
                       'k_percent',
                       'p_earned_run', 'p_era', 'p_run', 'p_win', 'p_loss',
                       'win_loss_pct',
                       'barrel_batted_rate', 'hard_hit_percent'])
```

	barrel_batted_rate	hard_hit_percent	home_run	k_percent	p_earned_run	p_era	p_loss	p_run	p_win	win_loss_pct
lack_of_run_support										
False	6.85	36.3	20.0	21.2	71.0	4.28	8.0	76.0	10.0	1.25
True	6.30	35.3	18.0	22.2	65.0	3.65	9.0	71.0	9.0	1.00

Fig. 9: This pivot table measures each remaining statistic separated by whether a pitcher receives enough run support.

The results of this pivot table were a bit surprising. For most factors, a pitcher with a higher chance of low run support pitched to better numbers than those with a more reliable offense. Two of the more intriguing observations were related to strikeouts and runs allowed.

Strikeouts A strikeout is the pitcher's equivalent of a hitter hitting a home run. It's one of the biggest flexes a pitcher can do in-game. And yet, according to this pivot table, more strikeouts equates to more likelihood to receiving low run support.

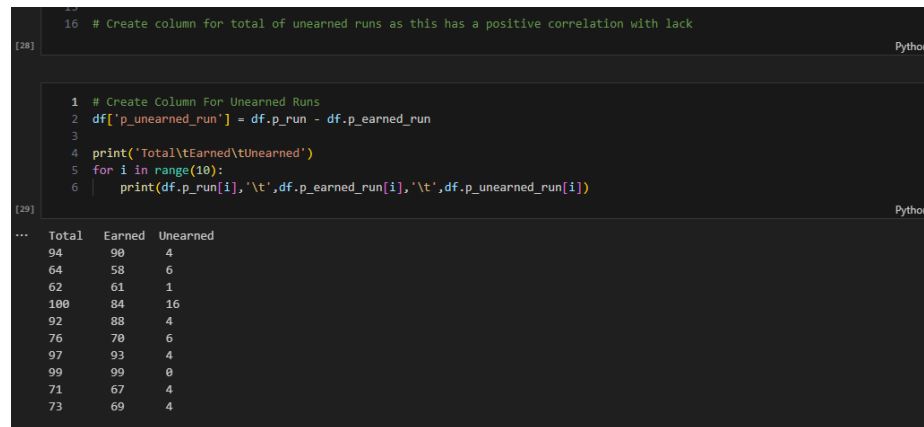
Runs Allowed So far, the data set has kept tabs on earned runs allowed and total runs allowed. But what's the difference? There are some occasions where an opposing player scores due to some sort of error committed by the defense. One such example would be dropping a fly ball when, as a professional athlete who gets paid quite a bit of money, that player is supposed to catch it. So if that batter ends up scoring, it's counted as a run, but it's not considered an earned run because it wasn't earned by the pitcher. Appropriately enough, they are called "unearned runs". The above pivot table shows that, for those who receive less run support, more unearned runs score on average.

4.3 Additional Cleaning

Using the observations addressed above, the data set can be further cleaned and simplified.

Adding New Variables Since there is potential in measuring the total number of unearned runs, a new column was created by subtracting the total number of runs allowed by the total number of earned runs allowed.

```
# Create Column For Unearned Runs
df['p_unearned_run'] = df.p_run - df.p_earned_run
```



The screenshot shows a Jupyter Notebook interface. The top part is a code cell with the following Python code:

```
16 # Create column for total of unearned runs as this has a positive correlation with lack

[28]

1 # Create Column For Unearned Runs
2 df['p_unearned_run'] = df.p_run - df.p_earned_run
3
4 print('Total\tEarned\tUnearned')
5 for i in range(10):
6     print(df.p_run[i], '\t', df.p_earned_run[i], '\t', df.p_unearned_run[i])

[29]
```

The bottom part of the screenshot shows the output of the code, which is a table with three columns: Total, Earned, and Unearned. The table contains 10 rows of data:

	Total	Earned	Unearned
94	90	4	
64	58	6	
62	61	1	
100	84	16	
92	88	4	
76	70	6	
97	93	4	
99	99	0	
71	67	4	
73	69	4	

Fig. 10: The code creates an additional column that calculates how many unearned runs a pitcher lets up in a year.

Drop Unwanted Columns The first column to go is an index column that was generated automatically when creating the cleaned data set.

```
# Drop the 'unnamed' column
df = df.drop(columns='Unnamed: 0')
```

There are also columns that don't relate to anything a pitcher can do during a game to improve his chances of receiving help from his offense... mainly the number of wins and losses per year. So these columns can be dropped as well.

```
# Drop columns that have an end result (wins, losses)
df = df.drop(columns=['p_win', 'p_loss'])
```

Finally, the columns used to create the lack-of-run-support column will have a ridiculously strong impact on determining the dependent variable. And since they are also end-of-game statistics, they can go as well.

```
# Remove variables that contributed with the creation of lack_of_run_support variable
df = df.drop(columns=['win_loss_pct', 'p_era'])
```

Remove Variables with High Correlation After adding removing all these variables, the heat map has changed quite a bit.

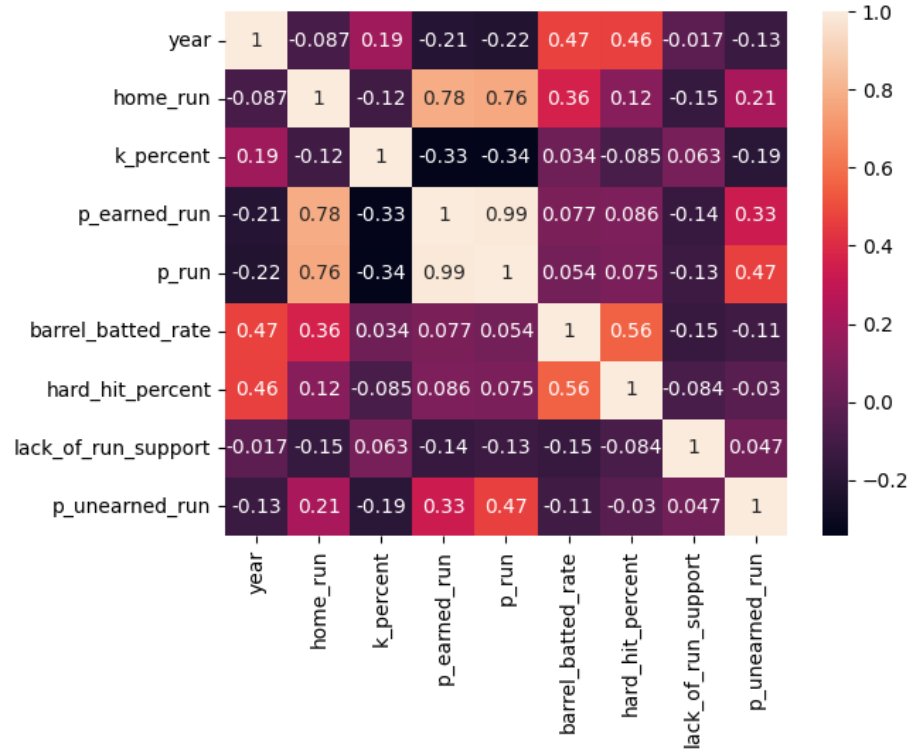


Fig.11: The heat map is used again for the same purpose. There exist some strong correlations between different variables that need to be addressed.

As mentioned before, there is a near-perfectly positive correlation between the number of total runs allowed and the number of earned runs allowed. So at least one of them is being dropped. But which one?

Observing the correlation between each of these variables and the lack-of-run-support column gives an idea of which to remove. Whichever one has a correlation closer to zero is to be less impactful if kept. So that's the one to drop.

Table 3: Correlations with lack-of-run-support Factor

Category	Correlation
Runs Allowed	-0.13
Earned Runs Allowed	-0.14

Since the number of runs allowed has a lesser correlation, that's the one to remove.

It was also mentioned earlier that the number of home runs allowed had a strong correlation with both runs allowed and earned runs allowed. Since the number of runs allowed is to be dropped, the next task is to decide whether to remove the number of home runs allowed or the number of earned runs allowed. The process is exactly the same.

Table 4: Correlations with lack-of-run-support Factor

Category	Correlation
Home Runs Allowed	-0.15
Earned Runs Allowed	-0.14

In this case, the number of home runs allowed is the winner. Therefore, the number of earned runs is also dropped.

```
df = df.drop(columns=['p_earned_run', 'p_run'])
```

4.4 Sort and Export

With an ample supply of data still at hand, the last bit before moving on is to send the revised data frame to its own CSV file. But first, it can be organized a bit better.

```
df = df.sort_values(['last_name', 'first_name', 'year'], ascending=True)
```

Now that the data is sorted more appropriately, it's time to export.

```
# Make sure to set index to False to avoid unwanted index columns
df.to_csv('stats_EDA.csv', index=False)
```

The use of Exploratory Data Analysis helped determine which variables are no longer necessary to our data set. It also brought forth the logic behind creating new variables. Each step in this process all compiles into creating a [new data set](#) to work with.

5 Machine Learning Processes

The data is nicely organized and ready for machine learning. All the code for machine learning can be found by [clicking here](#). Since the output variable of this data set is a boolean, the style of machine learning algorithms to consider using must involve classification.

5.1 Ready the Data

But before performing any sort of machine learning, the data itself can be better suited. Doing so leads to a smoother process while training and testing the models.

Independent Variables To avoid bias, all the input factors should be normalized. This was accomplished using feature scaling which ranges all the values from zero to one.

$$X_{normalized} = (X - X_{min}) / (X_{max} - X_{min}) \quad (3)$$

Performing feature scaling on every input variable lets each one have a fair say in determining how impactful it is to a lack of offensive run support.

Dependent Variable Being a boolean, the output variable has only two values: "True" and "False". But to help simplify any algorithms to be used, it would be more ideal to swap all the "True" values with ones and all the "False" values with zeros.

```
# Turn the True/False values of our output to 1/0
df_ml.lack_of_run_support = df_ml.lack_of_run_support.apply(lambda x: 1 if x==True else 0)
```

5.2 Split the Data

The data then gets split into two categories: one for training the machine learning model (80%) and the other for testing the model (20%). In order to do this using the desired library, Sklearn, the input variables need to be grouped into one array.

```
# Turn the independent variables into an array to allow splitting
indep_vars = df_ml[['home_run',
                    'p_unearned_run',
                    'k_percent',
                    'barrel_batted_rate',
                    'hard_hit_percent']].to_numpy()
```

Then the call to split the data can be made.

```
# Assign the training and testing variables
X_train, X_test, y_train, y_test = train_test_split(
    indep_vars,
    df_ml['lack_of_run_support'],
    test_size=0.2,
    random_state=11)
```

The random state is kept the same to provide consistency throughout the process.

5.3 Determine Appropriate ML Model

There are several models to consider. The type of outcome indicates what kind of models to look for, but there are still plenty of classification models to sift through.

Creating and Testing the Models Five different machine learning algorithms were selected: Logistic Regression, Decision Trees, Random Forest, Naive Bayes Theorem, and K-Nearest Neighbor (KNN). These were graded using the scoring methods of accuracy, precision, recall, mean squared error (MSE), root mean squared error (RMSE), mean absolute error (MAE), and R-Squared values [12](#).

For Logistic Regression, Decision Trees, and Naive Bayes, the approach was almost exactly the same. A model was fit using the training data, predictions were made on the testing data inputs, and those predictions were compared to the testing data outputs. Additional work was required from the other two methods. For the Random Forest algorithm, a sample of 1000 trees were taken. The depth (number of branches to go down) also needed to be taken into consideration. So using a loop, the scores of Random Forests were scored using one through ten branches.

Max Depth	Accuracy	Precision	MSE	RMSE	MAE	Recall	R2
1	0.79	0.0	0.21	0.46	0.21	0.0	-0.27
2	0.79	0.0	0.21	0.46	0.21	0.0	-0.27
3	0.79	0.0	0.21	0.46	0.21	0.0	-0.27
4	0.80	1.0	0.20	0.45	0.20	0.024	-0.21
5	0.8	1.0	0.2	0.45	0.2	0.048	-0.21
6	0.805	1.0	0.195	0.44	0.195	0.071	-0.18
7	0.805	1.0	0.195	0.44	0.195	0.071	-0.18
8	0.81	1.0	0.19	0.44	0.19	0.095	-0.15
9	0.805	0.67	0.195	0.44	0.195	0.14	-0.18
10	0.795	0.57	0.205	0.45	0.205	0.10	-0.24

Table 5: Scores for each Random Forest by Branch Count

The first three rows are exactly the same. The next two rows are identical to each other as well. In fact, there are only two inputs that don't have a precision value of zero (indicating no "True" values) or one (showing the possibility of overfitting). And out of the two, the input with nine branches scores better on every other scoring category. So the random forest of 1000 trees with nine branches moves on to the final testing.

The story is similar with the KNN algorithm. There are a plethora of options as to how many neighbors the model should have. So using the same for-loop structure, the model was tested using an increasing odd number of neighbors. Odd numbers are used to avoid the case of a tie. Afterward, the scores were

Num Neighbors	Accuracy	Precision	MSE	RMSE	MAE	Recall	R2
3	0.76	0.41	0.24	0.49	0.24	0.31	-0.45
5	0.785	0.46	0.215	0.46	0.215	0.14	-0.30
7	0.765	0.27	0.235	0.48	0.235	0.071	-0.42
9	0.77	0.17	0.23	0.48	0.23	0.024	-0.39
11	0.795	0.57	0.205	0.45	0.205	0.095	-0.24
13	0.785	0.4	0.215	0.46	0.215	0.048	-0.30
15	0.8	1.0	0.2	0.45	0.2	0.048	-0.21

Table 6: Scores for each KNN by Neighbor Count

While the row containing 15 neighbors seems the best, it also has a perfect precision value. So having 15 neighbors may be too many as there is potential for overfitting the model. In almost every score, the row with 11 neighbors is the next best. So the winning total of neighbors is 11.

Selecting the Most Appropriate Model All five algorithms have been created, tested, and graded using the seven scoring methods.

Algorithm	Accuracy	Precision	MSE	RMSE	MAE	Recall	R2
Logistic Regression	0.79	0.0	0.21	0.46	0.21	0.0	-0.27
Decision Trees	0.71	0.30	0.295	0.54	0.295	0.31	-0.78
Random Forest (9 branches)	0.805	0.67	0.195	0.44	0.195	0.143	-0.18
Naive Bayes	0.79	0.0	0.21	0.46	0.21	0.0	-0.27
KNN (11 neighbors)	0.795	0.57	0.205	0.45	0.205	0.095	-0.24

Fig. 12: Each machine learning model is scored using all the scoring methods listed. The one with the best combination of scores is to be declared the optimal model.

Out of the five models, two have major problems right off the bat. Both Logistic Regression and Naive Bayes have a precision value of 0.0. So because there are other options, these two won't make the cut. Ranking the other three models for each score shows which the winner is.

Score	Best	Middle	Worst
Accuracy	Random Forest	KNN	Decision Tree
Precision	Random Forest	KNN	Decision Tree
MSE	Random Forest	KNN	Decision Tree
RMSE	Random Forest	KNN	Decision Tree
MAE	Random Forest	KNN	Decision Tree
Recall	KNN	Random Forest	Decision Tree
R2	Random Forest	KNN	Decision Tree

Table 7: Rankings Show Random Forest as the Clear Winner

Out of the ones created and tested, the Random Forest model with 9 branches prevails as the best machine learning algorithm.

6 Results

Now that a model is created that can predict (with 80% accuracy) whether or not a pitcher's offense has his back, the leading cause of lesser run support can finally be revealed. The model validates the correctness of the results which can be found [here](#).

To find out which pitching statistic ties the strongest to low offensive support, correlations for each input variable are generated.

```

1 # Create correlation variables
2 corr_home_run = df.lack_of_run_support.corr(df.home_run)
3 corr_unearned_run = df.lack_of_run_support.corr(df.p_unearned_run)
4 corr_k_percent = df.lack_of_run_support.corr(df.k_percent)
5 corr_barrel = df.lack_of_run_support.corr(df.barrel_batted_rate)
6 corr_hard_hit = df.lack_of_run_support.corr(df.hard_hit_percent)

[3] ✓ 1.1s Python

1 print('Home Runs \t\t Unearned Runs \t\t Strikeout Percent \t Barreled Ball Rate \t Hard Hit Percent')
2 print(corr_home_run, '\t', corr_unearned_run, '\t', corr_k_percent, '\t', corr_barrel, '\t', corr_hard_hit)

[15] ✓ 0.0s Python

... Home Runs      Unearned Runs      Strikeout Percent      Barreled Ball Rate      Hard Hit Percent
-0.151351670225487      0.04706934633333991      0.06297838356519922      -0.14665210673139123      -0.08442802549283343

```

Fig. 13: The code lists the correlations of the five remaining factor with the lack-of-run-support variable.

The correlations are then [graphed](#) to compare the results.

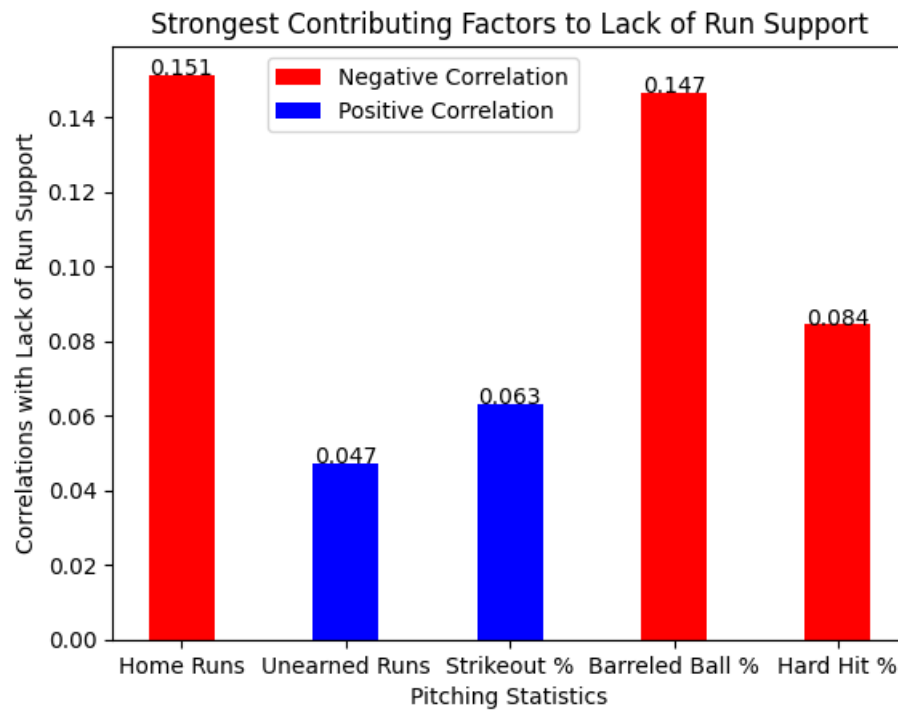


Fig. 14: Bar plot of each correlation shows home runs allowed to be the leading factor to receiving worse offensive support.

Though not super high (only 15%), letting up home runs has the strongest correlation with receiving lackluster run support. What makes this really interesting is that the correlation between the two variables is negative. Therefore, the fewer home runs a pitcher allows, the more likely he is to receive less help from his offense as shown below.

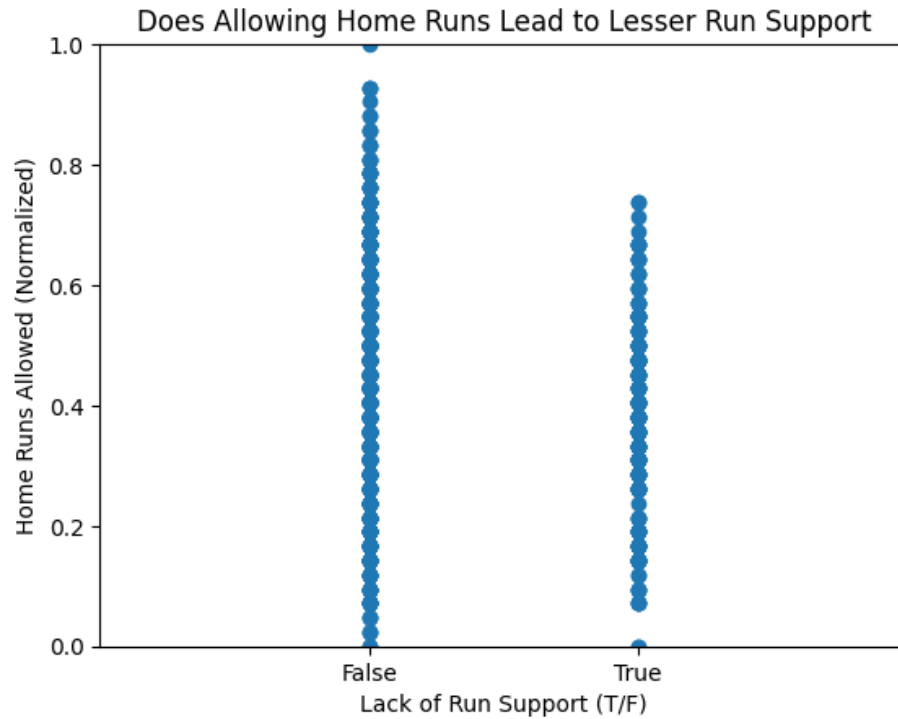


Fig. 15: The dots indicate that allowing MORE home runs increases one's chance of having a better supporting offense.

Many different factors can contribute to this such as the pitcher constantly throwing strikes and letting up solo home runs (nobody is on base when the home run is hit). Thus that pitcher isn't walking any batters, hitting any batters, or even running into deep counts (high number of balls and strikes in each at bat). All of these add up to make an inning last much longer which can mentally fatigue the pitchers teammates. Another possible reason is that the pitcher is just downright bad at pitching and lets up too many runs for even a good offense to recover from. Alternatively, there could be two juggernaut offenses going at each other resulting in high-scoring games. Whatever the reason, the numbers show that, as a pitcher, it's ok to let up home runs every now and again. There is no need to be afraid of letting up hard, barreled contact. The team's offense is there to provide support.

7 Conclusion

No matter how hard a pitcher tries, there are times in which he feels like his offense is letting him down and there is nothing he can do about it. But through

this study, it is determined that this is not always the case. By actually allowing more pitches to hit the barrel of opposing bats, leading to home runs, a pitcher puts his offense in a more likely situation where they score as well. It can also be determined, through a trained Random Forest machine learning model, which pitchers appear to be falling into a lack-of-offense funk. Through the insights gained in this project, teams can be better equipped to select appropriate pitchers for their roster, provide fixes for pitchers who are struggling to receive support, and most importantly, win more games.

References

1. Bradbury, J.C.: Does the baseball labor market properly value pitchers? (2007)
2. Gitter, S.R., Rhoads, T.A.: If you win they will come: fans care about winning in minor league baseball (2010)
3. Goold, K.L., Aniga, R.N., Gray, P.B.: Sports under quarantine: A case study of major league baseball in 2020 (2020)
4. McElroy, L.: Computer vision in baseball: The evolution of statcast