Minato Sky Myers, Zach Hracho, and Eric Powers

MLB Statistics Report

Introduction

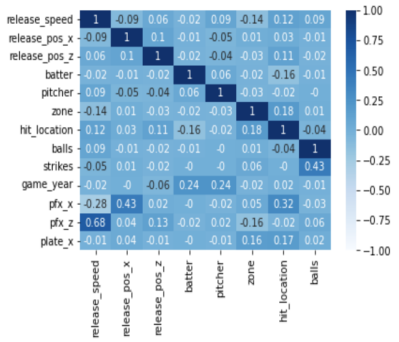
There were three main ideas throughout the completion of the MLB Statistics Capstone Project. Main topics of code include data visualization, web scraping, and machine learning fundamentals. Zach Hracho was responsible for data visualization, Eric Powers worked on web scraping, and Minato Myers completed the machine learning fundamentals section. Our group focused on a general overview of MLB statistics and its applications. This project can be helpful in determining drafts for fantasy baseball and useful for overall analysis of the top MLB players. All of the code was run through Jupyter Notebook using Anaconda. Certain modules may have already been present on our computers, but running on another computer may not have specific libraries present. We tried to take care of this problem by installing these libraries with the pip install magic command through Jupyter Notebook. However, if there are additional Module Not Found Errors, the library will have to be installed manually. BeautifulSoup, Seaborn, Matplotlib, Numpy, Requests, PyBaseball, scikit-learn, and Statsmodels were the libraries utilized for this project.

Data Visualizations

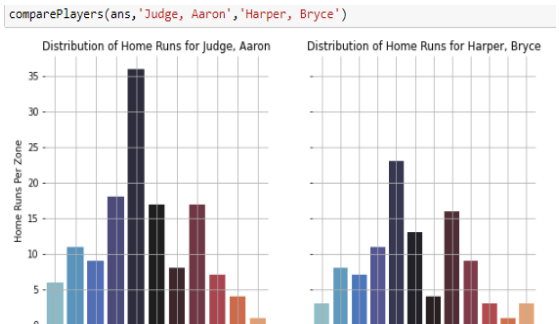
Most of the data visualization and analysis focused on home runs. In order to collect data, BaseballSavant ([13]), a MLB statistics website, was used to filter the data automatically by home runs. Initially, our goal was to include hitting data for multiple hit types. However, the

website would not support this functionality, so the analysis of homeruns was decided upon. There was one downside to this website: it would often time out when it was downloading the data for multiple seasons. Upon research, it was discovered that BaseballSavant only supported capacity for two seasons. To overcome the flaw in this website, multiple csv files of seasons were downloaded and concatenated into one large pandas dataframe (approximately 20,000 rows). Additionally, many of the columns of data had missing values (NaN) which had to be removed before the process of data visualization and machine learning had started.

To analyze the data, a correlation matrix in matplotlib was created. Since there were so many rows and columns present in the dataframe, the heatmap was nearly impossible to read. Therefore, only certain rows and columns were included to make the matrix easier to read and understand. A correlation matrix allows us to examine the relationships between variables based on a negative one to one scale where a positive one exhibits a strong positive relationship. Once these relationships were visualized, we can examine the correlation between home runs and the strike zone.



Our group was curious with the comparison of home runs by zone of two players. In order to compute these instructions, a function was created which required a pandas dataframe as well as two player names in the form of strings. Subplots were created with a shared y-axis to properly see the difference in home runs. Initially, the sharey attribute of the subplots function was not included, which made it difficult to distinguish a comparison. A countplot of the first player's data was created, counting the number of events (homeruns) in each zone. The next

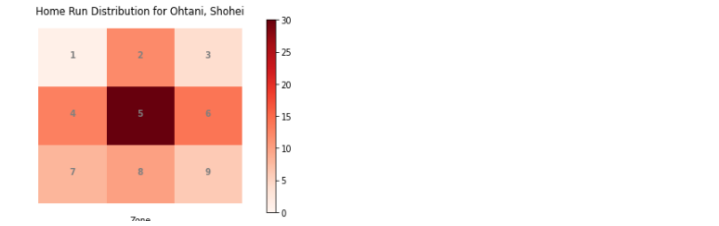


step was repeated, to produce the output on the right. This function allows for code reuse when we want to compare the homeruns and hitting statistics of other players of interest.

The next portion of interest, and one of the most complex models, was to create a heatmap visualization of the strike zone consisting of all the home runs of a specific player. Similar to how websites like MLB and BaseballSavant made their distributions, it allows the user to visualize the strike zone and formulate which zones correlate to the most homeruns. The plotHeatMap function allows users to type any fragment of a player's name, returning the corresponding heatmap of the player. The built-in seaborn heatmap function did not produce the desired result, so the DataCamp (https://projects.datacamp.com/projects/250) website was used to gather logic that a corresponding cartesian coordinate system relevant to each zone would be needed. Besides this site, only seaborn and matplotlib documentation was used to assist in visualization. The despine method and ax.set functions helped remove any white spaces and labels initially messing up the formatting of the heatmap. Text was also added to each bin in order to label each strike zone. A majority of time was spent trying to include the foul zone as well, overlapping the two heatmaps to visualize what an actual strike zone would look like in baseball. However, after many

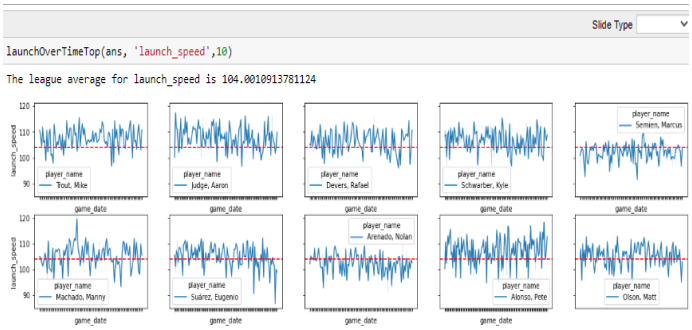
```
def y_coord(row):
    if row['zone'] in [1,2,3]:
        return 3
    elif row['zone'] in [4,5,6]:
        return 2
    elif row['zone'] in [7,8,9]:
        return 1


#Assigns new columns with x and y coordinates in order to graph the heatmap of homeruns
df['xcoord'] = df.apply(x_coord, axis = 1)
df['ycoord'] = df.apply(y_coord, axis = 1)

#Using seaborn, creates a histogram plot (heatmap) with a bar for the user to translate the color on the
ax = sns.histplot(data = df, x  = 'xcoord', y = 'ycoord', discrete = True, cmap ='Reds',cbar = True)
#Removes the left and bottom spine of the graph
sns.despine(left=True, bottom = True)
#Removes the x and y ticks from the graphs
ax.set(yticks=[], xticks = [])
ax.set_xlabel('Zone')
ax.set_ylabel("")
ax.set_title(f"Home Run Distribution for {playerName}")

#Count initializes in order to for the strike zone to be illustrated by the graph
count = 0

#Iterates through the cartesian plane (x and y axis to annotate the heatmap with the strikezones)

for y in reversed(range(1, 4)):
    for x in range(1,4):
        count += 1
        plt.text(x, y, count, horizontalalignment='center', size='medium', color='grey', weight='heavy')
plt.show()
```

```
plotHeatMap(ans, 'Shohei')
```

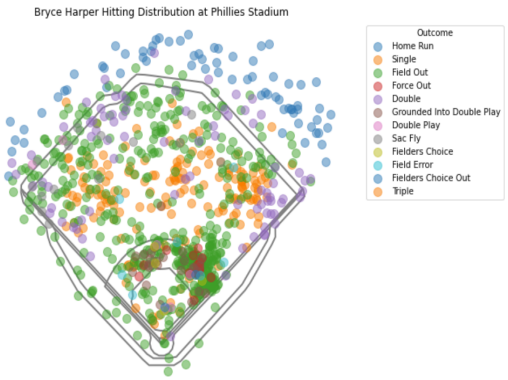


attempts, our group was unable to figure out how to properly overlap these two graphs, instead focusing only on the strike zone.

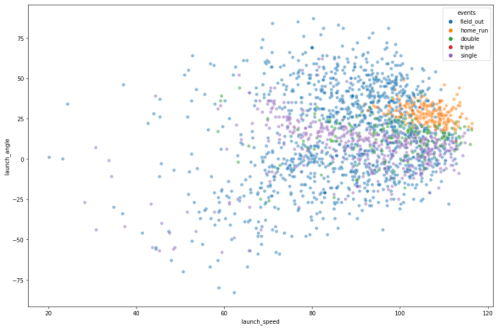
Next, I was curious in researching the relationship between certain quantitative variables that may change throughout the season between the top hitters. I wanted to analyze what makes these players stand out in leading home runs. A function was created which analyzed the top n players' (determined by the user) specified hitting statistics throughout their four seasons. Judging from their launch speed, we can determine that it is variable throughout the season, and there really is no correlation. A horizontal line was added in each subplot showing the league average for the specified variable, allowing for further comparisons to be made. To make this model more accurate, more hitting data could be included instead of data consisting of only home runs. There are some downsides with this function, however. Our group was unable to figure out how to deal with the top n players which are odd numbers. While there are still subplots provided, it does not produce all nine players for example, and it was difficult to add the remaining odd graph below.



Nearing the end of this project, a small user-made library named PyBaseball ([14]) was discovered. This library allowed for data to be scraped from BaseballSavant and downloaded locally to the computer. While this method took a large amount of time to properly gather the data, it was one possible solution to avoid constantly concatenating multiple data frames as previously mentioned. We were curious in looking at overall hitting statistics for players, mostly their distribution throughout a specific season. With this



library, we could search for a player by id and find their total hit distribution in a given stadium. It even provides a graphic of the stadium, including where all of the hits were marked. Our group made a function that allows for a user to type in any player, the seasons they want to analyze, and the stadium in order to receive a graph similar to the one above. As a Phillies fan, I also wanted to include a graphic of Bryce Harper's hitting statistics based on launch speed and launch angle ([15] was inspiration). With PyBaseball, I initiated a request for his data for his entire career, only focusing on the launch angle, speed, and hit event. A scatterplot was then created with all of this information to determine the relationship between these three variables. As seen by the graph, Bryce Harper's sweet spot for a home run is a launch speed of 110 and a launch angle of thirty degrees.



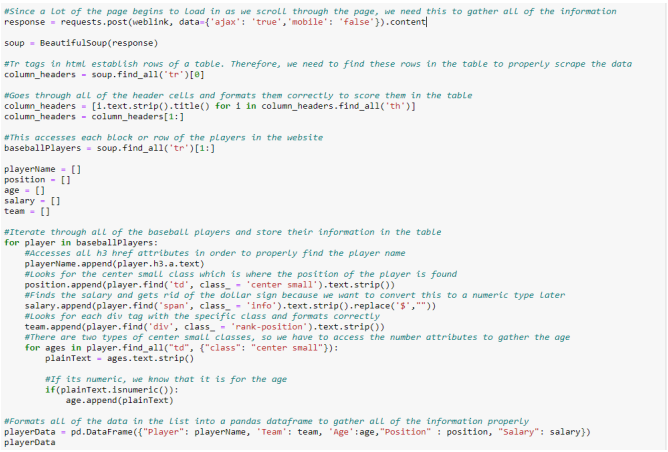
Web Scraping

The web scraping portion of this assignment was utilized to gather data to compare a player's salary with other statistics. We wanted to examine the relationship between a player's salary and their hitting/pitching/fielding statistics. With this newfound information, visualizations and rudimentary machine learning models (linear regression and an SGDC Classifer) were created. For the purposes of this assignment, BeautifulSoup and pandas read_html function were helpful in scraping the data.

The web scraping journey started on a website called spotrac.com. This website keeps track of MLB player's salaries as well as a few other identifiers like age and position. The code to scrape the 2020 season salary data from this website follows below. One of the main problems

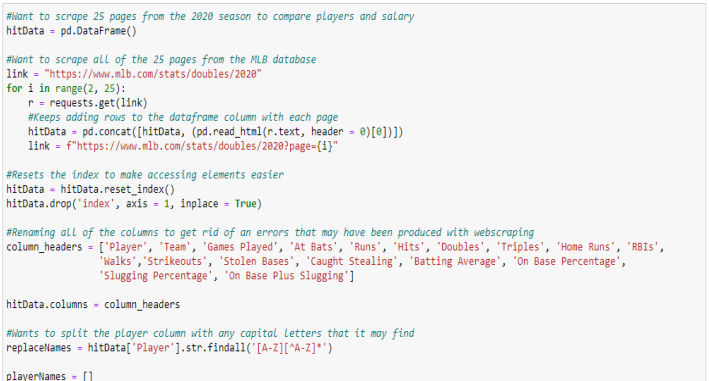
encountered was that this website loaded the data as the user continually scrolled down the page. With this error, web scraping the data would only return the first one-hundred rows of the table. To fix this problem, we had to fix the HTTP request, simulating browser activity so the website would load all of the information instead of the first 100 rows.

Column headers were scraped and properly formatted using a simple list comprehension, searching for all tr tags (defines a row of cells in a table in HTML). Empty lists were then initialized in order to store the information that was soon to be scraped, later being converted to a pandas dataframe. Each cell, or tr tag, contained all of the information for each baseball player, stored in a list named baseballPlayers. Iterating through the entire baseball roster allows us to break the scraping into individual players, appending their information to the lists and moving to the next tr tag once complete. The find() method was used within the for loop instead of find_all() because we only wanted to access one element for the player. Specific tags and class ids were passed in to the find function in order to retrieve the HTML text of that section. Once stored, an additional for loop was needed due to a problem with the player position and age being stored under the same class id. The workaround for this was to check each element under the 'center small' id and then to check if it was numeric to break up the position data from the age data. Ultimately, this information was stored within a pandas dataframe named playerData.



```
#Since a lot of the page begins to load in as we scroll through the page, we need this to gather all of the information
response = requests.post(weblink, data={'ajax': 'true','mobile': 'false'}).content

soup = BeautifulSoup(response)

#Tr tags in html establish rows of a table. Therefore, we need to find these rows in the table to properly scrape the data
column_headers = soup.find_all('tr')[0]

#Goes through all of the header cells and formats them correctly to score them in the table
column_headers = [i.text.strip().title() for i in column_headers.find_all('th')]
column_headers = column_headers[1:]

#This accesses each block or row of the players in the website
baseballPlayers = soup.find_all('tr')[1:]

playerName = []
position = []
age = []
salary = []
team = []

#Iterate through all of the baseball players and store their information in the table
for player in baseballPlayers:
    #Accesses all h3 href attributes in order to properly find the player name
    playerName.append(player.h3.a.text)
    #Looks for the center small class which is where the position of the player is found
    position.append(player.find('td', class_ = 'center small').text.strip())
    #Finds the salary and gets rid of the dollar sign because we want to convert this to a numeric type later
    salary.append(player.find('span', class_ = 'info').text.strip().replace('$',""))
    #Looks for each div tag with the specific class and formats correctly
    team.append(player.find('div', class_ = 'rank-position').text.strip())
    #There are two types of center small classes, so we have to access the number attributes to gather the age
    for ages in player.find_all("td", {"class": "center small"}):
        plainText = ages.text.strip()

        #If its numeric, we know that it is for the age
        if(plainText.isnumeric()):
            age.append(plainText)

#Formats all of the data in the list into a pandas dataframe to gather all of the information properly
playerData = pd.DataFrame({"Player": playerName, 'Team': team, 'Age':age,"Position" : position, "Salary": salary})
playerData
```

The next step was to scrape the hitting data from [mlb.com](mlb.com). This data would allow us to



```
#Want to scrape 25 pages from the 2020 season to compare players and salary
hitData = pd.DataFrame()

#Want to scrape all of the 25 pages from the MLB database
link = "https://www.mlb.com/stats/doubles/2020"
for i in range(2, 25):
    r = requests.get(link)
    #Keeps adding rows to the dataframe column with each page
    hitData = pd.concat([hitData, (pd.read_html(r.text, header = 0)[0])])
    link = f"https://www.mlb.com/stats/doubles/2020?page={i}"

#Resets the index to make accessing elements easier
hitData = hitData.reset_index()
hitData.drop('index', axis = 1, inplace = True)

#Renaming all of the columns to get rid of an errors that may have been produced with webscraping
column_headers = ['Player', 'Team', 'Games Played', 'At Bats', 'Runs', 'Hits', 'Doubles', 'Triples', 'Home Runs', 'RBIs',
                  'Walks','Strikeouts', 'Stolen Bases', 'Caught Stealing', 'Batting Average', 'On Base Percentage',
                  'Slugging Percentage', 'On Base Plus Slugging']

hitData.columns = column_headers

#Wants to split the player column with any capital letters that it may find
replaceNames = hitData['Player'].str.findall('[A-Z][^A-Z]*')

playerNames = []
```

analyze the correlation between runs batted in (RBI's), home runs, and strikeouts for each player. The code to scrape the website is shown on the right.

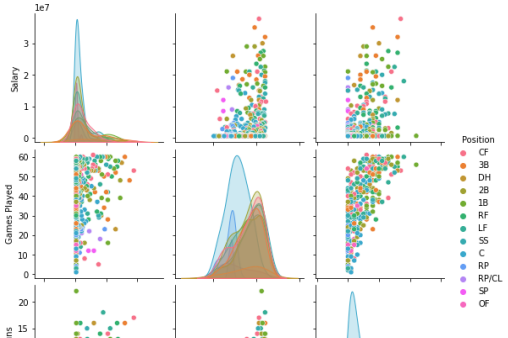
Since the website consisted of twenty five pages, a table on each page, the pandas read_html was useful. The requests library was used to constantly update the web link because the data was stored on different web pages. An empty dataframe was initialized, with the tables constantly being appended to the end. Once a similarity was found in the webpage, a for-loop was utilized to update the web link and send a request to read the next page. The Selenium library would have been useful when working with multiple pages, but was difficult to implement with pandas. There were some minor errors in formatting, which were fixed with regular expressions.

Having two separate dataframes would not make it easy to analyze and visualize the data. The two dataframes were merged using an inner join with the keys being their name and team. With this inner join, the pandas library looks for similarity between the two datasets (on the player and team column) and creates a resulting dataset with these matches. The final set will contain the salary of each player as well as their hitting statistics. Salary data was also changed from integers to floats to make calculations easier.

```
#Inner join to add the salary column to properly make calculations on the data
overallData = pd.merge(playerData, hitData, on = ['Player','Team'], how = 'inner')

#Convert the salary column to a float to make performing calulations on easier
overallData['Salary'] = overallData['Salary'].str.replace(',', "").astype(float)
overallData
```

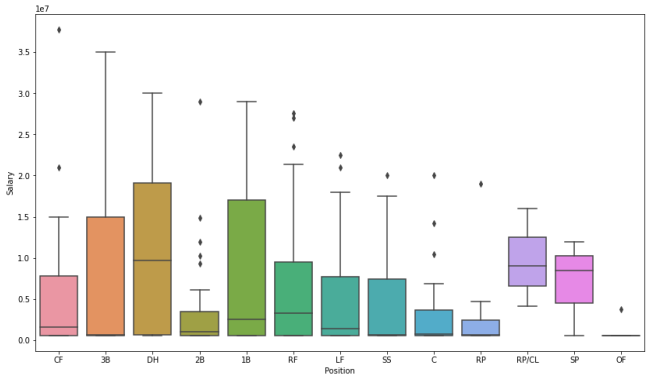
Additional Visualizations

Now it was finally time to make some visualizations with this data. The first graph created was a pairplot using the seaborn library.

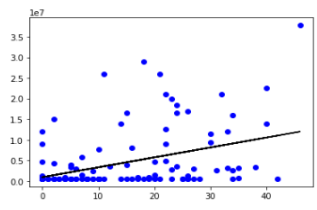


The pairplot is a fun way to see correlation between different variables. Similar to the correlation matrix above, it is able to measure the strength of relationships with scatterplots and KDE plots. In this case, correlation between salary, games played, and home runs is being visualized through scatterplots. Interpreting the top right graph, we can determine that as the number of home runs hit by a player increases, their salary is likely to increase (positive correlation). The data is also being categorized by position as shown through the legend on the right. The diagonals of this grid use a layered kernel density estimate, showing how closely clustered points of the same category are.

One of the next visualizations created was a boxplot. Utilizing the seaborn library once again, a box plot was created for each position that shows their respective salary distributions. This graph was especially interesting to analyze. It clearly demonstrates that the lowest paid position is an OF (a utility outfielder). Since they are utility players, they are likely to serve as a backup in case a player gets injured, hence the low salary. It is also interesting to see that even though center fielders (CF's) do not get paid that much respectively, Mike Trout still got paid the highest in the league for that year (the outlier of roughly forty million). This outlier is most likely due to the success he has brought the Angels and the excitement fans get watching him play.



Predictive Modeling



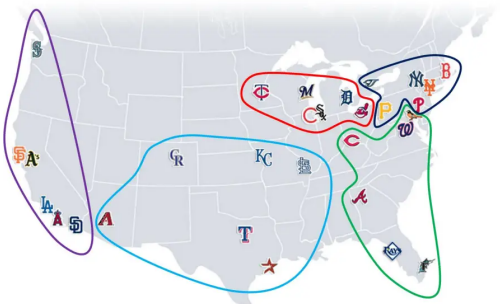

```
rbiRang = np.arange(1,20)
for rbis in rbiRang:
```

Next, we wanted to broach the topic of machine learning to predict the salary of a player given certain hitting statistics with the data web scraped (in this example RBIs). Firstly, we had to break up the dataset into training and test sets. This method allows the LinearRegression model to find a line of best fit with the data points. A test_size of .3 indicates that thirty percent of the dataset will be used to test the machine learning model to make predictions. While it did not receive a high R^2 value, there was not a large correlation between the two variables in the beginning. In the future, more data points would allow for the model to become more accurate. Additionally, the use of a neural network would have performed much better in this scenario if extra time permitted. As we can see, the higher the RBI count between players indicates a higher salary.

Predicting MLB Division using Machine Learning

Previously, we used a simple linear regression model to predict a player's salary solely using their RBI, but since this was unsatisfactory, we then considered other predictions that can be made by utilizing models. The first is predicting what division a given MLB team belongs in, whether it be the East, Central, or West division. While this may seem trivial, as given the temperature and location of a match the division would be obvious, we would like to determine if there are other factors that can help conclude the division.

Regarding the data collection, we retrieved a subset of data on MLB teams from Lahman's Baseball Database [3,4]. This dataset contains statistics of each team in each year they played, including the number of games played and total attendance. The first step we took was to retain only relevant features for the prediction

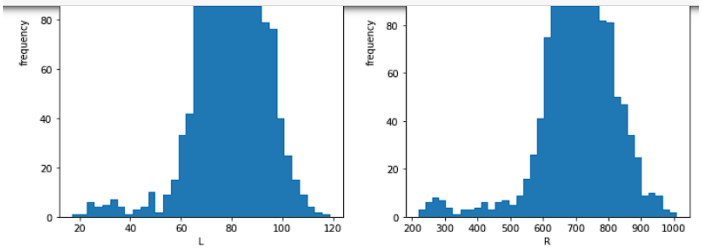


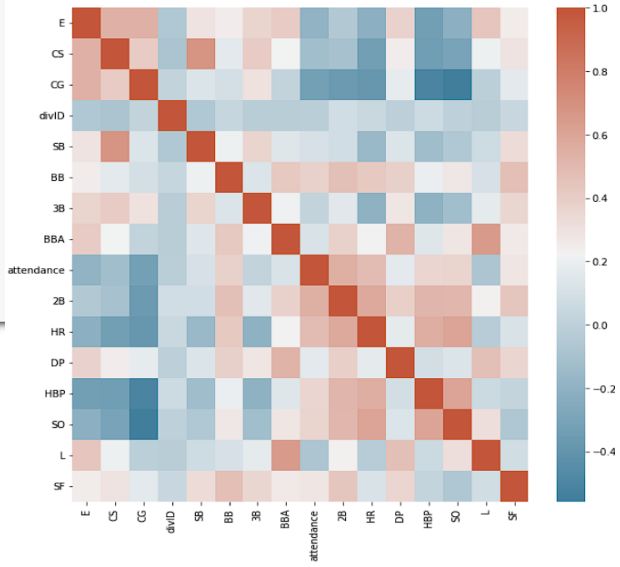
```
# Let's count how many teams we have for each division throughou
# It shows that the division for each team can change throughout

print(df_teams_no_NA[df_teams_no_NA['divID'] == 'W'].shape[0])
print(df_teams_no_NA[df_teams_no_NA['divID'] == 'C'].shape[0])
print(df_teams_no_NA[df_teams_no_NA['divID'] == 'E'].shape[0])
```

model. We ignored variables like **teamID** and **name** as we are not interested in these. We then further cleaned the data by converting numerical values, removing duplicates, and removing null values. Before performing feature selection, we began exploratory data analysis, identifying how many teams for each division in each year we have to work with. We identified that there were roughly twice as many teams in the West and East divisions than the Central division, which might affect the performance of our model later as the data is somewhat unbalanced.

The next step we took was to perform feature selection, where certain input variables are specifically chosen for the model to be used as predictors. First, we explored the distributions of the data as we may want to normalize or standardize the data later, which can also reveal outliers. One of the most important components of feature selection is identifying the correlation between variables [6]. We plotted the correlation matrix, pointing out certain features that have high multicollinearity.

```
# Let's explore the distributions of the feature data

# configure max open warning
plt.rcParams.update({'figure.max_open_warning': 0})
cols = list(df_teams_no_NA.columns)
print(cols)

plt.figure(figsize=(10,100))
n = 1
for c in cols:
    if c != 'divID':
        # plots either on left/right subplot based on parity
        plt.subplot((len(cols) - 1)//2 + 1, 2, n)
        plt.hist(df_teams_no_NA[c], bins = 'auto')
        plt.xlabel(c)
        plt.ylabel('frequency')
    n += 1

plt.tight_layout()
plt.show()
```





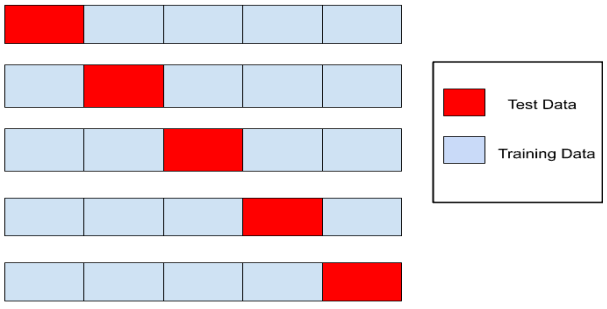
We also included two other methods of feature selection, including Univariate Analysis for classification as well as Recursive Feature Elimination with

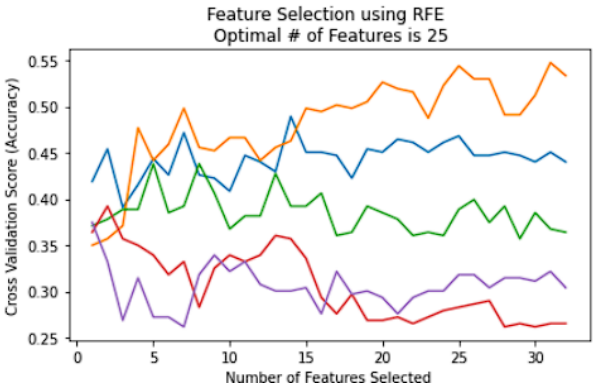
```
from sklearn.feature_selection import SelectKBest, chi2

X = df_teams_no_NA.drop(['divID'], axis = 1).copy()
y = df_teams_no_NA['divID'].copy()

feature_selector = SelectKBest(chi2, k = "all")
fit = feature_selector.fit(X,y)

p_values = pd.DataFrame(fit.pvalues_)
```

Cross-Validation (RFECV) [2]. In Univariate Feature Selection, the **SelectKBest()** method is utilized to determine whether there is a statistically significant difference between the observed and expected results. It uses Pearson's chi-squared test statistic to conclude whether a relationship exists.

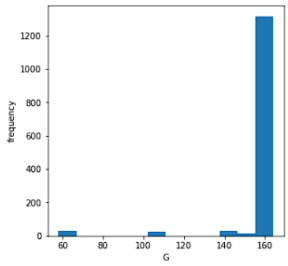
The code above calculates the χ^2 score for each variable as well as their respective P-values. We reject the null hypothesis, that the variables are independent, if the resulting P-value is less than a significance level of 0.05, thereby conditionally selecting only those features. RFECV works similarly in that it removes features that have the weakest association with the target variable. However it performs this step iteratively until it finds an optimal selected number of features. RFECV also implements cross-validation, which splits up the train/test data five different times than computes an average score.

```
from sklearn.feature_selection import RFECV
from sklearn.tree import DecisionTreeClassifier

feature_selector = RFECV(DecisionTreeClassifier()) # u
fit = feature_selector.fit(X, y)

optimal_feature_count = feature_selector.n_features_
```

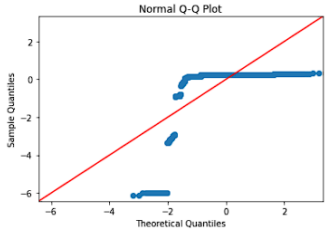





```
['G', 'Ghome', 'W', 'L', 'R', 'AB', 'H', '2B', '3B', 'HR', 'BB', 'SO', 'SB', 'CS'
'SV', 'IPouts', 'HA', 'HRA', 'BBA', 'SOA', 'E', 'DP', 'FP', 'attendance', 'WSWin_
Please enter a feature you want to analyze: G
Checking results for feature - G...
```



<Figure size 360x360 with 0 Axes>



In addition, we created an interface that takes in user input and outputs the distribution, Q-Q plot (shows distribution of data against the normal distribution), and

correlation matrix depending on what feature the user wants to analyze (shown to the right).

We can now begin training a classification model. Since we became comfortable with what features we wanted to include in the model, we split up the data into 80% training and 20% testing data, which helps avoid overfitting the data. We then scale our data to standard normal by subtracting off the mean and dividing by the standard deviation. Because we are classifying division type for data that includes the true division, we will choose a supervised classification model. In this case, we used the *k*-Nearest Neighbor algorithm [7], which is better suited for lower-dimensional data and has a short training time. *k* represents the number of neighbors to choose from. The algorithm calculates the distances between a test point and all other training points using a simple Euclidean distance metric, sorts the distances in ascending order, and selects the first *k* points. Finally it returns the mode of the *k* labels. Later, we will try to optimize this *k* value.

```
knn = KNeighborsClassifier(n_neighbors = 20)
knn.fit(X_train, y_train)

print(knn.score(X_test, y_test))
```

```
0.5492957746478874
```

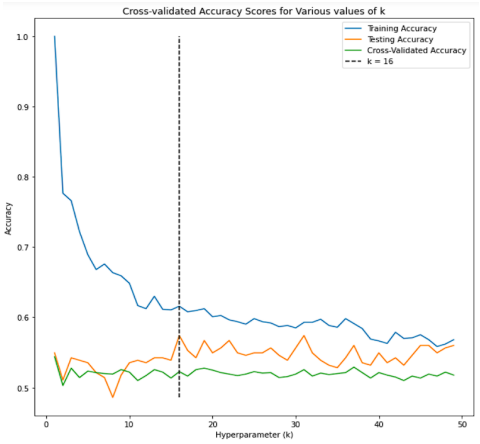
Our initial accuracy is roughly 55% so we can work on improving this result. We created a method called **parameterize()** that takes in a total number of neighbors to test, *n*, as well as the training and testing features and target variable. For each *k* number of neighbors in a range from 1 to *n*, the function trains a *k*NN with *k*, calculates the train, test, and cross-validated accuracies,

```
# Training KNN with optimal k value with cross validation

knn_final = KNeighborsClassifier(n_neighbors = grid_search.best_params_['n_neighbors'])
cv_scores = cross_val_score(knn_final,
                            X_scaled,
                            y,
                            cv = ms.KFold(shuffle = True,
                                          random_state = 42
                                         )
                           )
print(f"Accuracy with cross validation: {round(np.mean(cv_scores) * 100, 3)}%")
knn_temp = KNeighborsClassifier(n_neighbors = grid_search.best_params_['n_neighbors'])
knn_temp.fit(X_train, y_train)
print(f"Accuracy for the testing dataset with tuning: {round(knn_temp.score(X_test, y_test) * 100, 3)}%")
```

```
Accuracy with cross validation: 54.378%
Accuracy for the testing dataset with tuning: 54.93%
```

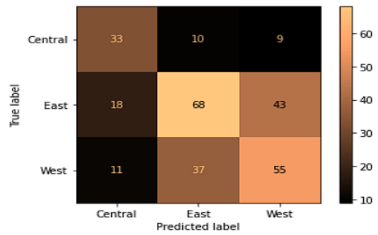
and plots the results as shown below. In this case, the optimal *k* value that maximized the cross-validated accuracy was found to be around 16. Therefore, we will retrain the *k*NN model with $k = 16$ and continue analyzing our results.

Despite having a better *k* value, our cross-validated accuracy is approximately the same. By plotting a confusion matrix, we can identify which categorical divisions were predicted better than others.



From the confusion matrix output, we can see that the K-nearest neighbors classifier had an easier time classifying the Central division than the East and West Divisions. Also, between the East and West Divisions, the East Division was slightly more accurate despite there being a relatively equal number of teams in the East and West Divisions in the input data. From pure guessing, these individual accuracies would equal 33% across the three divisions. Therefore, an accuracy of roughly 55% is somewhat decent. We created an additional method that predicts the division of

```
labels = np.array(['Central','East','West'])
conf_mat_disp = ConfusionMatrixDisplay.from_estimator(knn_temp,
                                                      X_test,
                                                      y_test,
                                                      display_labels = labels,
                                                      cmap = 'copper')
conf_mat = confusion_matrix(y_test, knn_temp.predict(X_test))
plt.show()
print(classification_report(y_test, knn_temp.predict(X_test)))
```



	precision	recall	f1-score	support
C	0.53	0.63	0.58	52
E	0.59	0.53	0.56	129
W	0.51	0.53	0.52	103
accuracy			0.55	284
macro avg	0.55	0.57	0.55	284
weighted avg	0.55	0.55	0.55	284

```
print(f"Predicted Division: {knn_temp.predict(np.array(X_temp_standardized))}")
print(f"True Division: {true_div}")
```

```
Please enter a year the team played in: MIA
Please enter a valid integer.
Please enter a year the team played in: 2019
Please enter the teamID of the team: MIA
Predicted Division: ['E']
True Division: E
```

a given team from the data, as well as from raw manual data.

Division Prediction Model Results

Apart from the quantitative results, we can consider other external factors that the model does not illustrate. There may have been more East and West division teams in earlier time periods than Central division teams as baseball started in New England and spread to the West eventually. The most recent change took place after the 2012 season, when the Houston Astros moved to the AL West division. We should also consider the potential bias of including more East and West Division teams in the model than Central teams (earlier we found that there were about twice as many teams within the East and West divisions as there teams in the Central division).

Predicting MLB Pitch Types using Machine Learning

From the division classification results above, we were unsatisfied with the mere 55% accuracy, so we tried following the same process to predict pitch type from MLB pitching data [1]. For the purpose of demonstration and reducing training time, we only collected the first 10,000 data points as the size of the entire dataset was nearly 1 GB. After preparing the data, optimizing hyperparameters, and training a *k*NN model, we get the following results.

```
# Training KNN with optimal k value with cross validation

knn_temp = KNeighborsClassifier(n_neighbors = 9)
knn_temp.fit(X_train, y_train)
print(knn_temp.score(X_test, y_test))
```

```
0.7682619647355163
```

```
conf_mat_disp = ConfusionMatrixDisplay.from_estimator(knn_temp,
                                                      X_test,
                                                      y_test,
                                                      cmap = 'copper')

plt.show()
print(classification_report(y_test, knn_temp.predict(X_test)))
```

	precision	recall	f1-score	support
CH	0.81	0.82	0.81	202
CU	0.75	0.86	0.80	147
FC	0.67	0.56	0.61	107
FF	0.82	0.93	0.87	671
FS	0.73	0.22	0.33	37
FT	0.63	0.61	0.62	274
KC	0.50	0.17	0.26	35
SI	0.71	0.50	0.59	203
SL	0.80	0.86	0.83	309
accuracy			0.77	1985
macro avg	0.71	0.61	0.64	1985
weighted avg	0.76	0.77	0.76	1985

During the data preparation phase, we removed pitch types that were exceedingly rare, in this case those that frequented less than 100 times. From the confusion matrix, we can see that the test dataset included some pitch types that had relatively low frequencies, including **KC** and **FS**. Had we removed rare types like these, our accuracy would have undoubtedly increased substantially. We also found from the results that factors like **vy0** (initial velocity on a relative y axis), **start_speed** (initial speed of the pitch), and **spin_rate** (pitch's spin rate, measured in RPM) were significant in the model. Also, from the classification report above we see that interestingly **FF** (four-seam fastball) was the easiest pitch type to predict, as a proper fastball can only be achieved with enough speed.

Conclusions

To conclude the machine learning section of this project, we were quite satisfied with the pitch type classification results. Several ways we could have improved the feature selection stage include utilizing different features, identifying features that were the most significant, and implementing a dimension reduction technique like PCA to reduce overfitting the model. When analyzing our results, we could have included performance metrics specifically for imbalanced classification as well as an ROC curve to compare the true positive and false negative rates. Finally, in terms of predicting the target variable, we could have outputted some kind of probability-based prediction plot so that instead of a single predicted response value we can get a probability tied to that result.

References (Code Research and Assistance)

These sources were used to research the machine learning process and pipeline. [1,3,4,13,14] include the data sources used for data visualizations and predicting the division and pitch type. [2,6,7] were used as a reference extensively for the feature selection process. The other sources assisted in the research process and basic function calling through library documentation.

[1] Schale, Paul. "MLB Pitch Data 2015-2018." *Kaggle*, 17 May 2020, https://www.kaggle.com/datasets/pschale/mlb-pitch-data-20152018.

[2] Luna, Zipporah. "Feature Selection in Machine Learning: Correlation Matrix: Univariate Testing: RFECV." *Medium*, Geek Culture, 10 Aug. 2021, https://medium.com/geekculture/feature-selection-in-machine-learning-correlation-matrix-univariate-testing-rfecv-1186168fac12.

[3] "Major League Baseball Teams Data." *Data Sets*, https://www.openintro.org/data/index.php?data=mlb_teams.

[4] Lahman, Sean. "Download Lahman's Baseball Database." *SeanLahman.com*, 9 Mar. 2022, https://www.seanlahman.com/baseball-archive/statistics/.

[5] "Recursive Feature Elimination with Cross-Validation." *Scikit*, https://scikit-learn.org/stable/auto_examples/feature_selection/plot_rfe_with_cross_validation.html.

[6] Gusarova, Maria. "Feature Selection Techniques." *Medium*, Medium, 1 Sept. 2022, https://medium.com/@data.science.enthusiast/feature-selection-techniques-forward-backward-wrapper-selection-9587f3c70cfa.

[7] "K-Nearest Neighbor Algorithm in Python." *GeeksforGeeks*, 23 Aug. 2022,

https://www.geeksforgeeks.org/k-nearest-neighbor-algorithm-in-python/.

[8] Allibhai, Eijaz. "Building a K-Nearest-Neighbors (K-Nn) Model with Scikit-Learn." *Medium*, Towards Data Science, 2 Oct. 2018, https://towardsdatascience.com/building-a-k-nearest-neighbors-k-nn-model-with-scikit-learn-51209555453a#:~:text=k%2DFold%20Cross%2DValidation,scored%20on%20the%20test%20set.

[9] Okamura, Scott. "GridSearchCV For Beginners." *Medium*, Towards Data Science, 30 Dec. 2020, https://towardsdatascience.com/gridsearchcv-for-beginners-db48a90114ee.

[10] Mudugandla, Sivasai Yadav. "10 Normality Tests in Python (STEP-by-Step Guide 2020)." *Medium*, Towards Data Science, 26 Sept. 2020, https://towardsdatascience.com/normality-tests-in-python-31e04aa4f411.

[11] "Fall 2022 CS305." *CS305: Machine Learning*, https://cs.wellesley.edu/~cs305/.

[12] Loukas, Serafeim. "How Scikit-Learn's Standardscaler Works." *Medium*, Towards Data Science, 8 Oct. 2021, https://towardsdatascience.com/how-and-why-to-standardize-your-data-996926c2c832.

[13] "Statcast Search." *Baseballsavant.com*, https://baseballsavant.mlb.com/statcast_search.

[14] Jldbc. "PyBaseball." *GitHub*, https://github.com/jldbc/pybaseball.

[15] McDaniel, RJ. "The Physics of Hard-Hit Balls." *The Hardball Times*, 18 Aug. 2016, https://tht.fangraphs.com/the-physics-of-hard-hit-balls/.

[16]"Beautiful Soup Documentation." *Beautiful Soup Documentation - Beautiful Soup 4.9.0 Documentation*, https://www.crummy.com/software/BeautifulSoup/bs4/doc/.