Introduction

Our dataset analyzed Bryce Harper's hitting stats from the 2021 season and 2019 season, provided by Baseball Savant. The 2020 season wasn't included because the season was cut short due to the COVID19 pandemic. Because there are many factors that went into this, we decided it was best to leave that season out so the numbers wouldn't throw off the data. The attributes that we looked at were exit velocity (MPH), launch angle (degrees), distance (ft), direction, pitch (MPH), pitch type and the result of the at bat. The result was our classification column. Most of the categories were continuous, however pitch type and direction were discrete. Pitch type is the type of pitch that was thrown, and includes changeup, sinker, split finger, fastball, cutter, and curveball in the dataset. Direction is the direction of where the ball went, and includes straightaway, opposite and pull in the dataset.

The overall goal for this project was to use different algorithms to predict whether Bryce Harper would hit a single, double, triple, homerun or get out. We hope to investigate which algorithm would be the best by comparing the correctness percentages of each algorithm we used. Naïve Bayes and k-nearest neighbor were the two main algorithms used, however our group also looked at decision trees and multi-layer perceptron's.

In order to clean the dataset, the 2019 and 2021 hitting results for each at-bat for Bryce Harper were placed into JMP. Everything from the website was used except the date of the game played, hitting and fielding team, and the pitcher. The program was used to randomly select 85% of the data to be our training set then the remaining 15% to be our testing set. The data sets were then exported to excel where they could be utilized by python.

Naïve Bayes

In order for the dataset to work properly with the Naïve Bayes algorithm, the data had to be discretized. Exit velocity, launch angle, distance and pitch velocity were changed to use variables of very low, low, average, high and very high. To determine the ranges for these variables, we used the minimum, maximum and median values, as well as the first and third quartile.

The algorithm used code from Project 1. Without changes, the algorithm resulted in 44.62% correctness and the best classification was "out". A modification that was made was changing all the values of 0 to something slightly bigger than 0. This included adding a simple if-statement in the code. The amount added to 0 was modified to see which gave the best correctness. The results are in the table below.

Table 1: Trial 1

Value	Correctness	Classification
1	1.7857142857142856%	Triple
0.1	58.03571428571429%	Out
0.01	59.82142857142857%	Out
0.0…01	59.82142857142857%	Out
0	44.6%	Out

The smallest correctness percentage was 1.79% and the classification was triple. From this, we know that 1 isn't the best value. By using a value of 0.1, the correctness percentage went up to 58.04% with a classification of out. The best percentage occurred when the value was 0.001. After playing around with the values, we found that it didn't matter how many 0's were after that, we still received the same correctness percentage of 59.82% with a classification of out.

Although the correctness percentage did increase with this modification, we wanted to see if we could increase this even more. Another modification that we did was changing how we discretized the data. The second trial used only three discrete values, rather than the original five. Exit velocity, launch angle, distance and pitch velocity were changed using variables of low, average and high. To determine the ranges for these variables, we used the minimum, maximum and median values. See *Table 2* for the results of the second trial.

Table 2: Trial 2

Value	Correctness	Classification
1	42.857142857142854%	Triple
0.1	55.35714285714286%	Out
0.01	55.35714285714286%	Out
0.0…01	55.35714285714286%	Out
0	54.46428571428571%	Out

Again, the smallest correctness came when the value was 1. The correctness percentage is 42.86% with an incorrect classification of triple. Furthermore, the best correctness percentage happened when the value was 0.01 with a correctness percentage of 55.36% and a classification of out.

We still wanted to see if we can improve the correctness percentage, so we investigated one last trial with different discretized values. We brought back the five variables: very low, low, average, high, and very high. Very low, average, and very high were the minimum, median and maximum. To get low, we found the halfway point between the minimum and the median. Likewise, to get the high value, we found the halfway point between the maximum and the median. See Table 3 for the results of trial 3.

Table 3: Trial 3

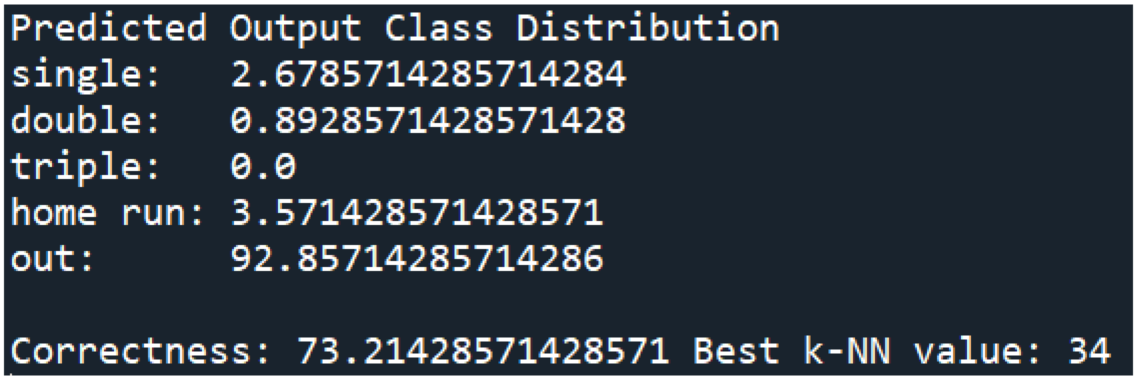
Value	Correctness	Classification
1	2.6785714285714284%	Triple
0.1	62.5%	Out
0.01	65.17857142857143	Out
0.0…01	65.17857142857143%	Out
0	59.82142857142857%	Out

Again, the smallest correctness came when the value was 1. The correctness percentage is 2.68% with an incorrect classification of triple. Again, the best correctness was when the value was 0.01 with a correctness percentage of 65.18% and a correct classification of out.

After all the trials were complete, the best correctness percentage that was recorded happened in the third trial. This confirms that by changing the values of 0 to something slightly bigger than 0 will increase the correctness percentage. Furthermore, by adjusting the ranges, a higher correctness percentage can be found. Compared to k-nearest neighbor, naïve bayes originally had a significantly lower correctness percentage. But after the modifications were made, they had about equal correctness percentages.

K-Nearest Neighbor

The code was originally set up using an automated selection for the best K value. Once the best K value was determined, it was used to determine the correctness of the testing set. By running the program multiple times, the best K-Values were in the range of 20-40 with correctness of about 70%. In baseball, 70% right is pretty good. But this got the team thinking, what was the predicted output class distribution.



```
Predicted Output Class Distribution
single:   2.6785714285714284
double:   0.8928571428571428
triple:   0.0
home run: 3.571428571428571
out:      92.85714285714286

Correctness: 73.21428571428571 Best k-NN value: 34
```

Figure 1: First Test with K-NN

Figure 1 displays the distribution of the predicted output class using a K value of 34. This means, based on the testing set, the algorithm predicted about 3% of the data were singles, about 1% were doubles, 0% were triples, 4% were home runs, and 93% resulted in an out. This data tells us that Bryce Harper wasn't even batting 0.100. This means that, based on this algorithm, he got a hit 1 out of 10 times.

In baseball, to be successful, a good batting average is .300, or getting a hit 3 out of 10 times. Baseball is a sport where you fail more than you succeed, so getting out 70% of the time doesn't make you a bad player. In fact, during the 2021 and 2019 seasons, Bryce Harper was hitting about .300. So, from this analysis, by generating a class that consisted of mostly outs, of course, this makes sense as to why the correctness would be mostly right. But this isn't an accurate representation of what actually happened.

For the two seasons he averaged a .280 batting average, meaning he got out about 72% of the time. From figure 1, we know something with the algorithm isn't right, so modifications needed to be made. Therefore, for the next modification, we were looking for an out percentage around 72% to create more variability for each result. This simulates a real-life hitting percentage.

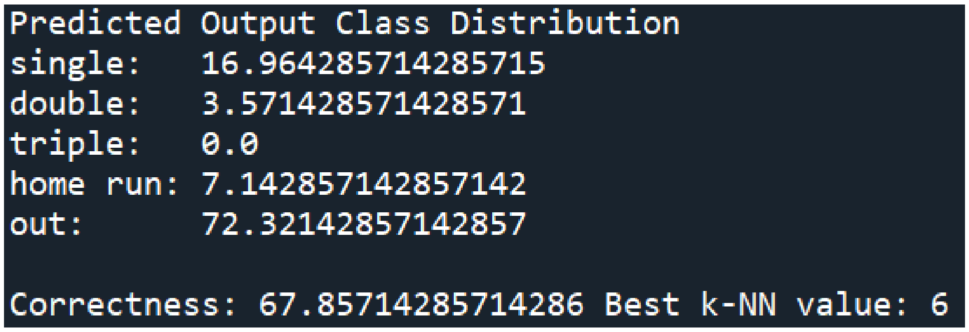

```
Predicted Output Class Distribution
single:   16.964285714285715
double:   3.571428571428571
triple:   0.0
home run: 7.142857142857142
out:      72.32142857142857

Correctness: 67.85714285714286 Best k-NN value: 6
```

Figure 2: 2nd Modification to Algorithm

Going through each K-Value manually, the best one was a value of 4 with an out percentage of about 72% and correctness of about 68%, Figure 2. This correctness was lower but provided a more realistic prediction because there was more variation in the results.

Decision Tree

For the decision tree code, we implemented the code provided by Dr. Thompson. The decision tree was created with the line, "root.create(myTrain, myTrainClass, namesData, bestValue)". Figure 3 shows the root node (Launch angle) and its follow up leaf nodes to find the results.

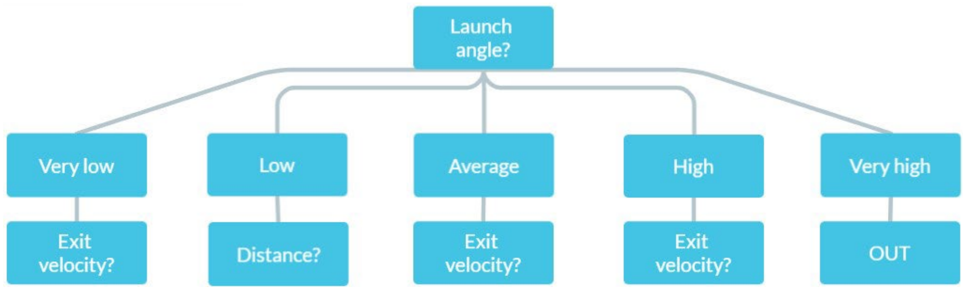
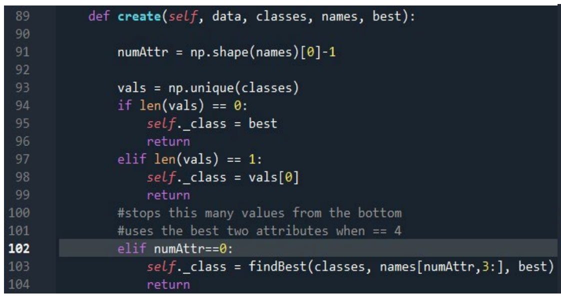
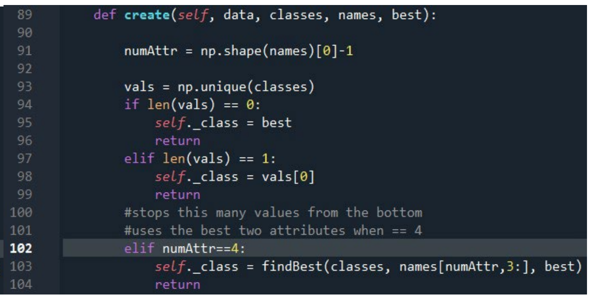


Figure 3: Decision Tree

We then created a for-loop to run each row in our testing data through the decision tree which gave an output for each row. From this we were able to compare it to our predicted output and count each value that was correct. To calculate the correctness, we divided the correct outputs by the total number of rows in the testing data. This gave us a correct percentage of 58.0357%. We were able to output the best two decision nodes by modifying the `elif` statement for `numAttr` in the create function, figure 4. After modifying, this gave us a new correct percentage of 62.5%, figure 5.



```
    def create(self, data, classes, names, best):

        numAttr = np.shape(names)[0]-1

        vals = np.unique(classes)
        if len(vals) == 0:
            self._class = best
            return
        elif len(vals) == 1:
            self._class = vals[0]
            return
        #stops this many values from the bottom
        #uses the best two attributes when == 4
        elif numAttr==0:
            self._class = findBest(classes, names[numAttr,3:], best)
            return
```



```
    def create(self, data, classes, names, best):

        numAttr = np.shape(names)[0]-1

        vals = np.unique(classes)
        if len(vals) == 0:
            self._class = best
            return
        elif len(vals) == 1:
            self._class = vals[0]
            return
        #stops this many values from the bottom
        #uses the best two attributes when == 4
        elif numAttr==4:
            self._class = findBest(classes, names[numAttr,3:], best)
            return
```

Figure 4 & 5: original 58.0357%(left), modified 62.5% (right)

Perceptron

A calculation with a single-layer perceptron can only predict an outcome of 0 or 1, meaning two possibilities. The perceptron program made predictions about whether a player will get a hit or an out based on the inputs of exit velocity, launch angle, distance, direction, pitch velocity, and pitch type. Direction and pitch type are discrete variables, so those had to be normalized. To do this, we used the same normalize function that was used in k-nearest neighbor.

There were some tricky issues with the dimensions of some arrays not matching up, but when that was finally resolved, we tried seeing how the algorithm would work when trying different values for iterations and eta values. The results were not very promising. No matter how much we changed these values, the results always appeared random. This is likely because the weights at the beginning were completely random. One may think that just using more iterations would lead to higher accuracy but doing too many iterations leads to overfitting the data. No matter what modifications were made the correctness stayed in the range of 30% to 75%.

	0.0001	0.25	0.5	1	50
1	75.00%	67.85%	48.21%	48.21%	68.75%
2	49.11%	43.75%	36.61%	48.21%	41.07%
3	50.89%	66.96%	67.86%	67.85%	68.75%
4	35.71%	48.21%	67.86%	65.18%	44.64%
5	67.86%	41.07%	65.18%	67.86%	63.39%

Table 4: Tests when modifying eta. 5 tests were run with each eta value. There were 100 iterations for these tests.

	1	50	100	200	1000
1	32.14%	67.86%	48.21%	39.98%	68.75%
2	50.89%	70.54%	36.61%	47.34%	50%
3	63.39%	37.50%	67.86%	41.96%	47.32%
4	66.96%	36.61%	67.86%	68.70%	65.17%
5	33.03%	71.43%	65.18%	43.75%	49.12%

Table 5: Tests when modifying the number of iterations. 5 tests were run with each iteration value. An eta value of 0.5 was used for these tests.

Best Algorithm

Overall, k-nearest neighbor gave the best correctness of 67%. This was done by individually testing all the numbers from 1 to 50 in order to find a realistic distribution between singles, doubles, triples, home runs and outs. Although Naïve Bayes gave a correctness of 65%, k-NN is still the better algorithm because most of our data uses continuous values. The problem we run into with Naïve Bayes is that by discretizing the data we have to choose a range. The range we chose is completely bias and may not be entirely accurate.

Although the correctness for Decision Trees was lower than Naïve Bayes and K-NN, it was the most visually interesting and agreed with many sports analysts who say launch angle is the first attribute to look at followed by exit velocity. Perceptrons had the chance of leading to a very accurate result, but it was far too random and difficult to test, likely due to the randomization of weights.

https://baseballsavant.mlb.com/savant-player/bryce-harper-547180?stats=gamelogs-r-hitting-statcast&season=2019