# Modeling MLB Player Performance to Predict All-Stars for the Following Year

DST 475

Sean Littmann & Vincent Rupp

5/7/2025

# Introduction

Our goal for this project was simple: create a machine learning model that can utilize an MLB player's season statistics to accurately predict if they will be an All-Star the following season. We quickly learned that achieving this goal was going to be a lot more complex and difficult than we originally anticipated. Some of the challenges that presented themselves during our work include, but were not limited to data cleaning, class imbalance, model tuning, and threshold optimization. The All-Star selection process is a complicated process that involves player performance, fan voting, popularity, and even more outstanding factors that are difficult to incorporate into a basic dataset. Although this did make our target variable challenging to predict, it led to interesting findings and conversations about the nature of MLB All-Stars.

We began our process with a raw dataset containing thousands of entries spanning over a decade of MLB seasons. We filtered out all pitchers, and all players with limited playing time. The resulting dataset contained roughly 4,500 observations, and had 30+ features ranging from traditional stats like batting average and on base percentage, to more less traditional statistics like OPS+, as well as positional and team data. Our diverse variable list presented both opportunities and obstacles for modeling, especially considering the high variance in feature scales and types.

One of the most immediate issues we encountered when beginning our model testing was the extreme class imbalance in our dataset, only about 9% of players were All-Stars. This extreme imbalance posed a major threat to our models' learning process and forced us to come up with some creative ways to combat this class imbalance. Accuracy, for example, would deceivingly reward models for blindly predicting "No" on every player (this would result in around a 91% accuracy rate). Instead we ranked and optimized our models based on their F1 score, a metric that balances precision and sensitivity, making it a great metric for datasets like ours, where the rare event is the important one.

Our strategy for creating these models emphasized tuning, experimentation, and depth. We implemented every applicable machine learning algorithm learned in class to our dataset, including logistic regression, decision trees, naive Bayes, KNN, SVM, bagging, boosting, JRip, XGBoost, and neural networks. Each model was trained using 10-fold cross validation and evaluated using F1 score as the optimization metric. For probability based models we conducted threshold tests to tune and find the optimal decision cutoff beyond the default 0.5.

The portfolio that follows documents our full process, from data preparation to model comparison, and walks through our attempt to not only build a working All-Star prediction tool, but also understand how these different models can handle our real-world dataset.

## The Dataset

The dataset that we used for this project was gathered from Baseball Reference and Stathead, two widely used and publicly accessible resources for MLB statistics. Our final dataset contains season-level batting statistics for MLB hitters from 2012-2023, excluding 2019 and 2020 due to COVID skewing those years' data. The data from 2019 was excluded because there was no All Star Game in 2020, meaning that our target variable was not really available for the 2019 year. Removing these years ensured our dataset was consistent and easily comparable to the rest of the seasons. Unfortunately, the dataset that we extracted from Stathead did not have the option to add an "All-Star Next Year" column, so we had to go through and add that column, as well as our "All-Star This Year" variable manually.

Our final dataset contains 4,514 observations, each row representing a single qualified season from a hitter. To better understand this, our identification variable looked something like this: JoeMauer2013, where more variables like JoeMauer2014, and JoeMauer2015 also exist. After all of our cleaning (which we will explore more shortly) we ended up with 25 variables for our models to explore. The binary target variable named AS_next_year contained the value "Yes", or "No", and indicated whether the player in question was an All-Star the season following the one represented in their row.

The AS_next_year variable was used as our response label, encoding future All-Star status. Initially we had this column formatted as 0 (non All-Star), and 1 (All-Star), but we later changed this to "No" and "Yes" to meet the optimal requirements for many of the classification algorithms, particularly Decision Trees and others.

Before we dove into creating any of our models, we did some preprocessing steps to the data to ensure that each model was being run with the best dataset we could provide them with. We ran into an initial problem with our Team variable, because many players played for multiple teams in a season, and each of those teams were represented in the Team column. For lack of a better idea, we simply took the team that the player had the most games played with in that year, and designated that team as the only team represented. We turned the Team variable (containing variables like "MIN", or "SF") from a character string to a factor.

On Stathead, the position column is quite complicated, it will contain something similar to: *978/P3D. Explaining the entire format of this column is complicated and unnecessary, but long story short, the first number listed in this list is the player's primary position, and that is the position we chose to keep for each player. We also turned the Pos variable from numeric to categorical. The Pos variable was important to change because positions in baseball are represented by numbers (such as "2" meaning catcher, "3" meaning first baseman, "4" meaning second baseman, and so forth through "9" meaning right fielder) but the scale of each number doesn't actually mean anything, it just represents the position they play. In the rare instance in our dataset that a player only played Designated Hitter (DH) all year, we assigned that player position "0", because a DH does not have a traditional number assigned to it.

We wanted to be conscious of not using redundant and unimportant variables in our dataset, so we did some cleaning to ensure that our dataset contained only necessary variables. Here are the variables we eliminated from our dataset:

Player_ID: Simply an identifying variable, completely useless in training a machine learning model

AB & G (At Bats & Games): These both measure the amount of which a player plays, but we already have PA (Plate Appearances) which measures the same thing, and is the most representative of the three mentioned variables, so we elected to only use PA in our dataset.

H & TB (Hits & Total Bases): These are both statistics that can be derived from the variables 1B, 2B, 3B, and HR, which are all in our dataset.

For models such as KNN and Logistic Regression, variables like Team and Pos had to be one-hot coded, and we did this using the fastDummies package in R.

For the naive Bayes model, all of the numeric features had to be converted to categorical. We did this by binning each feature and found that 4 bins gave us the best result.

To account for our extremely wide range in variable scales, such as OBP ranging from 0.2-0.5, to PA ranging from 100-800+, we normalized all of our variables. We did this using min-max scaling, and this was applied to the full dataset before we began our model creation step.

Although this couldn't be directly addressed in the data cleaning stage, it is important to note that our dataset contained a very extreme class imbalance (~91% non All-Stars). To address this we implicated some strategies that combated this imbalance, some of which include: F1 as our optimization metric, Upsampling, and threshold testing.

## Algorithms Considered

All of the algorithms that we used in our project were used because they were classification algorithms that we learned in class, and were listed in the project description.

Here is our list of tested algorithms:

- Naive Bayes
- K-Nearest Neighbors (KNN)
- Logistic Regression (glmnet)
- Decision Tree (C5.0)
- Rule-based Models (JRip)
- Support Vector Machine (Radial)
- Bagging (treebag)
- Boosting (adaboost.M1)
- Random Forest
- XGBoost
- Neural Network (nnet)

## Model Ranking

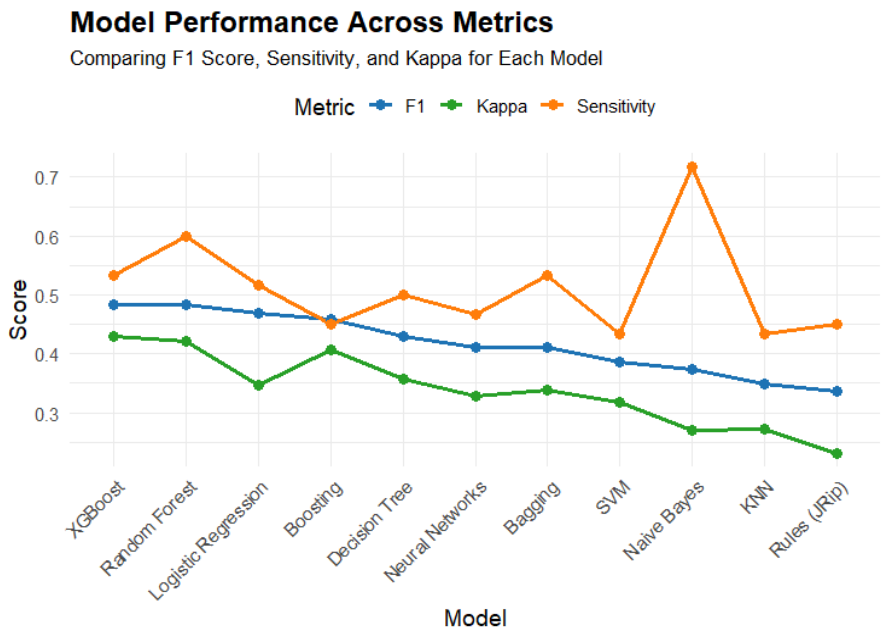
Disclaimer: **All of these results are based on the output we received *before* rendering our document. Our rendered code produced slightly different results for a couple of our models, despite us using the same seed (162). We are not completely sure what is causing this, but we believe that when we rendered our code (and in turn re ran the models), there could have been some data leakage from the test sets, so all of our numbers are true to our original model outputs, despite the attached code possibly having different answers.**

After training, tuning, and retraining to optimize the results of each model, we evaluated each model's final performance by using F1 score as previously discussed. We tracked other metrics as well such as sensitivity and kappa, but F1 score is the metric we chose to optimize and therefore rank our models on.

The following table is our full ranking sorted from best to worst by F1 score:

| Model | F1 | Sensitivity | Kappa |
|---|---|---|---|
| XGBoost | 0.4844 | 0.5333 | 0.4296 |
| Random Forest | 0.4832 | 0.6000 | 0.4219 |
| Logistic Regression | 0.4681 | 0.5167 | 0.3469 |
| Boosting | 0.4593 | 0.4500 | 0.4058 |
| Decision Tree | 0.4294 | 0.5000 | 0.3566 |
| Neural Networks | 0.4110 | 0.4667 | 0.3287 |
| Bagging | 0.4103 | 0.5333 | 0.3379 |
| SVM | 0.3852 | 0.4333 | 0.3179 |
| Naive Bayes | 0.3733 | 0.7167 | 0.2694 |
| KNN | 0.3487 | 0.4333 | 0.2718 |
| Rules (JRip) | 0.3360 | 0.4500 | 0.2315 |

Below is a chart showcasing the relationship between these metrics across all models:




In addition to the data table with all of our raw rankings, the line graph above provides a visual comparison of F1 score, sensitivity, and kappa across all models. Sensitivity was what we initially based our models on, but we quickly realized that optimizing on sensitivity (our 9% class) caused our model to sacrifice the accuracy of our other class (91%), causing very poor overall statistics. F1 balanced this issue, while still keeping our sensitivity relatively high.

This shows how our top performing models based on F1 score, like XGBoost, Random Forest, and Logistic Regression, also all performed fairly well in the other metrics as well, proving that F1 is a valid metric to use in our context.

We have some interesting things happening in the line graph, such as Naive Bayes having the 3rd lowest F1, the 2nd lowest kappa, yet the highest sensitivity by a wide margin. Depending on the application of our model, it could be argued that Naive Bayes was our best model. If we were purely interested in predicting every All-Star, and completely uninterested in predicting non All-Stars as well, we could utilize the Naive Bayes model. However, although we are most interested in correctly predicting All-Stars, we are also somewhat interested in correctly predicting non All-Stars as well.

Models like Boosting and Decision Trees resulted in competitive F1 scores, while also posting decent sensitivity and kappa values. Neural networks, despite their complexity, unfortunately gave us underwhelming results, performing only slightly better than Bagging, and performing worse than models like Logistic Regression in both F1 and Kappa, proving that model complexity does not necessarily automatically yield better results.

The JRip and KNN models ranked the lowest in F1 and Kappa, to no surprise. These models struggle to handle high dimensionality and class imbalance, two things that our dataset certainly has. As expected, these models are not suitable for this dataset, and shouldn't be used in any future predictions.

# Model by Model Discussion

Each of our models was run individually, and therefore could be tuned separately. We will go through each model, how we tuned it, and the results. We optimized all models on F1 score.

**k-NN:**

The k-NN model was not predicted to do very well. This does not handle class imbalance very well and the model performance showed that. For our tuning parameters for the k-NN model, we only were able to change the k value. Our initial parameters were testing k = c(3, 5, 7, 9, 11), which we adjusted to k = c(1, 2, 3, 4, 5) and included sampling = "up" after our first test of the model.

By including increased sampling, our F1 score improved from 0.241 to 0.349. Seeing this significant improvement, we chose to include increased sampling for all future models since we already had issues with predicting enough all stars.

k-NN did not allow for threshold testing, which was a large part of how we optimized our models on F1 score, therefore we also predicted this model would not perform well.

**Logistic Regression:**

Logistic Regression was expected to be a decent model for our dataset given its interpretability, on structured datasets, which we have. Our dataset is full of numeric features (OBP, AVG, PA, HR, more), so we can expect Logistic Regression to have some decent success. Some limitations we knew to be aware of was the fact that Logistic Regression is inherently a linear based model, and our data is likely not all linear, which could cause discrepancies in our results.

We had to apply one-hot coding for this model so that each level for Team and Pos were binary indicators (TeamMIN: "1" for yes, "0" for no). We then used 10-fold cross validation and evaluated and optimized on F1 score. We then used threshold tuning to find the optimal probability cutoff for this model.

The hyperparameter tuning for Logistic Regression is limited, but we used the following parameter grid to find the optimal parameters for our model:

alpha = c(0, 0.5, 1)
lambda = 10^seq(-4, 1, length = 10)
Sampling = up

With these parameters set up, and using the threshold found in our threshold testing, we landed an F1 of 0.4681, which placed Logistic Regression 3rd overall.

We had mid-level expectations for this model, we knew it could handle our diverse data well, but were aware of the limitations that the linearity of Logistic Regression would cause. We were pleasantly surprised with our great results for Logistic Regression, and would consider this a top option for future testing.

**Naive Bayes:**

Naive Bayes was considered a reasonable model for our dataset due to its simplicity, speed, and strong performance on high-dimensional structured data. Given that our dataset contains multiple numeric features, naive Bayes provided a straightforward and interpretable approach to classification. However, a known limitation of Naive Bayes is its strong assumption of feature independence, which is unlikely to hold true in our dataset. This could lead to biased probability estimates and affect overall model performance.

To prepare our dataset for Naive Bayes, we applied binning to numerical variables, converting each numerical feature into a categorical feature with 4 bins. This allowed the Naive Bayes classifier to handle numerical inputs effectively.

We used 10-fold cross-validation to train and evaluate the model, optimizing for F1 score to ensure balanced performance across precision and recall, especially important due to the class imbalance in our dataset. Like logistic regression, Naive Bayes required threshold tuning to select the optimal probability cutoff for predicting positive class membership.

Our initial parameters were laplace = 0, usekernel = FALSE, and adjust = 0.5. This gave us an F1 score of 0.368. When applying threshold testing, we found that the threshold value of 0.9 gave us the best F1 score of 0.373.

Naive Bayes underperformed for what we were expecting, however the results do not shock us based on how we had to bin the data before training. This F1 score ranked second to last in our final rankings.

**Decision Trees:**

Decision Trees, and the C5.0 algorithm in particular, were a natural choice for our dataset considering that it handles both numerical and categorical variables very well, and can handle our high dimensionality. We expected the decision tree model to uncover patterns that possibly other models wouldn't be able to, as long as we tuned and prepared the model well.

No special adjusting was needed to be made to the dataset for this model, so we went straight into encoding 10 fold cross validation, and optimized on F1, as usual. Our tuning grid went through a couple of refining stages, and settled on this:

trials = 12, 13, 14, 15, 16
model = "tree", "rules"
winnow = TRUE, FALSE,

with the following parameters being selected in the final model

Trials = 14,
Model = tree,
Winnow = FALSE.

We then went through threshold testing, as we did on every applicable model, and this resulted in us using the probability threshold of 0.3.

We ended with an F1 score of 0.4294, resulting in our 5th (of 11) best model. While the decision trees did a solid job of predicting All-Stars, other more powerful ensemble models like XGBoost and Random Forests still were able to outperform Decision Trees.

**Rules:**

We did not expect this model to perform well because our dataset is very difficult to separate into binary categories. Oftentimes in baseball, statisticians overlook one statistic in favor of another, therefore we would not expect a computer to be able to do this.

Again we used cross validation on our dataset. Our initial parameters for the JRip model were NumOpt = c(1, 3), NumFolds = c(2, 3), and MinWeights = c(2.0). This produced an F1 score of 0.32. We chose our final parameters as NumOpt = 1, NumFolds = 2, and MinWeights = 2.0, and added sampling = "up". This gave us a final F1 score of 0.323.

This model met our low expectations and for similar unbalanced datasets in the future, we will not be considering this model. This F1 score ranked last in our final rankings.

**Support Vector Machines:**

We did not expect this model to perform well because our dataset is very difficult to separate into binary categories. In many cases, especially in sports analytics like baseball, subtle patterns can be difficult for a computer to capture, particularly when one class significantly outweighs the other.

Again, we used cross validation on our dataset. Our initial parameters for the SVM model included C = c(2, 3, 5), sigma = c(0.005, 0.01, 0.015), and sampling = "up". This produced an F1 score of 0.336. We updated our parameters to C = c(2.5, 3, 4), sigma = c(0.015, 0.02), and sampling = "up". We chose our final parameters as C = 2.5, sigma = 0.015, and sampling = "up". After threshold testing, we chose threshold = 0.35 with an F1 score of 0.385

This model met our expectations, and for similarly unbalanced datasets in the future, we will likely explore alternative models better suited to handle imbalance, such as tree-based methods or ensemble approaches. This model did however surprise us a little bit after threshold testing and provided us with slightly better results than we were expecting to see. This F1 score ranked 8th (of 11).

**Bagging:**

Bagging is particularly useful for reducing variance in high variance models like decision trees, and given our datasets structure (many numeric features with a rare positive case), we expected bagging to give us promising results. The main appeal of bagging with our dataset was the ability of bagging to stabilize boundaries and avoid overfitting without having to do too much fine tuning.

We used the treebag method in caret to implement bagging, and this approach doesn't involve many tunable parameters, in fact, the only thing we really changed was turning on upsampling (as we did for all of our models to fight back against our imbalance classes). We continued to use 10-fold cross validation and optimizing on F1.

Our initial model predictions without threshold testing resulted in a sensitivity of just 0.23, with a low F1 score as well. Threshold testing was a major component for this model, and after doing that, we saw F1 rise to 0.4103, and sensitivity rise to 0.5333.

Ultimately, bagging placed 7th overall, outperforming some very basic models, but not much else. We were slightly disappointed with the results of bagging, considering it appeared to have the ability to handle our dataset well. More ensemble learning methods, and models with more opportunity for tuning proved to be more reliable with our dataset.

**Boosting:**

Boosting, using AdaBoost, is designed to convert multiple weak learners into a strong learner by sequentially fixing each mistake of the previous models. Given the complexity of our dataset, this certainly has promising attributes. Boosting also is known to work well in contexts where certain patterns are hard to predict all at once, like when the positive class is small, as it is in our case.

We implemented boosting into caret using the fastAdaboost package, using adaboost caret method. We set up 10-fold cross validation, turned on upsampling, and once again set F1 as the optimization metric.

Our tuning grid once again was not large, the main focus of our tuning grid was to determine whether Adaboost.M1 or real adaboost would be the more optimal option. Despite our expectation that real adaboost would be the winner, considering it is more dynamic and works better with high dimensionality, the results with Adaboost.M1 were better.

Initial results were a bit underwhelming, but we got used to this being the case before we ran our threshold tests. After our threshold tests, we found that 0.4 was the optimal probability cutoff for this model, and that maximized F1 to 0.4593, boosting our 4th best model. This small adjustment raised our results significantly

**Random forests:**

Random forests also introduce random feature selection at each split, which typically helps reduce correlation between individual trees and improves generalization. Given our dataset's structure (many numeric features and an imbalanced target), we expected random forests to perform well by stabilizing decision boundaries while still maintaining flexibility to capture complex relationships.

We implemented random forests using the caret package, focusing our tuning on the MTRY parameter, the number of predictors sampled at each split. We tested MTRY values using mtry_vals <- unique(pmax(1, round(seq(1, sqrt(num_predictors) * 1.5, length.out = 5))))

Additionally, to help with class imbalance and improve classification of the minority class, we again applied upsampling, used 10-fold cross validation, and optimized directly for F1 score. We also tested threshold adjustments, ultimately choosing a threshold of 0.7 to better balance precision and recall.

This approach proved relatively successful. After threshold tuning, random forests achieved an F1 score of 0.4832, placing 2nd overall among our models. Overall, random forests exceeded our expectations for this type of problem providing one of the highest F1 scores we saw in this project.

**XGBoost:**

XGBoost has become one of, if not the top leader in machine learning algorithms through its dominant success in machine learning competitions and practical applications. XGBoost is well known to be one of the best ensemble models in the world, so we had high expectations coming into this model.

We trained XGBoost using the xgbTree method in caret with 10-fold cross validation, upsampling, and F1 score as the optimization metric. We ran this model a few times, and reachedon the following hyperparameter grid for our final model run:

nrounds: 15, 30, 50, 65
max_depth: 7,11,15
eta: 0.01, 0.05, 0.15
Gamma, colsample_bytree, min_child_weigh, and subsample were all fixed at reasonable defaults to reduce noise, 0, 1, 1, 1 respectively.

After tuning the best combination turned out to be nrounds = 50, max_depth = 7, and eta = 0.05.

Our initial model showed high sensitivity, around 0.633, but mediocre in terms of F1, kappa, and balanced accuracy. So as we have been doing we went through and conducted threshold tests to find the best probability threshold for the XGBoost model.

The threshold test yielded results showing that 0.6 was the optimal threshold for F1, and gave us an F1 score of 0.4844, the best of any model. It also yielded sensitivity of 0.533, and kappa 0.4296, both high scores relative to the rest of our models.

XGBoost was unsurprisingly our best model, though only by a narrow margin over random forests. Its strengths in ability to pick up nonlinear patterns, easily deal with non similar variables, and fully incorporate our high dimensionality, proved to be useful. We believe that fine tuning even further would result in even better numbers, but unfortunately XGBoost is an expensive model in terms of computational power and time, so we settled with the good results we got.

**Neural Networks:**

Neural networks are typically quite powerful in terms of uncovering complex relationships, particularly in large and noisy datasets. We expected neural networks to struggle a bit with our dataset because our dataset is not the type of structure that neural networks typically excels at. Our relatively low observation count (4,514), and mostly numerical, or at least simple variable types, make our dataset a little on the simple side for using Neural Networks.

We trained our neural network model using nnet in caret. Once again, we set up 10 fold cross validation, used upsampling, and chose F1 as our optimization metric. We included the following tuning grid in our final model after doing a couple rounds of parameter testing to zone in on the parameter levels:

Size = 4, 5, 6
Decay = 0.5, 0.7, 0.9

Size = 6, and decay = 0.9 were the final chosen parameters for our model.

We trained only one relatively small hidden layer, which we believed balanced overfitting and underfitting on our relatively structured and simple dataset.

Initial performance showed decent sensitivity (0.6) but particularly low F1 scores, so we once again ran threshold testing. The threshold testing for neural networks resulted in a probability cutoff of 0.75, leading us to a final F1 of 0.411, ranking 6th overall.

Expectedly, neural networks was not a top performer, and showcases that complexity is not the only factor that makes a model good or bad, each dataset will have different models that work better on it, and for ours, neural networks was not one of them.

## Results

| Model | F1 |
|---|---|
| XGBoost | 0.4844 |
| Random Forest | 0.4832 |
| Logistic Regression | 0.4681 |
| Boosting | 0.4593 |
| Decision Tree | 0.4294 |
| Neural Networks | 0.4110 |
| Bagging | 0.4103 |
| SVM | 0.3852 |
| Naive Bayes | 0.3733 |
| KNN | 0.3487 |
| Rules (JRip) | 0.3360 |

Our results show that XGBoost and Random Forest outperformed other models, followed closely by Logistic Regression and Boosting. These rankings make sense given the nature of our unbalanced dataset. Tree-based ensemble methods like XGBoost and Random Forest are particularly effective on complex and imbalanced datasets because they combine multiple decision trees, reducing variance and capturing non-linear relationships. Similarly, boosting methods incrementally improve weak learners, which can help model subtle patterns in noisy data.

Logistic Regression also performed reasonably well, which is expected since it can handle binary classification problems effectively, especially when combined with regularization to manage noisy features. However, its linear nature limits its ability to capture complex interactions, which may explain why it did not surpass the tree-based models.

Models like Decision Trees, Neural Networks, Bagging, and SVM followed. Decision Trees are prone to overfitting on imbalanced data, while Neural Networks require large amounts of data and careful tuning, which was not optimal here. Bagging improves stability but does not address imbalance well. SVMs, though powerful for clear-margin classification, can struggle when classes are not easily separable or when the dataset is unbalanced.

Finally, simpler models such as Naive Bayes, KNN, and Rules (JRip) performed the worst. Naive Bayes assumes feature independence, which is rarely true in real-world data. KNN

suffers from issues with class imbalance and irrelevant features affecting distance calculations. JRip, a rule-based learner, performed predictably poorly as it lacks the flexibility to capture patterns in data.

# External Comparison

Doing a search online, we found a few projects from other individuals with very similar goals of predicting all stars. Two projects that stood out to us were a GitHub project optimizing on F1 score as well, and an individual using neural networks to predict all stars as well.

The first project was created by Kelvin Jiang. They used a dataset very similar to ours with almost all of the same features. This individual used Python to do their project running a random forests model. We thought this was intriguing because in our testing, we found random forests gave us our second best F1 score. It was nice to see another project optimize on this metric since we did not discuss it in class. Their model was able to produce an F1 score of 0.6 on their test data, and when tested on their evaluation data produced an F1 score of 0.69. These values are significantly higher than our findings. Some reasons for this may be because this individual used defensive stats as well in their predictions, and they also used data beginning in 1933.
Link to project: https://github.com/kelvin-jiang/MLBAllStarClassifier

The second project was created by James Wyllie. They specifically used neural networks, along with only offensive data, like we did. They used a slightly larger dataset, including data from 2000 to 2011, whereas we began our data in 2012. While this model optimized on overall accuracy, it produced similar results to our accuracy values when optimizing on F1. Their accuracy on the normalized dataset came to be 0.88, and ours on our best model was 0.90.
Link to project:
https://medium.com/%40jameswyllie43/building-a-nurel-network-to-predict-mlb-all-stars-db24c0cb4b20

Overall, our results seem similar to others produced online. While there are not "published" projects we could find, we feel that our results are representative of the dataset in comparison to online results as well using similar data and models. We recognize that baseball is an unpredictable game with uncertainty at every turn whether it be unlucky streaks of hitters hitting the ball right at defenders, injuries, or even fan voting when it comes to all stars.

# Conclusion

Overall, our project was a success in that we were able to test and compare a wide variety of machine learning models on a very challenging dataset. Although no model achieved perfect predictions, several performed reasonably well considering the difficulty of the task. Models like XGBoost and Random Forest stood out, as they handled the imbalanced and complex nature of the data better than others.

One of the main takeaways from our modeling process is that there is no "one size fits all" solution in machine learning. Different models have strengths and weaknesses, and it became clear that models capable of handling non-linearity and imbalance performed best for our data. Simple models, or those that rely on strict assumptions like Naive Bayes or Rules, struggled. We also learned that data preprocessing and tuning were just as important as model selection. Techniques like upsampling to handle imbalance and threshold adjustment to balance sensitivity and precision had a big impact on results. Without them, even our best models would have performed poorly.

Finally, this project showed us the potential and the limitations of machine learning in real-world baseball problems. While models can help spot trends, make predictions, and support decision-making, they can't fully capture the complexities of the game. Factors like player mentality, team dynamics, and in-game strategy are difficult to quantify. This means that while machine learning is a powerful tool that can improve analysis and provide helpful predictions, it should be seen as a complement to and not a replacement for expert knowledge and human judgment.

# Bibliography

Wyllie, J. (n.d.). *Building a neural network to predict MLB All-Stars*. Medium. https://medium.com/@jameswyllie43/building-a-nurel-network-to-predict-mlb-all-stars-db24c0cb4b20

Jiang, K. (n.d.). *MLB All Star Classifier* [GitHub repository]. GitHub. https://github.com/kelvin-jiang/MLBAllStarClassifier