Predicting Pitch Outcomes in Major League Baseball

Micah Jona
mjona@wisc.edu

Nathan Kolbow
nkolbow@wisc.edu

Roshan Poduval
rpoduval@wisc.edu

Abstract

In this study, we evaluate the abilities of numerous machine learning algorithms to predict the outcome of a pitch in Major League Baseball (MLB) using real time data. This is a good task for testing the efficacy of various machine learning methods because it is a highly random and complex task that is accompanied by a plethora of data. This study focuses chiefly on the relative abilities of each model in performing the classification task, but as a consequence of tackling a novel problem we also offer performance baselines for any future endeavors. We compare the performances of four gradient boosting machines, k-Nearest Neighbors (kNN) clustering, as well as neural networks with and without convolutional layers preceding fully connected multiple perceptron layers. To this degree, when comparing learning algorithms we varied hyperparameters and attempted to achieve the best possible local maxima for each method. Extensive optimization of each algorithm revealed that XGBoost performed best on this dataset.

1. Introduction

1.1. Baseball and its Technologies

Since its inception in 1869, millions of people have watched and attended MLB games across the United States. In recent years, as technology has progressed, MLB has been making greater use of advanced methods to capture detailed, real-time data.

Perhaps the most notable of these technologies are PITCHf/x and Trackman: systems that can track a pitch's speed, trajectory, and location in real-time. If you've tuned into any MLB games in the past ten years, you've likely seen one of these systems utilized in pitching Zone Evaluation as shown in Figure 1. PITCHf/x was MLB's first move towards gathering detailed game data, while Trackman is a more recent improvement on the system used from 2015-2019, which is the period our data is from[1].

The box shown in Figure 1 is known as the strike zone: an imaginary box within which pitches are considered strikes, and outside of which pitches are considered balls. The decision as to whether a pitch that is not swung at is a strike or a ball, though, is left to the umpire who stands directly behind the catcher. This leaves the final decision up to the imperfection of the umpire's vision, which can be affected by something known as catcher framing. According to ESPN[9], "catcher framing is an act of subtlety, [...] turning pitches that nick the border of the zone – or at least appear to – into called strikes."

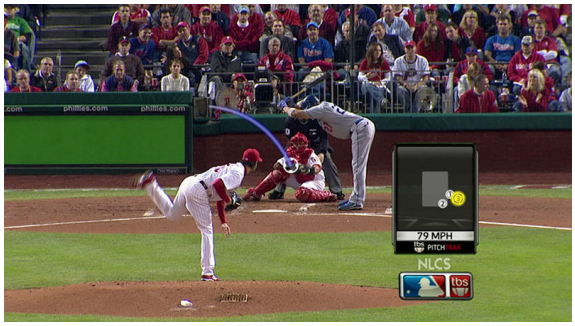

Figure 1. PITCHf/x data being used live. Image source: https://awfulannouncing.com/mlb/mlb-pitchfx-pitch-tracking-system-lawsuit.html.

1.2. Our Interests

In this project, we use real time data to predict whether the batter makes contact with the pitch, and if they do not, whether the pitch is ruled a strike or a ball for any given pitch. This leaves us with a classification task with three possible outcome classes: ball, strike, or hit[2]. There is an enormous amount of random chance that goes into the outcome of each pitch: umpire imperfection and catcher framing, as mentioned above, but also the batter may simply choose not to swing at a strike, or the batter may be having an off day, and so on. This random chance makes perfectly predicting the outcome of each pitch impossible, but it makes the prospect of maximizing predictive performance

[1] Even more recently, MLB has stopped using Trackman and begun using a system called Hawkeye.

[2] This is different from MLB's official definition of a hit which is "when a batter strikes the baseball into fair territory and reaches base without doing so via an error or a fielder's choice" [15].

very interesting. In a perfect situation, the large volume of data that we use should, in theory, give our learning algorithms enough information to solve the nonrandom portions of the classification task at hand. Our interests, and the results to follow, then, focus on the accuracy of various algorithms in solving the problem of classifying the outcome of pitches in MLB in this manner.

2. Related Work

Our task of predicting the outcome of any given pitch is novel, but there is a lot of existing work that has applied similar data to classifying the type of pitch thrown by a pitcher, i.e. fastball, curveball, etc. Two such applications are *Baseball ProGUESTus*[8] and *PitchNet*[13].

Baseball ProGUESTus is a simple, unsupervised approach to classifying pitch types. This model uses a k-means clustering algorithm to group pitches into 14 different classes using only horizontal and vertical movement as input. As it turns out, these two inputs are not enough to accurately distinguish pitch types from one another, so clustering accuracy in this model is not very good. Nonetheless, this project presents an interesting preliminary approach to a question very similar to our own.

More complex than *Baseball ProGUESTus*, PitchNet is a simple neural network which is able to classify the type of pitch that a pitcher throws with 96.25% accuracy when the pitcher is known, and 78.30% accuracy when the pitcher is new. The neural network takes release speed, horizontal break, vertical break, and custom pitcher embeddings[3] as input to a single dense layer which feeds into a softmax layer. The classified pitch is then the most likely pitch as determined by the softmax layer. Unlike a pitch's outcome, a pitch's type is nearly linearly separable from pitcher to pitcher based on a small number of features, which explain's PitchNet's high classification accuracy. Despite this difference though, PitchNet serves as a good example of a neural network being effectively utilized in a similar line of study.

Further, a plethora of work in comparing various classifiers' performance in a wide variety of domains has been done. However, little of this work has directly compared a wide variety of classifiers at once, and of this work even less has used highly dimensional data. The most similar work we were able to find was a study from 1989 that sought to predict defects in disk drive manufacturing[2]. This study found each of its methods, excluding kNN, to have error rates within 1% of one another, which our experiments do not reflect.

In summation, there is no work that seeks to either predict the outcome of a pitch and little work that seeks to compare a wide range of classifiers on high dimensional data, so in this way both our work and our results are unique.

[3]In our experiments we did not find any improved performance when distinguishing between different pitchers or batters.

3. Proposed Method

In order to predict if a pitch is a hit, ball, or strike, we will be testing a variety of machine learning algorithms. We have chosen scikit-learn's KNeighborsClassifier[4] for our kNN algorithm in order to test a clustering method as well as a relatively simple method. We also wanted to test many gradient boosting models; we chose scikit-learn's GradientBoostingClassifier (GBM)[5], XGBoost, and LightGBM (LGBM) because there is a lot of prior work done with these models, so they can be trusted to perform relatively well. Finally, we also tested the performance of neural networks because they are often considered the state of the art in regards to machine learning, and we are interested to see if they offer any higher performance than our other models.

In order to compare the predictive performance of our models, we will implement the holdout method by splitting our data into training and test sets and then comparing each model's predictive performance. With the amount of data we have, this should give us a reasonably good idea of how well each model generalizes. Data for training and testing each model will be split using the holdout method, and model hyperparameters will be optimized using various methods including grid searching, random searching, and Latin hypercube sampling. Due to time constraints, we were unable to explore every model as deeply as we would have liked to which is why some results to come are discussed in more depth than others. Likewise, model training was very costly, so even though we would have liked to generate confidence intervals for each of our best model settings we only did so for various gradient boosting model.

There are many options for comparing model performance, but we decided simply to use accuracy:

$$Accuracy = \frac{\#\ Correctly\ classified\ pitches}{\#\ Total\ pitches}$$

4. Experiments

4.1. General Experimental Setup

When building our models, in accordance with the holdout method, we always split our whole dataset into a training set and a test set[6], but we differed which observations were apart of the test set with each run in order to avoid

[4]https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

[5]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

[6]The size and members of the test set varied between runs; the size ranged from 10-30% of the whole data set.

favoring models with a bias towards our test set. Each algorithm was then trained and tested on different sets of hyperparameters in order to find the optimal model for each individual algorithm. The technique for optimizing hyperparameters varied between models and is discussed at an individual level below.

4.1.1 LGBM

One experiment that was performed for LGBM is dealing with early stopping. In order to have early stopping implemented, the Python package for LGBM requires you to pass an evaluation set along with a training set when fitting the model. The evaluation set is used to check if the model being trained is improving, and with the early stopping parameter, you can specify how long the model should train without seeing any improvement. When specifying this evaluation set, we tried different splits to see if this affected performance, and found that using 24 percent of our original dataset yielded good results.

4.1.2 XGBoost

Another experiment was within the XGBoost modeling. There are dozens of different hyperparameters within XGBoost[7]. Further, running XGBoost is expensive both in time and compution so we decided to experiment with a Latin hypercube to tune our XGBoost model hyperparameters. The Latin hypercube is a space-filling design that aims at filling the hyperparameter space as well as possible, so it's not necessary to run a model for every unique hyperparameter combination. This experiment did not go well, as the best hyperparameters from the latin hypercube ending up having a substantially smaller test accuracy than the model with the default hyperparameters.

4.2. The Dataset

4.2.1 Dataset Description and Acquisition

The original dataset is provided through Baseball Savant[8] and acquired through the *baseballr* package, created by Bill Petti[9]. We focused on the 2019 MLB season because including more than one season would have left us with too much data to handle, and this was the most recent year with complete information. The 2019 data contained approximately 730,000 pitches each with 89 features before any data pre-processing or cleaning. Included in each pitches features are data such as who the pitcher, batter, and fielders are, how fast the pitch was, where the pitch was released, where the pitch crossed home plate, and more. All features, included with descriptions, are available to view in Baseball Savant's Statcast Search documentation[10].

4.2.2 Dataset Cleaning and Pre-Processing

Firstly, there are a lot of features in our dataset that give information about the game state; this includes information such as the score of both teams before and after the pitch, what number at bat is currently taking place, whether it is the top or bottom of the inning, and more. We didn't want to include features like these both because they don't have an affect on the outcome of any given pitch and we wanted to focus solely on data detailing the physical characteristics of each pitch. Other features such as what teams are currently playing and the fielding alignment were also excluded for similar reasons.

Next, we needed to refine/simplify the outcome of each pitch down to our three classes; the description variable provided by Baseball Savant includes 15 unique pitch outcome classes. By grouping the pitch descriptions into three categories instead of fifteen we are able to make the classification process easier without losing potential insight. This is possible because many of the classes provided by Baseball Savant are functionally equivalent; for example, a *called strike*, a *swinging strike*, and a *swinging strike blocked* are all pitches which the batter did not make contact with, a swing and miss, or were called strikes by the umpire.

Lastly, we turned factor variables into dummy variables so that they would be immediately usable by all of our learning algorithms[11].

4.3. Software

This project would not have been possible without all of the following applications and packages: R[11], RStudio[12], tidyverse[16], caret[5], Python[14], XGBoost[3], scikit-learn[10], TensorFlow[6], NumPy[4], and Pandas[7]. The code for all of the experiments done in this study can be found at: https://github.com/NathanKolbow/MLBPredictions/.

5. Results and Discussion

5.1. kNN

Interactions between the variables that we have are complex, so it is no surprise that kNN performed worse than any other algorithm with an accuracy of 60.82%. To find the best kNN model, we trained the algorithm both with and without normalized input data, and varied the value of k. We found that the best value for k was 50 and that normalizing our data was an important part of improving predictions.

[7] https://xgboost.readthedocs.io/en/latest/parameter.html
[8] https://baseballsavant.mlb.com/
[9] http://billpetti.github.io/baseballr/
[10] https://baseballsavant.mlb.com/csv-docs
[11] TensorFlow, for example, does not have any built-in way of interpreting factor variables.

Despite the poor model performance, we can still extract valuable information from our results. Firstly, we can see the importance of normalizing the data that kNN is fit on. The distance measurement that we used for our kNN algorithm was Euclidean distance. Our input data comes on all different scales, which the default Euclidean distance metric completely ignores; this means that some features were arbitrarily overvalued by the kNN algorithm. After normalizing our data, predictive performance improved across the board, which can be seen in Figure 2.

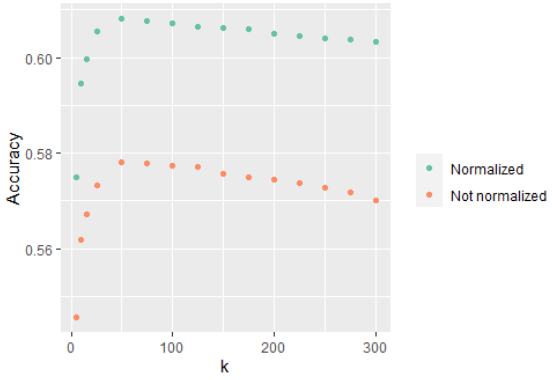

Figure 2. kNN accuracy with respect to k.

Another valuable takeaway from this experiment is the role that k plays in the model. As Figure 2 shows, the performance at different values of k follows a non-arbitrary curve, and demonstrates over- and under-fitting in the context of kNN. The low performance at $k < 25$ can be attributed to model under-fitting, and the steady decrease in performance after $k = 50$ can be attributed to model over-fitting. Despite its simplicity, we can see from this experiment that proper pre-processing of data and hyperparameter selection is imperative to getting the most out of a kNN model.

Lastly, we note that, despite its poor performance, kNN was orders of magnitude faster to train and test than any other learning algorithm we used. For example, XGBoost models took around 30 minutes to train, whereas kNN models took no more than two minutes. In our case we only care about performance, but this difference in speed is important to keep in mind for time sensitive applications.

5.2. XGBoost

Our baseline XGBoost model was fit on the train set with default hyperparameters; this model achieved an accuracy of 74.22% on the test set.

Next, we ran our aforementioned Latin hypercube on a five-fold cross validation of the training set. The best model in the twenty that were cross-validated performed with an 69.64% accuracy on the evaluation set, within the cross validation. We fit an XGBoost model with these discovered hyperparameters on the entire training set, and this model performed with an accuracy of 58.84% on the test set. It is clear that this model was over-fit on the training set, leading it to perform worse on the test set than the baseline model.

Since the hyperparameter tuning performed worse than the baseline model, we returned to the baseline model. Rounds of five-fold cross validation with default hyperparameters, except for the number of estimators, were run on the baseline model in order to find the optimal number of estimators. This cross validation revealed that the optimal number of estimators was 239, but fitting a new model with this hyperparameter left us with an accuracy of 74.18% on the test set which is still below the baseline.

The most curious part of our XGBoost modeling was the lack of performance improvements achieved from hyperparameter tuning. We have a few theories of why it may have performed as poorly as it did. One possibility is that, because the grid of hyperparameter values is generated randomly, we could have happened upon some very unfortunate choices for the values of the hyperparameters. Further, as previously mentioned, there are many different hyperparameters that can be tuned in an XGBoost model, and we may have poorly selected the hyperparameters that we tuned. However, perhaps most importantly is the observation that the values created by the Latin hypercube tended to prefer a more conservative model. For example, the hypercube's best performing model required more loss reduction and more child instance weight in order to partition a left node[12].

The final model's feature importance is provided in Figure 3. XGBoost calculates importance by measuring how much splitting on each feature improves model performance in each of the model's trees.

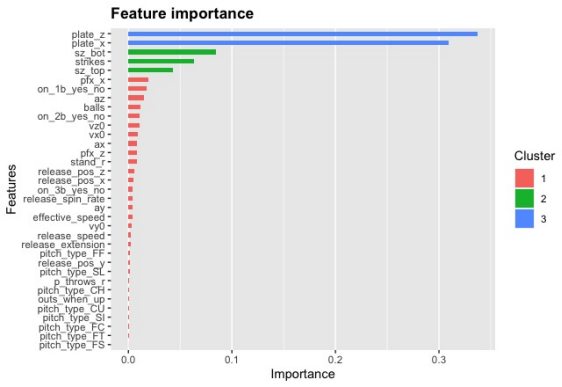

Figure 3. Feature importance of final XGBoost model.

We can see from Figure 3 that the two most important

[12]This discovered model had $gamma = 8.5$ (default 0) and $min\_child\_weight = 18$ (default 1).

features are *plate_x* and *plate_z*; these are the horizontal and vertical position at which the pitch crosses home plate. This makes sense, because, when the batter fails to contact the pitch, the position of the pitch when it crosses home plate *should* be the only factor in determining whether it is a strike or a ball. Some other highly important features also make intuitive sense, for example *sz_top* and *sz_bottom*: the top and bottom of the batter's strike zone, but others are interesting to see ranked highly in importance. For example, the number of strikes the batter already has is an important variable in determining the outcome of a pitch, and we cannot say for certain why this is.

5.3. GBM

scikit-learn's GBM was tested with only one hyperparameter modified throughout its testing[13]. A 10-fold cross validation was performed to create a 95% confidence interval for this GBM model. The 95% confidence interval was [68.81% - 69.22%]; this cross validation took 24 minutes to complete. This model performed markedly worse than the default models for XGBoost and LGBM, so in the interest in time we decided not to refine this model.

5.4. LGBM

LGBM has multiple options for its underlying boosting algorithm, and we were specifically interested in three of them: Gradient Boosting Decision Trees (GBDT), Dropout meets multiple Additive Regression Trees (DART), and Gradient-based One-Side Sampling (GOSS). We first wanted to compare how the performance of these algorithms compared to one another, so we built a simple model[14] for each of them and generated 95% bootstrap confidence intervals[15] for each model. We also thought it important to compare the speed of each algorithm, so we measured how long each model took to generate its 95% confidence interval.

The confidence intervals for GBDT, DART, and GOSS can be seen in Figure 4, Figure 5, and Figure 6 respectively. GBDT, DART, and GOSS had interval means of 71.93%, 69.36%, and 71.69%, and took on average 5.39, 11.07, and 4.69 seconds for each round of bootstrapping respectively. From these results, we can see that the algorithm with the best accuracy was GBDT, but the fastest algorithm was GOSS. Further, GOSS was faster than GBDT despite their accuracy intervals overlapping, and DART both had the worst performance and runtime by a wide margin; we found these comparisons very interesting, and possibly worth further validation in the future.

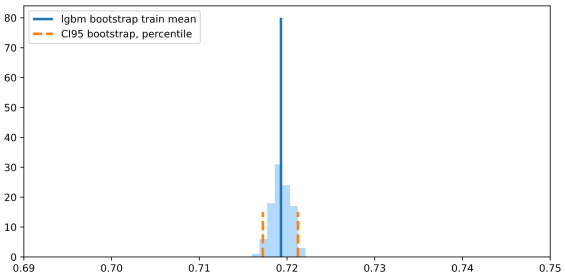

Figure 4. 95% confidence interval for gbdt model.

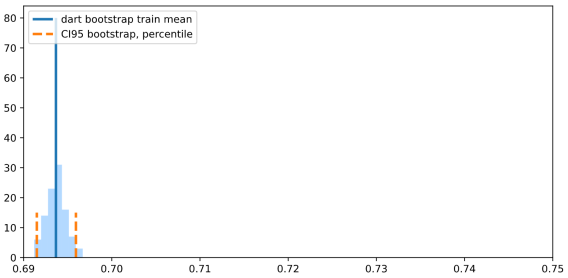

Figure 5. 95% confidence interval for dart model.

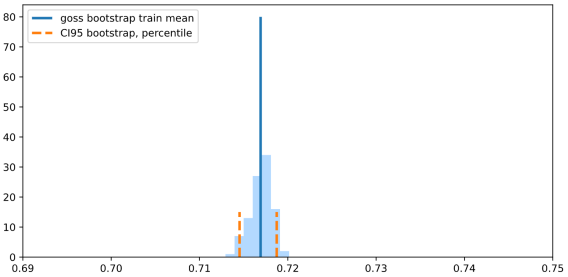

Figure 6. 95% confidence interval for goss model.

After identifying that the GBDT algorithm was the best, we moved to hyperparameter tuning its model. This process was very involved because there were many hyperparameters that we wanted to tune[16]. We used scikit-learn's RandomizedSearchCV (RSCV) in order to make hyperparameter tuning more feasible. We took advantage of RSCV's option to specify hyperparameter distributions by first having a wide range of values to search from and its option to limit the number of hyperparameters being searched for in

[13] sub_sample was set to 0.9.

[14] *subsample* was set to 0.9, *subsample frequency* to 1, *feature importance type* to Gini, *minimum split gain* to 0.25, and *feature fraction* to 0.9. GOSS had issues with some of these hyperparameters, so only *minimum split gain* was changed for its model.

[15] Code adapted from Professor Sebastian Raschka's work: *https://github.com/rasbt/stat451-machine-learning-fs20*

[16] This included the number of leaves, the learning rate, l1 and l2 regularization, the number of iterations run, early stopping, and maximum tree depth.

any given run. In the end, the final GBDT model[17] had a 95% bootstrap accuracy confidence interval of [72.38% - 72.95%] which is a small but noticeable increase from the GBDT model without tuned hyperparameters, mentioned earlier.

5.5. Neural Networks

Though they are often considered the state of the art in machine learning, our best neural network model[18] was not our top performer. In our experiments, we varied the number of hidden layers, nodes per hidden layers[19], learning rate, activation function, total epochs, batch size, and presence of convolutional layers. Our final model achieved 70.01% accuracy on the test set.

In order to utilize 2D convolutional layers, we mapped our one-dimensional feature vectors to uniform two-dimensional matrices; the positioning of each input feature in these two-dimensional matrices was determined by the arbitrary ordering of these features in their original one-dimensional feature vectors. This arbitrary ordering is likely why neither deep nor shallow systems of convolutional layers markedly improved model performance as they are known to do in computer vision tasks[1]. Still, we found that one convolutional layer preceding multiple layers of fully connected perceptrons improved model performance.

Each of our neural network models discussed from now had a single convolutional layer preceding a fully connected perceptron section. We found that the architecture of this fully connected section was the most important factor in model performance. Specifically, the number of hidden layers and nodes per hidden layer had the largest impact; this can be seen in Figure 7. This figure illustrates how too many nodes and layers can lead to intense over-fitting, while too few can lead to under-fitting.

Along with changing the underlying network architecture, we also tuned various model hyperparameters. As for the activation function, two commonly used activation functions are rectified linear unit (ReLU) $f(x) = max(0, x)$ and softplus $f(x) = ln(1 + e^x)$, so we chose to compare these two functions. For the other aforementioned hyperparameters, we chose arbitrary values along a continuum; all of the different hyperparameters were then compared using a 3-fold cross-validated grid search. Our best performing model was trained over 200 epochs with a learning rate of 5e-4, a batch size of 256, and five hidden layers, each consisting of 64 nodes using the ReLU activation function.

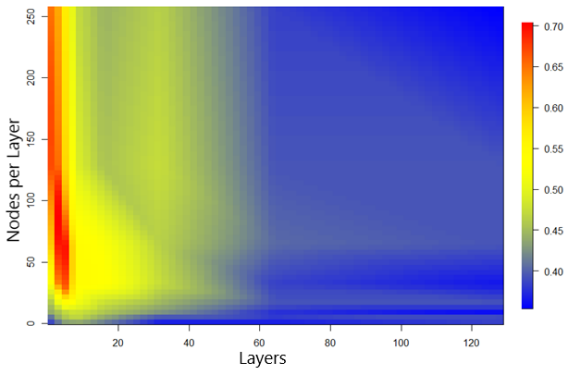

Figure 7. Neural network performance with respect to hidden layers and nodes per hidden layer

6. Conclusions

The models we used, in ascending order of accuracy were, kNN, DART, Neural Network, GOSS, GBDT, and XGBoost[20]. With XGBoost, we were able to achieve 74% prediction accuracy, which is far better than the 33.33% accuracy we would achieve with a totally naive, random classifier. Still, with respect to the accuracy that machine learning can help us achieve on other datasets, 74% is not that high. Why are we not able to do any better? What does this tell us?

As discussed earlier, there are many human elements at play when any given pitch is thrown. For example, if the batter doesn't swing, the umpire is left to call the pitch as either a strike or a ball. Our data contains the exact location of the ball when it crosses the plate, so we can definitively tell you if the baseball was within the strike zone or not, but a human umpire does not have this same information. These types of random effects are riddled throughout each pitch, so it makes sense that we were only able to achieve an accuracy of 74%. If we were able to somehow measure some of these random effects we may be able to improve model performance further but, given the imperfect nature of humans, we suppose that to do so would not be possible. Further, determining the detailed underlying reasons behind each model's ability or inability to accurately classify the data is beyond the scope of this study, so we offer only the prediction accuracy intervals and point estimates provided in the Results and Discussion section.

6.1. Future Direction

Given more time to work on this project, we had a plan to utilize our hit, ball, strike predictions to enhance the predictions of another separate model, such as a model that

[17] The final model had tuned parameters: *min_split_gain = 0.25*, *num_leaves = 100*, *learning_rate = 0.01*, *subsample = 0.9*, *subsample_freq = 1*, *feature_fraction = 0.9*, *lambda_l1 = 1.545*, *lambda_l2 = .215*, *n_iter_no_change = 60*, *num_iterations = 10000*, and *max_depth* = 2000.

[18] In building and training our neural networks, we used TensorFlow 2.0's high level Keras API: https://www.tensorflow.org/api_docs/python/tf/keras.

[19] In all of our models each hidden layer had the same amount of nodes.

[20] Long may she reign!

tries to predict the number of runs scored off of any given pitch. This would result in a type of layered ensemble learning that could experience various advantages over a single model. One such advantage could be reduced prediction variance, a feature that many ensemble learning methods enjoy. Another such advantage would be the ability to more easily develop a more interpretable model if our predicted pitch outcomes lead to the simplification of the second layer of this model.

Before determining that it would not be feasible given our time constraints, we had initially planned to create a model that predicts the number of runs that will be scored on each pitch. We would test the accuracy of this model both with and without the predicted pitch outcome labels discussed in this study and observe whether or not performance differed. There are a few ways that we could try to check if our intermediate prediction has actually made any improvement to our prediction of runs. One way would be to bootstrap and check performance accuracies to see if the raw predictive power of the model changed at all. Another method would be to compute the importance of each of the features in this new model and observe whether or not our predicted labels had high feature importance. If our intermediate predictions have low importance, then it is probably not very helpful, but if it is high or results in the obsolescence of various other features then we can see this as an improvement caused by introducing this layering.

One of the many aspects of this project that were challenging was the amount of time each model took to run on our machines. We were unable to utilize GPUs during our model computations, but future work would heavily benefit from leveraging GPUs. This would speed up training by a significant margin which would allow for better hyperparameter tuning and the ability to explore more learning algorithms than were explored in this study.

7. Contributions

7.1. Data Collection and Pre-Processing

Micah was responsible for acquiring the original data from Baseball Savant and contributed to the majority of the data pre-processing. Nathan and Roshan also helped with the data pre-processing.

7.2. Models

Micah was responsible for the XGBoost modeling and hyperparameter tuning. Nathan handled the kNN and NN modeling. Roshan contributed with the GBM and LGBM models. All group members were also more than willing to lend a hand with another group member's model or be part of the necessary problem solving that comes with a project of this nature.

7.3. Report

All group members contributed equally to the project report.

References

[1] Md. Zahangir Alom et al. "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches". In: *CoRR* abs/1803.01164 (2018). arXiv: 1803.01164. URL: http://arxiv.org/abs/1803.01164.

[2] C. Apte et al. "Predicting Defects in Disk Drive Manufacturing: A Case Study in High-Dimensional Classification". In: (Oct. 1999). DOI: 10.1109/CAIA.1993.366608.

[3] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. URL: http://doi.acm.org/10.1145/2939672.2939785.

[4] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[5] M. Kuhn. "Caret package". In: *Journal of Statistical Software* 28.5 (2008).

[6] Martın Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[7] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[8] Ethan Moore. *Baseball ProGUESTus: A Potential Alternative for Public Pitch Classification*. Sept. 2019.

[9] Harry Pavlidis. *MLB: How catcher framing is becoming essential*. June 2014.

[10] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[11] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2017. URL: https://www.R-project.org/.

[12] RStudio Team. *RStudio: Integrated Development Environment for R*. RStudio, PBC. Boston, MA, 2020. URL: http://www.rstudio.com/.

[13] Sam Sharpe. *MLB Pitch Classification*. July 2020. URL: https://technology.mlblogs.com/mlb-pitch-classification-64a1e32ee079.

[14] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[15] *What is a Hit (H)?: Glossary*. URL: http://m.mlb.com/glossary/standard-stats/hit.

[16] Hadley Wickham et al. "Welcome to the tidyverse". In: *Journal of Open Source Software* 4.43 (2019), p. 1686. DOI: 10.21105/joss.01686.