Building Machine Learning Models to Predict the Outcome of a Pitch in an MLB Game

Authors: Zachary Becker, Zachary Armand, Zhaofeng Li

Northeastern University

DS 5500 Capstone: Applications in Data Science

Spring 2025

April 22, 2025

Contents

Chapter 1

Introduction

For our Capstone Project, we wanted to answer the questions of how to find the most optimal way for hitters and pitchers to approach an at-bat, and what is the most likely outcome for a given pitch profile and batter? Understanding the most likely outcomes for each offering in a pitcher's repertoire helps a baseball team game plan and develop more in-depth approaches to each at-bat, thus providing a competitive advantage and putting teams in a better position to win games. In our work, we implement three different machine learning models to predict estimated wOBA using speed angle values of various at-bats in the 2024 MLB season.

Using machine learning and statistical methods to take advantage of the large amount of data associated with baseball to make useful predictions is hardly a new idea, with numerous other studies leveraging Major League Baseball (MLB) and Statcast data. However, most of these studies have different targets or aims. For example, some studies used baseball data to predict pitch type class[1] or the probability of a certain pitch outcome, like a strikeout.[3][18][24] Some papers have created similar architectures, such as linear neural networks to predict game matchup outcomes,[10] or LSTM recurrent neural networks and XGBoost models to predict pitch class type.[24] In one case, Statcast data and neural networks were combined with the aim of creating a model to improve real-time pitcher performance.[16] While these previous works were interesting and informative as to how baseball data and machine learning have been combined, as far as we are aware our work is one of, if not the only, to pair Statcast data with XGBoost and neural networks in order to predict wOBA. As a result, there were no other comparable baseline metrics that we could reference when evaluating our results. We therefore used common error metrics for regression-based tasks, such as root mean squared error and mean absolute error.

Chapter 2

Data Collection and Processing

2.1 Data Source

Our project exclusively used data from Major League Baseball's Statcast system. Statcast is a baseball tracking system that utilizes high-resolution and high-speed cameras in every MLB stadium to track the location of the ball and every player at all times.[13] Data collection for Statcast began in 2015, eventually fully replacing the previously widespread PITCHf/x data collection system before the start of the 2017 season. Today, MLB Baseball Operations and R&D departments utilize Statcast data on a daily basis and in a variety of ways to improve their club's performance on the diamond.

Using the Statcast system, a variety of different pitch (e.g. pitch velocity, spin rate, release point) and batted-ball (e.g. exit velocity, launch angle, wOBA) metrics are available for nearly every pitch and hit of every MLB game. As of the 2024 season, bat tracking data is also available using Statcast. The general public can access this data through Baseball Savant, MLB's Statcast clearinghouse.[15] For our project, we used Statcast data from the 2024 MLB season, the latest year with a complete season's worth of data. To access this data, we employed the Python package *pybaseball* to directly access the Baseball Savant API via Python,[12] efficiently bulk downloading the necessary data. A high-level overview of our data workflow and training process can be seen in Figure 2.1, including data acquisition, preprocessing, and model training and inference.

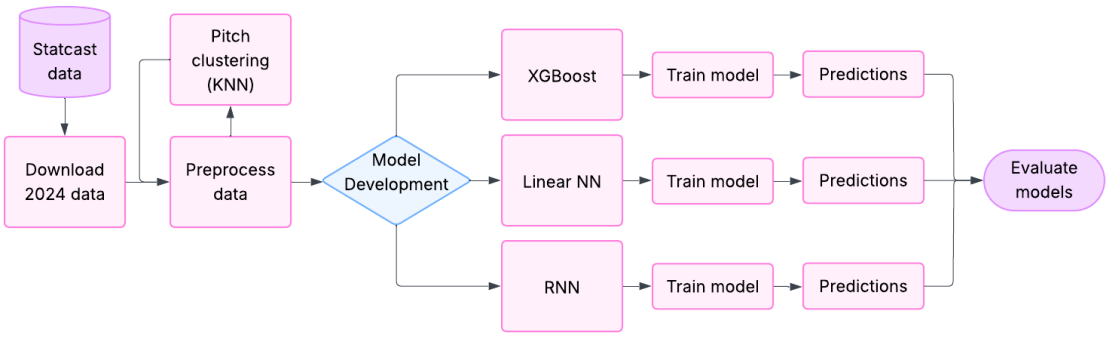

Figure 2.1: Project Workflow

2.2 Preprocessing

Our Statcast dataset contained a row for each pitch thrown in the 2024 MLB season and a wide selection of variables that described player and game contexts, pitch data, batted ball data, and more. More information about the variables found in the dataset can be found on Baseball Savant's documentation.[14]

While there were around 750,000 pitches thrown in the 2024 season, we only used pitches or pitch sequences that were put into play. For example, any at-bats that resulted in a strikeout were removed. To further preprocess the data, we only used regular season games, removed any at-bats with a score difference greater than 6 and removed any pitches with less than 3 appearances in the 2024 season. The reason for this is that, when a game is out of balance, teams begin to shift away from their typical pitching strategies. They don't want to use their best relief pitchers in a game that is not close, and will urge their pitchers to attack the middle of the strike zone more than usual. Any rows with missing values that came as a result of a miscalculation from the Hawk-Eye pitch tracking technology were also dropped. Because of the high volume of data and relatively low number of missing values due to the professional nature of Statcast, removing any at-bats with missing values did not remove an impactful volume of data.

Some of the key features used to do the pitch profile segmentation included the pitch type, the velocity, spin rate, spin axis, vertical break, horizontal break, release extension, and release height. For the hitter-specific data, we looked at the most common launch angles and exit velocities by pitch location (divided into 16 zones) as well as by pitch type. Some hitters will fare better against specific types of pitches and specific locations of the strike zone, so we of course needed to take that into account. We predicted the exit velocity and launch angle of a batted ball, which were then used in calculating the expected weighted on-base average (wOBA) of the play.

The pitch segmentation was done to uncover different pitch shapes that were thrown under the same label. A given MLB pitcher may throw a fastball, slider, and changeup, but the shapes and velocities of the pitches can vary quite a bit when compared to a separate pitcher with the same arsenal. For hitters, some of them may be very skilled at hitting a fastball with plenty of late life, but struggle with a slow sweeper, and some may struggle with high induced vertical break fastballs since they tend to target offspeed pitches more often.

The clustering process began by selecting a set of features from the Statcast dataset, standardizing them, and using KMeans to explore the optimal number of clusters for each pitch type using inertia, silhouette scores, and the Davies-Bouldin index. Once the optimal number of clusters for each pitch was determined, the final KMeans clustering was applied, assigning each pitch to a specific cluster.

After finishing the pitch segmentation, we determined the features that we wanted to include in our predictions:

- **Release speed**: Velocity of the pitch at the point of release (MPH)
- **Release position X**: Horizontal release location
- **Release position Z**: Vertical release location
- **PFX X**: Horizontal movement of the pitch
- **PFX Z**: Vertical movement of the pitch
- **Plate X**: Horizontal location of the pitch when crossing home plate
- **Plate Z**: Vertical location of the pitch when crossing home plate
- **Spin axis**: Axis of rotation of the ball (degrees)
- **Release extension**: Distance from the rubber where the pitcher releases the ball (feet)
- **Estimated wOBA using Exit Velocity and Launch Angle Zone Average**: Average expected wOBA for a hitter on pitches in a given plate zone (only using Exit Velocity and Launch Angle)
- **Estimated wOBA by pitch type**: Average expected wOBA for a hitter on each pitch type
- **Cluster label**: Cluster ID assigned to the pitch determined in the pitch segmentation process
- **Number of times through order**: How many times the pitcher has faced the current lineup in the current game

- **Balls**: Number of balls in the count
- **Strikes**: Number of strikes in the count

Summary statistics for the preprocessed estimated wOBA using speed angle values can be seen in Table 2.1, including mean, median, standard deviation, minimum, and maximum. These values fall within the range [0.0-2.0], with a mean value of 0.365 and standard deviation of 0.384. As can be seen in Figure 2.2, these values roughly comprise an exponential distribution, all in the positive range.

Mean	Median	Std.	Min.	Max
0.365	0.238	0.384	0.0	2.0

Table 2.1: Target Summary Statistics

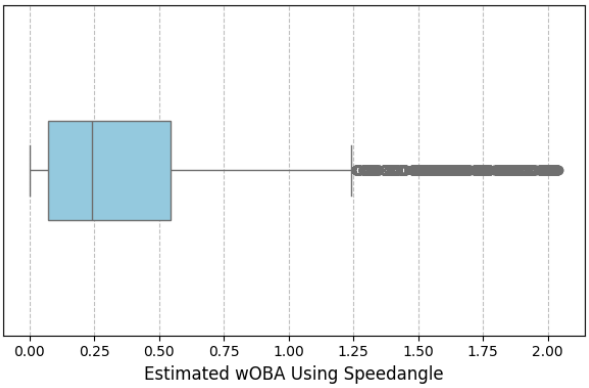

Figure 2.2: wOBA (Target) Distribution

When preparing the data for the XGBoost model, only the individual pitches in each at-bat that were hit into play were preserved, with all other pitches in the sequence dropped. Meanwhile, for the neural network models, all pitches from the entire pitch sequence were used. At-bats were grouped by taking rows with the same game date, at-bat number, and home team. This resulted in a rectangular input matrix, with a row for each pitch in the sequence. Only the final row had a corresponding a target value, representing the outcome of the entire pitch sequence (given that the final pitch in a sequence was the one that was actually hit into play).

Incorporating pitches that weren't hit into play provided the models with a larger lookback window and more contextual data. Furthermore, recurrent neural works are designed to process and understand sequential data more effectively than one-off observations. Special care was taken to ensure that groups of at-bats were preserved when preprocessing the data for the neural networks. When any rows were removed, the entire at-bat sequence was removed. That way only fully intact at-bats with no missing pitches were included in the input data.

Any at-bats with more than 9 pitches total were removed in order to reduce the dimensionality of the final dataset while keeping 99% of the total data. In order to ensure a dimensionally homogeneous input dataset, at-bat sequences were padded with 0 values if needed. When passing the input data sequence into the recurrent neural network, the last row of data to be processed was the last pitch in the at-bat, i.e. the one that was hit into play. For the Linear Model, the rectangular input data was flattened before being passed into the network, resulting in a long one-dimensional input vector.

We partitioned our cleaned data into an 80/20 train/test split for the XGBoost models, using random state 4400 for sci-kit-learn's random train-test split function. The neural network models necessitated an additional validation set for model evaluation during training. These models used an 80/10/10 train/test/validation split using PyTorch's random split function.

Chapter 3

Model Selection and Evaluation

3.1 Model architecture

Three models were implemented in our work: an eXtreme Gradient Boosting Regression model (XGBoost), a linear neural network ("Linear Model"), and a recurrent neural network ("Recurrent Model"). All three models were trained to predict the estimated wOBA using the speed angle values. Please see Figure 3.1 for an overview of the architecture of the two custom neural network models. For both neural networks, a Leaky Rectified Linear Unit activation function, or Leaky ReLU, was applied after each fully connected layer (or normalization layer, if applicable) in both models. The Leaky ReLU had a negative slope of 0.02. Note that in Figure 3.1, Leaky ReLU is shortened to just ReLU.

Dropout, a popular regularization technique,[8] was also applied after the activation function in several layer blocks in both models (see Figure 3.1 for precise location). The precise value of p, or probability of an element to be zeroed during dropout, was a hyperparameter tuned before training. Within a model, the value of p was consistent across all locations.

Gradient norm clipping was also introduced during training. The value at which to clip gradients, or whether to clip them at all, was a hyperparameter that was tuned before training. Gradient clipping is typically used to stabilize gradients during training and avoid any vanishing or exploding gradients, especially in recurrent neural networks.[17] Although modern gated recurrent architectures like Long Short-Term Memory (LTSM) units were designed to overcome gradient-based issues when training recurrent neural networks,[9] gradient clipping can still prove to be a powerful tool in preserving gradient stability.[17] Additionally, because gradient clipping was introduced as a hyperparameter, we could experimentally decide whether or not to include gradient clipping if it improved performance. After tuning, gradient clipping was still included alongside weight decay and dropout, suggesting that it was still a helpful regularization technique.

3.1.1 XGBoost

To start, raw pitch tracking data from the 2024 season was preprocessed to get our relevant columns, standardize feature scales, and perform feature engineering to create a handful of new columns. Hitter "hot zones" and pitcher "zone success" metrics were creating using K-Nearest Neighbors modeling of launch angle and exit velocity outcomes, and the cluster labels for each pitch type were merged back into the full dataset. Figure 3.2 displays the launch angle and exit velocity combinations that lead to the best results in terms of wOBA.

The model was trained using an XGBoost regressor, which was an ensemble gradient-boosting tree algorithm that was optimized for minimizing MSE. Categorical characteristics such as pitch type were encoded one-hot prior to training, and hyperparameters such as maximum tree depth, learning rate, regularization strength, and sub-sample ratios were tuned using Grid Search. For the model accuracy, we checked the root mean squared error, mean abolute error, and R-squared metrics, as well as SHAP values to view the relative importance of feature groups such as pitch release characteristics, movement, and location.

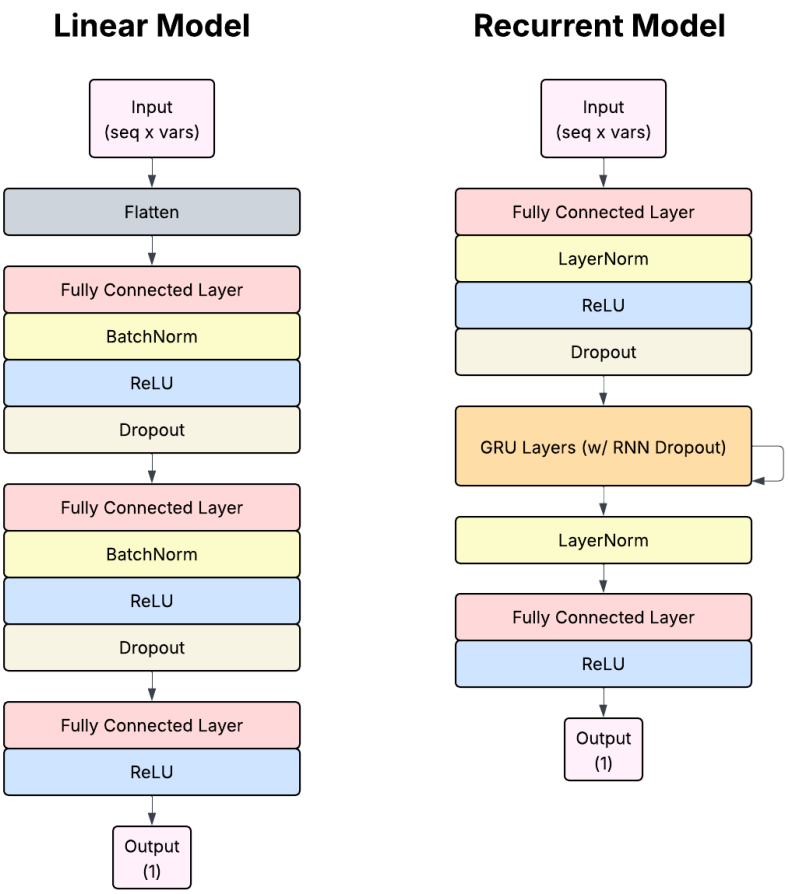

Figure 3.1: Neural Network Model Architectures

We also paid attention to how model performance varied across pitch clusters and pitch types. The variance that came from the model mainly resulted from typically weak batters hitting home runs, or the occasional extra-base hit on a pitch that a hitter has historically struggled with.

3.1.2 Linear Model

The Linear Model consisted of three layer blocks. To start, the sequential input data was flattened to a long one-dimensional input vector of mn, where m is the number of pitches (sequence length) and n is the number of input variables. The first layer took this input vector and increased the dimensions of the data into the hidden size, a hyperparameter that was tuned. No change in dimensionality occured in the middle layer, and the final output layer condensed the dimensions from this hidden size into a singular floating point value suitable for regression.

Aside from the change in number of neurons as data passed through the layers, the input and middle layer blocks were identical in structure. They took an input tensor, passed it through a fully-connected linear layer, applied batch normalization to the output, sent this normalized data through a nonlinear activation function, and then applied dropout. That is, the output of one of these blocks was $x = Dropout(LeakyReLU(BatchNorm(Linear(x))))$. The final block was simpler, consisting of a fully-connected linear layer and a nonlinear activation function (Leaky ReLU). The output of the final layer block was the network's prediction based on the input data.

Batch normalization, also known as BatchNorm,[11] which was applied to the first two blocks of the

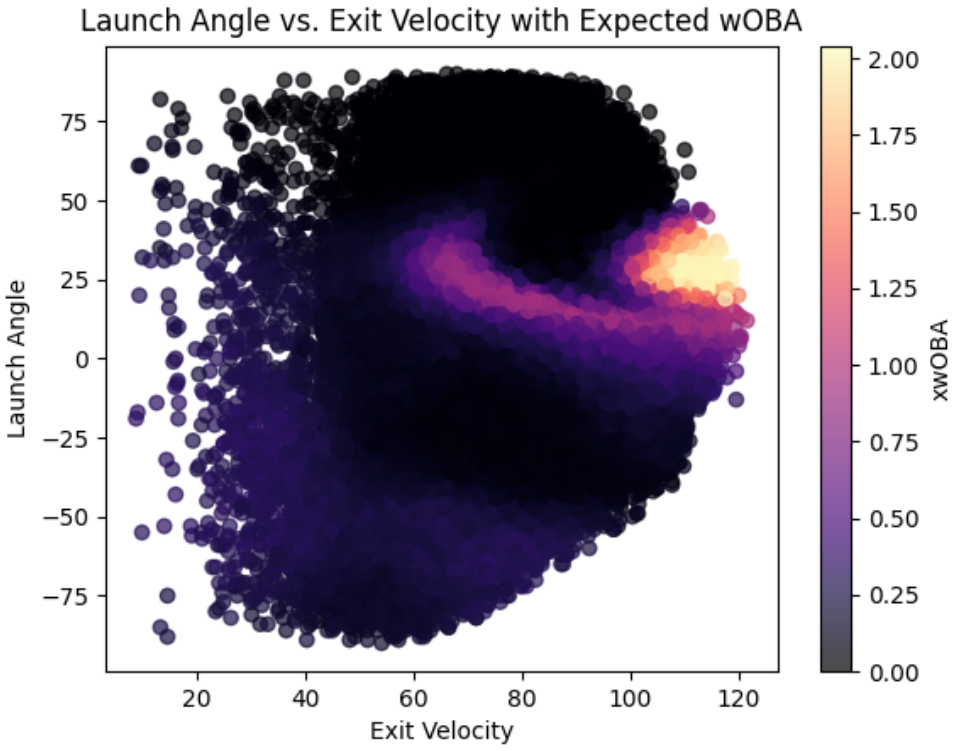

Figure 3.2: Exit Velocity and Launch Angle wOBA Interaction

Linear Model, aims to improve the regularization of networks and optimization convergence during training. Layer inputs are normalized during training using unit-variance standardization by calculating the mean and standard deviation of each mini-batch and applying the transformation to the layer. Running estimates of the mean and variance are computed across all mini-batches in order to normalize inputs during model inference. Additionally, because the implementation of BatchNorm includes two learned additive parameters to scale and shift the normalized value,[11] the bias term in the preceding fully connected layer was redundant and therefore removed from the model architecture.

3.1.3 Recurrent Model

Recurrent neural networks are particularly well-suited to handle sequential data, such as pitches in an at-bat. The output from each RNN cell is fed into the following cell as a hidden state representation, until all input vectors are processed. In our case, we were using a many-to-one RNN model, where multiple inputs ultimately produced a singular output value. Currently, there are several different RNN architectures that aim to handle issues with processing sequential data, such as exploding/vanishing gradients or limited lookbook capacity. Examples include Long Short-Term Memory (LTSM)[9] units or Gated Recurrent Units (GRU)[5]

In our work, we implemented GRUs in the Recurrent Model's recurrent layer(s). GRU cells, seen in Figure 3.3,[22] incorporate gating structures that learn to forget or remember features for later processing with reset, update, and new gates. In Figure 3.3, σ representing a logistic function, $tanh$ the hyperbolic tangent activation function, r_t the reset gate, z_t the update gate, x the input vector, h the the output vector at timestep t, and $\hat{h}$ the candidate activation vector. For a concise summary of the mechanisms and formulas that underlie the GRU architecture, see PyTorch's GRU cell documentation.[21]

Although there is some debate as to whether LTSMs or GRUs are better when handling sequential data in general (although both are more effective than non-gated "vanilla" RNNs),[6] evidence exists suggesting that GRUs are better at handling simpler sequential sequences.[4] Given that our data had a maximum sequence length of less than 10 with less than 20 variables, and common sequential tasks such as text processing can extend into the thousands, GRUs appeared more promising for our work. GRU

layers can also be stacked natively in PyTorch, and we experimented with the number of GRU layers as a hyperparameter.

The Recurrent Model in our work consisted of a fully connected linear layer that takes an input vector of shape mn, where m was the number of pitches (sequence length) and n is the number of input variables. This first layer increased the dimensions to the hidden size, a hyperparameter that was tuned. The output of this layer was then fed into the GRU cells, which had the same hidden layer dimensionality. After passing through L-GRU layers, with the number of layers L being tuned before training, the output of the last time step was then sent to a final fully connected layer. This reduced the dimensionality to a single floating point value suitable for regression.

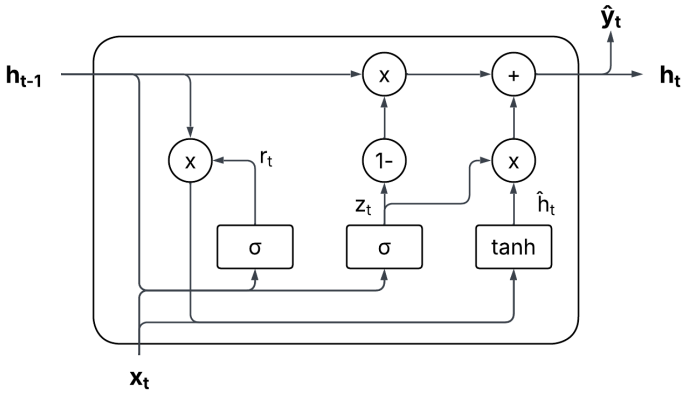

Figure 3.3: GRU Cell Diagram

A Leaky ReLU activation with a negative slope of 0.02 was applied to the output of each fully connected layer. Before training, the hidden state values of the GRU layers were initialized with a Xavier uniform distribution, also known as Glorot initialization.[7] Within the GRU layers, RNN Dropout[25] was applied, which differs in implementation slightly from the normal dropout that was applied after fully connected layers in both neural network models. However, in this model, the dropout percentage was the same across all instances within the same model.

Layer normalization, also known as LayerNorm, was applied to the output of the first fully connected layer as well as the final GRU layer output. Layer normalization is similar to the batch normalization used in the Linear Model, but solves some of the issues that make batch normalization difficult to apply to recurrent neural networks by applying normalization to each input sequence individually. It also applies the same transformation during training and testing, which can help account for sequences of varying lengths.[2] The first fully-connected layer output therefore appears as $x = Dropout(LeakyReLU(LayerNorm(Linear(x))))$.

3.2 Experimental Setup

3.2.1 Hardware and Software

Our work exclusively used Python 3 for data retrieval, preprocessing, and model creation and training. To retrieve Statcast data, we took advantage of the *pybaseball* python package,[12] which allows for seamless connection to the Statcast API to download large volumes of Statcast data. The XGBoost model used XGBoost's eXtreme Gradient Boosting Regressor (XGBRegressor), a common XGBoost library available in multiple languages.[23] The neural network models were implemented in PyTorch.[20] Data preprocessing and model evaluation took advantage of common python packages like sci-kit-learn, numpy, pandas, and seaborne.

The neural network models were trained on two GPU-capable devices: an Apple M1 Pro with 16 GB of memory, taking advantage of the device's integrated GPU and PyTorch's MPS backend,[19] and a NVIDIA GeForce GTX 1060 with 3 GB of memory.

3.2.2 Hyperparameter Tuning

Hyperparameters in all models were tuned. The XGBoost model used a grid search and three stratified folds for cross validation, tuning the maximum depth [2, 4, 6, 8], learning rate [0.04, 0.1, 0.2], number of estimators [100, 200, 300], subsample [0.85, 0.9, 0.95], sample of features used for each tree [0.6, 0.7, 0.8], as well as the L1 (alpha) [0.1, 1, 10] and L2 (lambda) [0.1, 1, 10] penalties.

The two neural network models were tuned using a random search, with each combination being tuned for 100 epochs with early stopping, and MSE as a tuning loss function. Similar hyperparameters were tuned for these two networks, with a complete list of parameters and their values listed in Table 3.1. For each model, 500 different random combinations were chosen for tuning, with the combination producing the lowest loss value chosen for final training. A full list of all tuning combinations and their loss values can be found in the GitHub repository.

Parameter	**Values**	**Model(s)**
Batch size	16, 32, 64, 128, 256	Linear, Recurrent
Optimizer learning rate	1e-2, 1e-3, 1e-4, 1e-5, 1e-6	Linear, Recurrent
Dropout percentage	0.0, 0.1, 0.2, 0.3, 0.4, 0.5	Linear, Recurrent
L2 weight decay penalty	0.0, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5	Linear, Recurrent
Hidden layer dimensions	8, 16, 64, 128, 256	Linear, Recurrent
Gradient clip value	None (no clipping), 0.5, 1.0, 2.0, 5.0	Linear, Recurrent
GRU layers	1, 2, 3, 5	Recurrent

Table 3.1: Neural Network Tuning Parameters and Values

3.2.3 Training and Evaluation

Our three models all aimed to predict the same metric after training on our preprocessed data: estimated weighted on-base average (wOBA). This variable was a floating-point value, so our models were all designed to perform a regression task with a single output value, which was then compared against the true wOBA value from our source dataset. Accordingly, we used standard regression performance metrics to compare our model outputs against the true values. These included root mean squared error (RMSE), mean squared error (MSE), mean absolute error (MAE), and the coefficient of determination, or R^2 score.

During training, both the Linear Model and the Recurrent Model used the ADAM optimizer (with L2 weight decay penalty and learning rate chosen after hyperparameter tuning) and MSE as a loss function. Both models were trained for 150 epochs with early stopping with a patience of 15 used to halt training if validation loss did not improve sufficiently. After hyperparameter tuning, the Linear Model used a batch size of 32, a learning rate of 1e-4, dropout percentage of 0.2, weight decay of 1e-4, 16 hidden dimensions, and gradient clipping value of 1.0. The Recurrent Model used a batch size of 32, a learning rate of 1e-4, dropout percentage of 0.3, weight decay of 1e-4, 128 hidden dimensions, a gradient clipping value of 0.5, and 1 GRU layer.

Chapter 4

Results and Discussion

4.1 Neural Network Training

Using the tuned hyperparameters listed above, the final Linear Model and Recurrent Model configurations were trained for 58 and 47 epochs, respectively. Both models were allowed 150 epochs to train with early stopping. The results of the training and validation losses of both models, along with final test loss, can be seen in Figure 4.1. Training appears to be relatively stable, with no major spikes or fluctuations in loss occurring. The training and validation loss curves of both models do not appear to diverge from each other in any major way, suggesting that the models were both sufficiently generalized. Early stopping also appeared to be beneficial, as the validation loss of both models were leveling out at the time of early stopping.

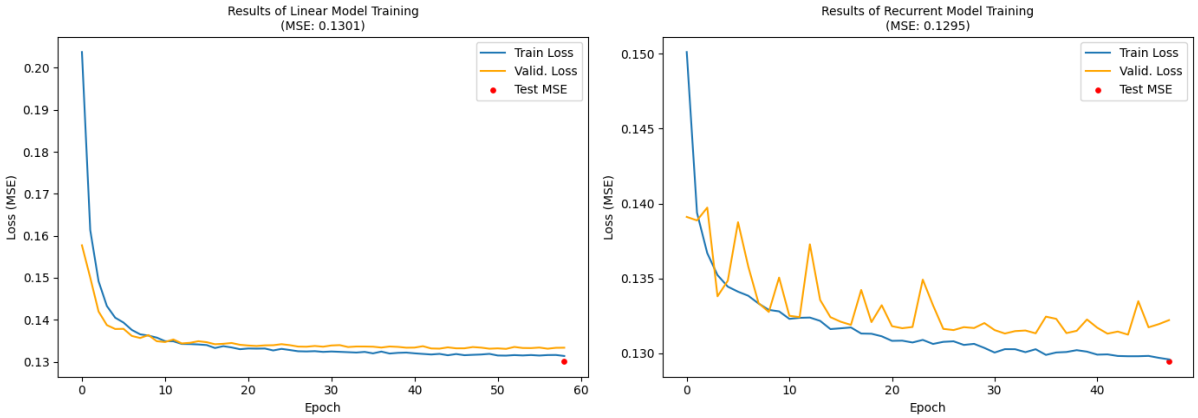

Figure 4.1: Neural Network Training Loss

4.2 Quantitative Results

Table 4.1 shows the performance of each of the three models on the test set across three different metrics: root mean squared error, mean absolute error, and R^2 score. These values were all calculated when comparing the predictions of each model against the true values of expected wOBA using the speed angle found in the test set. All three models performed relatively equally (RMSE of ± 0.007).

The fact that the Linear Model and the Recurrent Network performed so similarly (RMSE of 0.360 vs. 0.359, respectively) might suggest that the recurrent structure didn't provide any major benefits when learning the structure of the dataset they are trained on. Recurrent neural networks require context (provided by sequences) in order to understand the dataset and update their hidden representations sufficiently. In our case, given that the longest sequence was less than 10 in length, the at-bat sequences may not have been sufficiently long enough to provide the GRU cell the context needed to significantly improve its predictions.

Model	RMSE	MAE	R2 Score
XGBoost	0.363	0.273	0.11
Linear Model	0.357	0.274	0.10
Recurrent Model	0.356	0.269	0.10

Table 4.1: Testing Set Performance

The similarity in model performance can also be observed in Figure 4.2, which shows the plot of the residuals of each model, or the difference between the true values and the predicted values. As seen in Figure 4.2, the plots of the three models are not substantively different, suggesting that they were able to learn and predict the data with similar capacity. Further investigation into whether the models are similar due to similar capacity or whether it is a reflection of the inherent trends in the data is warranted.

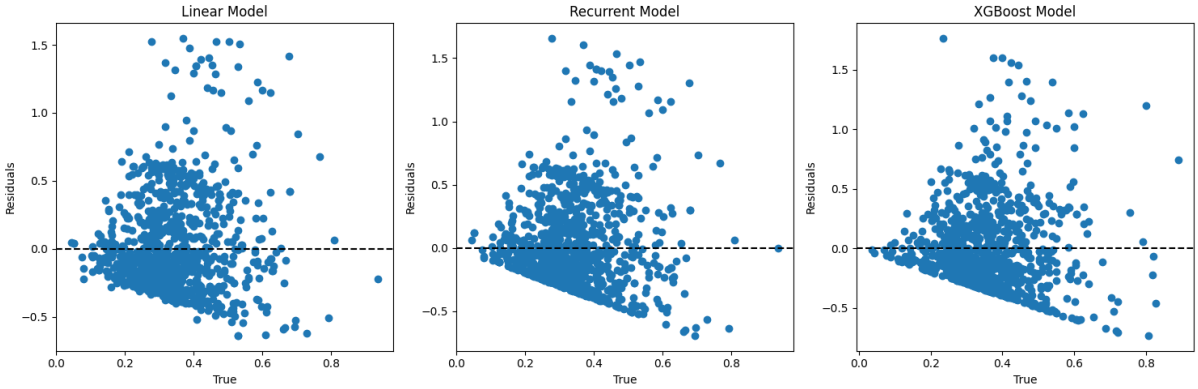

Figure 4.2: Model Residuals

Chapter 5

Future Work

As data capture improves within MLB parks and new Statcast data continues to become publicly available, more and more future work should become available. Baseball Savant recently released new bat tracking data, which could allow us to look at deviations in bat speed and bat paths through the strike zone for various pitches and locations that a batter sees. If we are able to combine previous success with more granular bat tracking metrics, that will allow us to get a better sense at the types of pitches that are deceiving specific hitters, providing teams with more information to work with when game planning.

With our current findings and the data available to us right now, we can add more years of data, and test on the 2025 season. It would also likely be beneficial to incorporate external data about weather, ballpark dimensions and altitude, or try and predict other batted ball metrics. There are several other ways to gauge quality of contact made, such as the barrels per batted ball event percentage, or hard hit percentage.

Bibliography

[1] Adam Attarian et al. "A Comparison of Feature Selection and Classification Algorithms in Identifying Baseball Pitches". In: *Proceedings of the International MultiConference of Engineers and Computer Scientists*. Vol. I. Lecture Notes in Engineering and Computer Science. IMECS 2013, March 13-15, 2013. International Association of Engineers (IAENG). Hong Kong: IAENG, Mar. 2013, pp. 263–268.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML]. URL: https://arxiv.org/abs/1607.06450.

[3] Jasmine Barbee. "Prediction of Final Pitch Outcome in MLB Games Using Statistic Learning Methods". MA thesis. California State University, Long Beach. Department of Mathematics and Statistics, 2020. URL: https://link.ezproxy.neu.edu/login?url=https://www.proquest.com/dissertations-theses/prediction-final-pitch-outcome-mlb-games-using/docview/2492950717/se-2?accountid=12826.

[4] Roberto Cahuantzi, Xinye Chen, and Stefan Güttel. "A Comparison of LSTM and GRU Networks for Learning Symbolic Sequences". In: *Intelligent Computing*. Springer Nature Switzerland, 2023, pp. 771–785. ISBN: 9783031379635. DOI: 10.1007/978-3-031-37963-5_53. URL: http://dx.doi.org/10.1007/978-3-031-37963-5_53.

[5] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL]. URL: https://arxiv.org/abs/1406.1078.

[6] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE]. URL: https://arxiv.org/abs/1412.3555.

[7] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Vol. 9. PMLR. 2010, pp. 249–256.

[8] Geoffrey E. Hinton et al. *Improving neural networks by preventing co-adaptation of feature detectors*. 2012. arXiv: 1207.0580 [cs.NE]. URL: https://arxiv.org/abs/1207.0580.

[9] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[10] Mei-Ling Huang and Yun-Zhi Li. "Use of Machine Learning and Deep Learning to Predict the Outcomes of Major League Baseball Matches". In: *MDPI* 11.10 (2021), p. 4499. URL: https://www.mdpi.com/2076-3417/11/10/4499.

[11] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[12] James LeDoux. *pybaseball*. GitHub repository, available at GitHub. 2023. URL: https://github.com/jldbc/pybaseball.

[13] Major League Baseball. *Statcast Glossary*. Accessed: 2025-04-12. MLB Advanced Media, LP, 2025. URL: https://www.mlb.com/glossary/statcast.

[14] Major League Baseball Advanced Media. *Statcast Search CSV Documentation*. Accessed: 2025-04-15. MLB Advanced Media, LP, 2025. URL: https://baseballsavant.mlb.com/csv-docs.

[15] MLB Advanced Media. *Baseball Savant*. 2025. URL: https://baseballsavant.mlb.com/.

[16] Stephen Eugene Otremba Jr. "SmartPitch: Applied Machine Learning for Professional Baseball Pitching Strategy". MA thesis. Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science, 2022. URL: https://hdl.handle.net/1721.1/145144.

[17] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG]. URL: https://arxiv.org/abs/1211.5063.

[18] Yifan PI. *Predict Next Baseball Pitch Type with RNN*. [Semester project, Stanford University]. 2018. URL: https://cs230.stanford.edu/projects_spring_2018/reports/8290890.pdf.

[19] PyTorch Team. *MPS backend*. Version 2.6.0. Accessed: 2025-04-15. PyTorch Foundation. 2025. URL: https://pytorch.org/docs/stable/notes/mps.html.

[20] PyTorch Team. *PyTorch*. Accessed: 2025-04-15. PyTorch Foundation. 2025. URL: https://pytorch.org/.

[21] PyTorch Team. *torch.nn.GRU*. Version 2.6.0. Accessed: 2025-04-15. PyTorch Foundation. 2025. URL: https://pytorch.org/docs/stable/generated/torch.nn.GRU.html.

[22] Wikipedia contributors. *Gated recurrent unit — Wikipedia, The Free Encyclopedia*. [Online; accessed 2025-04-15]. 2024. URL: https://en.wikipedia.org/wiki/Gated_recurrent_unit.

[23] XGBoost Developers. *XGBoost Python Package*. Version 3.0.0. Accessed: 2025-04-15. DMLC. 2025. URL: https://xgboost.readthedocs.io/en/stable/python/index.html.

[24] C.-C. Yu, C.-C. Chang, and H.-Y. Cheng. "Decide the next pitch: A pitch prediction model using attention-based LSTM". In: *2022 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*. 2022, pp. 1–4. DOI: 10.1109/ICMEW56448.2022.9859411. URL: https://doi.org/10.1109/ICMEW56448.2022.9859411.

[25] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. *Recurrent Neural Network Regularization*. 2015. arXiv: 1409.2329 [cs.NE]. URL: https://arxiv.org/abs/1409.2329.