CS 4641 Final Project

Machine Learning MLB Predictor

Marcus Ma and Cole Johnson

December 10, 2021

Abstract

The second most watched sport in America, baseball, attracts approx. $1 billion in bets every year and, as such, is a prime financial application of predictive models. Historically, people have relied on statistical and purely mathematical models for this prediction. In this study, we take an alternative approach to explore the efficacy of machine learning models to the problem of predicting game results based on player and team features. To do this, we compare Random Forest, ElasticNet, and XGBoost models.

1 Introduction

Over the course of its two-hundred year history, baseball has accrued many superstitions. Perhaps most well known is the Curse of the Billy Goat, in which a disgruntled Cubs fan was asked to leave Wrigley Field after his accompanying pet goat was bothering nearby attendees; as he was escorted from the premises, he vexed the Chicago Cubs into a playoff pennant drought for the next 71 years. Put frankly, baseball's superstitions arise from the sheer randomness of the game. In a sport where the best batters get a hit 3 times out of 10, a team's performance can vary wildly from day to day. A winning percentage of 60% is fantastic in baseball, where a similar win rate in other sports would fail to turn any heads.

The paradox of baseball comes from its depth of analytical data despite the inconsistent results. There are many things that enable baseball to be suitably encapsulated as numbers; every play is a discrete event with almost perfect independence between them (a data scientist's dream!). Leveraging this dearth of data, we aim to create a baseball prediction model that guesses the outcome of the game given both teams' historical and current season stats. Using publicly available baseball records, we model each game as an independent list of features ranging from the team's winning percentage to individual players' career and season batting stats.

2 Materials & Methods

Data Collection & Preprocessing

The first step of any machine learning project is to collect the dataset. We decided to incorporate all regular-season games from 1980-2019, a range of forty years. For each game, we wanted to aggregate the following stats as features (reference here for baseball terms):

1. **Current season team stats (7 features per team)**

   (a) The team's cumulative season Wins, Games Played, OBP, SLG, ERA, Runs/game, HR

2. **Historical team stats (4 features per team, per year)**
   (a) The team's Win%, OBP, SLG, ERA of the three previous years
3. **Batter stats (14 features per player, per team)**
   (a) The batter's career OBP, SLG, HR, BB, RBI, K, AB
   (b) The batter's cumulative season OBP, SLG, HR, BB, RBI, K, AB
4. **Pitcher stats (12 features per player, per team)**
   (a) The pitcher's career Outs, BB, K, BAOpp, ERA, HR
   (b) The pitcher's cumulative season Outs, BB, K, BAOpp, ERA, HR

We chose these stats as a good representation of the performance of a team. There is a mix of rate stats (i.e. OBP and SLG) and counting stats (i.e. K and HR) to ensure that a player who hits poorly but has many at-bats does not outperform a platoon player, for example, who hits well but has limited plate appearances. We chose to disregard fielding statistics as they are often negligible when compared to batting and pitching metrics. Since we are only including the starting roster for individual player features, we added the collective team performance to account for substitute players and bullpen pitchers.We also considered more advanced sabermetric stats such as OPS+ and WAR, but we found that these were a lot harder to calculate and deemed that these features were enough to create an accurate model.

There are many baseball archives online, and the two databases we decided upon were https://retrosheet.org/ and http://www.seanlahman.com/baseball-archive/, with the former having downloadable .txt files and the latter having an SQL database, both of which simplified our data creation pipeline considerably over web scraping. Retrosheet provided play-by-play logs of every season while the Lahman Database was used to aggregate career statistics. As such, we will outline below our process for each database.

Lahman Database

Querying the Lahman Database was quite simple. For example, the SQL query for retrieving the batter stats is:

```
SELECT playerID, (sum(H) + sum(BB) + sum(HBP) + sum(IBB))
/ CAST((sum(AB) + sum(BB) + sum(IBB) + sum(HBP) + sum(SF))
as REAL), (sum(H) + sum("2B") + 2*sum("3B") + 3*sum(HR)) /
CAST(sum(AB) as REAL), sum(HR), sum(BB), sum(RBI), sum(SO), sum(AB)
FROM batting WHERE {y1} <= yearID AND yearID < {y2}  GROUP BY playerID;
```

This collects all of the player's career stats into a single table, where each row corresponds to a given player's cumulative career statistics to a given year. For example, if we were evaluating the game on 6/12/16 where the SF Giants played the LA Dodgers at home, then the career stats of Buster Posey, the Giants catcher, would be aggregate from his rookie year, 2008, until the previous season of 2015. We produced similar queries for career pitching stats and team history stats.

Retrosheet

The Retrosheet process was a lot harder. While we can uncover historical data via the Lahman Database, we want to give more weight to the stats from the current season. However, we cannot simply aggregate the entire season into those particular features; for example, the Chicago Cubs went 103-58 in the 2016 season, so if we just predicted "Win" for every Cubs game, we would get an impressive accuracy of 64%. To keep this an accurate model, however, if we're evaluating a game that took place on May 8, we only want to count season stats up to May 7. The only way to evaluate season stats up to a certain point is to look at the play-by-play logs and game logs provided by Retrosheet. The play-by-play logs for each season looked like the following:

```
play,5,1,sandp001,21,CBBX,6/P
play,5,1,poseb001,11,BCX,HR/7/L
play,5,1,pench001,11,BFX,63/G-
```

Each line corresponds to a particular play that occurred. We can decipher the above example as Pablo Sandoval popping out to the shortstop, our good friend Buster Posey hitting a home run, and Hunter Pence grounding out to shortstop. In terms of our stats, Sandoval's AB and PA would increase by one and Posey's AB, PA, HR, H, and RBI would increase by one and his TB would increase by four. We can then calculate the features based on these counting stats. We used a dynamic programming approach to improve efficiency by creating examples for games sequentially while we iterated through the play-by-play logs, such that we only had to iterate through a season's event logs once. This dynamic programming approach meant we could create data examples for a given season (approx. 2500 games and 300000 plays) in seven minutes on a standard laptop, while a more naive approach took upwards of three hours. Using the methods outlined above, we iterated through the 1980 to 2019 seasons to produce a total of 90,519 examples.

Machine Learning Models

In this study, we chose to compare three different models: a random forest classifier, ElasticNet, and the XGBoostClassifier, all trained and tested used a standard 80-20 split with random permutation of entries.

We predicted that classifiers with architecture similar to decision trees would perform the best as the features we chose to feed into the models were direct representations of the quality of each team. Thus, this suggests that a series of simple comparison calculations would be sufficient to predict the outcome of a given match. Hence, the first model we chose was a popular ensemble decision tree – scikit-learn's Random Forest Classifier.

Another model we thought would perform relatively well was scikit-learn's ElasticNet. This model is generally used when the features form groups that contain correlated independent variables, which, in the case of baseball statistics, they do.

Finally, we evaluated the XGBoostClassifier. Similar to the random forest, we predicted that the XGBoost model would perform well as we had an extremely large number of observations in our training data. In addition, the XGBoost model tends to train well when the data consists of both numerical and categorical features, which our set does. In comparison with the random forest approach, although similar, XGBoost is more appealing as it uses similarity scoring to prune the data before modeling. An idea we originally had was to use principle component analysis to do this in a preprocessing step, so XGBoost could validate this approach.

Although minimal tuning of the hyperparameters was required, we used cross validation and grid search to find the optimal parameters. For each of the aforementioned models, we first did a broad parameter search to find the approximate neighborhoods of optimal parameters, then narrowed our field and performed an additional grid search. In all of these procedures, the objective function we used was the minimization of absolute error. Over the course of optimization, we used a development data set to validate that we were not overfitting the training data and finally, a test set to compare the different accuracies of the models.

3 Results

Our baseline that we will compare our results against are the Las Vegas betting lines for each game (which we scraped from covers.com). Over the forty-year timespan, Las Vegas was right 56% percent of the time; while this accuracy may seem only a smidge better than a coin flip, this is due to baseball's inherent randomness as addressed in the introduction. It is important to note that in the last decade or so accuracy has improved to about 59%, so evidently Las Vegas must also be benefiting from some of machine learning advancements.

When evaluated on the test data, XGBoost performs significantly better than both alternatives without any tuning of the hyperparameters.

	Random Forest	ElasticNet	XGBoost
Mean Squared Error	0.686544	0.496830	0.677766
Mean Absolute Error	0.471343	0.492769	0.459367
r^2 Error	-0.895712	0.007227	-0.847545
Accuracy Score	0.528657	0.507231	0.540633

Table 1: Default Model Metrics

We then optimized the models per the produced detailed above. This resulted in an odd accuracy score for ElasticNet as any optimization yielded a worse accuracy than previously obtained used the default parameters. Other than that, the results followed our initial hypothesis: despite the Random Forest approach improving more than the XGBoost model, the XGBoost model out performs by approx. .1%, crossing the 55% accuracy threshold.

	Random Forest	XGBoost
Accuracy Score	0.548902	0.550898
Improvement	0.023382	0.010265

Table 2: Optimized Model Accuracy

The optimal hyperparameters we found were {bootstrap: True, max_depth: 10, max_features: 'sqrt', min_samples_leaf: 2, min_samples_split: 5, n_estimators: 200} for the random forest and {max_depth=2, n_estimators=98} for the XGBoost model. This is interesting as both the number of estimators and the max_depth are very low compared to the default values of the model, perhaps suggesting that only a handful of the metrics we provided the models are highly influential in accurate classification.

Conclusion

When compared to the 56% accuracy Las Vegas baseline, which represents world-class sports prediction models, we are very happy with our 55% accuracy produced from XGBoost. We opted to implement rudimentary sabermetric stats to keep modelling and data collection simple. However, more advanced stats like Pythagorean run differential and Wins Above Replacement have been shown to be better indicators of performance than classic stats like ERA and OBP, so we are confident that if we changed our pipeline to include those stats we would have a model that could cross the 56% accuracy threshold. Regardless of these improvements, we are satisfied with our preprocessing pipeline and XGBoost model and we think this project lays the groundwork for any further sabermetric analysis in the future.