Predicting College Football Scores

Cameron Chamberlain, Matt Feist, Devan Gwynn, Kaiden Williams
CS 270, Fall 2024
Department of Computer Science
Brigham Young University

Abstract

This experiment aimed to learn which machine learning model, parameters, and features can predict the score of a college football game the most accurately. We trained the models from per-game team statistics from the 2022 to 2024 seasons. To improve the features, we removed any stats having to do with points so the model would not rely on them. We tested a variety of machine learning models, trying to reduce the mean absolute error for each. The best-performing model was the Multi-Layered Perceptron with a mean absolute error of 4.71. This means we can predict the score of a game within five points for each team, on average.

1 Introduction

American football is one of the most popular sports in the United States. A large part of that popularity stems from its apparent unpredictability. As a casual observer, it seems like anything can happen during the 60 minutes the game is played on the field. A promising drive, with a team bringing the ball 90+ yards down the field, can end with a turnover. A 20-yard chip-shot field goal can be missed. All of this is especially true when it comes to college football. Thanks to the popularity of the sport, a wide variety of data exists for most college football games. This means that after every game, two things are left: a final score and a collection of statistics detailing what happened in the game. As college football fans, our team was curious to see how the outcome of a game related to the statistics gathered from that game. More specifically, we wanted to see how accurately the final score of college football games could be predicted by using the statistics from that game.

2 Methods

In this section, we discuss the methods we used to predict the score of a college football game.

2.1 Layout

We obtained the data from the ESPN website as well as an API. ESPN is one of the most well-known sources for game statistics and scores, including college football. The website tracks information well into the past for all games played. To get the data from the website, we scraped web pages containing team stats for each game using Python's Beautiful Soup package. From this, we extracted the scores and amount of yards for passing, rushing, etc.

The API we used was the College Football Data API. Using the /games/teams endpoint, we gathered data containing a row for each game a team played with various team stats for that game. Since we wanted to predict the scores of both teams for any given game, we then combined the two rows corresponding to an individual game into one row that contained both teams' stats for that game.

One of the critical decisions we made was regarding which games to include in our dataset. More specifically, we discussed the pros and cons of using more than just the current season's data, particularly as it relates to making team-specific predictions. Since team strength varies across seasons, including past seasons' data could provide misleading information to the model when trying to make team-specific predictions. Ultimately, we decided the benefit of having additional instances outweighed the cost of possibly having less accurate team-specific predictions. Further discussion on incorporating past seasons' data while accounting for differing team strengths can be found in the Future Work section.

The other decision regarding what data to include dealt with the different classifications within the college football system: Division I—which is split into the Football Bowl Subdivision (FBS) and Football Championship Subdivision (FCS)— as well as Division II and Division III. Our biggest concern was that the effect of certain stats might be different for teams outside the FBS, especially when the game includes teams from two different divisions. Additionally, some team stats would likely be more extreme for games in which two teams from different divisions play each other since there would be a more significant skill gap. Therefore, we considered focusing solely on FBS games but ultimately decided to include games from all divisions to give us more data to work with.

As part of our effort to account for any given team's change in strength from season to season, we decided to only use data from 2022 to 2024 for all three divisions. This gave us just over 4,500 instances of games with team stats for both teams to work with.

2.2 Data Instances

Our initial data set had 4,528 instances with 85 features each. Each instance had: game ID, year, week, and team features. For each team it contained school team ID, school team, conference, if they are home team, total points (used for testing), completion attempts, defensive TD, first downs, fourth down efficiencies, fumbles lost, fumbles recovered, interception TDs, interception yards, total interceptions, kick return TDs, kick return yards, number of kick returns, points from kicking, net passing yards, passed deflected, passes intercepted, passing TDs, possession time, punt return TDs, punt return yards, punt returns, QB hurries, rushing attempts, rushing TDs, rushing yards, sacks, tackles, tackles for loss, third down efficiencies, total fumbles, total penalties with their yardage, total yards, turnovers, yards per pass, and yards per rush. We used the points scored by each team as the target for our models.

2.3 Selected Models

The goal of predicting college football game scores based on team stats alone is a problem that is difficult to examine. As a result, we decided to test multiple different learning algorithms. Our goal for this was to see which model could produce the best predictions. Then, using the best model from each algorithm, we would combine them in an ensemble. We wanted to do this so that the strengths of each model could compensate for the weaknesses of another model.

We chose to test multi-layered perceptron (MLPRegressor), decision tree, k-nearest neighbors (kNN), and random forest models. Additionally, we combined these into ensemble models using stacking and voting regressors, both of which can be found in scikit-learn, a Python package.

To gauge the effectiveness of each model, we decided to focus on the mean absolute error (MAE) of each model. In this case, the MAE tells us the average difference between the model's prediction and the actual score of each team. Thus, the lower the MAE, the more accurate each model is.

3 Initial Results

Initially, we tested each of the non-ensemble models on a random 80% split of all the data, leaving the rest for testing our MAE. We initially used most of the original 85 features of the data as input, with each team's score as the two target outputs. We expected this to give us somewhat of a baseline before performing feature selection and other improvements to each model.

3.1 MLP Regressor

Initially, the MLPRegressor had an average MAE of 5.091. This was calculated by training 10 different MLPRegressors using the default parameters for the scikit-learn model and averaging the MAE for each. This means that on average, these models predicted the score within around five points of each team's actual score.

3.2 Decision Tree

With the decision tree's default parameters, the model was heavily overfitted. The training MAE was 0 because there were no overfit avoidance measures. As a result, the test MAE was very high—8.48.

3.3 K-Nearest Neighbors

Using default parameters, the KNeighborsRegressor achieved an average test MAE of 8.578. This was calculated by averaging the test MAE scores computed from training a default KNeighborsRegressor using n-fold cross-validation, with n being set to 5.

4 Improvements

4.1 Feature Improvements

Within the initial dataset were many features that were unnecessary for our models. The main features we needed to get rid of were anything to do with touchdowns, field goals, and safeties. It was vital to remove these because otherwise, the models could simply learn to count these, apply the proper points for each, and get 100% accuracy. Our goal wasn't to have it count the number of touchdowns but to predict the score based on the other stats. Using a linear regressor and the function SelectKBest from scikit-learn, we determined the worst features and removed them. These features include things such as team ID, the week the game was played, the year the game was played, and game ID since none of them would aid in predicting scores. However, removing these features only slightly decreased the MAE.

After removing unnecessary features, we further adjusted the data in several ways. We started by one-hot encoding the features that weren't numbers. We chose one-hot encoding so that each team would have equal importance and could be accounted for by the model. For example, if a certain team is more likely than another to score based on the same stats, the model could discover that. We also converted all stats that consisted of two numbers (such as passing attempts/completions) into ratios. Additionally, we converted the time of possession to seconds.

Once all the data was adjusted, we used a MinMax scaler to normalize the data between 0 and 1, making the minimum of a given column 0 and the max 1. We did this to prevent features such as time of possession (which was generally a relatively large number) from overshadowing other features that could be more important.

4.2 Model Improvements

Decision Tree Improvements

Since the MAE for the training set is so low and the error for the test set is high, we could reasonably conclude that the tree was overfitting to the training data. To help with this, there are a few different hyperparameters that can be changed to avoid overfitting. The first one we tested was the min_sample_leaf. This determines the minimum number of samples required to be at a leaf node, so leaf nodes have a variety of samples to average against. As can be seen in

Figure 1, the test MAE greatly dropped at first before leveling out when this hyperparameter reached around 25. Each different x value was tested 5 times, with the average being shown. Seeing this, we decided that the minimum samples in each leaf should be 25, which resulted in the lowest error of 6.84. Further testing showed that having the max depth set to 9 further reduced the error on average.

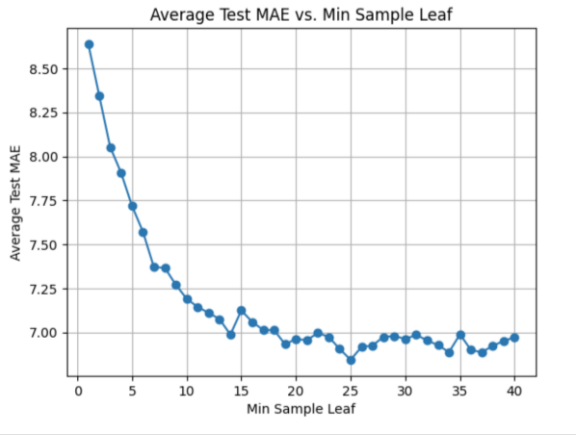

Figure 1: Decision Tree performance based on different minimum sample leaf parameters.

In addition to a Decision Tree model, we decided to try a Random Forest Regressor. Using randomized search, we determined the best random forest model was composed of 200 estimators. Adjusting other hyperparameters to reduce overfit of the individual trees made the error worse. We theorize this is because each tree model is slightly different, overfitting on different features. Therefore, combining them all decreases the overall test error. In the end, the best random forest model obtained an MAE of 5.58, which was a reduction of 1.26 points compared to the Decision Tree.

MLP Regressor Improvements

With an initial MAE of 5.091, we decided to change hyperparameters to see if we could get a better result. We used the scikit-learn function GridSearch, which allows you to enter a grid of hyperparameters, from which it will test a given model using every possible combination. Upon doing this, it determined the best hyperparameter combination to be the following: an alpha (L2 regularization term) of .001, one hidden layer with 100 nodes, a constant learning rate, and an initial learning rate of .01.

After training a new MLPRegressor using the hyperparameters from GridSearch, the model obtained an MAE of 5.085, which was slightly better than the initial approach. After achieving this MAE with the new hyperparameters, we tested different numbers of max iterations. We did this separately due to the large amount of time it would take if we included both of them in the first GridSearch. When testing for the optimal max iterations, we searched in 500 iteration increments from 0 to 5,000, except for testing 100 instead of 0: [100, 500, 1000, … 4500, 5000]. During this process, we tested both the initial MLPRegressor and our best MLPRegressor. In doing so, we found that the best max iterations for each of them were 2,500 and 4,000, respectively. Upon running these models with this hyperparameter updated for each, the base MLPRegressor had an MAE of 5.061, while the best MLPRegressor had an MAE of 5.01.

After obtaining these updated results, we decided to do a semi-ensemble for the MLPRegressors by averaging the results over 10 MLPRegressors with the same hyperparameters. The base model had an average MAE of 5.09, and the best model had an average MAE of 5.08. However, as you can see in Figure 2, all the other metrics were higher for the best model.

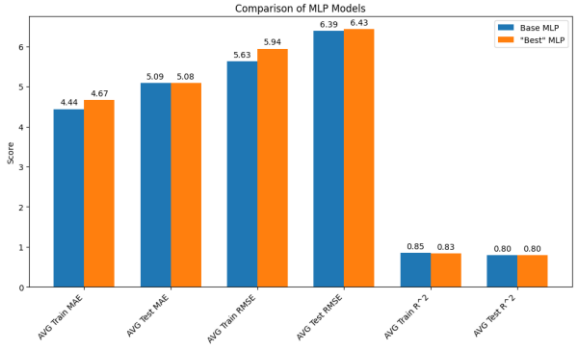

Figure 2: Comparison of base MLP and best MLP

This discrepancy shows that while minimizing error may be the main goal, it's important to take everything else into account as well. Something positive about this graphic is that it highlights a lack of overfitting using these hyperparameters.

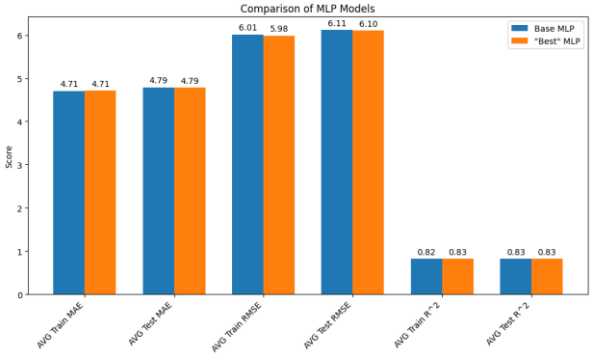

Figure 3: Metrics for base MLP with 2500 iterations and updated best MLP

After obtaining these results, we decided to do more testing on max iterations. Upon further testing, we found that on average, 3,000 iterations were ideal for the best MLP, while 2,500 iterations were ideal for the base MLP. After training each with these hyperparameters, they seemed extremely similar, with both models having an average of 4.79 test MAE. As you can see in Figure 3, the other metrics

were also almost identical, unlike the previous renditions. In this case, it would be best to use the base MLP with 2500 iterations as it would take less time to train.

The updated best MLPRegressor also had an average of 90.3% accuracy in predicting which team won any given game.

K-Nearest Neighbors Improvements

As stated in section 3.1, the test MAE for the default KNeighborsRegressor was 8.578. To increase the model's performance, a variety of different hyperparameters were tested in an iterative process until an optimal combination was found.

The first thing tested was the normalization approach used on the data. While not technically a hyperparameter of the model, the technique used to normalize the input data can lead to drastically different results and is important to test. MinMaxScaler, StandardScaler, RobustScaler, and MaxAbsScaler from scikit-learn were all tested to see the difference in results for the KNeighborsRegressor test MAE. The StandardScaler led to the best improvement, resulting in a test MAE of 6.676, a large improvement from the default of 8.578.

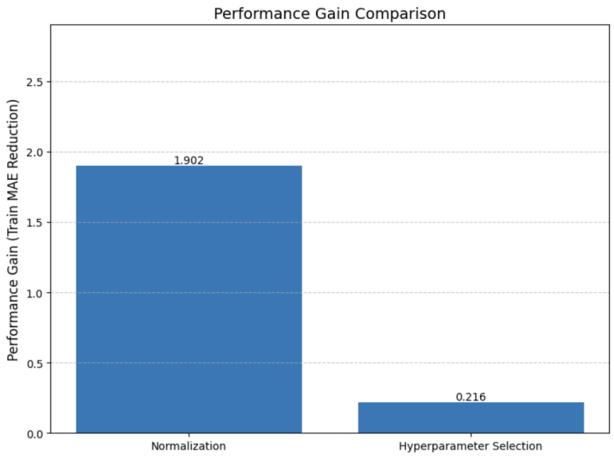

Figure 4: Comparison of Train MAE reduction achieved from Normalization and Hyperparameter Selection

Once the normalization approach of using StandardScalar had been determined, it then became important to test different hyperparameters. Using GridSearch, a variety of different hyperparameter combinations were tested, which are too exhaustive to list in full here. Upon completion of GridSearch, it became apparent that two parameters, n_neighbors and weights, were of the greatest importance. Other hyperparameters tested, such as algorithm or leaf_size, did not lead to a noticeable differentiation in test MAE scores. The models with the best MAE scores had the weights parameter set to "distance" and the n_neighbors parameter set at the highest end of the grid search, 15. Additional testing of n_neighbors beyond the GridSearch was done until it was found that a value of 16 was optimal for n_neighbors—any larger and the test MAE started increasing. It is of note that the majority of the test MAE improvements were found by normalization of the input data, not hyperparameter selection, as shown in Figure 4. Ultimately, the optimal KNeighborsRegressor achieved a test MAE of 6.459. It was trained with data normalized using StandarScaler() and had the hyperparameter values of 16 for n_neighbors and "distance" for weights.

5 Final Results

For our final model, we trained an ensemble using the best model from each of the algorithms mentioned above. We hoped that by training an ensemble using our best models thus far, we could take the strengths and eliminate the weaknesses from each model. We trained one voting ensemble as well as several stacked ensembles, each with a different final estimator.

The voting ensemble performed worse than all the stacked ensembles (with a test MAE of 5.19), other than the one that used a random forest as the final estimator. Therefore, we decided to focus our attention on the stacked ensembles.

To find the best stacked ensemble, we tried several different final estimators: random forest, gradient boosting, linear regression, ridge regression, lasso regression, and elastic net regression. We wanted to use simpler models as final estimators to prevent overfitting. This proved to work well, seeing as the stacked ensemble with linear regression as the final estimator performed the best, yielding a test MAE of 4.93.

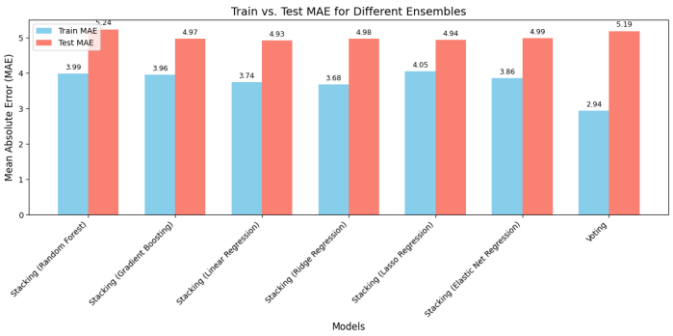

Figure 5: Train/Test MAE for ensembles

Ultimately, our best model ended up being the multi-layer perceptron, with a test MAE of 4.71. Additionally, when approaching the problem from a classification standpoint of predicting the winner of a game, our best model achieved 90.3% accuracy.

6 Conclusions

In conclusion, our best model was able to predict the score of a college football game with an average error of 4.71 points for each of the true scores. When using our model to predict the winner of a game, an accuracy of 90.3% was achieved. This is a major success and much better than the baseline accuracy of 50%, which is what the model would obtain if it were to randomly guess the scores. While the problem has not been completely solved, this paper shows

that it is possible to predict college football scores using team game stats with fairly high accuracy.

7 Future Work

Given more time and resources, we believe it would be beneficial to focus on two main areas for future work: predicting game scores based on average team stats up to that point in the season and accounting for differences in team strength across seasons.

7.1 Incorporating Average Team Stats

Under our current framework, the model requires game stats as input, which means they either have to be put in after the game (which defeats the purpose of prediction but gives insight into which stats were most influential) or predicted before the game (which could be accomplished by using other regression models or inputting the season average stats). By incorporating a team's average stats as part of our input, it would take into account how the team has been performing so the model has a baseline to work with. The average stats could then be supplemented by predictions for the game stats, which would most likely help the model do better with particularly noisy instances in which a team performed much better or worse than expected based on the game stats.

7.2 Accounting for Differences in Team Strength Across Seasons

Additionally, given the dynamic nature of college football, accounting for a team's changes in strength from season to season is paramount, particularly for team-specific predictions. For example, BYU's football team in 2023 had a 5-7 record but the very next season went 10-2, showing the extreme change in team strength from consecutive seasons. By accounting for these changes, it would be easier to incorporate past seasons' data without poorly informing the model when trying to make team-specific predictions. One possible way this could be implemented is by taking the absolute value of the difference in win percentages between the current season and any given past season. That value would then be subtracted from 1 and used as a weight for stats from that season. This way, past seasons that have a very similar win percentage to the current season would be given more weight than those in which the win percentage is quite different from the current season.