Predicting NFL Offensive Plays

Hasan Khan
hakhan6@wisc.edu

Rohit Menon
rvmenon@wisc.edu

Joey Robers
jrrobers@wisc.edu

Abstract

The role of analytics in sports has no doubt changed tremendously in recent years. Deep analytics in sports arguably first picked up traction back in 2002, when Oakland A's general manager Billy Beane hired a team of analysts to assist their old-school player acquisitions team [1]. Their strategy of using analytics to sign on undervalued players resulted in a surprisingly successful team of misfits. The 2002 Oakland A's famously won 20 consecutive games and finished first in the American League West Division - an incredible feat, despite having the third lowest payroll in the entire MLB. Although the use of analytics in baseball was novel then, it seemed to be only a matter of time before academics made their way into strategy. Anyone who has seen the back of a baseball card can tell you that the game is run by numbers: batting averages, on-base percentages, slugging percentages, and countless other metrics have been meticulously kept track of for decades. Analytics for strategy in other sports have not been as relied upon as baseball, but that certainly does not render them useless. There have even been applications to soccer, a sport where the low number of scores per game might seem to undermine the usefulness of analytics in player acquisitions. However, American Football is, by far, the most popular sport in the United States, and it has had a share of analytics in strategy. Using machine learning to predict what play the opposing team may run could be invaluable in choosing the type of defense necessary to stop them. This could improve teams' performance and help them win more often. The whole goal of this project is to find out what type of plays offenses mainly run and to eventually predict the play types teams run on offense to not only assist defenses but help sports bettors and analytical teams. Using analytics could revolutionize the game because teams will be able to use hard data to make educated decisions in regards to drafting, play calling, and overall team chemistry.

1. Introduction

American football consists of two teams trying to score either touchdowns or field goals to gain points. A touchdown is six points and a field goal is three points. Each team has four downs to go ten yards. If they get past those ten yards, the number of downs resets back to one. Each team will drive down the football field either trying to score or kick a field goal. In football, the ideal circumstances would be to keep scoring touchdowns in the minimal amount of plays on a drive to not tire out the offense too much. The purpose of this project is to help viewers better predict and classify what play types would be called on any given down during a drive based on yards to go, play direction, and other variables with a machine learning model based on historical play by play data. By predicting these plays, it can help the defensive team's strategy when playing against teams with certain types of offensive plays (an example includes plays that run more than throw and vise versa) Viewers will be able to predict what play type will be called during any part of the game.This machine learning algorithm can assist in sports betting or any sports analytics. The dataset that we used for this project was obtained from a website with play by play data for every year, but we are only basing our analysis on the year of 2020. Doing this will allow us to make better predictions and classification which can be used in future years as well as past years.

2. Related Work

With the rise of sports analytics in many franchises in several different sports, teams are able to make more data driven decisions regarding players, strategies, etc. In 2014, the NFL, specifically, developed Next Gen Stats technology and began embedding RFID tags in shoulder pads tracking all players in all venues with the help of Zebra Technologies and Wilson Sporting Goods. In 2017, the NFL teamed up with AWS to venture into advanced analytics with the help of AWS machine learning technologies and all football teams were tagged for the first time. By 2020, new machine learning algorithms were being run in AWS to help predict expected rush yards, win probability and receiver route detection [2]. The main goal the NFL had getting into more analytics was to enhance the fan's experience at the games or at home. The more information and data the viewers receive, the better they will be able to understand the game and stay intrigued. This motivated us to also look at NFL

data to see if we could also use machine learning methods and models to come up with predictions on other aspects of the game.

3. Proposed Method

3.1. Decision Tree

In trying to predict the offensive play, we used many different strategies and algorithms. The first algorithm tried was the decision tree algorithm, splitting according to entropy. The decision tree was a natural first step, since it is the simplest and most intuitive algorithm. Also, most of the other algorithms used are based on the decision tree, so a good understanding of the decision tree will be important for understanding the rest of the analysis described in this paper. Intuitively, think of entropy as a measure for a subset of data that represents the heterogeneity of the class labels: homogenous class labels produce an entropy of 0, and perfectly mixed class labels produce an entropy of 1. Essentially, a decision tree will create subsequent boundaries which minimize the entropy and therefore maximize the information gain. Although we expect to get better results from other algorithms, sometimes simple algorithms like decision trees perform surprisingly well. Also, since it is easy to interpret the decision tree boundary, we use a simple two-feature (Down and Yards To Go) decision tree to check to see if the boundary matches our intuition of predicting the play type. Following this, we fit a tree with more features to try and get better prediction accuracy.

3.2. K-Nearest Neighbors

Another simple model used in the analysis is the k-nearest neighbors algorithm. Essentially, the algorithm calculates the distance between the feature columns of the row to predict between each of the feature columns of the training rows. After this, it determines which training observations are closest to the feature observations to be predicted. Using the class labels of its closest neighbors, the algorithm predicts the unknown class label like a majority vote. For example, if k is set to equal 5, then the algorithm predicts the class label of the unknown observation according to the most prevalent class label in the 5 nearest neighbors to the unknown observation. Similar to the analysis done with the decision tree, we started with a simple two-feature (Down and Yards To Go) algorithm to plot and check to see if the decision boundary matches our intuition. After this, we applied the algorithm to all of the features to hopefully create more robust predictions. One thing to note is that, since this classifier relies on euclidean distance, it is important to normalize the feature columns.

3.3. Random Forest

The second classification model that we tried was the random forest. Random forests make up the average of decision trees tested on varying portions of the training dataset to attempt to reduce the overall variance. When utilizing this algorithm in our project, we kept the default criterion as "gini" and chose to vary the maximum depth of the decision trees that are part of the overall random forest. Because greater depths allow for more precise accuracy through increased splitting of nodes in the decision tree, also known as feature randomness, we chose to test various maximum depths on the training and validation sets of our data with use of a for loop.

An important aspect of the random forest classifier is how it uses bagging and the idea of feature randomness to grow and develop unique decision trees. Bagging is utilized in a random forest through random sampling with replacement. This is further discussed in the section on the bagging algorithm below. Additionally, as briefly mentioned above, feature randomness is achieved through random splitting of nodes in decision trees to design unique decision trees with varying maximum depths. Through both of these elements, each decision tree is built to be different from the next as to create an accurate resulting score for the classifier.

3.4. Bagging

The third algorithm tried was the bagging classifier algorithm, splitting according to entropy. Bagging is the ensemble learning method that is used to help reduce variance within a decision tree classifier and noisy dataset. This algorithm selects data from the training set with replacement and uses bootstrapping sampling techniques to reduce the variance of the estimate from the bagging classifier thus increasing its bias. Bagging leverages the bootstrapping sampling techniques to create unique samples by generating various subsets of the training dataset by selecting data points with replacement and randomly. "With replacement" means that one can select the same instance multiple times. Next, the algorithm is then trained independently using weak or base learners. Finally, since in our project, we are looking for classification, a majority of the predictions from the estimators are taken to compute a more accurate estimate and then we look at the highest majority breakdown of votes to determine which classes are accepted.

3.5. Adaptive Boosting

One of the more complex algorithms used in this modelling was Adaptive Boosting. Adaptive Boosting is an ensemble method used with a weak learner as a base algorithm. Generally - and in the case of this paper - the base weak learner is a decision tree with a low maximum depth. The general Adaptive Boosting algorithm is as follows: first, fit the weak learner with uniform weights on

each observation in the training set; second, weight the observation with incorrect predictions higher; third, fit the model again; fourth, repeat the second and third steps many times to attain many weak-learning models. The final predictions of the Adaptive Boosting algorithm are a majority vote among the many weak-learning models fitted.

3.6. Stacking

The last algorithm used to attempt to predict PlayType was a stacking classifier. Stacking can be thought of as a more complicated majority voting ensemble algorithm. As a whole, stacking is a very adaptable algorithm, but it also can be prone to overfitting. Essentially, the data scientist chooses multiple algorithms, each which predicts the outcome of each observation. Then, a meta-classifier is chosen to predict the class label of each observation based on the predictions of the original classifiers. The stacking algorithm performs well when there are hidden trends in the way that the original classifiers predict, or when significantly more information is obtained by looking at multiple classifiers at once instead of independently. In this case, the original classifiers used were a random forest, a decision tree, a k-nearest neighbor algorithm, and an adaptive boosting algorithm. The meta-classifier used was logistic regression.

4. Experiments

The data we collected was from NFL Savant (http://nflsavant.com/about.php), which is a data science project geared toward the NFL. The site has play by play data for NFL seasons all the way back to 2013. Each observation in the dataset corresponds to a play in an NFL game. Each play has 41 features Since the most recent year of data is theoretically the most relevant to learn from, this project uses the dataset corresponding to the latest full regular season available - 2020. Since there are many plays per NFL game and many games per NFL season, the dataset is not small: before cleaning, the data set had over 46,000 observations. However, not all of these observations are essential to the central question of this paper: will the opposing team run the ball or pass the ball? This question focuses on the 'normal' plays in football: the algorithm will specialize in predicting the play type of drives before scoring or punting. Said another way, this approach assumes that extra-point and two-point conversions are a separate issue, not to be grouped with the drives before scoring. Also, a more or less arguable assumption this approach makes is the deterministic nature of punting. This approach assumes that it is perfectly predictable whether or not a team is going to punt on fourth down. While this might not be true 1 percent of the time or so, this assumption does simplify the analysis, and we found it in our best interest to move forward with it.

Given these choices, our dataset was filtered to exclude extra-points, two-point conversions, punts, timeouts, and penalties. This cut the dataset down to 32,612 observations. After filtering the rows, filtering the columns was necessary to streamline the training and testing process. For columns, we kept only the rows that could help us predict the play-type, without using any future information. In order for this project to be useful in actual football scenarios, it is important that the model gets only the information available to a coach or player in a real world scenario before predicting the play type. For example, it would be cheating for the model to take into account how many yards the play gained. Even though it is in the same row as the play type, it is after the play occurs that a coach or defensive player would know this information. Thus, it should not be available to our model. Keeping this in mind, the main predictors used for the models are as follows: Down - how many plays a team has run without getting a first down yet; ToGo - how many yards is necessary for a first down; YardLine - how close on the field a team is to scoring; Time Left - a column we created to represent the amount of seconds left in the half; Formation - a string representing how the offensive team has lined up. The Formation column was one-hot encoded to represent which one of the four formations the team was in. All of these columns would be known by a coach or player before the ball is snapped, and is fair game for the models to use for prediction.

In the spirit of the modelling being fairly assessed against a coach or player in a real-world situation, the train-test split might be out of the ordinary. Instead of randomly splitting the train and test data or using stratas, we split the train and test data by date. Since, in the real world, this tool would be used to predict future play types from past information, it might be misleading to report testing accuracy any other way. For example, if there is some large trend towards passing more in the league as time goes on, our model would not be able to take that into account during actual use, so it should not be able to take it into account here either. Said another way, it seems dishonest to train our models on an entire season of football, when in the real world it can only be trained on a part of a season to predict the rest of the season. So, the last two weeks of the NFL season were reserved for testing: 4,046 rows. Our train and validation set were then randomly split 75% and 25%, respectively, making 21,424 training examples and 7,142 validation examples. We decided on the holdout method because we have such a large dataset, and cross validation would have been computationally long and expensive. We used the training set to train our data, then evaluated its performance on the validation set and tuned the model accordingly. Finally, we train the best models on the training and validation data and test its performance on the testing data. This should give a reasonable estimate to how well our models would work in

the real world.

4.1. Software and Hardare

While we did not use any specific hardware in this project, all three of us used the same software to undergo this project. First, we collectively decided that Python would be the best and most realistic language to use for this project: many of the packages that we used in this project have been given thorough demonstrations in lecture. If we were still feeling unsure, we were able to quickly read up on the documentation of various commands in these packages to continue onward. As a result, we all felt confident in our abilities to utilize the machine learning algorithms in relation to our project with Python.

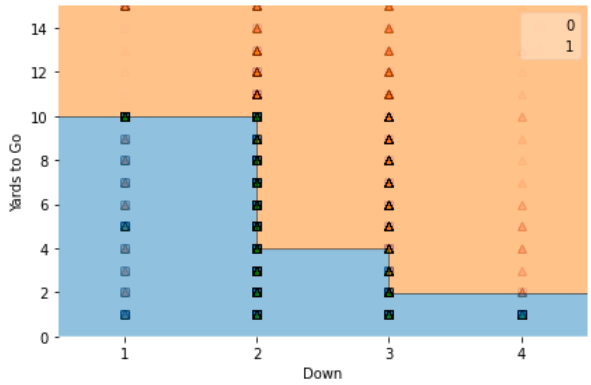
With regards to Python packages, our data analysis for this project was centered around three specific packages: pandas, sklearn, and matplotlib. We used pandas to effectively read in the original dataset and filter out specific columns through boolean conditional statements. Next, we used sklearn because it comes with many machine learning algorithms that were relevant to this project. As discussed above, we created machine learning models based on the decision tree, random forests, k-nearest neighbors, bagging, adaptive boosting, and stacking classification algorithms. Finally, we used matplotlib to create visually appealing and easy to understand figures that best illustrated the results of our data analysis.

5. Results and Discussion

In order to establish a bottom line for the performance of the models' predictions, it is useful to examine that about 61% of the observed data ended in a play of type pass. That means that if our models predict a pass every time, it will be correct 61% of the time. That number provides a benchmark for the analysis. However, it is difficult to define what a successful prediction accuracy would be: it is hard to tell how NFL coaches do predicting plays without machine learning models. Also, we expect attaining a high prediction accuracy will be difficult, since the idea of choosing plays for the offense is to be as unpredictable as possible.

5.1. Decision Tree

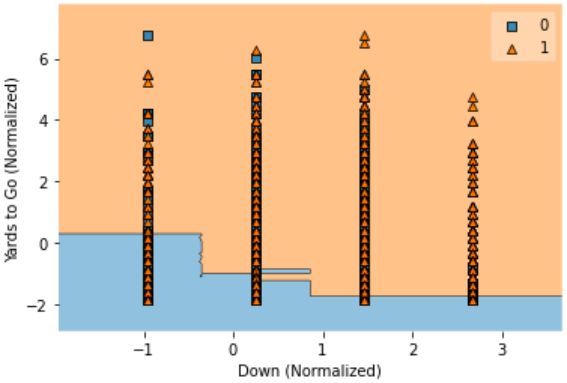
The first decision tree we fit was a simple, two-feature tree with down and yards to go. This was to mainly check that the decision boundary matches our intuition. The following figure shows the results of the decision tree. Orange corresponds with Pass and blue Rush.



The figure does indeed match our intuition: the more yards to go, the more likely a team is to pass the ball. Also as the down increases, this means the offense has less chances to make a first down, and they are therefore more likely to pass the ball. Tuning this model boiled down to finding the ideal maximum depth, which was determined to be 6, using the validation set. This model reported a testing accuracy of 66.09% - a bit better than the baseline. The second decision tree was fit with all of the features, and the same maximum depth of 6. As expected, this one had a much better testing accuracy: 73.38%.

5.2. K Nearest Neighbor Classifier

Before tuning and training a nearest neighbor classifier, it is important to standardize the columns first. For this, we used Sci-kit Learn's StandardScalar. Similar to the decision tree analysis, we started with a simple two-feature k-nearest neighbor classifier. After tuning with a validation set, it was determined that the optimal k is 90. The figure below shows the results of the 90-nearest neighbor model.

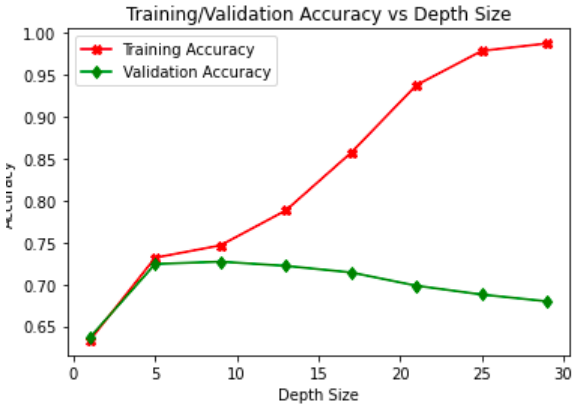


Since k-nearest neighbor relies on standardized columns, the graph is a bit more difficult to interpret, but we can see that the boundary is similar in shape to the decision tree boundary. The testing accuracy for this model is 65.52%, a bit worse than the simple decision tree. In an attempt to make a more robust nearest neighbor estimator, we included all of the features to predict the play type. After tuning, the optimum k was found to be 100. As expected, this clas-

sifier performed better on the test set, with an accuracy of 72.91%.

5.3. Random Forest

To create the random forest classifier model, we first used the Ensemble library from sk-learn to import the RandomForestClassifier. When considering how to properly tune the hyperparameters of this algorithm, we decided that determining the ideal number of splits (max depth) would be most advantageous. In doing so, we created a list of numbers ranging from 1 to 32 with a step size of four. We then iterated through this list of numbers and fit each random forest model with the corresponding maximum depth of the current number in the list. After this, we decided to compare the training and validation set accuracy with each increase in depth so we created two dictionaries: the keys for both of these dictionaries were the numbers for tuning the maximum depth that we iterated through the list, but the values were for the corresponding training and validation set accuracy. After creating these two dictionaries, we decided to plot the relationship of maximum forest depth against the training and validation set accuracy.



Upon graphing these features, we immediately noticed that as the maximum depth of the random forest increased, the training accuracy increased. However, this was not the case for the validation accuracy. While the validation accuracy began to increase as the depth size for the forest increased, it eventually began to fall at a depth of 13 and continued to fall as the depth size increased. In doing so, we concluded that the random forest is vulnerable to overfitting and that the proper depth size for this model would be 13. Consequently, we noted that the test accuracy at a depth of 13 was about 72.89%, which was slightly less than the k-nearest-neighbor, stacking, bagging, and adaboosting classifiers.

5.4. Bagging

For the Bagging classifier, we used Ski-kit's Learn's ensemble library to import the BaggingClassifier. Before getting into the classifier, I had to create a DecisionTree. The criterion used for it was entropy which is basically a measure of the disorder of the features within the target. The random state stayed at 1 and for max depth, I played around with the numbers and realized as I increased it, it was getting better accuracy, however, the testing and training accuracy didn't change after 5 so I stuck with 6 for the most optimal max depth. Again, like most of the other classifiers, we stuck with the holdout method and found that it provided a testing accuracy of only 73.38% which was a tad better than k-nearest neighbor and a little worse than the stacking classifier.

5.5. Adaptive Boosting

For the Adaptive Boosting classifier, we used Ski-kit Learn's AdaBoostClassifier. We attempted to use the bootstrap_point632_score function from mlxtend in tuning, but the sheer amount of data proved to computationally take too much time, so we resorted back to the holdout method. Tuning with the validation set, we found the optimal parameters to be a base tree with a max depth of 3 and a learning rate of 0.06. Although this was the algorithm we were most confident in, it only provided a testing accuracy of 73.08%, barely better than the k-nearest neighbor classifier and a bit worse than the decision tree with maximum depth of 6.

5.6. Stacking

The final algorithm trained and tested was the stacking classifier. Since stacking is so open-ended and prone to overfitting, this classifier was very difficult to tune. Also, since there is so much data, performing cross validation would be expensive and time consuming. As a result, a limited amount of hyperparameters were tried in tuning this classifier. The final testing accuracy was 73.41%, just barely better than the decision tree with a maximum depth of 6, and probably a result of noise in the data.

5.7. Shortcomings and Future Considerations

A shortcoming in predicting playtype using these methods is the lack of features we had access to. A coach in real time would have access to more information such as the talent of the players at each position, if there are any injuries in the current game, the opposing coach's style, and much more information either difficult to quantify or difficult to efficiently collect data on. If there were better ways to quantify or efficiently collect the features we did not have access to, the performance of the model would likely be significantly better. Otherwise, using the model as an aide rather than the conclusive deciding factor would be classified as being properly used.

In the future, we could consider looking at a team's win-loss ratio and the current score of the game. For instance, if a team that has a significant lead on their opposition is known for being overtaken at the last minute, it would be

interesting to see if our model could be adjusted to predict the plays that caused this. This would especially be fascinating if the team that had a major comeback had a lower win-loss ratio than their opponent because our model would then be able to predict the plays that led to an upset.

6. Conclusions

The best testing accuracy we received from our classifier algorithms was 73.41% whereas our baseline accuracy was 61%. This increase in accuracy shows an improvement in our classification model. While the initial decision tree, the simplest classification algorithm, came out with a test accuracy of approximately 65%, adjusting the maximum depth to be 6 increased this to about 73%. This accuracy is comparable to the random forest, bagging, and k-nearest-neighbors, which proves that suitable tuning of hyperparameters can lead simple models to perform up to the standards of the advanced models such as adaboosting and stacking.

Because there has been limited work done regarding machine learning models with respect to NFL play by play data, we found it to be difficult to properly relate our testing accuracies with previous work. However, seeing as our testing accuracy improved over the baseline accuracy, we have clearly manifested the idea that machine learning can be applied to historical sports analytical data.

7. Acknowledgements

We would like to express our gratitude towards the authors of the website NFL Savant that we obtained our dataset from. This website provided us with invaluable information that we used in our data analysis.

8. Contributions

At meetings, all members contributed to the preprocessing steps, and formed a general outline of where the project would be headed. We also did the train-validation-test split together as a team

Joseph trained and tuned the decision tree, k-nearest neighbor, adaptive boosting, and stacking algorithms. He wrote the sections in the proposed methods and the results/discussions that pertained to these algorithms. He also wrote the experiments section as well as the shortcomings and future considerations sections.

Rohit trained and tuned the bagging classifier algorithm. He also wrote the sections in the Abstract and Introduction as well as sections within Proposed methods, results/discussions, and related work.

Hasan trained and turned the random forest classifier algorithm. He wrote the portion in the proposed methods and the results and discussion section that pertained to this algorithm. He also worked on the conclusion section, acknowledgements section, software and hardware sections in the Experiments section, and did an overall review of the report to maintain clarity and consistency in all sections.

References

[1] “The Real Story Behind ‘Moneyball’”.
URL: https://filmschoolrejects.com/moneyball-true-story/

[2] “NFL Next Gen Stats”. URL: https://operations.nfl.com/gameday/technology/nfl-next-gen-stats/