UDACITY

MACHINE LEARNING ENGINEER NANODEGREE

CAPSTONE PROJECT REPORT

HANNES ENGELBRECHT

13 DECEMBER 2019

1. OVERVIEW

Machine learning in sports is an ever-growing industry with more and more raw data becoming available daily. According to Bunker and Thabtah [1], machine learning has shown promise when it comes to the domains of classification and prediction in sports, where club managers and owners desire accurate machine learning models to help formulate strategies to help them win matches, thereby maximizing profits and enhancing their club's reputation.

Competitions for proposing solutions to problems in sports using machine learning common and can be found on a variety of online data science communities. This report documents the work done in the application of a machine learning model in the National Football League (NFL), as presented by Kaggle NFL Big Data Bowl [2].

2. PROBLEM STATEMENT

The Kaggle NFL Big Data Bowl competition, as set out on their website [3], reads as follows:

"American football is a complex sport. From the 22 players on the field to specific characteristics that ebb and flow throughout the game, it can be challenging to quantify the value of specific plays and actions within a play. Fundamentally, the goal of football is for the offense to run (rush) or throw (pass) the ball to gain yards, moving towards, then across, the opposing team's side of the field in order to score. And the goal of the defense is to prevent the offensive team from scoring.

…

In this competition, you will develop a model to predict how many yards a team will gain on given rushing plays as they happen."

The original goal of the challenge was to have competitors make predictions on live games *as they happen*. Seeing that the cut-off for this Kaggle data science challenge has expired and no new live game data will be available, predictions will be made on historical data as opposed to live data. This requires the provided dataset to be split into a training and testing set, the latter of which will be used to evaluate the model.

3. DATASETS AND INPUTS

A dataset has been provided by the NFL and can be found at https://www.kaggle.com/c/nfl-big-data-bowl-2020/data. This dataset contains 49 columns and roughly 510 000 rows. Each row in the dataset represents a single player's involvement in a single play for a specific game. This translates into roughly 500 'games' and 23 000 'plays'.

4. METHODOLOGY

Each of the steps involved in this project are listed below along with a brief description of what that step entailed:

i. **Obtaining the dataset** – The dataset was obtained from the source described in section 3 and uploaded to Amazon Sagemaker as a CSV file.
ii. **Exploring the data** – General summary statistics of the dataset were looked at in this section. A significant component of this section was exploratory data analysis, where visualizations were created in the *Exploratory Public Desktop* app to show the relationship between the target variable (yards) and independent variables/features. This section also looked into missing values that need to be imputed, as well as features that need some cleaning before using that data in the models.
iii. **Data preprocessing** – This step was the most time-consuming of all. This section included:
    i. Imputation of missing values
    ii. Feature 'cleaning', for example text fields with inconsistent data entries
    iii. Consolidating the data
    iv. Dropping redundant features
    v. Correlation analysis and the removal of heavily correlated features
    vi. Encoding categorical variables
    vii. Splitting the dataset into a training, testing and validation set
    viii. Feature scaling using MinMaxScaler
iv. **Creating the models** – This section focused on the following:
    i. Creating a benchmark model (simple linear regression model)
    ii. Creating an XGBoost model, which was the model of choice for this project
    iii. Fitting the models to the training data
v. **Evaluating model performance** – The XGBoost model's performance was compared to that of the benchmark model
vi. **Model refinement** – In this section an attempt was made to improve the performance of the chosen model
vii. **Summary of the results**
viii. **Limitations and recommendations**

5. EXPLORING THE DATA

General:

As mentioned in section 2, the data provided by Kaggle comprises of roughly 510 000 rows and 49 columns with size 181MB. Each row in the dataset represents the data relating to a unique combination of GameId, PlayId and NflId (the player's unique NFL identifier). The original data can be grouped into 3 sections:

i. Data relating to the game, for example where the game was played and in which season
ii. Data relating to the particular play, for example in what section of the field it took place
iii. Data relating to each player involved in a given play, for example his name and location on the field

For a full list of the features included in the dataset, please see the list provided by Kaggle [4].

Exploratory data analysis:

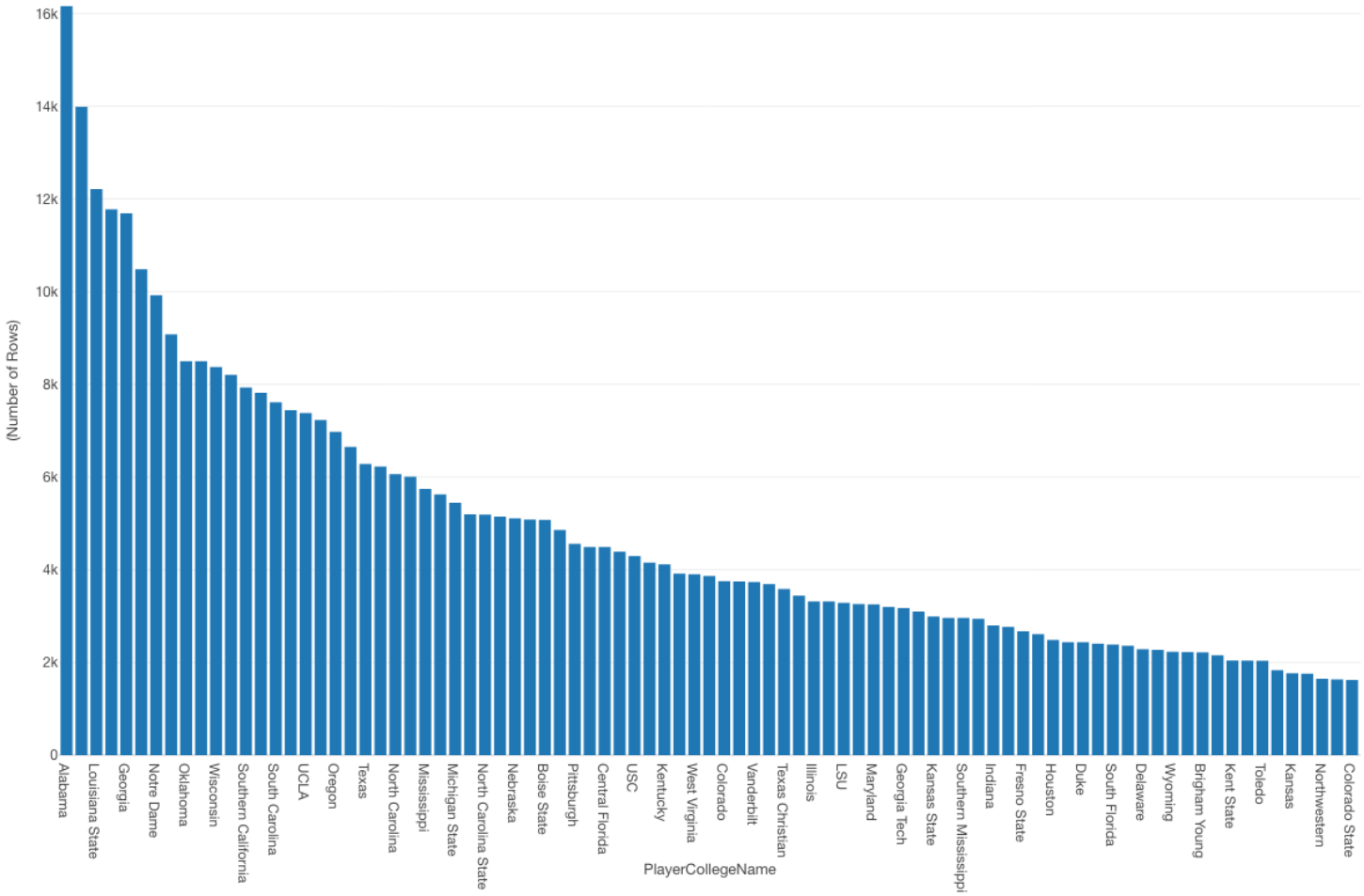
For this section, *Exploratory Public Desktop* app was used to visualize the data. *Exploratory Public* is a statistical software program that provides users with data science functionality such as data wrangling, visualization, machine learning, reporting and dashboards [5]. A link to the *Exploratory* analysis has been provided [6], where more visualizations can be viewed. Please note that in order to view the *Exploratory* project, the link provided in [6] has to be imported as a project in the *Exploratory Public Desktop app.*

It's worth reiterating here that the target variable was 'Yards', i.e. the goal was to predict the number of yards to be gained/lost in the next play. The resulting visualizations below are therefore focused on investigating the relationship between multiple independent variables and the 'Yards' variable.

- Average yards per play = 4.21 yards (across all records)

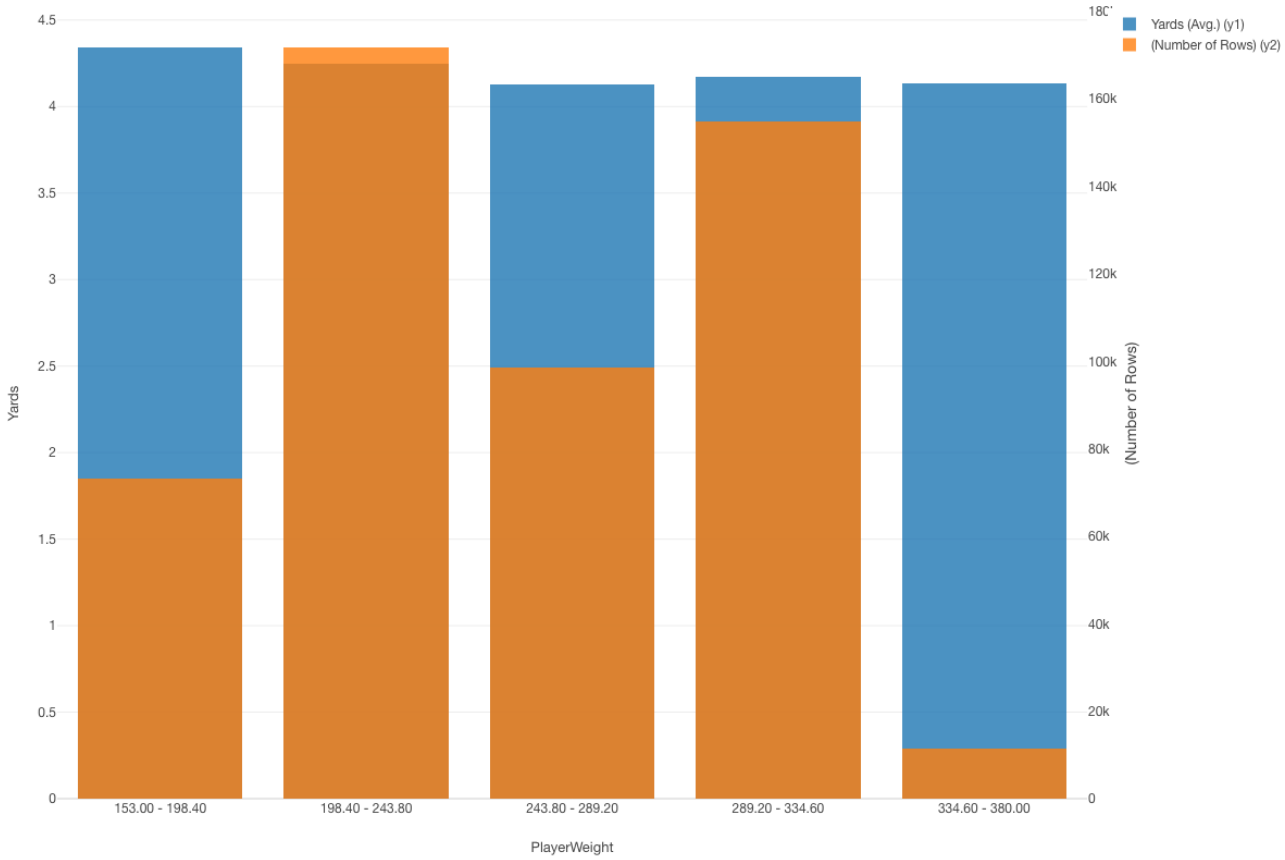
- Player names – the word cloud below depicts the player names associated with the most rows in the dataset. The larger the name displays, the more rows that name appeared in. This was investigated on the premise that when certain players are involved in a play, e.g. a celebrated quarterback, the average yardage for those plays might be slightly higher.

- Player college name – the image below provides some initial information on how many plays involved players from different colleges. Please see [6] for additional visualizations regarding the players.



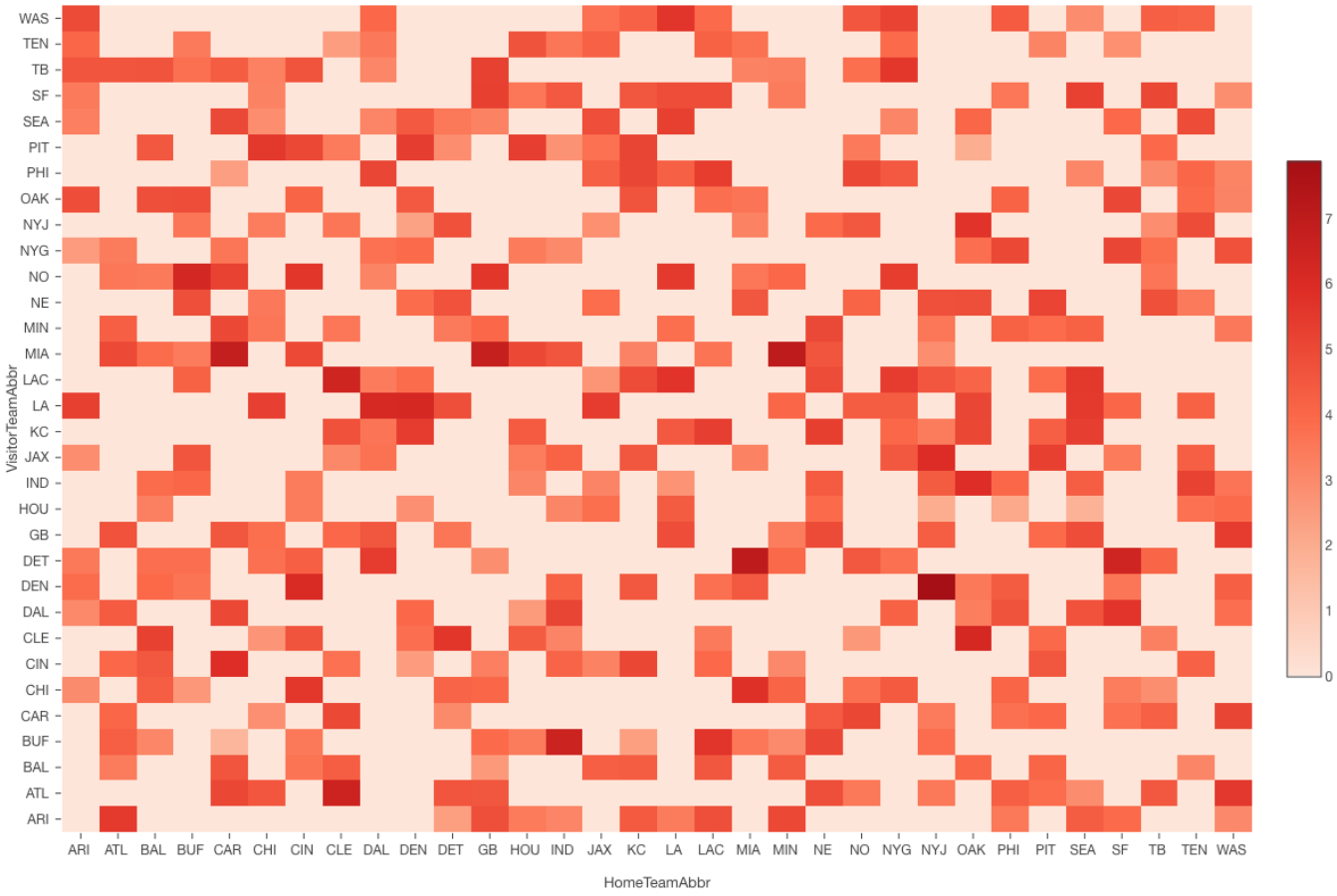
Notice how many players descend from Alabama College, Ohio State (second largest) and Louisiana State into the NFL.

- Player weight – here we look at the different weight groups within the NFL and the average yards gained/lost within that weight group. Additionally, we visualize how many rows were observed within each of the weight groups:



In the above image, the blue bars represent the average yardages per weight group, whereas the orange bars represent the total number of rows within each of the weight groups. We notice that the lighter the player involved in a play, the more likely he is to gain more yards. We also notice that most players fall within the weight range of 198.40lbs – 243.80lbs. Very few players exceed 335 lbs in weight.

- Teams – the goal of this section was to investigate pairs of opponents where the average yardage per play is particularly high or low. The heatmap below displays the average yardage per play for every combination of opponents (home team and visitor team combinations). Bright red blocks indicate high average yards per play, whereas faded colors represent plays with lower average yardages.



6. DATA PREPROCESSING

Missing values:

One of the first steps in the data preprocessing stage is measuring the extent of missing values. Below is a summary of the number of missing values per feature in the original dataset:

	Number
Orientation	18
Dir	14
FieldPosition	6424
OffenseFormation	110
DefendersInTheBox	66
StadiumType	32934
GameWeather	43648
Temperature	48532
Humidity	6160
WindSpeed	67430
WindDirection	80234

Each of the features with missing values were dealt with as follows:

- Orientation: Missing values were replaced with the mean value for Orientation (180.25)
- Dir: Missing values were replaced with the mean value for Dir (179.93)
- FieldPosition: Missing values were replaced with the mode value for FieldPosition ('BUF')
- OffenseFormation: Missing values were replaced with the mode value for OffenseFormation ('SINGLEBACK)
- DefendersInTheBox: Missing values were replaced with the mean value for DefendersInTheBox (6.94)
- StadiumType: Missing values were replaced with the mode value for StadiumType ('Outdoor')
- GameWeather: Missing values were replaced with the mode value for GameWeather ('Cloudy')
- Temperature: Missing values were replaced with the mean value for Temperature (60.44)
- Humidity: Missing values were replaced with the mean value for Humidity (55.65)
- WindSpeed: Missing values were replaced with the mode value for WindSpeed ('5')
- WindDirection: Missing values were replaced with the mode value for WindDirection ('NE')

Text preprocessing:

Another important aspect of data preprocessing is investigating alphanumeric columns and ensuring that the data entries are consistent. For example, if we look at the column StadiumType and observe some values: 'Closed Dome', 'Dome', 'Dome, closed', 'Domed', 'Domed, Open', 'Domed, closed' and 'Domed, open'. These values are all actually referring to the same stadium type and can therefore be grouped together. This ensures consistency in the data, improves model predictions and enhances the credibility of those predictions (because we will be making use of larger samples/exposures if the appropriate values are grouped together).

Text preprocessing was done for the following features:

- Stadium*
- Location*
- StadiumType
- Turf
- GameWeather
- WindSpeed

* Text preprocessing done using the *fuzzywuzzy* library, which helps calculate the similarities between each of the possible feature values and a given string. This, in turn, helps to group those feature values together that have similarities above a specified threshold.

Consolidating the data:

As mentioned earlier, each row in the original dataset represents the values for a particular GameId, PlayId and NflId. This means that each play, on which predictions will be made, has 22 rows. Many of the 49 columns therefore have the same values across a set of 22 rows (all related to the same play). However, no two rows are exactly the same since the feature values relating to specific players will be different (e.g. the player's name).

In order to achieve a yardage prediction per play, the data would need to be consolidated such that each row represents a unique play. There were two options for this:

i. Consolidate the data and use an aggregated, single observation per play to train the model on. This will produce a single yardage prediction per play (GameId-PlayId combination)
ii. Keep the data at the granular level and train the model. Predictions will be made on a GameId-PlayId-NflId level, after which 22 rows would have to be combined into a single prediction for a particular play (GameId-PlayId level)

The former of the two options above was chosen. Below are the steps that were followed to achieve this:

i. A function 'aggregation_func' was written (see the helpers.py file) that combines all 22 rows with a particular GameId-PlayId key into a single row. Each row in the training, validation and testing data therefore represents a single GameId-PlayId key. After the consolidation, two separate rows can relate to the same *game*, but no two rows can relate to the same *play*.
ii. For each observation, a distinction was made between the 'attacking' team and the 'defending' team.
iii. Many of the features retained for modelling purposes were split into two – one for the 'attacking' team and one for the 'defending' team. For example, player weights were split into two. PlayerWeight_off represents the average weight of players in the offensive team, whereas PlayerWeight_def represents the average weight of players in the defensive team for that play.
iv. Other features retained for the modelling were kept as is. For example, the stadium remains the same regardless of which team is being considered. This single value (across all original 22 rows of a play) was retained for the resulting single row that now represents the 22 rows.
v. More detail on how particular features were split/retained in the aggregation function can be found in section 3 ('Feature Engineering') of the accompanying Jupyter Notebook.

The resulting consolidated dataset contained 23171 rows and 64 columns.

Dropping redundant features:

Seeing that the consolidated dataset contained many features (64), a decision was made to reduce that number by omitting columns that were deemed to be less important for prediction purposes. The columns that were dropped include: GameId, PlayId, GameClock, TimeHandoff, TimeSnap, WindDirection, DisplayName_off and DisplayName_def.

The resulting dataset now contained 23171 rows and 56 columns.

Correlation analysis and the removal of heavily correlated features:

A correlation analysis was conducted to identify pairs of features in the consolidated dataset that had high correlations. Based on these results, additional features to be dropped from the dataset were selected. In modelling, removing highly correlated features may result in more accurate predictions and shorter runtimes.

Pandas' *corr* method was used to calculate the pairwise correlations (using the Pearson Standard Correlation Coefficient). Another function 'correlation' was written to return the list of features that had a correlation in excess of a specified threshold. This function can also be found in the helpers.py file. The threshold selected for this analysis was 0.7 and is rather subjective. The list of features dropped after the correlation analysis includes: Y_def, S_def, Dis_off and X_def.

The resulting dataset now contained 23171 rows and 52 columns.

Encoding categorical variables:

Categorical variables were encoded using Pandas get_dummies. The list of categorical variables that were encoded includes: PossessionTeam, FieldPosition, OffenseFormation, OffensePersonnel, DefensePersonnel, PlayDirection, HomeTeamAbbr, VisitorTeamAbbr, Stadium, Location, OffenseTeam, PlayerHeight_off, PlayerHeight_def, PlayerCollegeName_off, PlayerCollegeName_def, Position_off and Position_def.

After the label encoding, the resulting dataset now contained 23171 rows and 619 columns.

Splitting the dataset into a training, testing and validation set:

The preprocessed dataset was split into a training, validation and testing set. A split of 80%/20% was used in both stages of the splitting. After the split, each of the sets' dimensions were as follows:

Training: 14828 rows, 619 columns

Validation: 3708 rows, 619 columns

Testing: 4635 rows, 619 columns

A validation set was created in anticipation of hyperparameter tuning of the XGBoost model, which will be explained in a subsequent section.

Feature scaling using MinMaxScaler:

In order to fit the benchmark model (simple linear regression model), our data needed to be scaled. For our purposes, Sklearn's MinMaxScaler was used to normalize the data. Two scaler objects were created; one for scaling the features and another for scaling the target variable. The scalers were fit on the training data only (as opposed to on the full dataset), but were used to transform the training, validation and testing data. This was done in order to prevent 'data leakage', where some of the data in the validation/test sets may be used to scale the values in the training set [7].

7. MODELLING: SETTING UP THE MODELS

In this section, two models were constructed – a benchmark model for comparison purposes, and a 'model of choice' expected to outperform the benchmark model.

The benchmark model:

The benchmark model chosen for this project is a simple linear regression model using Amazon Sagemaker's LinearLearner algorithm. In order to train this model, a RecordSet needed to be created from the *scaled* training data. The model was fit to the *scaled* training data.

The 'model of choice' – XGBoost:

Our model of choice for this project was an Extreme Gradient Boosting (XGBoost) model. XGBoost and other tree-based ensemble models are 'considered best-in-class right now' for small-to-medium sized tabular/structured data [8]. Some of the advantages of XGBoost mentioned by Morde & Setty [8] include:

- Applied to a wide range of problems
- Supports multiple programming languages
- Easily integrated into the cloud
- Compatible with Windows, Linux and OS X

Another model that was considered for this project was the Artificial Neural Network (ANN). However, ANN's are known to overfit to training data in many cases if it's not explicitly taken care of (through early stopping or setting a dropout when training the ANN). Seeing that we ended up with 618 feature columns, which is a large number, it seemed more fitting to make use of an XGBoost with a slightly lower risk of overfitting to the training data. In addition, one is also able to use early stopping rounds to prevent XGBoost models from overfitting. This can be done quite easily in Amazon Sagemaker's high-level XGBoost API.

8. MODELLING: EVALUATING THE RESULTS

For this project the measure against which the two chosen models was compared is the Root Mean Square Error (RMSE). This is defined as:

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2}$$

where:

n = total number of plays on which predictions are made (4635 predictions)

y_j = actual yards gained/lost on the j'th play

$\hat{y}_j$ = predicted yards gained/lost on the j'th play

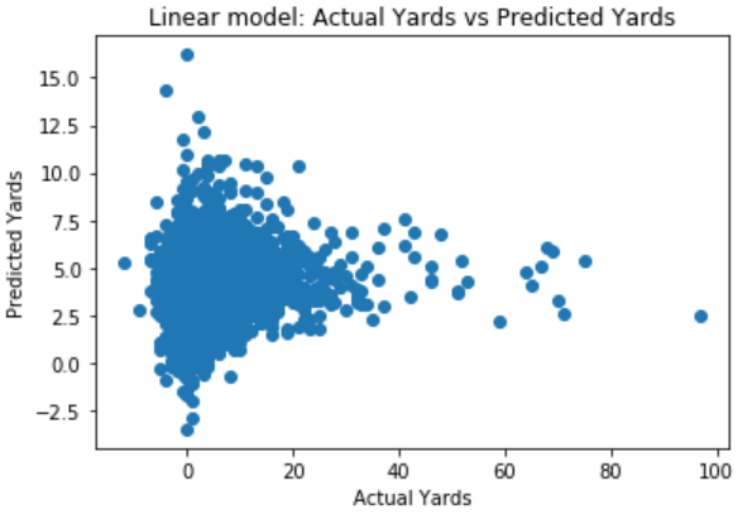
The RMSE therefore gives an indication of the extent of the model's misclassification across the whole test set, with larger values indicating poorer model performance. A function 'rmse' was created and stored in the helpers.py file, which was used to calculate the model performances.

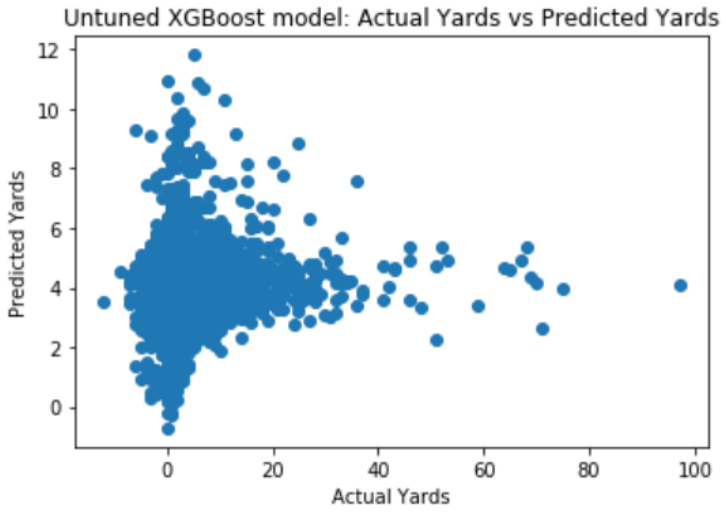
Simple linear regression:

The simple linear regression model had an RMSE value of 6.43.

XGBoost:

The XGBoost model has an RMSE value of 6.36 – a slight improvement on the benchmark model.





Another point worth noting is that the average yards gained per play in the training set (only) is 4.22 yards. If we created another simple model by simply predicting that each play in the test set were to gain the average (4.22) yards, which we'll call the 'average model', we arrive at an RMSE value of 6.41. So, using the average yards gained from the training set produces a better model than the simple linear regression model. The point here is that even though the XGBoost model outperformed the benchmark model by a small margin, it might not be a very accurate model to start with. As a result, an attempt was made to improve the XGBoost model's performance.

Additional visualizations of the model predictions against the actual observations can be seen in the accompanying Jupyter Notebook, section 7.

9. MODELLING: REFINING THE CHOSEN MODEL

A crucial tool in any machine learning engineer's arsenal should be 'hyperparameter tuning', i.e. the ability to select a very specific model based on which set of hyperparameters produce the best results on a validation set. To try and improve on the initial XGBoost model described above, a 'tuned' model was created. This model was trained on the training data and selected based on the performance on the validation data. The 'tuned' model was subsequently used to predict on the test data, after which we arrived at an RMSE score of 6.34. This is a small improvement on the initial/default XGBoost model, and we end up with an XGBoost model that outperforms the benchmark model by a slightly bigger margin.

These results may be regarded as a little surprising at first sight, seeing that XGBoost is regarded as one of the best performing models out there at the moment. One could have expected both of the XGBoost models, 'tuned' and 'untuned', to perform at a higher level than they did. Possible reasons for 'less than optimal' model performance are investigated in section 11.

10. SUMMARY

In conclusion, four models were built to predict the yards to be gained/lost for a given NFL play. The dataset provided by Kaggle was preprocessed and consolidated first, after which the data was split into training, validation and test sets. The models were trained on the training set, tuned on the validation set (where applicable) and tested on the testing set. The four aforementioned models and their performances, based on the RMSE, are:

- Simple linear regression model (benchmark model) produced an RMSE of 6.43
- XGBoost model with no hyperparameter tuning produced an RMSE of 6.36
- XGBoost model with hyperparameter tuning produced an RMSE of 6.34
- Average model where the average yardage in the training set was used as the prediction, with an RMSE of 6.41

Although the 'tuned' XGBoost model outperforms the other models there still seems to be quite some room for improvement.

11. LIMITATIONS

Potential limitations of this study along with possible points of action that could lead to improvement are set out below:

Different structuring of the data:

For this task, the author essentially split the data into four groups:

i. Game-related data, e.g. stadium at which it was played
ii. Play-related data, e.g. in which quarter of the game did the play take place
iii. Offensive team data, e.g. NFL unique ID of the offensive team's quarterback
iv. Defensive team data, e.g. NFL unique ID of the defensive team's quarterback

There is a well-known saying in data science: 'A model can only be as good as its data'. The way the data was structured may not be the best way to split the data for optimal model performance. Additionally, the way that missing values were imputed and 'dirty' data cleaned would undoubtedly filter through to the models' performance. This could be investigated further as well.

Potential overfitting:

The fact that the XGBoost models (both 'tuned' and 'untuned') performed worse than might have been anticipated could possibly be attributed to overfitting to the training data. It's important to note here that if the models did in fact overfit, this would also be the case for the benchmark model seeing that it used the exact same features as the XGBoost models. So, if overfitting was addressed per se, we would likely see an improvement in the benchmark model's performance too, thereby keeping the XGBoost models' performance constant relative to the benchmark. However, investigating the extent of overfitting could lead to better models, regardless of any comparison to the benchmark model in this case. One way this could be done is to create predictions on the training dataset using the XGBoost model, calculating the RMSE of those predictions and comparing to the RMSE (of the same XGBoost model) on the test set. If we notice that the performance is significantly better on the training data than on the testing data, overfitting may in fact be an issue.

Two possible ways of preventing overfitting even before it occurs include:

i. Principal Components Analysis (PCA) for dimensionality reduction.
ii. Setting a smaller value for the XGBoost 'early_stopping_rounds' parameter. By reducing the value of this parameter, the model stops at an earlier stage when it recognizes that the model isn't improving its performance on the testing set.

Fitting a different model:

Another possible way to improve results is to fit another model to the data. The first model that comes to mind is the ANN, which is known to perform very well across a wide range of machine learning problems. It was mentioned earlier that ANN's are prone to overfitting, especially if there is a large number of features. However, modern algorithms allow this to be addressed explicitly, even for ANN's. This can be done by setting early stopping rounds (as for the XGBoost) or by setting a dropout parameter when training the ANN. ANN's might therefore be a solid substitute for the XGBoost model.

12. REFERENCES

[1] Rory P. Bunker & Fadi Thabtah (2017). A machine learning framework for sport result prediction.

Applied Computing and Informatics, 15(1), 27-33

[2] https://www.kaggle.com/c/nfl-big-data-bowl-2020 (2019)

[3] https://www.kaggle.com/c/nfl-big-data-bowl-2020/overview (2019)

[4] https://www.kaggle.com/c/nfl-big-data-bowl-2020/data (2019)

[5] https://exploratory.io/ (2019)

[6] https://exploratory.io/git/aEz9ceQ3iQ/Udacity_Castone_Project_EDA_eXo1NTi1.git (2019)

[7] https://towardsdatascience.com/preventing-data-leakage-in-your-machine-learning-model-9ae54b3cd1fb (2019)

[8] Vishal Morde & Venkat Anurag Setty, XGBoost Algorithm: Long May She Reign! https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d (2019)