UDACITY

MACHINE LEARNING ENGINEER NANODEGREE

CAPSTONE PROJECT REPORT

HANNES ENGELBRECHT

16 DECEMBER 2019

1. DEFINITION

Project 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 are common and can be found on a variety of online data science community platforms. This report documents the work done to predict the number of yards gained/lost on given rushing plays in the National Football League (NFL). This problem, the *NFL Big Data Bowl*, is presented by Kaggle as an online data science challenge and can be found at https://www.kaggle.com/c/nfl-big-data-bowl-2020. A more detailed explanation of the problem is in the *Problem Statement* section below.

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.

The tasks required to complete this project include:

- Obtaining the data*
- Exploring the data
- Preprocessing the data

- Creating and training the models
- Evaluating model performances
- Improvements on models
- Conclusion and recommendations

Ideally, the solution will be a model that is able to accurately predict the yards gained/lost on given rushing plays. Additionally, we would prefer a model that is explicable and that can be implemented and updated by the NFL within reasonable timeframes. Some initial models that come to mind are tree-based models, for example Random Forests or XGBoost, as well as Artificial Neural Networks (ANN's). Model selection is explained in more detail in section 2, *Analysis – Algorithms, Techniques and the Benchmark*.

* A dataset was provided by the NFL and can be found at https://www.kaggle.com/c/nfl-big-data-bowl-2020/data.

Metrics

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

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

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.

Seeing that we are predicting a number (yards), the problem at hand can be regarded as one of *regression,* and not *classification.* RMSE is a common metric to use for such problems because it is easily interpretable and is measured in the same units as the target variable (yards, in this case). Other metrics that are used for similar problems include the Mean Square Error (MSE) and Mean Absolute Error (MAE).

2. ANALYSIS

Data Exploration

The provided 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'.

Below is a comprehensive list of the columns included in the dataset along with a short description of each field [4]:

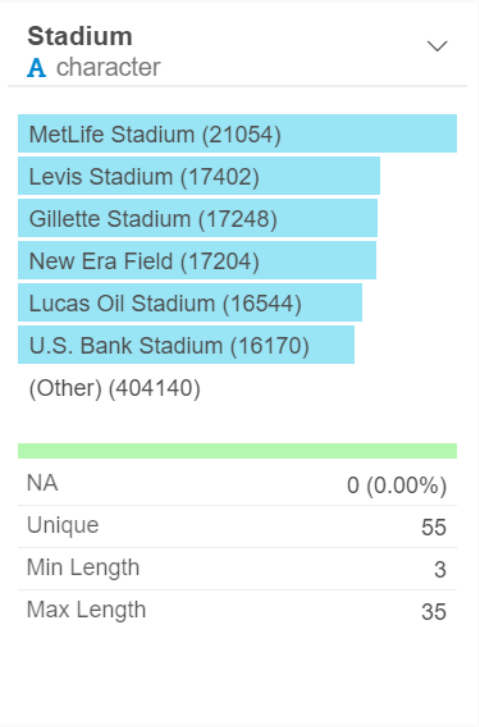
- GameId - a unique game identifier
- PlayId - a unique play identifier
- Team - home or away
- X - player position along the long axis of the field. See figure below.
- Y - player position along the short axis of the field. See figure below.
- S - speed in yards/second
- A - acceleration in yards/second^2
- Dis - distance traveled from prior time point, in yards
- Orientation - orientation of player (deg)
- Dir - angle of player motion (deg)
- NflId - a unique identifier of the player
- DisplayName - player's name
- JerseyNumber - jersey number
- Season - year of the season
- YardLine - the yard line of the line of scrimmage
- Quarter - game quarter (1-5, 5 == overtime)
- GameClock - time on the game clock
- PossessionTeam - team with possession
- Down - the down (1-4)
- Distance - yards needed for a first down
- FieldPosition - which side of the field the play is happening on
- HomeScoreBeforePlay - home team score before play started
- VisitorScoreBeforePlay - visitor team score before play started
- NflIdRusher - the NflId of the rushing player
- OffenseFormation - offense formation
- OffensePersonnel - offensive team positional grouping
- DefendersInTheBox - number of defenders lined up near the line of scrimmage, spanning the width of the offensive line
- DefensePersonnel - defensive team positional grouping
- PlayDirection - direction the play is headed
- TimeHandoff - UTC time of the handoff
- TimeSnap - UTC time of the snap

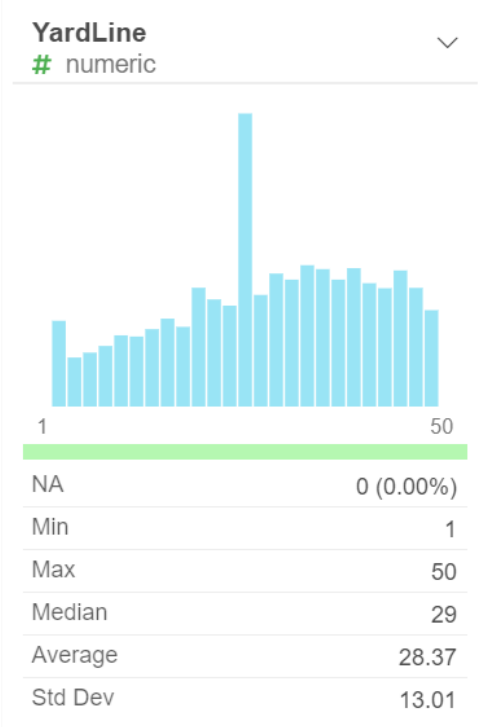
- Yards - the yardage gained on the play (you are predicting this)
- PlayerHeight - player height (ft-in)
- PlayerWeight - player weight (lbs)
- PlayerBirthDate - birth date (mm/dd/yyyy)
- PlayerCollegeName - where the player attended college
- Position - the player's position (the specific role on the field that they typically play)
- HomeTeamAbbr - home team abbreviation
- VisitorTeamAbbr - visitor team abbreviation
- Week - week into the season
- Stadium - stadium where the game is being played
- Location - city where the game is being played
- StadiumType - description of the stadium environment
- Turf - description of the field surface
- GameWeather - description of the game weather
- Temperature - temperature (deg F)
- Humidity - humidity
- WindSpeed - wind speed in miles/hour
- WindDirection - wind direction

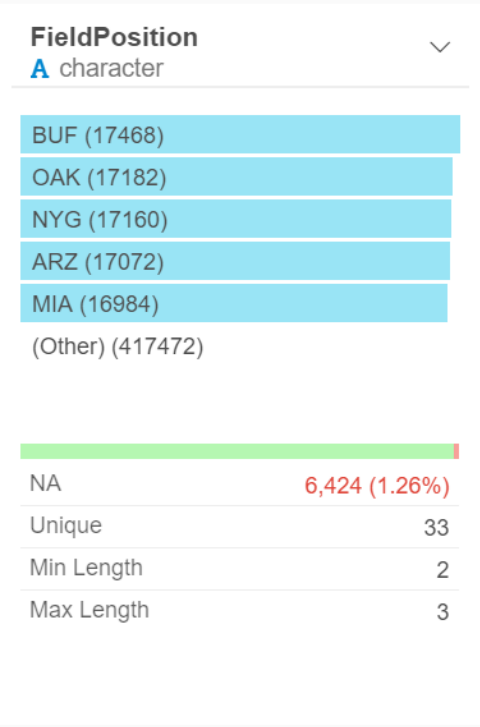
Exploratory Visualization

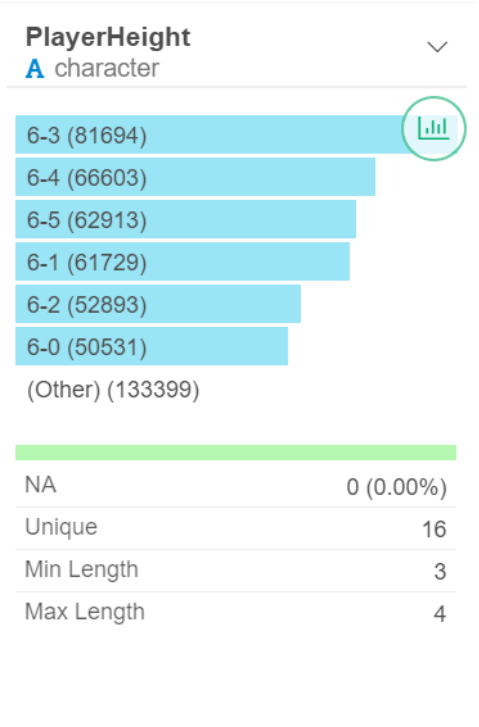
For this section, the *Exploratory Public Desktop* app was used to visualize the data. *Exploratory Public* is a statistical software program that provides users with data science functionality such as data wrangling, visualization, machine learning, reporting and dashboards [5].

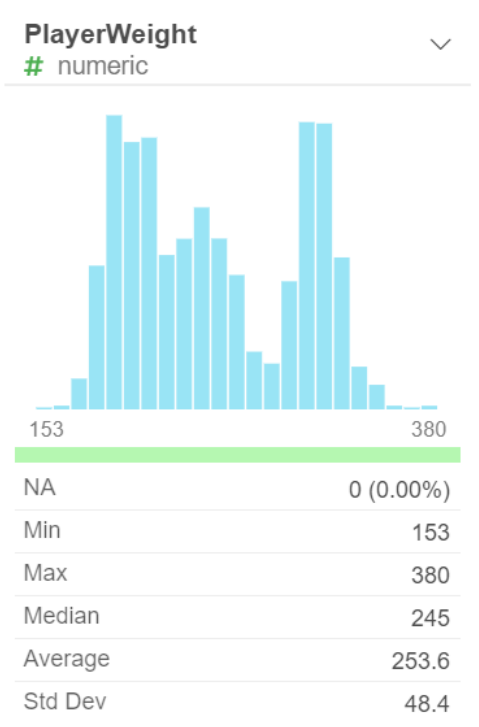
Below are some summary statistics and visualizations of individual columns. The bars represent the total number of rows in the dataset (split across the possible values for that column), with summary statistics of the column's actual values displayed below the horizontal bar charts.

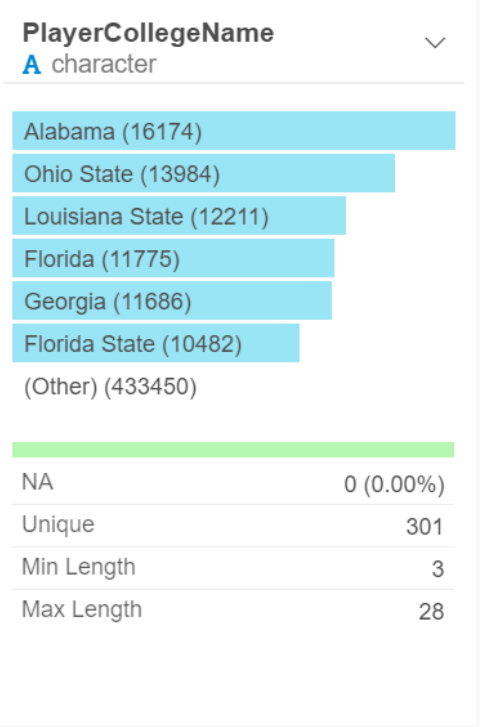












It's worth reiterating here that the target variable was 'Yards'. Most *Exploratory* visualizations therefore focused on investigating the relationship between multiple independent variables and the 'Yards' variable. Some of these visualizations can be seen in Appendix A.

Algorithms, Techniques and the Benchmark

Two models were constructed – a benchmark model for comparison purposes, and a 'model of choice' that was expected to outperform the benchmark model.

The benchmark model:

A more complex metric for model performance was specified by Kaggle for this competition, which will not be explained in this document. The model selected for this project could therefore not be compared to other Kaggle competitors' models using the RMSE. Consequently, a benchmark model needed to be constructed against which the final model's performance could be gauged. For this purpose, a simple model with relatively quick runtimes that is explicable was required.

The benchmark model chosen for this project was a simple linear regression model, which was constructed using Amazon Sagemaker's *LinearLearner* algorithm. Linear regression models are often used when working with regression problems, i.e. where the predicted variable takes on a numerical value. In addition, Amazon Sagemaker's built-in algorithm is simple to train and create predictions from. Simple linear regression models are also easy to interpret relative to other machine learning models, which adds to its selection as the benchmark. The predictions from the benchmark model were used to calculate an RMSE value, which served as the benchmark RMSE value against which the 'model of choice' was compared.

The 'model of choice' – XGBoost:

Our model of choice for this project was an Extreme Gradient Boosting (XGBoost) model using Amazon Sagemaker's high-level XGBoost API. 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 ANN. ANN's are known to overfit to training data in many cases if it's not explicitly accounted for (through early stopping or setting a dropout when training the ANN). Seeing that we ended up with 618 feature columns (after preprocessing the data – see section 3 below), which is a large number, it seemed more fitting to make use of an XGBoost with a 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.

The set of hyperparameters used for the initial XGBoost model are displayed below:

```
# Setting the hyperparameters - these will serve as default values
xgb.set_hyperparameters(max_depth=5,
                        eta=0.2,
                        gamma=4,
                        min_child_weight=6,
                        subsample=0.8,
                        objective='reg:linear',
                        early_stopping_rounds=10,
                        num_round=200)
```

3. METHODOLOGY

Data Preprocessing & Implementation

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].

Training the models:

Both the benchmark model and the XGBoost models were trained on Amazon Sagemaker using the '.fit' method:

```
%%time
# train the estimator on formatted training data
linear_model.fit(formatted_train_data_scaled)
```

```
# First, we fit a general model ('untuned') with predefined hyperparameters. T
# Untuned model is fit
xgb.fit({'train': s3_input_train, 'validation': s3_input_validation})
```

Refinement

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.

Here is the code that was used to vary the hyperparameters and select the best-performing set of parameters:

```
HyperparameterTuner(estimator = xgb, # The estimator object to use as the basis for the training jobs.
                    objective_metric_name = 'validation:rmse', # The metric used to compare trained models.
                    objective_type = 'Minimize', # Whether we wish to minimize or maximize the metric.
                    max_jobs = 20, # The total number of models to train
                    max_parallel_jobs = 3, # The number of models to train in parallel
                    hyperparameter_ranges = {
                         'max_depth': IntegerParameter(3, 12),
                         'eta'      : ContinuousParameter(0.05, 0.5),
                         'min_child_weight': IntegerParameter(2, 8),
                         'subsample': ContinuousParameter(0.5, 0.9),
                         'gamma': ContinuousParameter(0, 10),
                    })
```

4. RESULTS

Model Evaluation and Validation

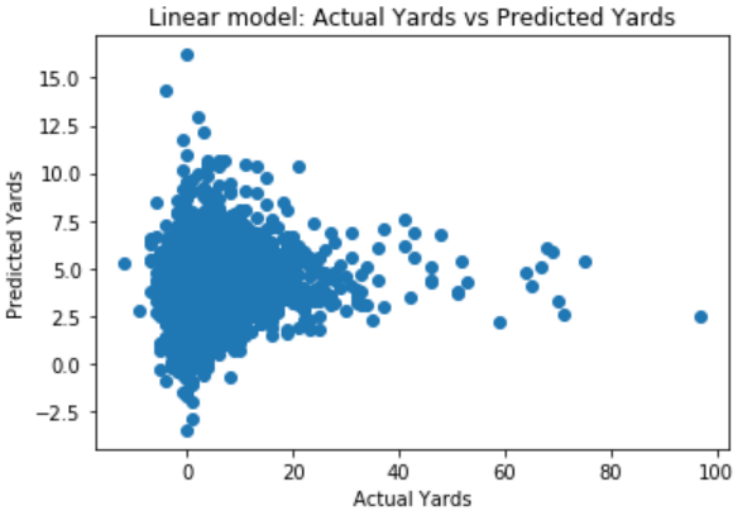
Simple linear regression:

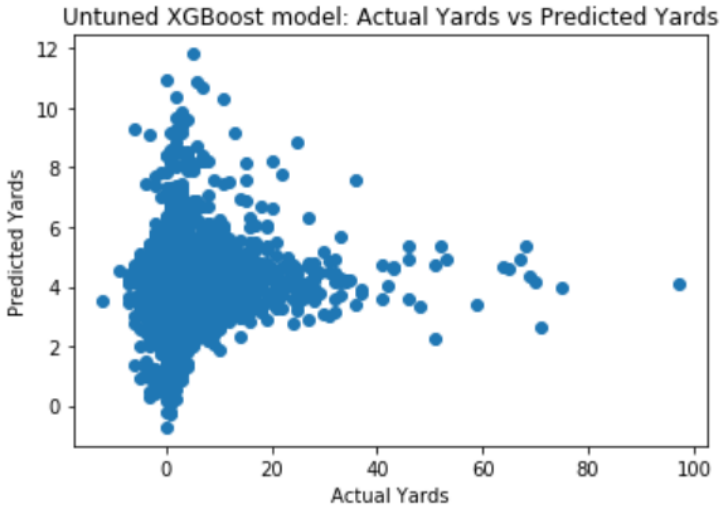
The simple linear regression model had an RMSE value of 6.43.

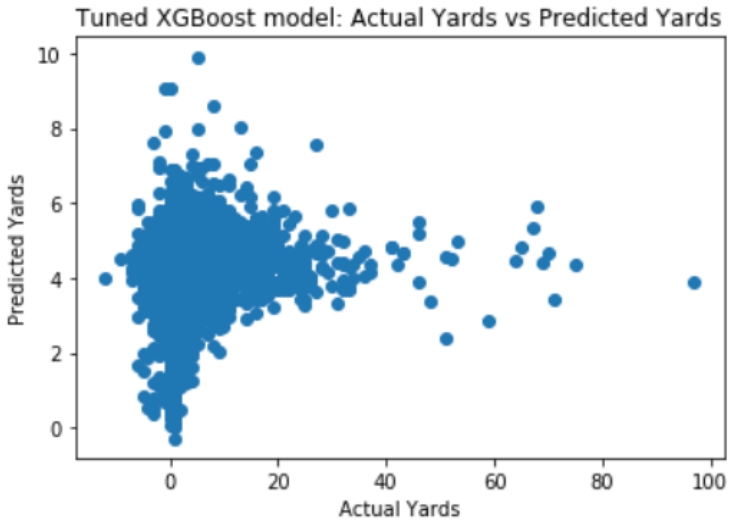
XGBoost:

The 'untuned' XGBoost model had an RMSE value of 6.36 – a slight improvement on the benchmark model. The 'tuned' XGBoost model had an RMSE value of 6.34.

From the above output we notice that the 'tuned' XGBoost model (RMSE of 6.34) performs better than the linear regression model (RMSE of 6.43), as was the case with the 'untuned' XGBoost model. Additionally, the 'tuned' XGBoost model does even beter than the 'untuned' version, albeit by a small margin (difference of 0.02 in RMSE).







Another point worth noting is that the average yards gained per play in the training set (only) was 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 benchmark model.

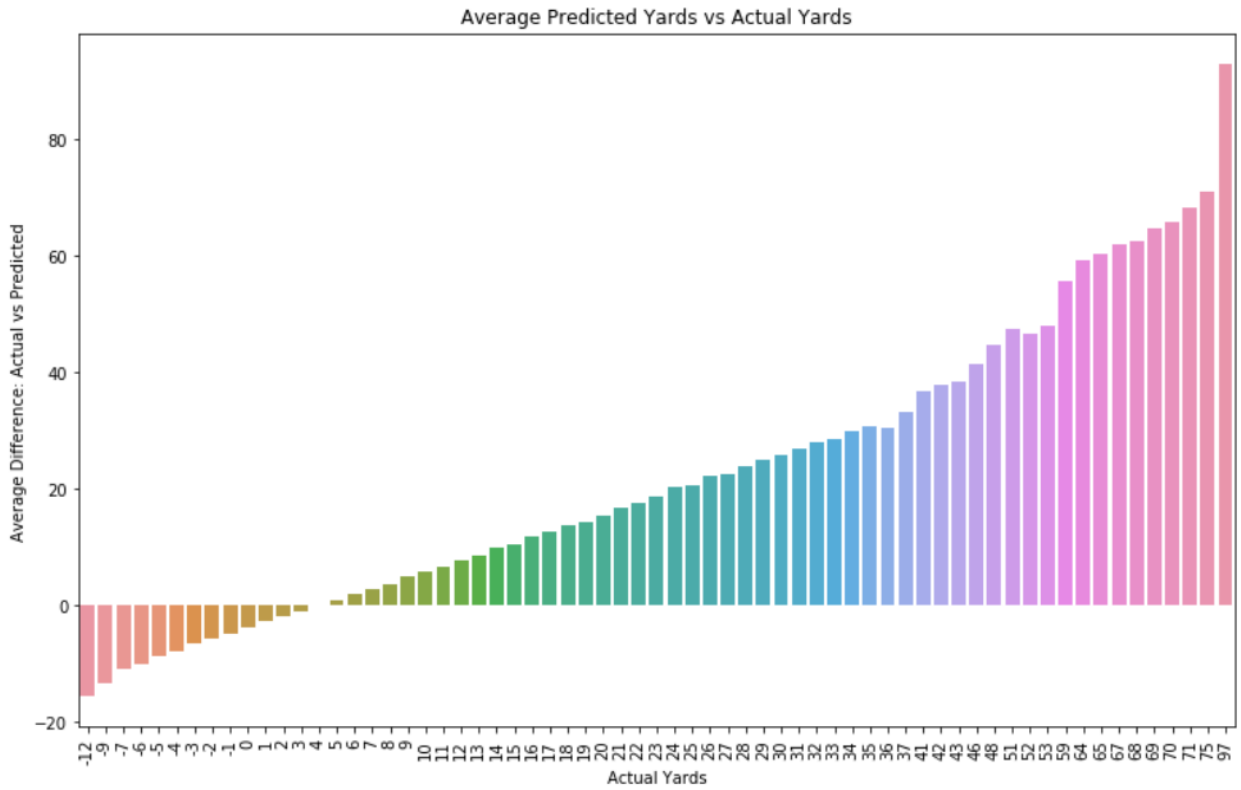
Justification

These results may be regarded as a little surprising at first sight, seeing that XGBoost is regarded as one of the best performing models in the industry 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 5.

5. CONCLUSION

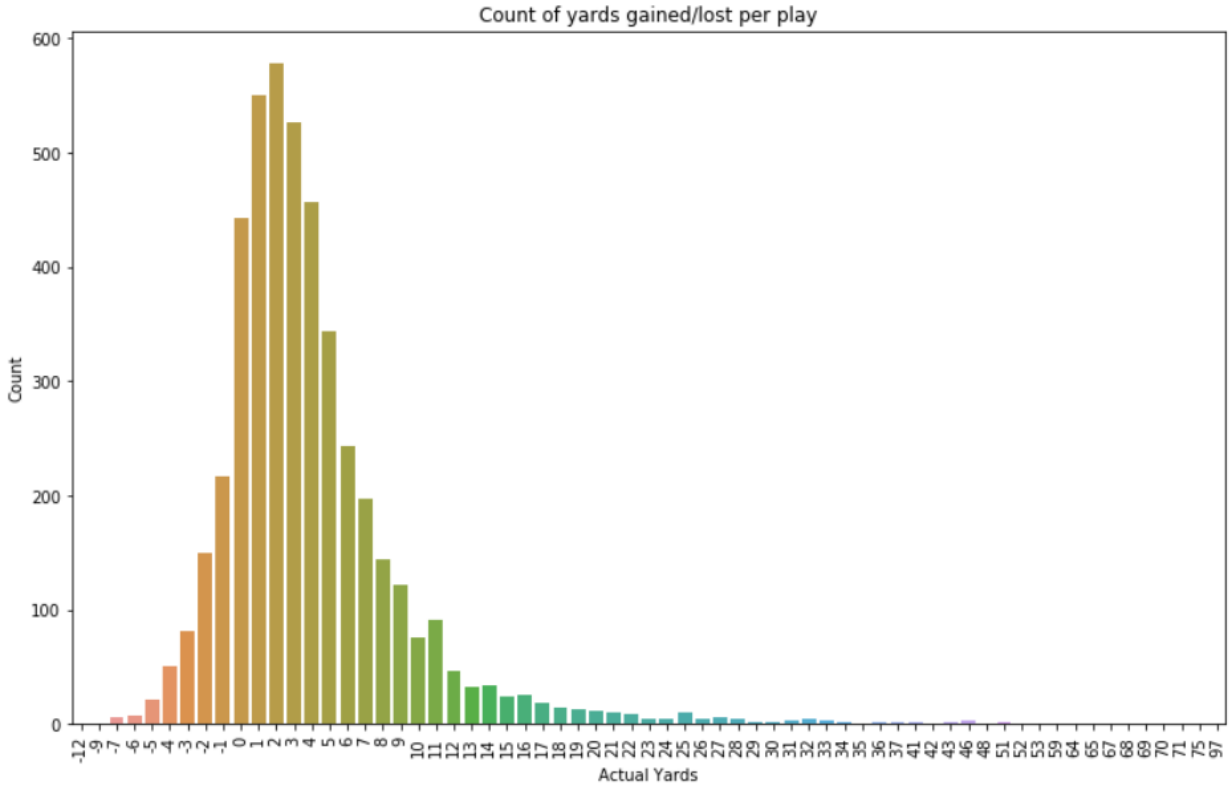
Visualization

Below is an image that depicts the average prediction errors made by the 'tuned' XGBoost model when compared to the actual yardage values. On the x-axis we have the actual yards gained, with the average difference between the predicted and actual values on the y-axis.



The ‘tuned’ XGBoost model predicts plays to gain 4.44 yards on average. Very few plays are predicted to gain more than 5 yards. The main takeaway from the above image is that our XGBoost model predicts most plays to gain between 3 – 5 yards, and that it does not perform well when the actual yards gained/lost is very large. This is the case for the ‘untuned’ XGBoost model as well as the benchmark model.

Suppose that the actual yards gained was 90 and the predicted yard number was 4. Our model missed by quite a margin in this example, which would lead to tall bars on the right-hand side of the above image. We need to keep in mind that most of the actual records saw plays gain between 0 - 6 yards (visible in the graph below). If our model consistently predicts 4.44 yards to be gained, it does not miss the mark by too much for the majority of the observations.



Reflection and possible areas of improvement

Potential limitations of this study along with proposed points of action for an improved model 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 in all models, 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 of determining the extent of overfitting is to create predictions on the training dataset using the XGBoost model, calculate the RMSE of those predictions and compare 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.

6. 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)

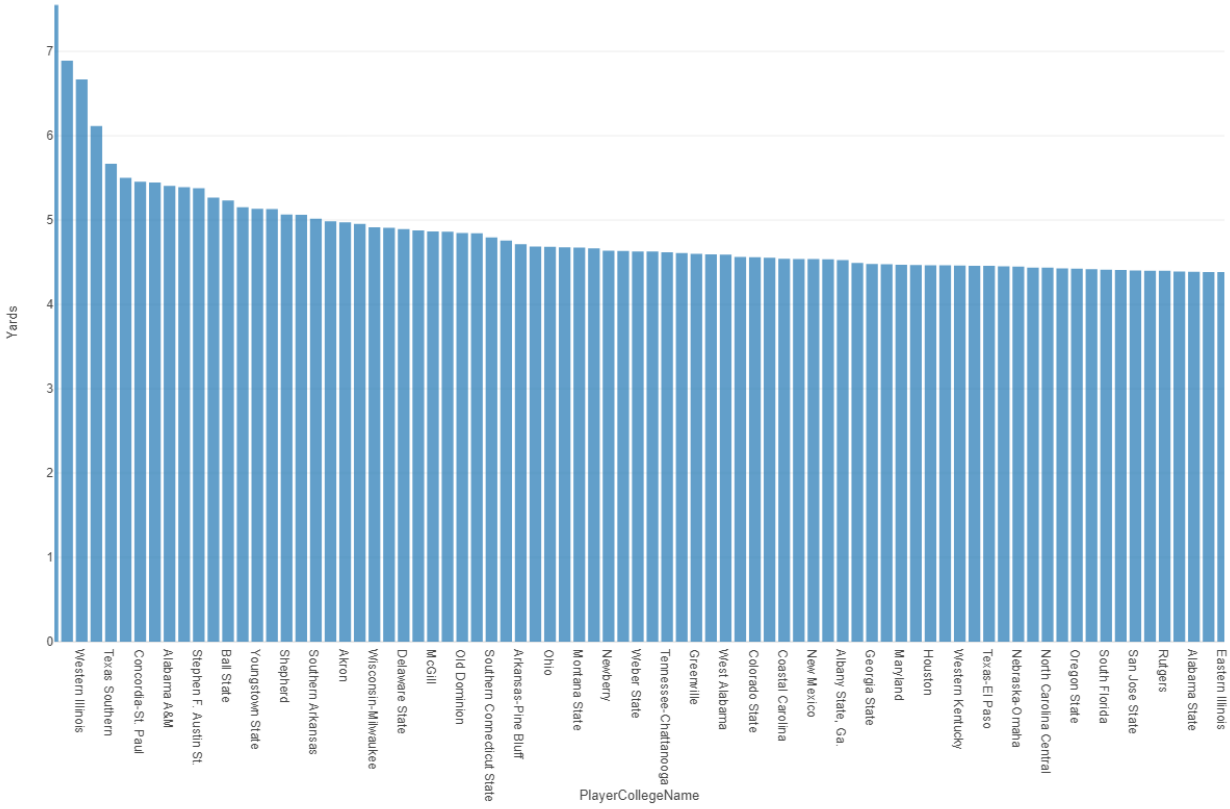
[6] https://exploratory.io/git/aEz9ceQ3iQ/Udacity_Castone_Project_EDA_eXo1NTi1.git (2019)

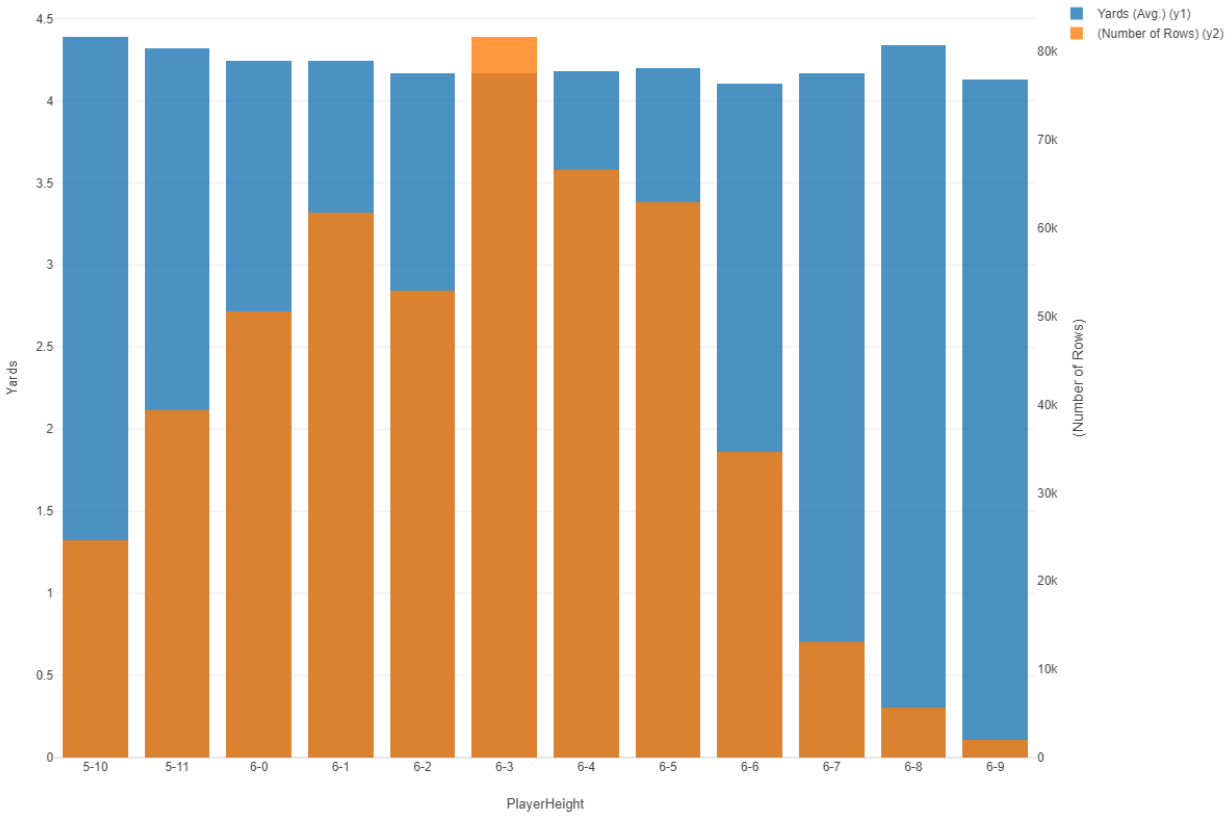
[7] https://towardsdatascience.com/preventing-data-leakage-in-your-machine-learning-model-9ae54b3cd1fb (2019)

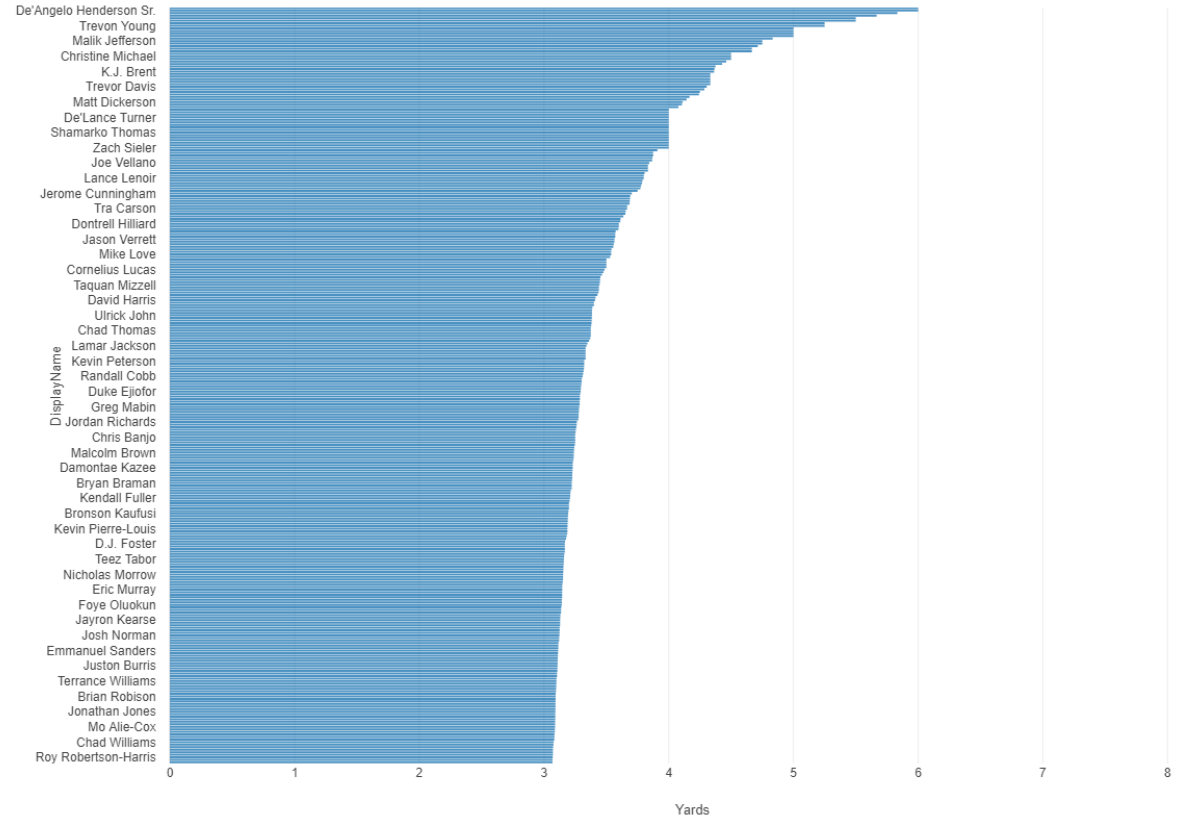
[8] Vishal Morde & Venkat Anurag Setty, XGBoost Algorithm: Long May She Reign! https://towardsdatascience.com/https-medium-com-vishalmorde-xgboost-algorithm-long-she-may-rein-edd9f99be63d (2019)

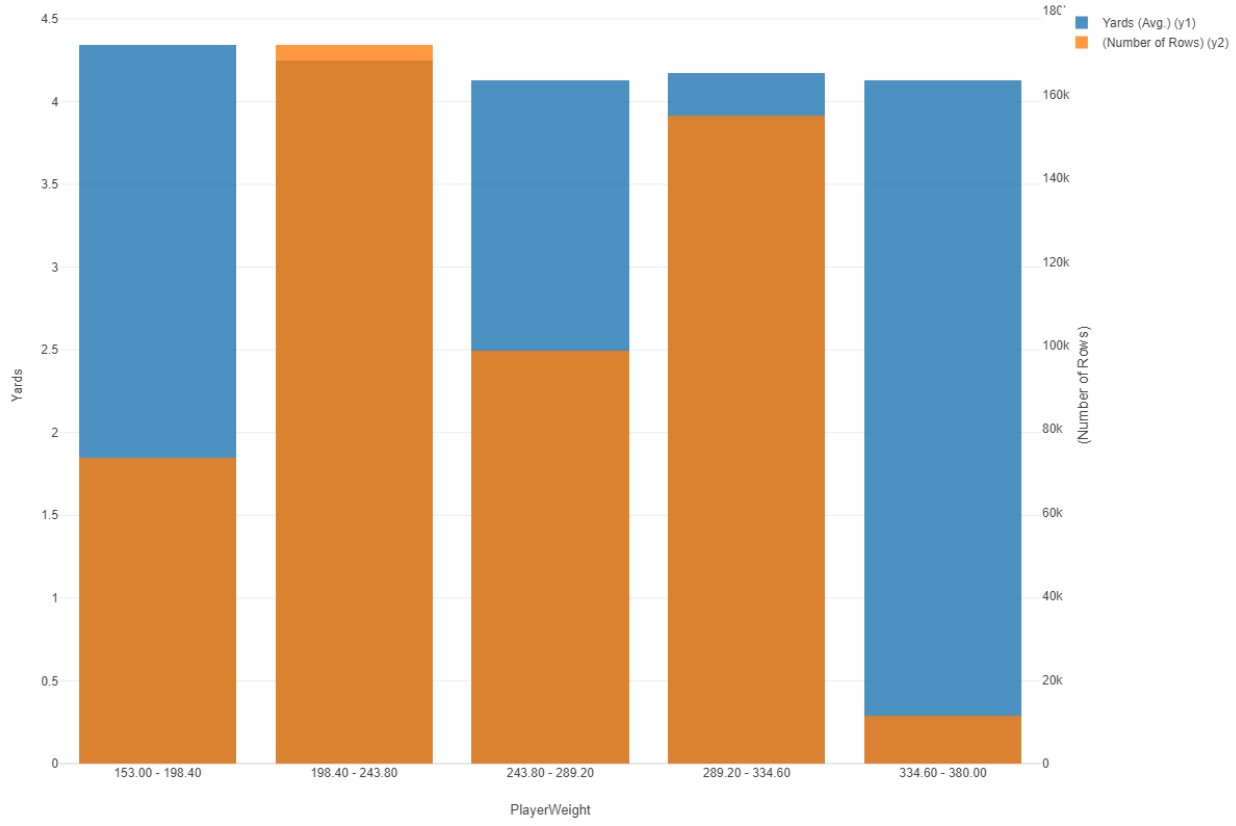
APPENDIX A:

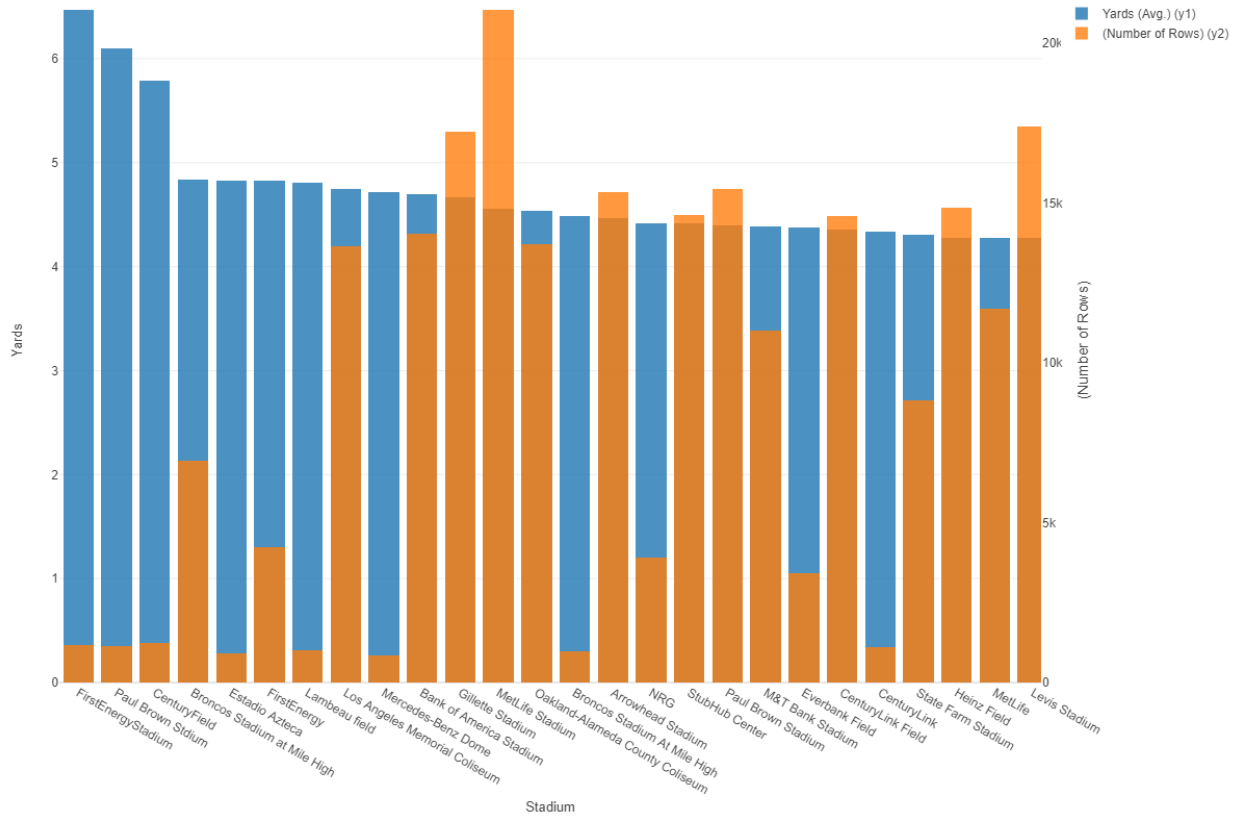
Below are the *Exploratory* visualizations mentioned in section 2, *Analysis.*

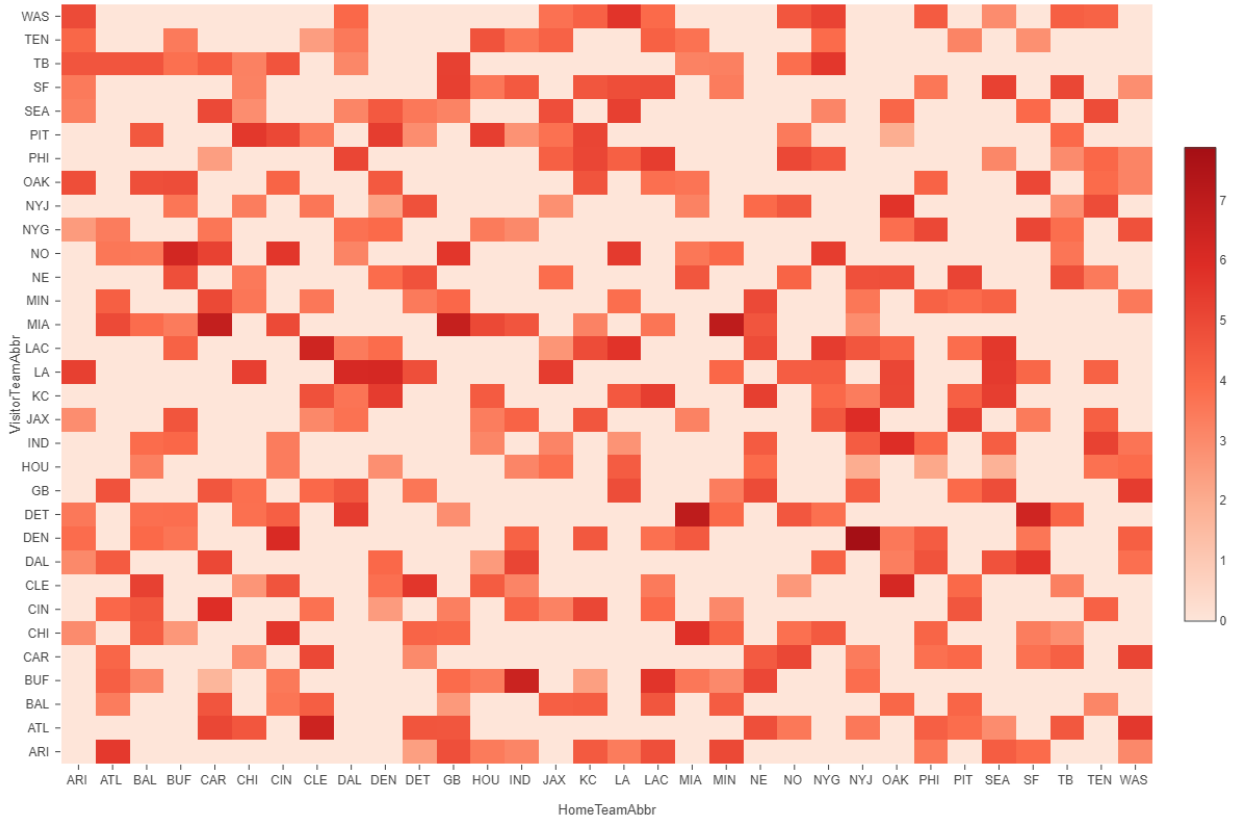


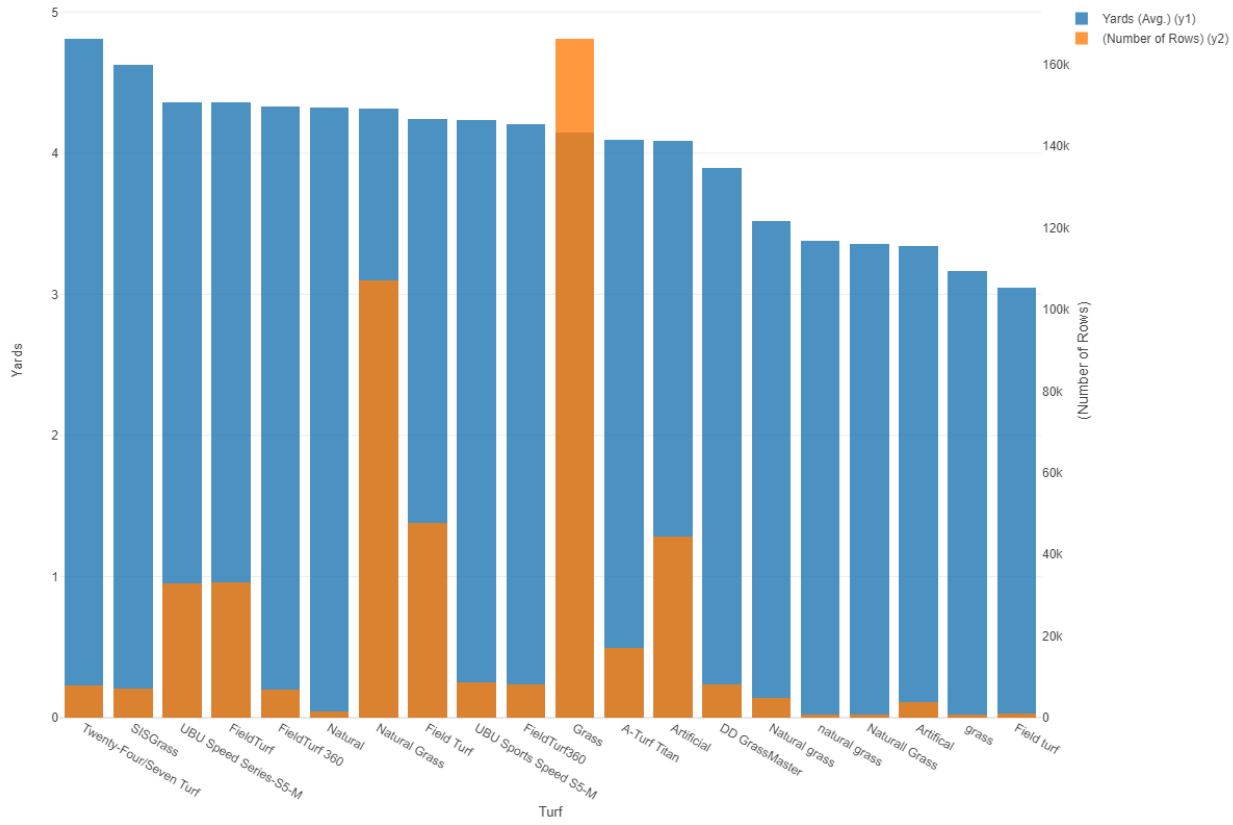

Yards (Avg.) (y1)
(Number of Rows) (y2)
Yards
(Number of Rows)
PlayerHeight
5-10
5-11
6-0
6-1
6-2
6-3
6-4
6-5
6-6
6-7
6-8
6-9


DisplayName
Yards
De'Angelo Henderson Sr.
Trevon Young
Malik Jefferson
Christine Michael
K.J. Brent
Trevor Davis
Matt Dickerson
De'Lance Turner
Shamarko Thomas
Zach Sieler
Joe Vellano
Lance Lenoir
Jerome Cunningham
Tra Carson
Dontrell Hilliard
Jason Verrett
Mike Love
Cornelius Lucas
Taquan Mizzell
David Harris
Ulrick John
Chad Thomas
Lamar Jackson
Kevin Peterson
Randall Cobb
Duke Ejiofor
Greg Mabin
Jordan Richards
Chris Banjo
Malcolm Brown
Damontae Kazee
Bryan Braman
Kendall Fuller
Bronson Kaufusi
Kevin Pierre-Louis
D.J. Foster
Teez Tabor
Nicholas Morrow
Eric Murray
Foye Oluokun
Jayron Kearse
Josh Norman
Emmanuel Sanders
Juston Burris
Terrance Williams
Brian Robison
Jonathan Jones
Mo Alie-Cox
Chad Williams
Roy Robertson-Harris


Yards (Avg.) (y1)
(Number of Rows) (y2)
Yards
(Number of Rows)
PlayerWeight
153.00 - 198.40
198.40 - 243.80
243.80 - 289.20
289.20 - 334.60
334.60 - 380.00


Yards (Avg.) (y1)
(Number of Rows) (y2)
Yards
(Number of Rows)
Stadium
FirstEnergyStadium
Paul Brown Stdium
CenturyField
Broncos Stadium at Mile High
Estadio Azteca
FirstEnergy
Lambeau field
Los Angeles Memorial Coliseum
Mercedes-Benz Dome
Bank of America Stadium
Gillette Stadium
MetLife Stadium
Oakland-Alameda County Coliseum
Broncos Stadium At Mile High
Arrowhead Stadium
NRG
StubHub Center
Paul Brown Stadium
M&T Bank Stadium
Everbank Field
CenturyLink Field
CenturyLink
State Farm Stadium
Heinz Field
MetLife
Levis Stadium


VisitorTeamAbbr
WAS
TEN
TB
SF
SEA
PIT
PHI
OAK
NYJ
NYG
NO
NE
MIN
MIA
LAC
LA
KC
JAX
IND
HOU
GB
DET
DEN
DAL
CLE
CIN
CHI
CAR
BUF
BAL
ATL
ARI
ARI ATL BAL BUF CAR CHI CIN CLE DAL DEN DET GB HOU IND JAX KC LA LAC MIA MIN NE NO NYG NYJ OAK PHI PIT SEA SF TB TEN WAS
HomeTeamAbbr
7
6
5
4
3
2
1
0


Yards (Avg.) (y1)
(Number of Rows) (y2)
Yards
(Number of Rows)
5
4
3
2
1
0
160k
140k
120k
100k
80k
60k
40k
20k
0
Twenty-Four/Seven Turf
SISGrass
UBU Speed Series-S5-M
FieldTurf
FieldTurf 360
Natural
Natural Grass
Field Turf
UBU Sports Speed S5-M
FieldTurf360
Grass
A-Turf Titan
Artificial
DD GrassMaster
Natural grass
natural grass
Naturall Grass
Artifical
grass
Field turf
Turf