NFL Play Call Prediction Using Sequential Neural Networks

Joseph W. Director
University of Colorado Denver
MS Statistics

Joshua French
University of Colorado Denver
Advisor

Steffen Borgwardt
University of Colorado Denver
Graduate Committee

Florian Pfender
University of Colorado Denver
Graduate Committee

Spring 2022

Abstract

The prevalance of data analytics in professional sports has significantly increased over the last 20 years. First popularized in *Moneyball: The Art of Winning an Unfair Game (2003)*, the use of advanced analytics is now mainstream in the four major U.S. sports and abroad. In the National Football League (NFL), millions of dollars are invested into analytics departments and data is being used to drive decision making at every level of an organization's operation. These departments can leverage statistical methods to learn the opposition's tendencies, providing a substantial competitive advantage. In particular, the defensive team can improve its strategy by accurately predicting the offensive team's play call (whether the play is a run or a pass). To this end, many prior works have implemented machine learning algorithms for play call prediction. However, none of the works encountered have treated play-by-play data as sequential. In Football, the offensive team's current play call is dependent upon the sequence of plays called before; therefore, there is a time series component that a modeling strategy must account for. In this work, we explore the ability of sequential deep learning models to predict NFL play calls. Namely, we compare the performance of Recurrent Neural Networks (RNNs) and Long Short Term Memory (LSTM) networks to baseline models (Logistic Regression and Gradient Boosted Decision Trees). Using classification accuracy and ROC-AUC as metrics, we found that sequential models outperform the baseline.

Contents

1 Introduction

1.1 Background

Gameplay of NFL football is seperated into a sequence of instances called plays. The two teams on either side of the ball are allowed to reposition themselves and prepare in betweeen these instances. Given this nature, there is immense opportunity to strategize when the game is not in play. Akin to moving pieces on a chessboard, the coaching staff decides their team's best course of action by anticipating their opponent's moves. For regular plays (i.e. not a kick-off, punt, or field goal attempt), there is a binary option for the type of play the offensive team can do; either a pass or a run. A strategic advantage is gained for the defensive team if they can accurately predict this outcome. As a simplistic example, if they predict pass they can put more players in pass coverage, or put more players near the line of scrimmage if they predict run.

There are a number of indicators that can inform play call prediction. Certain personnel packages (groups of players from various positions) and formations of the offensive team are more associated with either passes or runs. Unfortunately, the NFL does not publicly release data containing specific personnel or formations. However, there exists a general binary indicator for the formation; whether the play was from the shotgun (QB lines up a few yards back from the center) or under-center (QB lines up directly behind the center). Beyond this, there is the in-game context of the current play. This includes the down (how many plays can be used to gain the required distance), the distance (the amount of yards needed to gain in order to keep the ball), the score differential, the amount of time remaining, etc. The conditions of these factors all incentivize the offensive team to use either a run or a pass play. For example, if an offensive team is losing by a lot of points with little time remaining in the game, they are more likely to pass because they can gain more yards using less time. Lastly, tendencies of the offensive team can be studied. This is done by accumulating the relative frequency of passes to runs for the offensive team (pass to run ratio) as well as how successful they are at either passing or running (average yards gained per run or pass play).

An NFL coach combines experience and intuition to predict the play call. In this work, experience is replaced by labeled data points from the entirety of a single NFL season (2019-2020), and intuition by a supervised machine learning algorithm. A supervised machine learning task involves teaching a computer to learn the underlying patterns relating the response variable to the features. If the model can discern information about the features and response well enough, its predictions should generalize well to unseen instances. A popular domain of machine learning is the field of deep learning. Deep learning uses artificial neural networks (ANNs) that loosely resemble a biological brain. They contain networks of individual neurons or nodes, each with its own activation signal, that is each capable of sending and receiving signals to other nodes (through weights). Types of deep learning algorithms vary in complexity and structure. Here we examine recurrent neural networks (RNNs) and their variant, long short term memory (LSTM) networks. These kinds of networks were originally designed for speech and text recognition because of their ability to learn sequential patterns.

1.2 Problem Statement

Prior works have approached play call prediction with machine learning, many implementing complex classification algorithms such as ensemble models and multi-layer perceptrons. These methods only provide marginal performance increases from simplistic models such as logistic regression. This is because, as is often the case in machine learning tasks, the signal relating the response to the features is only so strong. Therefore, in these cases, the choice of a complex algorithm provides small return as there is little that the added complexity can pick up on. Overcoming these performance limits requires rethinking how the data itself is structured. In prior approaches, each singular play is treated as a sample point. Treating the data this way means the model isn't aware of any sequential patterns that may exist. In the approach proposed by this work, an individual sample point is represented by a play sequence of length k.

Table 1: Feature Descriptions

Feature	**Description**	**Type**
Posteam	Name of team on offense for the current play	Categorical
Defteam	Name of team on defense for the current play	Categorical
Yardline	Distance from the goal line (yards)	Numeric
Seconds remaining	Amount of time remaining in current half	Numeric
Yards to go	Yards needed to gain for a first down	Numeric
Down	Number of plays to get a first down	Categorical
Shotgun	Whether the offensive team lines up in shotgun formation	Binary
No huddle	Whether the offensive team used the huddle before the snap	Binary
Posteam timeouts	Timeouts remaining for offensive team	Numeric
Defteam timeouts	Timeouts remaining for defensive team	Numeric
Score differential	Difference in score between offensive and defensive team	Numeric
Temperature	On field temperature for the current play	Numeric
Windspeed	On field windspeed for the current play	Numeric
Posteam Home	Whether the offensive team is on its home field	Binary
Half	Whether the play is ran in first or second half	Binary
Dome	Whether the field has a dome or not	Binary
Open	Whether a domed field is open or closed	Binary
Outdoors	Whether the game is played outdoors	Binary
Cumulative run yards	Total yards gained per run play (at current point)	Numeric
Cumulative pass yards	Total yards gained per pass play (at current point)	Numeric
Pass to run ratio	Current proportion of passes ran to total plays	Numeric
Pass yards allowed	Total yards allowed per pass play (Defteam)	Numeric
Run yards allowed	Total yards allowed per run play (Defteam)	Numeric

1.3 Data

Abundant play-by-play data is made easily accessible by the `nflfastR` package. Data published officially by the NFL is available so no web-scraping is required as a step in preprocessing. A singular season (2019) was chosen as the sample because the offensive and defensive teams are used as features; teams are subject to change from year to year. Around 32,000 individual play instances occurred over the course of this season. A list of features chosen for modeling is included in Table 1. Some of these features are given directly by `nflfastR`, others are acquired through feature engineering. The data given is two dimensional (n, m): n samples (individual plays) and m features. These dimensions are fed into baseline models. For sequential models, three dimensional data is required (n, k, m): n samples, each sample consisting of k consecutive plays, and m features for each play in the sequence. This requires additional preprocessing.

2 Methodology

2.1 Aims

Our analysis aims to implement industry standard data science practices for building and testing models. This consists of building a scalable data pipeline for preprocessing, hyperparameter tuning, and model selection. To this end, cloud computing resources were provisioned from Google Cloud Platform (GCP). This platform allowed for distributed model training on multiple graphics preprocessing units (GPUs); a critical resource given the computational demands of model tuning. The overall workflow for the project is given by Figure 1. Additional steps such as feature engineering and data visualization are included to attain a wholistic view of the data used for modeling.

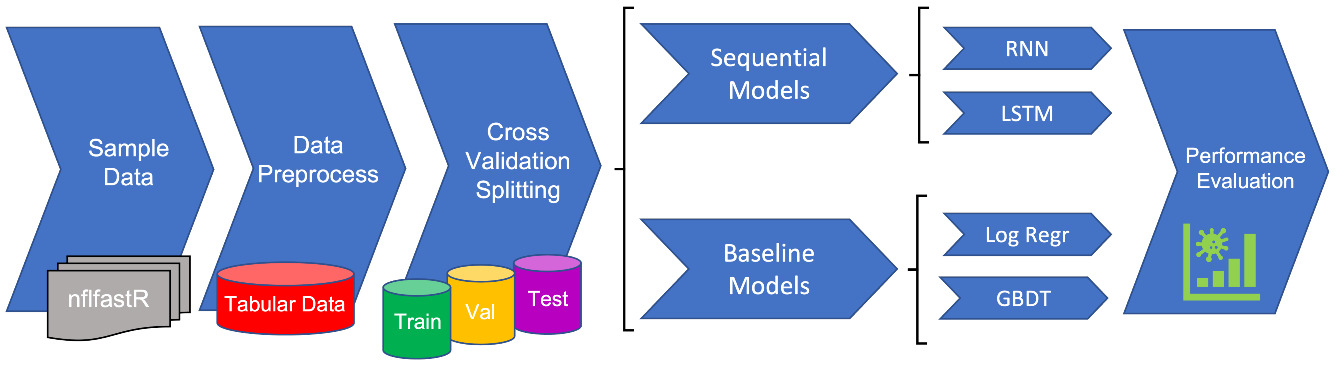

Figure 1: Overall Methodologic Workflow

2.2 Preprocessing

Data sampled directly from `nflfastR` contains all play types and outcomes from both the regular and post-season. For the purpose of this analysis, we are only interested in regular season plays that were either passes or runs. This meant removing all instances of special teams plays (kick-offs, punts, field goal attempts) and pre-snap penalties. It is important to note that some plays are intended to be passes but end up being runs (this is called a QB scramble). We consider these to count as passes since this was the intended play call.

Missing values were only encountered for some of the weather features selected. Temperature and wind are NA for games that were played inside of a dome. Inside NFL domes, the temperature is controlled and most often set to 72 degrees fahrenheit. Missing values for these features were set to 0 windspeed and 72 degrees fahrenheit for all play instances occurring inside a dome.

For all methods considered, binary and categorical features need to be one-hot encoded. This is because machine learning models aren't able to interperet text strings directly. One-hot encoding entails creating a new column for each level of a categorical variable. The column is a 1 if an instance falls under that category and 0 if not. One-hot endcoding was chosen over integer encoding to avoid creating any ordinal pattern in the categorical features.

2.3 Feature Engineering

Feature engineering is a vital aspect of the machine learning process. It is a method by which domain knowledge is leveraged to transform raw data into meaningful features that are capable of distinguishing classes in the response. In this case, features are constructed to describe historical tendencies of both the offensive and defensive teams. This information is designed to be accurate up until the current play instance.

Cumulative tendencies are not readily available in data loaded from `nflfastR`. However, enough information is included in order to build these features. For the cumulative pass to run ratio, we group the data by offensive team and calculate a running total of passes ran divided by total plays. This ensures the feature is current up to a given play; reflecting the information the defensive team would have before the snap. Similarly, we build in features communicating the offensive teams effectiveness at either passing or running. The available data includes how many yards were gained for a given play. Again, we group data by offensive team and calculate a runnning total of yards gained (for each type of play) divided by how many plays were ran of that type.

In addition to having a sense of the offensive teams tendencies and effectiveness, the defensive team will also be aware of its own weaknesses. This aspect is built into the data the same way as offensive tendencies. Instead of grouping by offensive team, we group by defensive team and add a running total of yards allowed per play type.

2.4 Cross Validation Splitting

Cross validation splitting is a method for assessing the performance of a model on unseen data. The original sample of data is partitioned into three non-overlapping subsets called training, validation, and testing. Training is used for teaching the model to learn the patterns within the sample. Validation is the first unseen partition and is reserved for evaluating which hyperparameter combinations perform best. Hyperparameters, unlike model parameters, are not learned during the training algorithm and therefore need to be pre-specified by the user. Given a number of candidate models performing well on the validation set, a final performance evaluation is done on the testing data. Metrics recorded at this stage are used for final reporting.

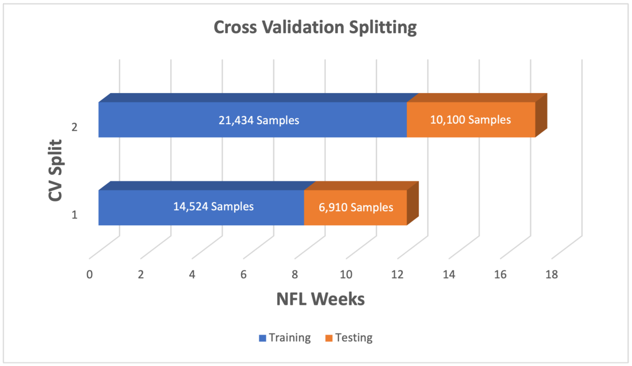

Figure 2: Cross Validation Splitting

The cross validation scheme is described in Figure 2. The first 8 weeks of the NFL season are used for the training set, with weeks 9-12 as validation for evaluating hyperparameters. For each model type, we then select the best 20 hyperparameter combinations for testing. Models with these combinations are retrained using weeks 1-12 and evaluated for final performance using weeks 13-17.

2.5 Exploratory Analysis

Exploring the data through table summaries and visualizations is an important step before applying any modeling strategy. At this stage, we analyze the distribution of the response variable as well as its relationship to the features.

Response Variable

Understanding the distribution of the binary response variable will inform crucial aspects of applying classification models. Mainly, it gives a sense of the class imbalance. Class imbalance exists in the data when one of the binary classes occurs much more often than the other. In cases with heavy imbalance (one class represents 80% or more the observations), undersampling of the dominant class or oversampling the non-dominant class may need to be applied. Table 2 shows the distribution of the response variable in the different cross validation folds.

Fold	**% Pass**	**% Run**
Train	59.50	40.5
Validation	60.01	39.99
Test	59.01	40.99

Table 2: Play Call Distribution in CV Folds

For the three splits, the play call distribution remains at a consistent 60-40 split in favor of passing plays. This kind of distribution is described as slightly imbalanced data. No undersampling or oversampling methods are recommended when this is the case. Note that the overall ratio of passes to runs is the same across the cross validation folds. The validation data needs to be a representative sample of the training data in order to ensure good results.

Features

For brevity, only the most illuminating insights relating play calls to the features are displayed. The first of which is play calls by down and distance, as illustrated in Figure 3. Notably, it is seen that runs are favored over passes on first downs, second and short, and fourth and short situations. Large disparities between passes and runs exist for third/fourth and 4+ yards as well as second and long scenarios. This matches what is expected as offensive teams are heavily incentivized to throw the ball on later downs and longer distances (need to gain more yards using only one or two plays). Perhaps the only surprising aspect of the plot is that passes are more common on third and short. A possible explanation is that passes are being ran on third and short when the offensive team already knows it is going to go for it on fourth down (if the third down pass attempt fails). They are running the riskier option (a pass attempt) knowing they will get another try and then running the safer option (a run attempt) when there is only one attempt available.

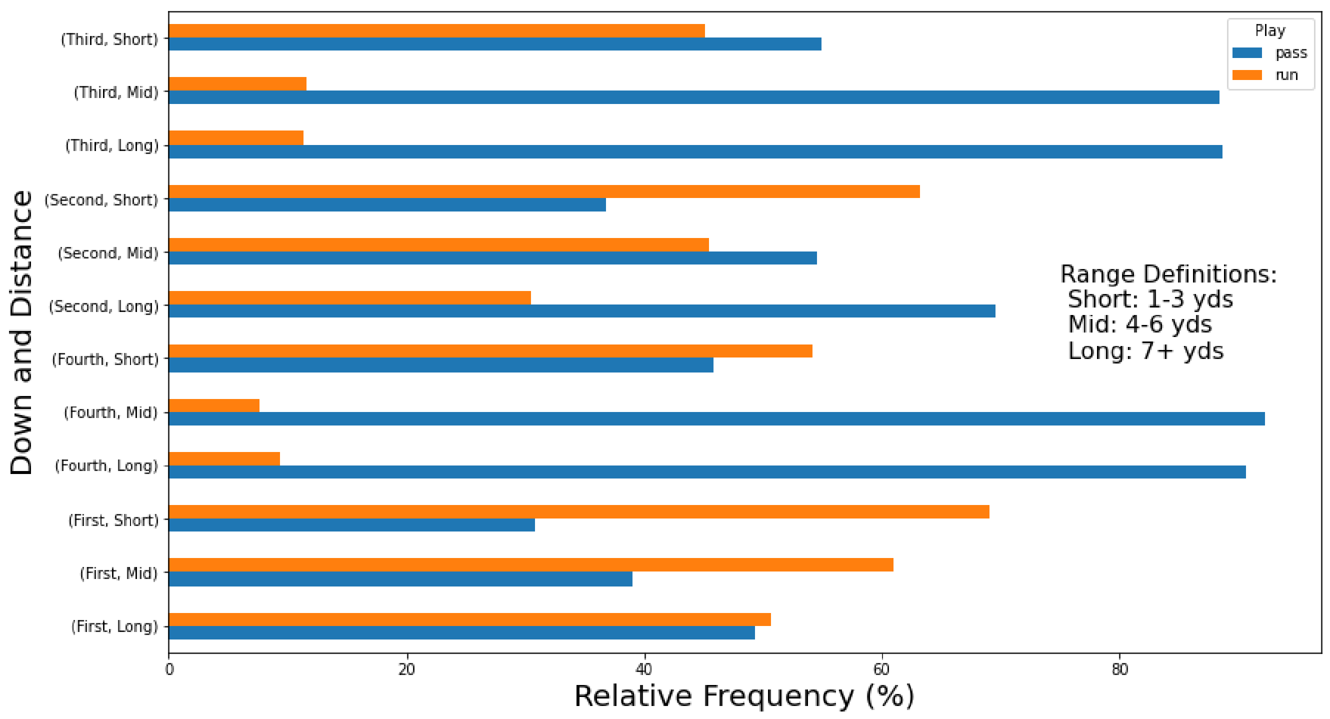

Figure 3: Relative Play Call Frequency by Down and Distance

As previously mentioned, there is little information released by the NFL pertaining to what formation the offensive team lines up in. However, they do release whether the play was from the shotgun formation or under center. In shotgun, the quarterback (QB) lines up farther back from the line of scrimmage. This is typically to allow more time for the QB to throw the ball before the defensive team can get to him. The relationship between shotgun formation and play type is displayed by Figure 4. For more context, the data is grouped by an additional category called huddle. This binary feature tells whether the hurry-up offense is being ran. In the hurry-up, the offensive team does not meet in a huddle before running the play; this is typically done late in games to avoid losing time in between plays. The figure confirms what is expected. Passes are much more common in the shotgun formation and runs are much more common under center. They remain more common for the hurry-up offense in the shotgun formation but the same result isn't true for under center, huddle plays. It is suspected more runs are ran in this scenario because these no huddle plays are ran in normal game conditions (i.e. not with little time remaining, down by many points).

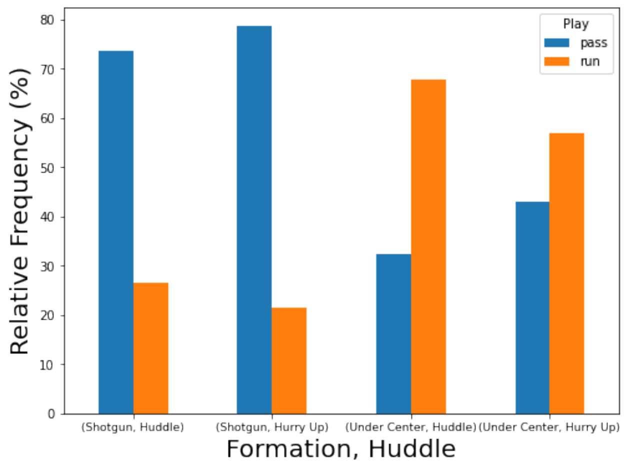

Figure 4: Play Frequency by Formation, Huddle

There is also some visible tendencies relating score differential and amount of time remaining to optimal play call decisions. This relationship is explored in Figure 5 where second half plays are examined (so that seconds remaining reflects time left in the game). Passes seem to be called overwhelmingly with less than 500 seconds remaining in the game when a team is down by 0-20 points. During this same time window, runs seem to be called more when a team is leading by 0-20 points.

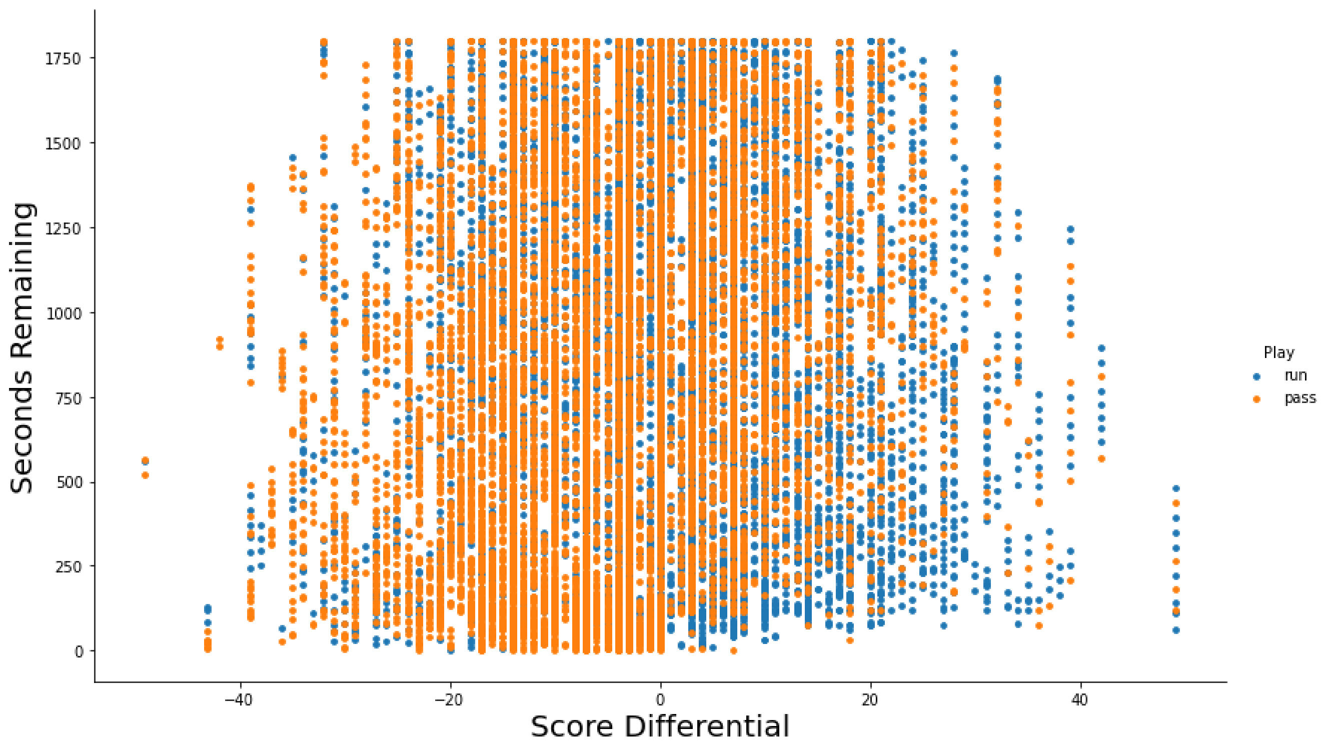

Figure 5: Play Call by Score Differential and Seconds Remaining

Finally, the evolution of play call tendencies as the season progresses are visualized in Figure 6. Teams are plotted by division (across conferences) for added visual clarity. The behavior for each team is generally sporaditic for the first 1000-2000 play calls. Beyond this threshold, a given team's overall tendencies begin to steady out. In the long run it is seen that all teams operate by using around 40-80% passing plays with most teams using about 60%. This matches the play call distributions seen in Table 2.

2.6 Sequential Models

Mathematical details for the sequential neural networks used in this analysis are given.

Recurrent Neural Networks

Recurrent neural networks are a special class of artificial neural networks where connections between nodes flow in a consecutive sequence. This allows them to exhibit memory-like behavior and learn to understand temporal patterns. While originally intended for speech and text recognition, they have been applied to a multitude of tasks where data represents a time series.

For the models used in this work, let the pair (X_i, y_i) for $i = 1, 2, ...N$ represent a training observation where:

- $X_i := X^l_{t-k}, X^l_{t-k+1}, ..., X^l_t$ i.e. a sequence (size k) of design vectors (dimension l, the number of features)
- $y_i := y_t$ i.e. the binary response at time t

Importantly, each input represents a sequence of feature vectors of length k while the target is the binary label at the end of each sequence. Hence, in the context of this analysis, the previous k play calls and current conditions are used to predict the play at timestep t. Now define a simple recurrent network connecting the sequence of inputs to the outputs:

- $h_t = \sigma_h(W_h X_t + U_h h_{t-1} + b_h)$
- $\hat{y}_t = \sigma_y(W_y h_t + b_y)$

Where the variables are given by:

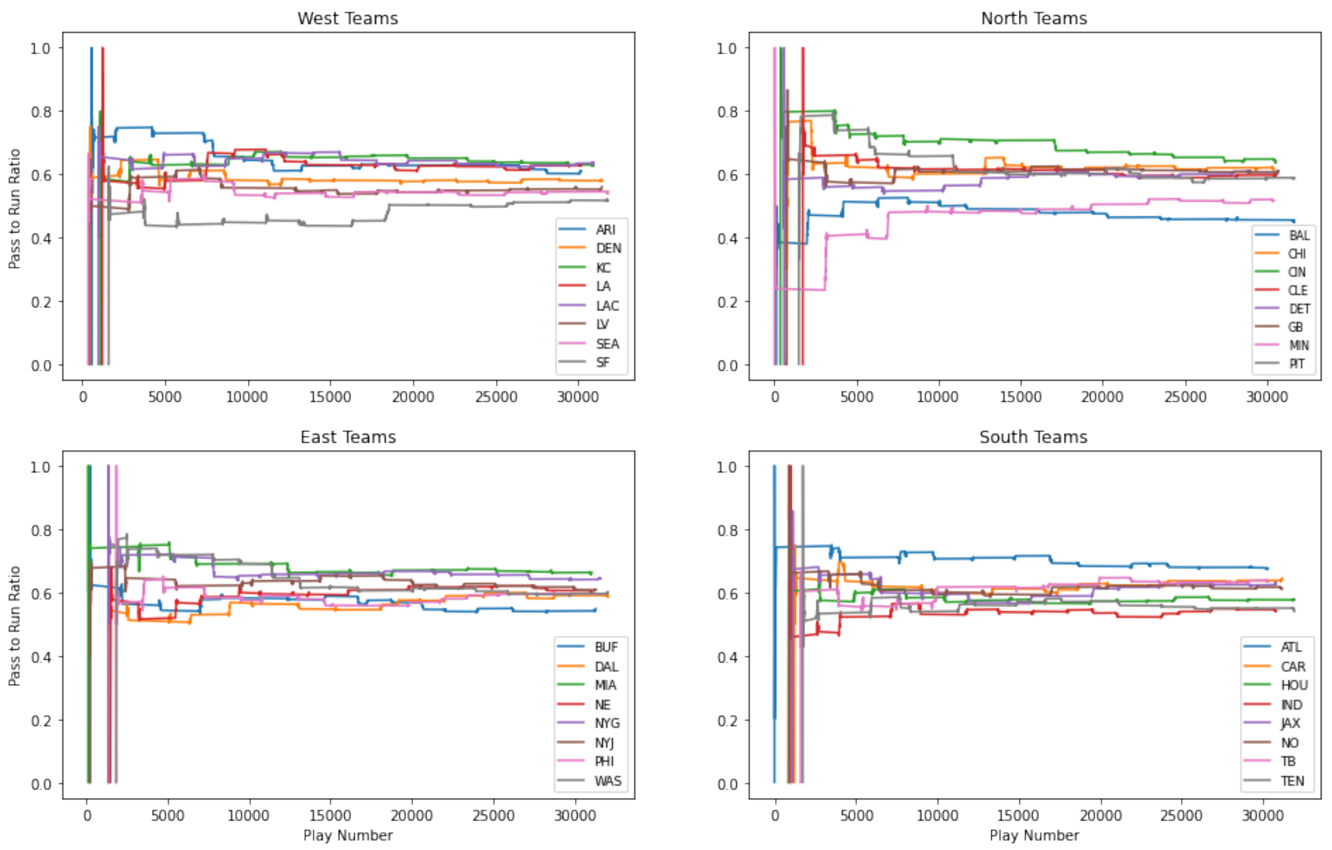

Figure 6: Pass to Run Ratio by Team

- X_t : Input matrix
- h_t := Vector of hidden states
- W_h := Weights connecting inputs to hidden states
- U_h := Weights connecting hidden states to previous hidden states
- b_h := Hidden state biases
- W_y := Weights connecting hidden states to outputs
- b_y := Output biases
- σ_h := Hidden state activation function
- σ_y := Output activation function
- $\hat{y}_t$:= Predicted target variable

A visual representation of the network is given in Figure 7. Note this is a many to one network.

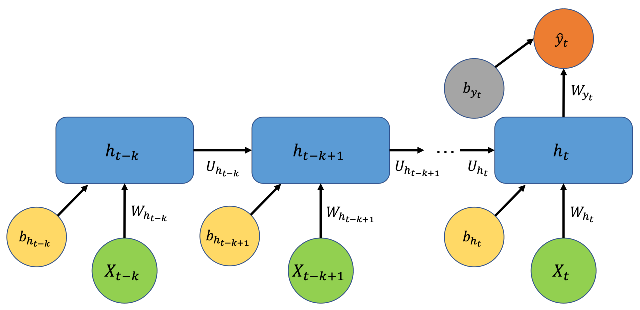

Figure 7: Basic RNN Topology

Activation functions are used to map resulting values (hidden or output states) to a normalized range (usually (0,1) or (-1,1)). For hidden state activation, the choice of hyperbolic tangent (tanh) and exponential linear unit (elu) functions are tuned as hyperparameters, while sigmoid is used for the outputs:

- tanh $:= \sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- elu $:= \sigma(z) = \alpha(e^z - 1) \;\; if \;\; z \leq 0, \;\; z \;\; else$
- sigmoid $:= \sigma(z) = \frac{1}{1+e^{-z}}$

While the simple RNN structure suffices for showing the mathematical details, it should be noted that added complexity is used for modeling in this work. This includes stacking multiple recurrent and dense layers before the output layer. When this is the case, the outputs of the previous layer simply feed into the next one. Many such aspects of the model topology are treated as hyperparameters, as will be discussed later on.

For training a recurrent network, a first order iterative optimization algorithm known as mini-batch gradient descent is used with backpropogation. First, define a cost function $C(y_i, \hat{y_i})$ relating model predictions to actual observed outcomes. The cost function is designed to be large when the model missclassifies an observation and small when it correctly classifies. In this analysis the binary cross entropy (also known as log-loss) cost function is used:

$$C(y_i, \hat{y_i}) = -(y_i log(\hat{y_i}) + (1 - y_i) log(1 - \hat{y_i}))$$

The value for $\hat{y_i}$ is between 0 and 1, interpreted as the predicted probability the observation belongs to a given class. As $C(y_i, \hat{y_i})$ is a function of model parameters (weights and biases), optimal parameters are found by minimizing the cost function.

Given a choice of cost function, the training algorithm begins by randomly sampling observations from the sample into mini-batches of size n. This is done for managing computational efficiency while ensuring convergence to a global minimum. It is common practice to set $n = 32$ as is done in this work. For each mini-batch the weights and biases must first be set to initial values, this is done by randomly sampling from a uniform distribution. A forward pass is completed given these initial values, hence all hidden state and output activations are computed. Given model outputs, the gradient of the cost function with respect to model parameters across all samples in the mini-batch is attained. A backward pass is then performed by updating the weights and biases according to the negative direction of the gradient and a pre-specified learning rate γ. Parameters are updated as follows:

$$\bar{\Theta}_{j+1} = \bar{\Theta}_j - \gamma(\nabla C(\bar{\Theta}_j))$$

When the model completes a mini-batch, it is called an iteration. The algorithm will then continue through all mini-batches to complete an epoch (i.e. it has seen all observations in the training sample). This process will repeat for a pre-specified number of epochs or until a stopping cirteria is met. Typically, this criteria is to stop when the cost function has not reduced over a given number of epochs.

LSTM Neural Networks

A problem with traditional RNNs is that they struggle to pick up on long term patterns. LSTM models have a similar structure with some added complexity that tries to account for long dependencies. Let the training observation pairs be represented by (X_i, y_i) as they were previously. Define a new network relating the input sequence to the output:

- $f_t = \sigma_g(W_f X_t + U_f h_{t-1} + b_f)$
- $p_t = \sigma_g(W_p X_t + U_t h_{t-1} + b_p)$
- $\tilde{c_t} = \sigma_c(W_c X_i + U_c h_{t-1} + b_c)$
- $c_t = f_t \circ c_{t-1} + p_t \circ \tilde{c_t}$
- $h_t = o_t \circ \sigma_c(c_t)$
- $o_t = \sigma_g(W_y X_t + U_y h_{t-1} + b_o)$

Where the variables are defined as follows:

- $f_t :=$ Forget gate
- $p_t :=$ Update gate
- $\tilde{c_t} :=$ Cell update
- $c_t :=$ Cell state
- $h_t :=$ Hidden state
- $o_t :=$ Output gate
- $W, U, b :=$ Weights and Biases
- $\sigma_g :=$ Sigmoid activation
- $\sigma_c :=$ Hyperbolic Tangent activation
- $\circ :=$ Pointwise vector multiplication

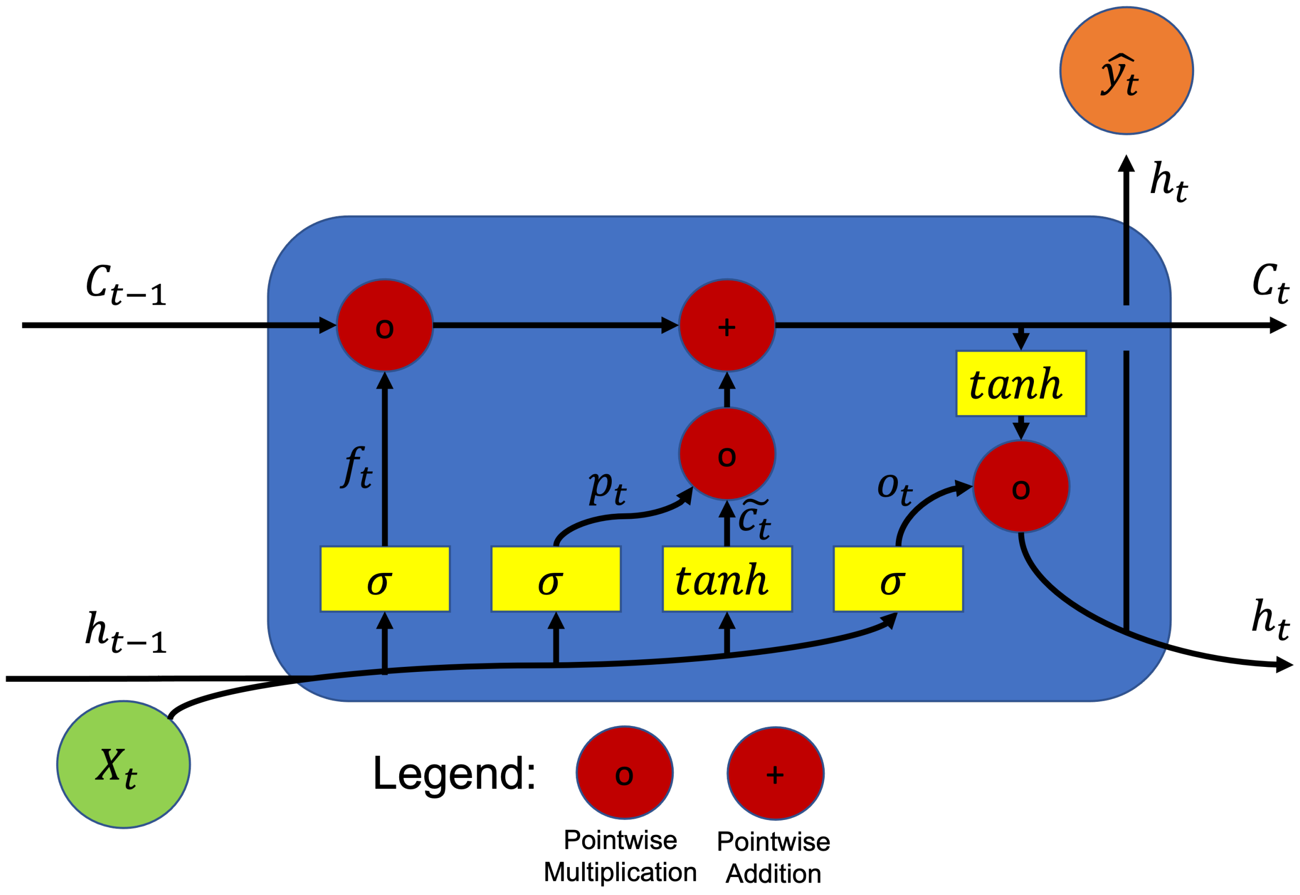

Figure 8: LSTM Topology

The structure of an LSTM cell is shown in Figure 8. The cell state c_t communicates the long term memory of the network and are updated at various points within the cell. f_t, the forget gate, is used for deciding what to discard. It combines information from the previous output and the current input and uses sigmoid activation to compress this into a value between 0 and 1. This is combined with c_{t-1}, the previous cell state, by pointwise multiplication. Values close to 1 mean keep this information while values close to 0 mean discard it.

Then, new information is added to the cell state by pointwise multiplication of an input gate layer p_t (sigmoid activation) and candidate cell values $\tilde{c}_t$ (tanh activation). The cell update values combine information from the previous cell and current input while the input gate scales how much influence the update should have on the cell state. After this is decided, this information is added to the cell state by pointwise addition.

Given that the cell state has been updated, the cell must now decide what information to output. The output gate, o_t, uses sigmoid activation to transmit information from the previous outputs and current input. This is merged with information from the updated cell state (tanh activation) by pointwise multiplication. Output is passed to the next LSTM cell (along with the updated cell state) and to the current target prediction.

The training process for LSTM neural networks is the same as in simple RNNs. Mini-batch gradient descent with backpropogation is used with the same cost function (binary cross entropy) to train all weights and biases. As is the case with RNNs, added complexity of LSTM topology is explored in this analysis. The structure of a sequential model, whether LSTM or RNN, is tuned via various a hyperparameters. A list of hyperparameters chosen for model tuning of sequential neural networks is given in Table 3.

Table 3: Sequential Model Hyperparameters

Parameter	Description	Range	Model
Sequence Length	(int) k, Number of plays to consider a sequence	[1,15]	Both
Sequential Layers	(int) Number of LSTM or RNN layers to stack	[1,4]	Both
Recurrent Units	(int) Number of hidden states in sequential models	[8,102]	Both
RNN activation	(choice) Hidden state activation for RNNs	[tanh, elu]	RNN
RNN dropout	(float) Fraction of units to drop from input	[0.1,0.25]	RNN
RNN rec dropout	(float) Fraction of units to drop from recurrent states	[0.1,0.35]	RNN
LSTM dropout	(float) Fraction of units to drop from input	[0.1,0.25]	LSTM
Dense layers	(int) Number of dense layers after sequential layers	[1,2]	Both
Dense dropout	(float) Fraction of input units to drop (dense layers)	[0.1,0.3]	Both
Learning rate	(float) Rate at which weights are updated in training	$[1e^{-5}, 1e^{-2}]$	Both

The use of dropout at various stages warrants further explanation. This is a regularization parameter. Regularization refers to measures in machine learning algorithms that counteract overfitting. Overfitting occurs when a model fits too closely to the training data and therefore doesn't generalize well to new observations. When dropout is added to a layer in a deep learning model, the fraction sets the amount of randomly selected inputs to turn off (set to 0). This helps to avoid keying in to closely to the training inputs and helps performance on test data.

2.7 Baseline Models

Logistic Regression

Logistic regression is a simple but powerful classification algorithm that is applied in a wide range of contexts. In this setting, each data point (X_i, y_i), $i = 1, ..., N$ is treated as an individual play call and feature vector (i.e. not a sequence). Arrange the feature vectors into an $N \times l$ design matrix X and define a column parameter vector β of length l. The parameters describe the relationship between each feature and the response. The probability that the response belongs to the positive class at each i is then given by:

$$\hat{y} = \sigma_g(X\beta)$$

Where σ_g is the sigmoid activation function defined earlier. Similarly to the training of sequential models, a cost function is defined $C(y_i, \hat{y}_i)$. Again, the binary cross entropy is chosen as the cost function. Gradient descent is applied in this context as well as once more the task is to find optimal parameters that minimize the cost function. The parameters are initialized at random. Then, at each training iteration, the parameters are updated by the gradient of the cost function given a specified learning rate γ:

$$\beta_{j+1} = \beta_j - (\gamma \nabla C(\beta_j))$$

This process is repeated for a specified number of iterations or until a similar stopping condition as in sequential models.

Gradient Boosted Decision Trees

Gradient boosted decision trees (GBDT) are a popular class of ensemble models. They are called an ensemble because they combine the predictions of many so called weak learners. Weak learners in this context are decision tree classifiers. Decision trees work by partitioning the feature space into subregions that seek to seperate the target classes. This is done by a series of binary decisions pertaining to certain values of the features. For an in context example, the model might first consider splitting observations based on time remaining in the game. All obervations falling under 250 seconds are sent down one branch of the tree and the rest another. For observations under 250 seconds, another split might be considered based on score differential, whether it is positive or negative. This example is illustrated by the tree diagram in Figure 9.

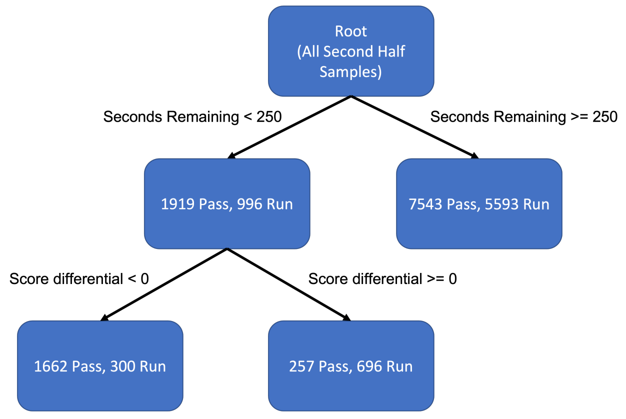

Figure 9: Simple Tree Diagram

The quality of a subregion is assessed by the binary cross entropy (as defined earlier) of observations that fall within its range. Determining optimal parameters (where to make splits, what features to use for a split, and what threshold of said features) is done by a greedy optimization algorithm. It starts from a single root branch and adds branches in an iterative fashion, making locally optimal decisions at each step. Predictions are made by majority voting in the leaf nodes and the probability of belonging to a class is given by its proportion of observations in the leaf.

Decision trees are considered weak learners because they exhibit a lot of variance based on the training data used. This means that using a different sample of training data will dramatically change the structure of the tree, causing the tendency to overfit. To overcome this defect, single decision tree models are rarely used, and instead multiple are combined in various ways called ensembles. Gradient boosting is one such ensemble method that builds multiple decision trees in an iterative fashion, with each iteration correcting the mistakes of it predecessor.

As the name suggests, another form of the gradient descent algorithm is used for optimization. Letting the data be represented by (X_i, y_i), $i = 1, .., N$, let γ denoted the log(odds) of the target variable i.e. $log(\frac{passes}{runs})$. Define a cost function as:

$$C(y_i, \gamma) = -y_i\gamma + log(1 + e^{\gamma})$$

The process begins by determining an initial model defined by a single constant leaf node for all N observations. This is done by minimizing γ over the sum of the cost function across N:

$$M_0 = argmin_{\gamma} \sum_{i=1}^{N} C(y_i, \gamma)$$

Where M_0 represents the initial model. Given this choice of loss function, the value for γ that minimizes this sum is simply the log(odds) based on observed values in the data. The model iterates by fitting decision trees to residuals between observed values and the previous probability predictions. For $j = 1, ..., J$:

1. Find the residual $r_{ij} = -\frac{\partial C(y_i, M_{j-1}(X_i))}{\partial M_{j-1}}$
2. Fit a decision tree using the features and the residuals (X_i, r_{ij})
3. Calculate output values in the new decision tree $\gamma_j = argmin_{\gamma} \sum_{i=1}^{N} C(y_i, M_{j-1}(X_i) + \gamma h_j(X_i))$
4. Update the model $M_j = M_{j-1} + \gamma_j h_j$

The final model ouput will be M_J. Step 4 can be seen to closely resemble the prior results of gradient descent where γ_j represents the learning rate.

As was the case with sequential models, there are a number of hyperparameters to tune in both logistic regression and gradient boosted decision trees. A list of hyperparameters for the baseline models is given in Table 4. Again, many of these are related to regularization. In logistic regression, this comes in the form of an elastic net. An elastic net is a way to penalize the cost function by mixing L_1 and L_2 norms. The ratio controls how much L_1 penalization to use relative to L_2. Penalization constrains the parameters during optimization and forces the algorithm to favor more simplistic models, which helps if the model overfits.

For GBDT, there are a number of regularization parameters. A random subsample (with replacement) of the original data is used during the bulding of each tree. This is akin to bootstrapping and greatly decreases the variance of predictions given a change in training data. Similarly, only a random subsample of features are considered at each split (the square root of the total features). Aspects of the tree structure such as the minimum

Table 4: Baseline Model Hyperparameters

Parameter	Description	Range	Model
Estimators	(int) Number of iterations	[100,500]	GBDT
Subsample	(float) Fraction of samples to consider for each tree	[0.5,1.0]	GBDT
Min samples split	(int) Min number of samples needed to split a node	[2,15]	GBDT
Min samples leaf	(int) Min number of samples needed in each leaf	[1, 10]	GBDT
Max Depth	(int) Max number of consecutive branches	[1,5]	GBDT
L_1 ratio	(float) Proportion of L_1 penalty to L_2	[0,1]	LOG

samples needed to make a split, minimum samples required in a leaf, and max depth all prevent the model from learning the training data too closely.

2.8 Tuning Algorithm

Model tuning is a procedure where different hyperparameter combinations are tested on validation data. There are a number of ways to conduct this process. Given extensive search spaces (possible parameter combinations) and limited computational resources, a tuning algorithm with optimal efficiency is favored. This analysis used a Hyperband search strategy.

Hyperband is designed to allocate more resources (i.e. training iterations) to models that are promising in their early stages. The algorithm builds off the idea of successive halving. In successive halving, a large number of randomly chosen combinations are ran for a few iterations. The top half performing combinations are chosen to continue training for more iterations and the rest are discared. This is repeated until one model remains. Successive halving uses a finite budget B (total training iterations) and fixes a number of combinations to consider n. A problem with this approach is that it does not optimize the tradeoff between trying many combinations (large n) and trying fewer combinations but for more iterations (large B/n). It is unknown at the time prior to training whether the model will converge slowly or quickly. If n is large, and the model converges slowly, then many good models will be discarded in the early stages. Conversely, when B/n is large, poor models will be trained for too many iterations and the budget will be wasted.

Hyperband addresses this tradeoff by testing different numbers of combinations for a fixed budget. Define r, the minimum amount of iterations that can be allocated to a single combination, and R, the maximum amount of iterations that can be allocated to a single combination. The algorithm chooses various values of n and r based on R (outer loop) and then runs successive halving on each fixed n and r (inner loop). A run of the inner loop is called a bracket. Each bracket uses a fixed budget B but tests different tradeoffs between n and B/n, as n is varied. By testing ways to balance resources and combinations, this algorithm is optimal for large search spaces and limited computing resources.

To scale up the tuning process, free to use resources were provisioned from GCP. Distributed training was performed across 2 GPUs on a single virtual machine. Tuning models in this way allows for multiple combinations to be tested simutaneously, greatly expediting the process. The top 20 performing hyperparameter configurations for each model type considered were saved and passed on to final testing.