NFL Injury Prediction

Abstract:

NFL Injury Prediction: Based on a variety of factors such as turf type, weather, stadium, player speed/direction, etc., can a machine learning model predict whether an NFL player is likely to suffer an injury?

Team Members:

Kushal Gourikrishna

Problem Statement:

The NFL has always been a very violent sport and injuries are an unfortunate byproduct of this. However, aside from injuries suffered due to the physicality of the game there are also a large number of non-contact injuries that are suffered every year. There has been a suspicion that turf type can have an impact on player injuries but there are a variety of factors on every play that could play a role such as the speed and direction of the player, weather conditions, stadium, etc.

Objective:

Build a classification model that can help predict injuries to NFL players based on a suite of factors present during a typical NFL game

Approach/ Methodology:

The ultimate goal of this project is to experiment with various different types of machine learning models in order to determine what the best prediction method could be for the chosen datasets. The first step was to investigate the datasets and join them together to create one dataset that could be used for model training and prediction. Once the datasets were joined, EDA and feature engineering was performed to get an idea of various trends in the data and identify inputs for the experimental models.

The primary takeaway from the data exploration and feature engineering was that the data that has been chosen for this project is heavily imbalanced. Hence, as part of the model building process, data augmentation and other techniques would be required in order to investigate.

The following methods were and models were chosen to be built and run for this project. The idea was to tackle the imbalanced data problem from all angles by experimenting with various machine learning models as well as experimenting with different data augmentation and weighting techniques.

Model Building Using Undersampled Data:

- KNN
- Logistic Regression
- Decision Tree
- Decision Tree (Hyperparameter Tuning)
- Random Forest
- Random Forest (Hyperparameter Tuning)
- XGBoost
- AdaBoost

Model Building Without Data Augmentation:

- KNN
- Logistic Regression
- Decision Tree
- Decision Tree (Hyperparameter Tuning)
- Random Forest
- Random Forest (Hyperparameter Tuning)
- XGBoost
- AdaBoost

Model Building Using Oversampled Data:

- KNN
- Logistic Regression
- Decision Tree
- Random Forest
- XGBoost
- AdaBoost

(Note with oversampled data, hyperparameter tuning was not performed due to time constraints as the tuning process took many hours.)

Artificial Neural Networks

- Basic ANN
- Basic ANN with Class Weights
- Basic ANN trained with undersampled data
- Basic ANN trained with oversampled data

Convolutional Neural Networks:

- Basic CNN
- CNN with Class Weights
- CNN trained with undersampled data
- CNN trained with oversampled data

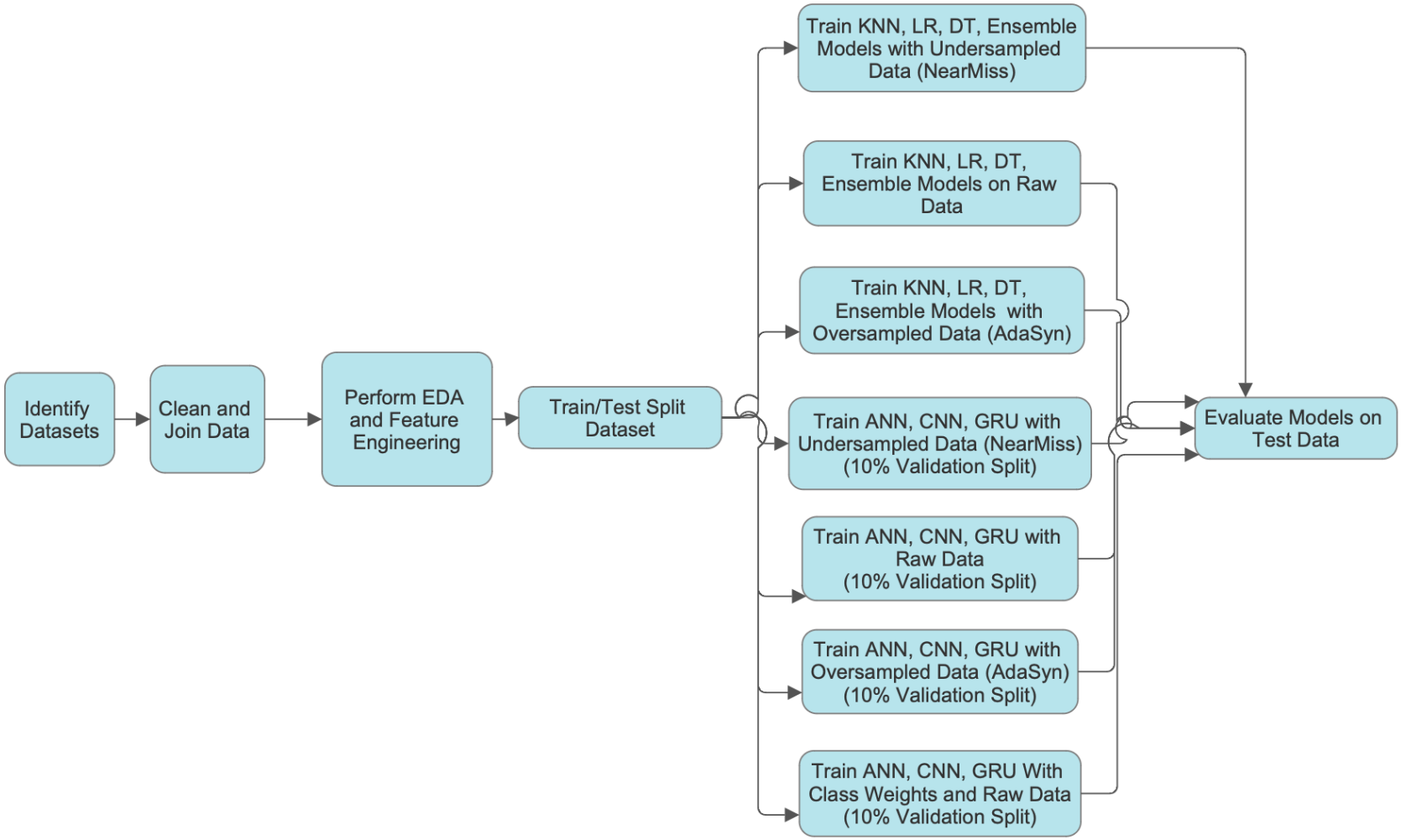
Gated Recurrent Unit Networks

- Basic GRU
- GRU with Class Weights
- GRU trained with undersampled data

- GRU trained with oversampled data

Finally, all of the results from each of these experiments are stored in tables to compare results and analyze outcomes.

Block Diagram:



Datasets:

Three total datasets were used for this project

- **Injury Record:** The injury record file in .csv format contains information on 105 lower-limb injuries that occurred during regular season games over the two seasons. Injuries can be linked to specific records in a player history using the PlayerKey, GameID, and PlayKey fields.
    - https://drive.google.com/file/d/16YOcHu6cYXq5Q-3mKl2a4YPgH2LTzMmK/view?usp=sharing
- **Play List:** The playlist file contains the details for the 267,005 player-plays that make up the dataset. Each play is indexed by PlayerKey, GameID, and PlayKey fields. Details about the game and play include the player's assigned roster position, stadium type, field type, weather, play type, position for the play, and position group.
    - https://drive.google.com/file/d/19LTDs6ZecNgCymioofI3RkEV-UZkBJcz/view?usp=sharing
- **Player Track Data:** player level data that describes the location, orientation, speed, and direction of each player during a play recorded at 10 Hz (i.e. 10 observations recorded per second).
    - https://drive.google.com/file/d/1R80OPOD_MX2v4rxCWDwiiAr1P_C9koSd/view?usp=sharing

The following tables describe the data in each of these datasets:

Field	Format	Description
PlayerKey	XXXX	Uniquely identifies a player with a five-digit numerical key
GameID	PlayerKey-X	Uniquely identifies a player's games (not strictly in temporal order)
PlayKey	PlayerKey-GameID-X	Uniquely identifies a player's plays within a game (in sequential order)
BodyPart	character string	Identifies the injured body part (Knee, Ankle, Foot, etc.)
Surface	character string	Identifies the playing surface at time of injury (Natural or Synthetic)
DM_M1	1 or 0	One-Hot Encoding indicating 1 or more days missed due to injury
DM_M7	1 or 0	One-Hot Encoding indicating 7 or more days missed due to injury
DM_M28	1 or 0	One-Hot Encoding indicating 28 or more days missed due to injury
DM_M42	1 or 0	One-Hot Encoding indicating 42 or more days missed due to injury

Table 1: Injury Data

Field	Format	Description
PlayerKey	XXXX	uniquely identifies a player with a five-digit numerical key
***GameID**	PlayerKey-X	uniquely identifies a player-game (this index is not strictly in temporal order) – see note below
PlayKey	PlayerKey-GameID-X	uniquely identifies a player's plays within a game (in sequential order within a game)
RosterPosition	character string	provides the player's roster position (i.e. Running Back)
*PlayerDay	integer	an integer sequence that reflects the timeline of a player's participation in games; use this field to sequence player participation
*PlayerGame	integer	uniquely identifies a player's games; matches the last integer of the GameID (not strictly in temporal order of game occurrence)
StadiumType	character string	a free text description of the type of stadium (open, closed dome, etc.)
FieldType	character string	a categorical description of the field type (Natural or Synthetic)
Temperature	float	on-field temperature at the start of the game (not always available - for closed dome/indoor stadiums this field may not be relevant as the temperature and weather are controlled)
Weather	character string	a free text description of the weather at the stadium (for closed dome/indoor stadiums this field may not be relevant as the temperature and weather are controlled)
PlayType	character string	categorical description of play type (pass, run, kickoff, etc.)
PlayerGamePlay		an ordered index (integer) denoting the running count of plays the player has participated in during the game
Position	character string	a categorical variable denoting the player's position for the play (RB, QB, DE, etc.) – may not be the same as the roster position.
PositionGroup	character string	a categorical variable denoting the player's position group for the position held during the play

Table 2: Play Level Data

Field	Format	Description	Additional Details
PlayKey	character string		
time	float	time in seconds (float) since the start of the NGS track for the play	this is the time index for the player track
event	character string	play details (character string) as a function of time during the play (huddle break, snap, etc.)	
x	numeric list	player position (float) along the long axis of the field (yards) over time index	0 - 120 yards
y	numeric list	player position (float) along the short axis of the field (yards) over the time index	0 - 53.3 yards
dis	numeric list	distance traveled (float) from prior time point over the time index	Distance (yards)
s	numeric list	estimated speed (float) at that particular point in time over the time index	yards per second
o	numeric	Orientation (float) - angle that the player is facing (deg)	0 - 360 degrees
dir	numeric	Direction (float) - angle of player motion (deg)	0 - 360 degrees

Table 3: Player Tracking Data

Success/Failure Criteria:

With a heavily imbalanced dataset, accuracy will not be a reliable metric for this project in terms of success or failure. Primarily, recall score will be the more important metric for the imbalanced data, but the model overall has to be robust as well.

As a result the following criteria will be considered:

Primary Metric:

- Recall Score > 70%

Evaluation Parameters:

In addition to recall score and F1 score, the following parameters will be used for model evaluation:

- Accuracy
- Recall
- False Negative Rate
- Precision
- F1 Score
- ROC/AUC Score

Experiment Description:

Before building any machine learning models, the chosen datasets had to be cleaned and joined. After that, some feature engineering was performed to narrow down the relevant features to be used as inputs for the machine learning models. For missing values, appropriate values were imputed based on contextual information in the rest of the dataset. Once the data was cleaned and joined, a correlation matrix was created to determine if any features could be dropped based on high correlation. In Figure 1, the pre and post correlation matrix for the numerical features is shown. Categorical variables were encoded so they could be interpreted by the machine learning models.

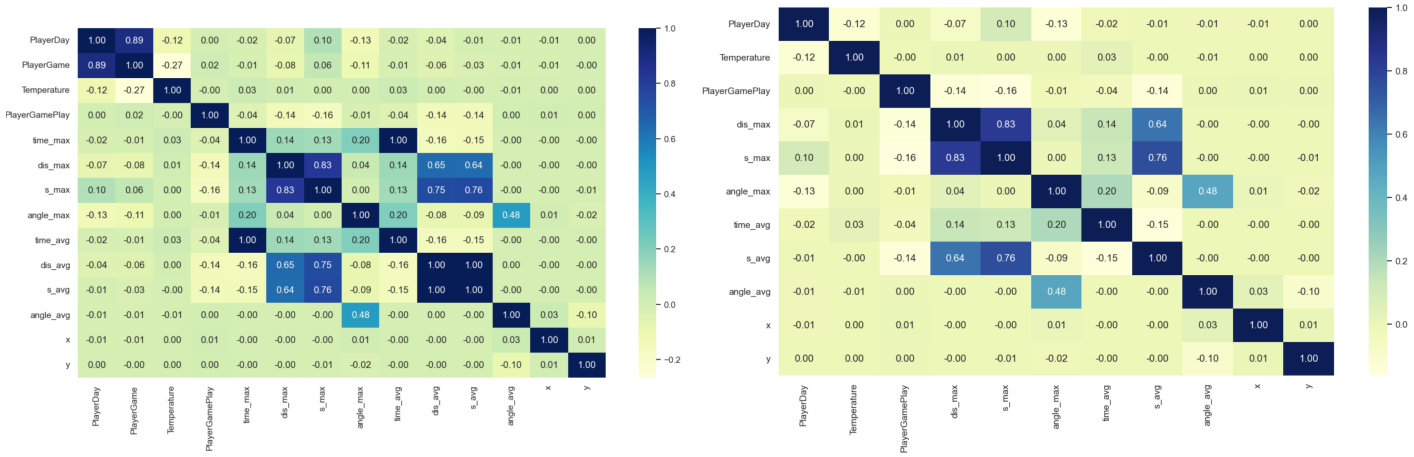


Figure 1: Correlation Matrix before and after dropping features (Before on Left, After on Right)

Finally, in Table 1 below the complete list of features used as inputs for all of the machine learning models built for this project is shown.

PlayerDay	**Number of games a player has played to this point**
Temperature	**Temperature at time of game**
PlayerGamePlay	**The number of plays the player has participated in up to this point**
dis_max	**Maximum distance traveled during play**
s_max	**Maximum speed reached during play**
angle_max	**Maximum body angle during play**
time_avg	**Average time of play**
s_avg	**Average speed during play**
angle_avg	**Average body angle during play**
x	**X position on the field**
y	**Y position on the field**
RosterPosition	**Player Roster Position: Cornerback, Defensive Lineman, Kicker, Offensive Lineman, Quarterback, Running back, Safety, Tight End, Wide Receiver**
StadiumType	**Stadium Type: Indoors, Outdoors, Roof Open, Roof Closed**
FieldType	**Field Type: Natural, Synthetic**
Weather	**Weather at time of play: Clear, Cloudy, Indoors, Precipitation**
PlayType_Rush	**Type of play run: Rush, Pass, Extra Point, Kickoff, Kickoff Not Returned, Kickoff Returned, Field Goal, Punt Not Returned, Punt Returned, Punt,**
Injury	**Whether a player suffered an injury on this play or not**

Table 4: Final Feature List

Once the data was cleaned and feature engineering was complete, model building could begin. As discussed in the approach and methodology section, due to the imbalanced nature of the chosen NFL injury dataset, multiple different types of data augmentation techniques and machine learning models were experimented with.

In terms of data augmentation, two different techniques were used along with using the raw data. For undersampling, the NearMiss method was used. This method selects examples from the majority class based on their (Euclidean) distance from examples in the minority class. For oversampling, the AdaSyn technique was used. This technique generates synthetic samples for the minority class to balance the overall set.

Overall, nine different machine learning techniques were experimented with for this project:

- KNN

- Logistic Regression
- Decision Tree
- Random Forest
- XGBoost
- AdaBoost
- Artificial Neural Network
- Convolutional Neural Network
- GRU Recurrent Neural Network

With decision trees and random forests, some hyperparameter tuning was done as well to optimize the models for the undersampled and raw data. Due to the large size of the oversampled data, hyperparameter tuning was not done for this set as the process took too much time.

An 80% training and 20% test data split was used for all models. For the neural networks, a 10% split within the training data was used as a validation set. All models were trained on undersampled, raw data, and oversampled data and then evaluated with unaugmented test data. Due to the heavy imbalance in the data, class weights were modified to adjust for this in the raw data.

Results:

	Model	Accuracy	FalseNegRate	Recall	Precision	F1 Score	ROC_AUC
0	KNN	0.858260	0.904762	0.095238	0.000265	0.000529	0.476899
1	Logistic Regression	0.417740	0.285714	0.714286	0.000483	0.000965	0.565954
2	Decision Tree	0.087688	0.285714	0.714286	0.000308	0.000616	0.400863
3	Decision Tree Tuned	0.060127	0.047619	0.952381	0.000399	0.000797	0.506078
4	Random Forest	0.074264	0.047619	0.952381	0.000405	0.000809	0.513149
5	Random Forest Tuned	0.048522	0.095238	0.904762	0.000374	0.000748	0.476473
6	XGBoost	0.075557	0.095238	0.904762	0.000385	0.000770	0.489996
7	AdaBoost	0.091850	0.047619	0.952381	0.000413	0.000825	0.521946

Table 5: Model Results Trained with Undersampled Data

	Model	Accuracy	FalseNegRate	Recall	Precision	F1 Score	ROC_AUC
0	KNN	0.999606	1.000000	0.000000	0.000000	0.000000	0.500000
1	Logistic Regression	0.704708	0.619048	0.380952	0.000508	0.001015	0.542894
2	Decision Tree	0.999306	1.000000	0.000000	0.000000	0.000000	0.499850
3	Decision Tree Tuned	0.463974	0.285714	0.714286	0.000524	0.001048	0.589081
4	Random Forest	0.999606	1.000000	0.000000	0.000000	0.000000	0.500000
5	Random Forest Tuned	0.999606	1.000000	0.000000	0.000000	0.000000	0.500000
6	XGBoost	0.999569	1.000000	0.000000	0.000000	0.000000	0.499981
7	AdaBoost	0.806513	0.857143	0.142857	0.000291	0.000581	0.474816

Table 6: Model Results Trained with Raw Data

	Model	Accuracy	FalseNegRate	Recall	Precision	F1 Score	ROC_AUC
0	KNN	0.999531	1.0	0.0	0.000000	0.000000	0.499962
1	Logistic Regression	0.006018	0.0	1.0	0.000396	0.000792	0.502813
2	Decision Tree	0.998913	1.0	0.0	0.000000	0.000000	0.499653
3	Random Forest	0.999606	1.0	0.0	0.000000	0.000000	0.500000
4	XGBoost	0.999588	1.0	0.0	0.000000	0.000000	0.499991
5	AdaBoost	0.994094	1.0	0.0	0.000000	0.000000	0.497243

Table 7: Model Results Trained with Oversampled Data

	Model	Accuracy	FalseNegRate	Recall	Precision	F1 Score	ROC_AUC
0	Basic Neural Network	0.999606	1.000000	0.000000	0.000000	0.000000	0.500000
1	Weighted Neural Network	0.407203	0.190476	0.809524	0.000537	0.001074	0.608284
2	Undersampled Neural Network	0.554306	0.380952	0.619048	0.000547	0.001093	0.586664
3	Oversampled Neural Network	0.998800	1.000000	0.000000	0.000000	0.000000	0.499597

Table 8: ANN Results

	Model	Accuracy	FalseNegRate	Recall	Precision	F1 Score	ROC_AUC
0	Convolutional Neural Network	0.999606	1.000000	0.000000	0.000000	0.000000	0.500000
1	Weighted Convolutional Neural Network	0.869190	0.809524	0.190476	0.000574	0.001145	0.529967
2	Undersampled Convolutional Neural Network	0.330971	0.238095	0.761905	0.000448	0.000896	0.546353
3	Oversampled Convolutional Neural Network	0.999119	1.000000	0.000000	0.000000	0.000000	0.499756

Table 9: CNN Results

	Model	Accuracy	FalseNegRate	Recall	Precision	F1 Score	ROC_AUC
0	GRU RNN	0.999606	1.000000	0.000000	0.000000	0.000000	0.500000
1	Undersampled GRU RNN	0.817256	0.619048	0.380952	0.000821	0.001639	0.599190
2	Oversampled GRU RNN	0.999156	1.000000	0.000000	0.000000	0.000000	0.499775
3	Weighted GRU RNN	0.628738	0.761905	0.238095	0.000253	0.000505	0.433494

Table 10: GRU Results

Discussion and Model Comparison:

Beginning with non-neural network models, one can see in the undersampled results that the overall accuracy of each model is very poor. However, as the experiment moves from the simple KNN model to more robust ensemble models, the recall scores increase substantially with the AdaBoost, Random Forest, and Tuned Decision Tree giving the best recall score at around 0.95. Nevertheless, with such low accuracy and precision, it seems undersampling with these models gave too much weight to the minority class. In our raw data results for non-neural networks, there is some interesting data. For the most part, the models tended to overfit or simply predict the majority class for all examples even with class weights. However, with Logistic Regression, the Tuned Decision Tree, and AdaBoost the class weights are shown to potentially have an impact. All three of these models gave us a Recall score greater than zero and specifically the Tuned Decision Tree was able to breach our threshold of 70%. Finally, with the oversampled non-neural networks it seems that all of the models aside from Logistic Regression overfit towards the majority class. Curiously, Logistic Regression actually seemed to lean heavily towards the minority class with a perfect recall score (1.0).

Moving on to neural networks, the first neural network that was built was a basic artificial neural network. Here only the weighted ANN gives a recall score above the 70% threshold (around 0.80). Both the ANN trained with raw data (without class weights) and oversampled data overfit to the majority class similar to the previous models. For exploratory purposes, a CNN and GRU RNN was built and tested with our data. The results show that the undersampled CNN gives a recall score above 70% but as expected these neural networks did not perform well overall. With RNNs being most effective with natural language processing

and CNNs being most effective with image data, it is not surprising that they did not perform well on this numerical classification problem. For the specific neural network architectures, reference Appendix A.

When looking at all the different models during the experimentation phase, and considering the primary metric of recall along with the identified secondary metrics, the **weighted neural network** looks to be the best performing model. This model, compared to the other models that were tested, has a high recall score (0.80) while not completely sacrificing accuracy (0.41) compared to the undersampled models. It also has one of the highest F1 scores compared to the other tested models.

As a final note, it is recognized that a subset of models were able to get over our desired threshold of 70% recall. However, with a closer look at the secondary metrics for each of the models the performance was overall not very acceptable. While high recall scores are seen, very low precision and F1 scores are seen as well. Even with a focus on recall, the lack of precision is not entirely desirable when looking to predict injuries. Recall is the focus as one would not want to overlook scenarios where injuries will happen but with such low precision a large amount of plays would need to be unnecessarily evaluated for potential injury.

Constraints:

With a one-man team, time was a big constraint in terms of testing the full breadth of machine learning models available. Some complexity had to be sacrificed in order to try various different data augmentation techniques along with a variety of machine learning models. WIth more time or a larger team, it would have been beneficial to dig deeper into each type of machine learning model, particularly the neural networks and investigate more complex architectures.

Software Standards:

Software:

- Python 3.9.13

Python Packages:

- Numpy 1.21.5
- Pandas 1.4.4
- Seaborn 0.11.2
- Tensorflow 2.9.1
- Sklearn 1.0.2
- Imblearn 0.0

Limitations of the Study:

Ultimately, the heavy imbalance of the dataset proved to be a very big hurdle. Furthermore, with the dataset being sourced from Kaggle, we were limited to the set of features that were provided as well as limited to the set of players and seasons provided. It was clear that the set of features did not differentiate the minority class enough from the majority class so more data granularity would be beneficial.

Future Work:

In a future study, it would be beneficial to gather more injury data and perhaps expand the scope beyond just lower leg injuries to balance the data a bit more. Furthermore, it would be important to investigate additional features specific to the player such as height, weight, number of injuries, etc. With injury data, there will always be an imbalance compared to the total number of plays during an NFL game but with more granular feature information one can hope to build a better overall machine learning model.

Appendix A:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 16)                672

 batch_normalization (BatchN  (None, 16)               64
 ormalization)

 dense_1 (Dense)             (None, 24)                408

 dense_2 (Dense)             (None, 24)                600

 dropout (Dropout)           (None, 24)                0

 dense_3 (Dense)             (None, 24)                600

 dense_4 (Dense)             (None, 1)                 25

=================================================================
Total params: 2,369
Trainable params: 2,337
Non-trainable params: 32
_________________________________________________________________
```

Figure 2: ANN Architecture

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv1d (Conv1D)              (None, 39, 64)            256

 batch_normalization_1 (Batc  (None, 39, 64)           256
 hNormalization)

 max_pooling1d (MaxPooling1D  (None, 19, 64)           0
 )

 conv1d_1 (Conv1D)            (None, 17, 64)            12352

 max_pooling1d_1 (MaxPooling  (None, 8, 64)            0
 1D)

 flatten (Flatten)            (None, 512)               0

 dense_5 (Dense)              (None, 64)                32832

 dense_6 (Dense)              (None, 1)                 65

=================================================================
Total params: 45,761
Trainable params: 45,633
Non-trainable params: 128
_________________________________________________________________
```

Figure 3: CNN Architecture

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 gru (GRU)                    (None, 64)                12864

 batch_normalization_2 (Batc  (None, 64)               256
 hNormalization)

 dense_7 (Dense)              (None, 64)                4160

 dense_8 (Dense)              (None, 1)                 65

=================================================================
Total params: 17,345
Trainable params: 17,217
Non-trainable params: 128
_________________________________________________________________
```

Figure 4: GRU RNN Architecture