Technical Guide

Lower Limb Sports Injury Prediction and Prevention

CA400

Lorcan Dunne - 19311511
Dylan Bagnall - 20413066

Table of Contents

Abstract

The National Football League (NFL) faces a major challenge in the form of lower limb injuries, which have a massive effect on player performance, player wellbeing and teams' output. The goal of this project is to create a predictive model for lower limb injuries using machine learning techniques.We use a dataset with information from 2 regular NFL seasons, that contains 267,005 player-plays and 105 cases of lower limb injuries.Through very thorough data pre-processing, feature engineering, hyperparameter tuning and cross-validation, we established a robust model. Our research shows how AI-powered technologies can improve professional players' injury prevention tactics.

1. Glossary

NFL - National Football League, The major professional American football league in the United States

CSV - Simple text format storing tabular data (rows and columns), with commas separating values.

Decision Tree - Model that makes predictions using a tree-like structure of decision rules. Easy to understand and visualise.

Random Forest - Combines multiple decision trees for improved predictions. Reduces overfitting and handles complex data well.

XGBoost - Optimised gradient boosting algorithm that builds sequential decision trees to correct previous errors. Known for speed and performance.

RandomisedSearchCV - Randomised Search Cross-Validation

GridSearchCV - Grid Search Cross-Validation

2. General Description

2.1. Motivation

The NFL serves as a prime example of a high-intensity sport where lower limb injuries are a major concern. These injuries not only have devastating consequences for players' physical and mental well-being but they can potentially cut careers short. On top of this, injuries can translate into substantial financial burdens for teams and the NFL alike, encompassing medical expenses, rehabilitation costs, the need for player replacements, and even potential revenue losses due to diminished fan engagement. Moreover, injuries disrupt team dynamics, hinder on-field performance, and can significantly impact a team's ability to compete at its best.

Therefore, injury prediction and prevention is crucially important within the NFL. By leveraging the power of machine learning to identify key risk factors and predict injury susceptibility, several significant benefits can be realised. Injury models can facilitate the development of targeted prevention programs, addressing the specific vulnerabilities of high-risk athletes and customising training regimens accordingly. These models enable early intervention for players demonstrating signs of increased risk, thereby optimising recovery periods and minimising the chances of severe injury. Additionally, teams can utilise injury predictions to inform strategic decision-making processes such as roster management, coaching tactics, and potentially even contract negotiations. Overall, a robust injury prediction model empowers team medical staff and coaches with data-driven insights. This fosters a proactive approach to athlete health in the NFL, protecting players, enhancing competitive edge, and ultimately increasing the overall fan experience.

2.2. Business Context

Injuries directly impact a team's bottom line through medical costs, rehabilitation expenses, and the need to replace injured players. The loss of star players due to injury can also translate into decreased ticket sales, merchandise revenue, and overall fan engagement, potentially leading to significant financial losses for the team and the NFL. Additionally, injuries can disrupt a team's strategic plans and on-field performance, hindering their ability to achieve success and the associated financial benefits. By investing in machine learning-driven injury prediction and prevention, NFL teams can gain a competitive edge. Proactive strategies informed by these models can optimise player health, maintain roster strength, and ultimately contribute to greater on-field success and financial gains

2.3. Data Overview

Dataset Source: NFL 1st and Future - Analytics (https://www.kaggle.com/competitions/nfl-playing-surface-analytics/data)

File Format: CSV

The dataset provided for analysis has 250 complete player in-game histories from two subsequent NFL regular seasons. We used two different files in .csv format which document injuries and player-plays. Over the course of the two seasons' regular season games, 105 lower-limb injuries occurred. The information on these injuries is contained in the injury record file in .csv format. Using the PlayerKey, GameID, and PlayKey fields, injuries can be connected to particular records in the PLayList file. The 267,005 player-plays that comprise the dataset are detailed in this PlayList file. PlayerKey and GameID fields index each play. The player's designated roster position, the type of stadium, the type of pitch, the style of play, the play's position, and the position group are included in this file.

InjuryRecord

PlayerKey	GameID	PlayKey	BodyPart	Surface	DM_M1	DM_M7	DM_M28	DM_M42
39873	39873-4	39873-4-32	Knee	Synthetic	1	1	1	1
46074	46074-7	46074-7-26	Knee	Natural	1	1	0	0
36557	36557-1	36557-1-70	Ankle	Synthetic	1	1	1	1
46646	46646-3	46646-3-30	Ankle	Natural	1	0	0	0
43532	43532-5	43532-5-69	Ankle	Synthetic	1	1	1	1
41145	41145-2	41145-2-60	Knee	Natural	1	0	0	0
46014	46014-10	46014-10-22	Ankle	Synthetic	1	1	1	1
44860	44860-5	44860-5-52	Knee	Natural	1	1	0	0
44806	44806-7	44806-7-61	Knee	Synthetic	1	0	0	0
45962	45962-8	45962-8-40	Ankle	Synthetic	1	1	0	0
46331	46331-4	46331-4-44	Ankle	Synthetic	1	1	1	1
36621	36621-13	36621-13-58	Foot	Natural	1	1	1	1
44492	44492-3	44492-3-23	Ankle	Natural	1	1	1	1
43505	43505-2	43505-2-49	Foot	Natural	1	1	1	1
41094	41094-1	41094-1-55	Knee	Natural	1	1	1	1
40474	40474-1	40474-1-8	Knee	Synthetic	1	0	0	0

Fig. 1: Example Injury Record File

PlayList

PlayerKey	GameID	PlayKey	RosterPosition	PlayerDay	PlayerGame	StadiumType	FieldType	Temperature	Weather	PlayType	PlayerGamePlay	Position	PositionGroup
26624	26624-1	26624-1-1	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	1	QB	QB
26624	26624-1	26624-1-2	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	2	QB	QB
26624	26624-1	26624-1-3	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Rush	3	QB	QB
26624	26624-1	26624-1-4	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Rush	4	QB	QB
26624	26624-1	26624-1-5	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	5	QB	QB
26624	26624-1	26624-1-6	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Rush	6	QB	QB
26624	26624-1	26624-1-7	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	7	QB	QB
26624	26624-1	26624-1-8	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	8	QB	QB
26624	26624-1	26624-1-9	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Rush	9	QB	QB
26624	26624-1	26624-1-10	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	10	QB	QB
26624	26624-1	26624-1-11	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Rush	11	QB	QB
26624	26624-1	26624-1-12	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	12	QB	QB
26624	26624-1	26624-1-13	Quarterback	1	1	Outdoor	Synthetic	63	Clear and warm	Pass	13	QB	QB

Fig. 2: Example Playlist File

As can be seen in figure 2, the PlayList.csv file is significantly larger than the Injury.csv file, with 267,005 entries spanning 14 columns. It provides extensive play-by-play data including player identifiers, game identifiers, and specific play identifiers. It also includes player positions, game sequence, the type of stadium and field, weather conditions, and temperature. Crucial to understanding game conditions are the StadiumType and Weather fields, though these exhibit some missing values, which could impact detailed environmental analysis. The PlayType field, describing the nature of each play, also has minor missing data. We use this dataset paired with the Injury file to have a comprehensive dataset on the players and their workload during the season.

2.4. Methodology

Our methodology begins with data cleaning to ensure the integrity of the datasets, InjuryRecord.csv and PlayList.csv. Fortunately, the Injury Record file has no missing data, however the PlayList file had to be cleaned, this included addressing missing values, correcting inconsistencies, and standardising data formats. The datasets are then merged on relevant keys such as PlayerKey and GameID to enrich the injury records with play-by-play context, enabling a holistic analysis. Feature engineering is employed usings out Dylannumplays file and this is done to develop new variables that better capture the nuances and interactions between different data aspects, such as player workload and environmental stress (combining weather conditions and temperature).

To address the challenge of class imbalance that we observed in our injury datasets, we implement random undersampling to prevent overfitting and ensure our models generalise well to new, unseen data. For our predictive modelling, we utilise a suite of machine learning algorithms including Decision Trees, Random Forest, and XGBoost. These models are chosen for their ability to handle the complex, non-linear relationships and feature interactions within our data. Validation of the models is performed using k-fold cross-validation, which enhances the robustness of our predictions by ensuring that each observation from the original dataset has the chance to appear in both the training and validation sets. Throughout the modelling process, we also analyse feature importance, which helps us to identify the most influential factors contributing to injuries.

2.5. Limitations and Constraints

This project faces several limitations and constraints that could impact the findings and their applicability. First, we are constrained by time, with only seven months allocated to complete the project, which may limit the depth of analysis and model refinement. Additionally, the datasets cover only two seasons, restricting our analysis to a narrow time frame and possibly affecting the generalisability of our results. Furthermore, external factors such as player conditioning, pre-existing health conditions, and other unrecorded environmental variables are not accounted for, which could influence the accuracy of our predictions. Together, these limitations necessitate careful consideration when interpreting the outcomes of the project and when applying the insights derived from our analysis.

3. Data Preparation

3.1. Data Cleaning

To ensure data quality and to prepare the datasets for modelling, we started the data cleaning and preparation process. This involved merging the two original CSV files, handling missing values, standardising inconsistent data, and transforming features.

We addressed inconsistencies by cleaning specific columns within the data. This can be seen in the figure below, where we removed '-' symbols from the GameID and PlayKey columns to ensure they were interpreted correctly during analysis.

```
19    # Remove '-' symbol from GameID and PlayKey columns
20    merged_df['GameID'] = merged_df['GameID'].str.replace('-', '')
21    merged_df['PlayKey'] = merged_df['PlayKey'].str.replace('-', '')
```

Fig. 3: Data cleaning

On top of this we also standardised weather descriptions, mapping various terms (e.g., "Partly Sunny", "Partly Cloudy") into broader categories like 'dry', 'wet', or 'indoor' to simplify analysis. Missing values in key columns were addressed using zero-fill for specific cases like the 'Days_Out' column (representing whether a player was injured or not). We did this in order to create a binary like feature, to make it easier to interpret and learn patterns from. We noticed that when a game was played indoors the temperature was set to -999 degrees fahrenheit, in order to fix this we replaced every value that was “-999” with the average indoor temperature at NFL games which is “70”. We did this because we knew that having extreme values of “-999” would skew our data and ultimately affect our models performance.

```
124
125    # Fill missing values in 'Days_Out' column with 0 (for non-injured players)
126    df['Days_Out'].fillna(0, inplace=True)
127
```

Fig. 4. Filling missing values

3.2. Feature Engineering

We engineered several new features to enhance the predictive power of our model. We hypothesised that workload metrics and environmental interactions would be significant predictors of injury risk, based on our established knowledge as fans of the NFL. As can be seen in the below figure, we developed metrics like WorkloadIncrease, DaysRest and WorkloadStress to quantify the relationship between a player's physical workload in previous games and their potential injury risk. These features were derived from existing data on game participation such as PlayKey_y and NumPlays, as well as the temporal aspect of games like PlayerDay and GameID.

```
246    # Calculate the number of days between games for each player
247    df['DaysRest'] = df.groupby('PlayerKey')['PlayerDay'].diff().fillna(0)
248
249    # Calculate the change in workload between consecutive games
250    df['WorkloadIncrease'] = df.groupby('PlayerKey')['PlayKey_y'].diff().fillna(0)
251
252    # Calculate the workload stress as the product of workload increase and inverse of days rest
253    df['WorkloadStress'] = df['WorkloadIncrease'] * (1 / df['DaysRest'])
254
```

Fig. 5: Feature Engineering

On top of this we also created interaction features that model the combined effect of field type (FieldType) with both weather conditions (Weather_Category) and temperature (Temp_... bins). These features aim to uncover more complex patterns regarding how playing conditions might influence injury likelihood. We did this because we believe that injuries rarely occur due to a single isolated factor. Environmental conditions often play a significant role in how a player's body responds to the physical demands of The NFL and by adding these interaction features we aimed to explore these combined effects, which is closer to how things happen in reality. We also believed that by doing this we could uncover some hidden patterns in relation to the weather. For example, Wet conditions on a natural grass field might present a very different injury risk than wet conditions on artificial turf. Aswell as that extreme heat or cold combined with a player's workload could

influence their susceptibility to injury in a way that looking solely at temperature or workload wouldn't reveal.

```python
def create_comprehensive_interactions(df):
    # Weather interactions (same as before)
    for weather_type in ['wet', 'damp', 'dry', 'indoor']:
        for field_type in ['Natural', 'Synthetic']:
            feature_name = f"{field_type}_{weather_type}"
            df[feature_name] = (df['FieldType'] == field_type) & (df['Weather_Category'] == weather_type)

    # Temperature bin creation
    df['Temp_VeryCold'] = (df['Temperature'] < 30)
    df['Temp_Cold'] = (df['Temperature'] >= 30) & (df['Temperature'] < 50)
    df['Temp_Mild'] = (df['Temperature'] >= 50) & (df['Temperature'] < 80)
    df['Temp_Hot'] = (df['Temperature'] >= 80) & (df['Temperature'] < 95)
    df['Temp_VeryHot'] = (df['Temperature'] >= 95)

    # Temperature bin interactions
    for temp_bin in ['Temp_VeryCold', 'Temp_Cold', 'Temp_Mild', 'Temp_Hot', 'Temp_VeryHot']:
        for field_type in ['Natural', 'Synthetic']:
            feature_name = f"{field_type}_{temp_bin}"
            df[feature_name] = (df['FieldType'] == field_type) & df[temp_bin]

    return df

```

Fig. 6: Example of Temperature bin creation from previous page

3.3. Data Splitting

To ensure unbiased model evaluation and prevent overfitting, we split our preprocessed data into training and testing sets. We used a stratified random split with a test size of 20%, we chose 20% because then the majority of our data is dedicated to training, which gives our model ample opportunity to learn complex patterns, which will be needed given the complexity of the problem we are trying to solve. This stratification is crucial for injury prediction, as injuries are likely a minority class, and a simple random split could lead to misleading performance evaluations. We set a random_state of 42 to ensure reproducibility of the split. This splitting approach ensures the model is trained on a representative dataset and its performance is rigorously evaluated on unseen data, simulating real-world deployment.

```python
# Target variable
y = merged_df['Injured']

# Split the data into training and testing sets with stratification
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

Fig. 7: Data Splitting, shows stratification, test size choice, and reproducibility

4. Modelling

4.1. Algorithms

Having prepared our data through cleaning, feature engineering, and splitting, we now turn our attention to the modeling process. Here, we explore the selection and training of machine learning algorithms to identify patterns within the data that can predict a player's risk of injury. The three models we chose to use in this project are Decision Trees, Random Forest and XGBoost. We started with Decision trees because Decision trees learn simple if-then-else rules from the data to make predictions. For example, a rule might be "If a player's workload increased by more than 20% and they played on artificial turf, they have a higher risk of injury." These rules can help understand what factors might contribute to injury. However we knew that Single decision trees easily overfit and this could cause a problem. So then we decided to also look at Random Forests which combine many decision trees to make better predictions. Each tree is trained on a slightly different version of the data. The final prediction is made by combining the results of all the trees, which makes the model more reliable and less likely to make mistakes due to quirks in the data. Finally we decided to use XGBoost, which is an especially powerful type of decision tree model. It builds a series of trees where each new tree tries to fix the mistakes made by the ones before it. XGBoost also uses special techniques to prevent overfitting. Which in turn leads to very accurate predictions, even on challenging problems like predicting injuries.

4.2. Model Training and Hyperparameter Tuning

Starting with our Decision tree model we used GridSearchCV which is a technique used to find the optimal hyperparameter settings for our model. As you can see in the below figure, we created a dictionary named param_grid for decision trees. This dictionary defines three hyperparameters;

- **max_depth:** which controls the maximum depth a tree can grow, which affects its complexity.
- **min_samples_split:** This sets the minimum number of samples required to split a node in the tree, which prevents overfitting.
- **criterion:** This determines how the tree chooses the best splitting feature.

When using GridSearchCV systematically tries out all combinations of hyperparameter values from your defined grid. After doing this, GridSearchCV picks the hyperparameter combination that achieved the best score on the validation sets.

```
# Define the parameter grid
param_grid = {
    'max_depth': [None, 5, 10, 15],
    'min_samples_split': [2, 4, 20],
    'min_samples_leaf': [3, 4, 8],
    'criterion': ['gini', 'entropy']
}

# Create the grid search object
grid_search = GridSearchCV(
    DecisionTreeClassifier(random_state=42, class_weight='balanced'),
    param_grid,
    cv=5,
    scoring='f1_macro',
    verbose=1
)

```

Fig. 8: The use of GridSearchCV in decision tree model

For our Random Forest and XGBoost models, we used RandomisedSearchCV which is an alternative to GridSearchCV for hyperparameter tuning. RandomisedSearchCV is slightly different from GridSearchCV because Instead of trying all combinations in a parameter grid, RandomisedSearchCV randomly samples a specified number of combinations. This can be more efficient when exploring a large search space, which is ideal for our more complex models to explore a wider search space and find the most optimal parameters. When using RandomisedSearchCV we defined distributions for each hyperparameter we wanted to tune, we did this in both our Random Forest model and our XGBoost model and an example of which can be seen in the figure below.

```

# Define the parameter grid for RandomizedSearch
param_grid = {
    'max_depth': [3, 5, 7, 9],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'n_estimators': [150, 200, 250, 400],
    'subsample': [0.65, 0.7, 0.85, 0.9],
    'colsample_bytree': [0.5, 0.6, 0.7, 0.8],
    'gamma': [0, 0.1, 0.2, 0.3],
    'min_child_weight': [1, 5, 10]
}
```

Fig. 9: Distributions from our XGBoost Model

After having determined the optimal hyperparameters through GridSearchCV and RandomizedSearchCV, we then retrain each model on the complete training dataset using those best settings. We did this to ensure that the models have access to the maximum amount of data for learning patterns.

After this is complete we then ensure that our models are evaluated based on their ability to generalise to unseen data, reducing the risk of selecting overfit models. We further integrate K-fold cross-validation within our codes as can be seen below where we use 5-fold cross-validation to evaluate our model's performance. K-fold cross-validation is a technique for getting a more reliable performance assessment of machine learning models. It starts by dividing your dataset into K equally-sized folds. Then, for each fold, the model is trained on all the other folds combined, and its performance is evaluated on the held-out fold. This process repeats K times, ensuring each fold gets a turn being the test set. Ultimately, the performance metrics from each iteration are averaged. This helps combat overfitting by giving a more realistic picture of how the model might perform on totally new data, and it's often used for hyperparameter tuning to find the best model configuration.

```
75
76  # Evaluate the classifier using k-fold cross-validation
77  skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
78  for train_index, test_index in skf.split(X_resampled, y_resampled):
79      X_train, X_test = X_resampled.iloc[train_index], X_resampled.iloc[test_index]
80      y_train, y_test = y_resampled.iloc[train_index], y_resampled.iloc[test_index]
81
82      xgb_classifier.fit(X_train, y_train)
83      y_pred = xgb_classifier.predict(X_test)
84
85      accuracy = accuracy_score(y_test, y_pred)
86      print("Accuracy:", accuracy)
87      print("Classification Report:")
88      print(classification_report(y_test, y_pred))
89      print("Confusion Matrix:")
90      print(confusion_matrix(y_test, y_pred))
```

Fig. 10: Evaluating Classifier using k-fold

4.3. Feature Importance Analysis

To understand the factors driving injury risk predictions, we examine feature importances. Our XGBoost model provides built-in feature importance scores based on the 'gain' metric, which reflects how much each feature contributed to improving the model's predictions during training.

```
70  # Feature importances
71  feature_importances = xgb_classifier.feature_importances_
72  feature_importance_df = pd.DataFrame({'Feature': X_resampled.columns,
73                                        'Importance': feature_importances}).sort_values(by='Importance', ascending=False)
74  print("Feature Importances:")
75  print(feature_importance_df)
```

Fig. 11: Feature Importance

By getting these feature importances we can pinpoint key risk factors that allow coaches and medical staff to potentially modify training regimens, focus on recovery strategies, or even adjust playing conditions to reduce injury likelihood. On top of this it improves the trust of the model by increasing its explainability by knowing the features driving predictions, that makes the model less of a "black box" and ideally will help build trust among sports medicine professionals. Feature importance also helps with collecting future data, it can highlight different types of data that might be beneficial for building even better predictive models in the future.

5. GUI Development

5.1. Overview

The feature is a versatile tool tailored specifically for the analysis and prediction of injuries using the machine learning models that have been trained. To make it easier to explore data on sports injuries in-depth, it combines interactive user inputs, model evaluation, and data processing. This GUI provides a clearer and more concise way to interpret and analyse the models performance.

5.2. Model Evaluation

The GUI consists of three main components:

First is the model evaluation window. The purpose of this window is to load the pre-trained models, evaluate them and display the performance metrics.

```
class ModelEvaluationWindow(QWidget): # This class is used to evaluate the model
    def __init__(self, model_paths):
        super().__init__()
        self.initUI()
        self.X, self.y = None, None

    def initUI(self): # This function initializes the UI
        layout = QVBoxLayout(self)
        self.model_dropdown = QComboBox()
        self.model_dropdown.addItems(["XGBoost", "Decision Tree", "Random Forest"]) # Add the models to the dropdown
        layout.addWidget(self.model_dropdown)

        self.load_button = QPushButton('Load and Evaluate Data') # Create a button to load and evaluate the data
        self.load_button.clicked.connect(self.load_and_evaluate_data)
        layout.addWidget(self.load_button)
```

Fig. :

As you can see from the code this is the initialisation of the model evaluation window. It uses ‘QWidget’, indicating that it can be used as a window or a component of an overall larger PyQt application. In the ‘def initUI(self)’, the layout and the UI components of the model evaluation window are created. This consists of the machine learning models being added to a dropdown list for the users selection. This is done by initialising the ‘model_dropdown’ feature. Load Button: A ‘QPushButton’ labelled 'Load and Evaluate Data' is created. This button is connected to the load_and_evaluate_data method, which is responsible for handling the loading of the dataset and initiating the evaluation process using the selected model.

Once the models are loaded into the dropdown feature the evaluation function is used. This obtains all the metrics of the models performance and displays them.

```
def evaluate_model(self, X, y): # This function evaluates the model
    model_name = self.model_dropdown.currentText().lower().replace(" ", "_")
    model_path = f"{model_name}_search.joblib"
    search = load_model(model_path)  # This loads the model
    if search:
        model = search.best_estimator_
        y_pred = model.predict(X)
        best_params = getattr(search, 'best_params_', 'Not available')
        best_score = getattr(search, 'best_score_', 'Not available')
        feature_importances = dict(zip(X.columns, getattr(model, 'feature_importances_', [])))
        self.display_results(X, y, y_pred, best_params, best_score, feature_importances, search)
    else:
        self.summary_text.setText("Failed to load the model.") # Display an error message if the model fails to load
```

From the code above you can see that the ‘model_path’ variable is initialised to hold the .joblib file of the selected model which is then loaded into the ‘search’ variable.

Now that the ‘search’ variable contains the model it can be called to retrieve the results of the model using ‘.best_estimator_’, and ‘.predict()’.

5.3. Filter/Search Data

The second window of the GUI is the ‘Filter/Search Data’ window. This allows the user to have a deeper look into the dataset. It provides a filter feature which allows the user to select a specific feature from the dropdown list, and input the feature they wish to search in the provided box.

```
def apply_filter(self):
    column = self.filterColumn.currentText()
    if column == "Select column" or not column:  # Check if a valid column is selected
        QMessageBox.information(self, "Error", "Please select a column to search.")
        return

    value = self.filterValue.text().strip()
    filtered_data = self.data[self.data[column].astype(str).str.contains(value, case=False)]
    self.update_display(filtered_data)

def clear_filter(self):
    self.update_display(self.data)
```

The functions above create this functionality. The text inputted into the provided feature box is retrieved using ‘self.filterValue.text().strip()’. This is stored and called in the ‘update_display’ function to display the filtered dataset. The ‘clear_filter’ function allows the user to clear the filter and see the original dataset, by using the ‘update_display’ function calling the original data. The ‘update_display’ function can be seen below.

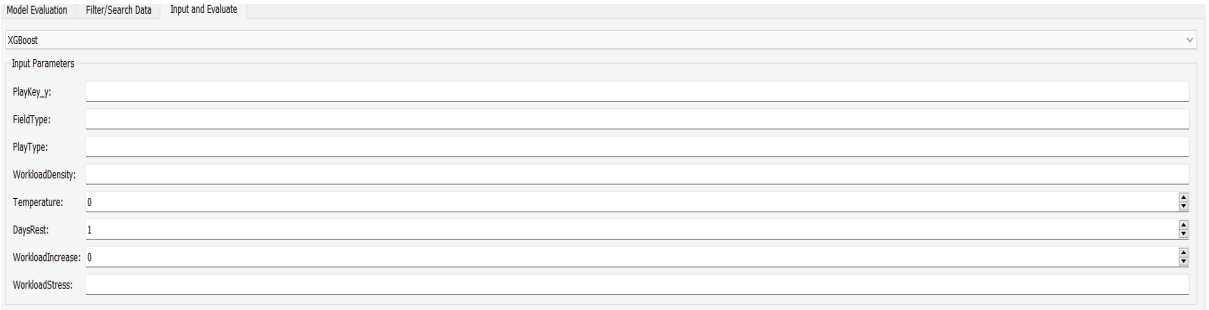
```
def update_display(self, data):
    if data.empty:  # Check if the DataFrame is empty
        QMessageBox.information(self, "Search Result", "Input not found.")
        self.results_table.setRowCount(0)  # Clear previous results if any
        self.results_table.setColumnCount(0)
    else:
        self.results_table.clear()
        self.results_table.setRowCount(len(data))
        self.results_table.setColumnCount(len(data.columns))
        self.results_table.setHorizontalHeaderLabels(data.columns.tolist())

        for row_index, row_data in enumerate(data.itertuples(index=False), start=0):
            for column_index, value in enumerate(row_data):
                self.results_table.setItem(row_index, column_index, QTableWidgetItem(str(value)))

        self.results_table.resizeColumnsToContents()
```

5.4. Input Form

Finally the input form window is created to allow the user to input their own data in the features to see what outcome the trained models predict.



There are text boxes created beside the features to allow the user to input. The data that is inputted is then retrieved and stored into an ‘input_data’ variable. This is then used to create a dataframe for the models to read. Pre-processing is used on the dataframe before the ‘model.predict()’ function is called on the new dataframe. The results of the prediction are then stored in a new variable and called in the display results function displaying whether or not the model predicted the player to be injured or not injured. These steps can be seen in the evaluate function below.

```
def evaluate(self): # This function evaluates the model
    model_key = self.model_dropdown.currentText()
    model = self.models.get(model_key)

    if not model:
        self.results_text.setText("Selected model is not available. Please check model loading.")
        return

    input_data = {feature: float(widget.text()) if isinstance(widget, QSpinBox) or widget.isReadOnly() else widget.text()
                  for feature, widget in self.input_fields.items() if widget.text()}

    try: # Try to predict the result
        input_df = pd.DataFrame([input_data])
        preprocessed_data = self.local_preprocess(input_df)

        result = model.predict(preprocessed_data)[0]
        prediction_prob = model.predict_proba(preprocessed_data)[0][1] * 100  # as percentage
        model_accuracy = self.model_accuracies.get(model_key, 'Unknown')  # Fetch accuracy dynamically

        result_text = (
            f"Predicted Result: {'Injured' if result == 1 else 'Not Injured'}\n"
            f"Model Used: {model_key}\n"
            f"Model Accuracy: {model_accuracy}%"
        )
        self.results_text.setText(result_text)
        self.update_graph(model_accuracy)  # Assuming you have a method to update some graphical display

    except Exception as e:
        self.results_text.setText(f"Error in prediction: {str(e)}")
```

6. Testing

6.1. Unit Testing

We employ several unit tests to ensure the individual components of our system function correctly. The following tests are employed in our XGBoost and Random Forest test files Data integrity is verified by the test_file_loading test, which validates correct loading and parsing of data files into Pandas DataFrames and Series. The test_required_columns test ensures all necessary data fields are present, safeguarding against missing feature issues. For the GUI, the test_initial_state test guarantees the InputFormWindow initialised properly, including the model dropdown, providing a stable starting point for users. The test_model_loading test simulates model loading and integration into the GUI's model store. While testing GUI interactions was complex, to combat this we directly invoke the evaluate method with test_evaluate_method_directly to confirm its basic functionality and ability to update GUI output displays, and this work-around can be seen below.

```
49      @patch('gui.load_model')
50      def test_evaluate_method_directly(self, mock_load_model):
51          input_form_window = self.main_window.input_form_window
52          mock_model = MagicMock()
53          mock_model.predict.return_value = [0]
54          mock_model.predict_proba.return_value = [[0.7, 0.3]]
55          mock_load_model.return_value = mock_model
56          input_form_window.models['XGBoost'] = mock_model
57          for widget_key, widget in input_form_window.input_fields.items():
58              if isinstance(widget, QLineEdit):
59                  widget.setText('1')
60              elif isinstance(widget, QSpinBox):
61                  widget.setValue(10)
62          input_form_window.evaluate()
63          expected_output = "Predicted result: Not Injured"
64          self.assertIn(expected_output, input_form_window.results_text.toPlainText())
65          self.assertIn("Prediction probability: 0.30", input_form_window.results_text.toPlainText())
66
```

6.2. Integration Testing

Our integration tests focus on the smooth interaction between components. The test_model_training_runs that can be seen on the next page integrates data loading, preprocessing, and XGBoost model training, ensuring the complete pipeline operates without errors. To assess model stability and adaptability to real-world data variations, the test_cross_validation_runs test employs k-fold cross-validation on the XGBoost model. This verifies the model's ability to train and produce reliable predictions across different subsets of data.

```python
def test_model_training_runs():
    X_filepath = 'src/tests/Xtest_data.csv'
    y_filepath = 'src/tests/ytest_data.csv'
    X, y = read_data(X_filepath, y_filepath)
    X, y = preprocess_data(X, y)
    model = RandomForestClassifier()
    model.fit(X, y)
    print("passed training runs")

def test_cross_validation_runs():
    X_filepath = 'src/tests/Xtest_data.csv'
    y_filepath = 'src/tests/ytest_data.csv'
    X, y = read_data(X_filepath, y_filepath)
    X, y = preprocess_data(X, y)
    model = RandomForestClassifier()
    skf = StratifiedKFold(n_splits=2)

    for train_index, test_index in skf.split(X, y):
        X_train, X_test = X.iloc[train_index], X.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]
        model.fit(X_train, y_train)
        print("passed cross validation test")

```

6.3. System Testing

We also conduct a full pipeline test (TestFullPipeline) to evaluate the system's overall functionality, this can be seen in the figure below. This end-to-end test encompasses data loading (read_data), data preprocessing (preprocess_data), model training, and prediction generation with the XGBoost model. This comprehensive test verifies that all components integrate seamlessly and the complete pipeline operates cohesively, from raw data input to final predictions. This system-level validation is essential for assessing the deployment-readiness of our injury prediction system.

```python
def test_full_pipeline():
    # Load data
    X, y = read_data('src/tests/Xtest_data.csv', 'src/tests/ytest_data.csv')

    # Preprocess data
    X_processed, y_processed = preprocess_data(X, y)

    # Train model
    model = xgb.XGBClassifier()
    model.fit(X_processed, y_processed)

    # Make predictions
    predictions = model.predict(X_processed)

    # Evaluate predictions
    assert predictions is not None, "No predictions were made"
    assert len(predictions) == len(y_processed), "Number of predictions does not match number of samples"

```

7. Learning Outcomes

Overall this experience trying to create the predictive models to the highest performance we could get was a massive learning experience. Firstly balancing the dataset provided a lot of issues which we had to overcome. This was both of our first times trying to balance a dataset of this size with such a drastic imbalance. We attempted various different balancing algorithms and techniques such as under sampling, over sampling and SMOTE. These techniques affected the performance of the models differently so we learned which technique suited our problem better. From many different experiments we concluded that the best technique for us was under sampling.

Another learning experience we faced was during the feature engineering process. We thought that using our knowledge of sports and the NFL itself would prove to be useful in deciding how and what features we created. For example, we created a 'DaysRest' feature which was the number of days a player had rest between playing games. We felt this would be a very useful feature as the more rest a player gets the less likely the player would be injured. So we thought. Through more experiments and testing, checking feature importances we discovered that this feature didn't provide much importance to the models at all. Improving the performance of the models marginally. We found this very interesting as firmly believed the correlation between the number of days rest a player got would significantly impact the results.

In our hypothesis test (can be found in our archive folder of the project) we found a statistical difference between field types in regards to injury. Because of this we believed that field type would play a significant role in our models predictions. Whilst it does play a role in the performance of the model, it's not as much as other features. We learned that through our feature engineering that workload features provided a significant impact on the metrics and accuracy of the models performance. Creating workload density, workload increase and workload stress provided a significant boost in the accuracy of our predictions.

In conclusion, throughout our rigorous testing, experimenting and feature engineering, we learned that knowledge of the sport itself in terms of creating features that we thought would be important, didn't provide much impact to the performance of the models. Predictive models don't incorporate logic or context into account. They take in the numbers and data and produce their result based on that.

8. Appendices

Link provided below to show all results from previous models, with changed parameters, features, etc.

https://www.notion.so/Ca400_testing_results-0bd5ff45d63b4f9d89c4d47a6d580d52?pvs=4