Transnational University Limburg is a unique collaboration of two universities in two countries: the University of Hasselt and Maastricht University.

UHASSELT KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

UHASSELT KNOWLEDGE IN ACTION

Maastricht University

Faculty of Sciences

School for Information Technology

Master of Statistics and Data Science

Master's thesis

Predicted Motion Pressure - Using Machine Learning Algorithms to Metricize Pressure Created by Defensive Linemen in the NFL, as well as to Predict Their Motion to Evaluate the Players Performance and to Increase the Players Safety

Christopher Patzanovsky

Thesis presented in fulfillment of the requirements for the degree of Master of Statistics and Data Science, specialization Data Science

Confidential

SUPERVISOR :
Prof. dr. Dirk VALKENBORG

2022
2023

Faculty of Sciences
School for Information Technology
Master of Statistics and Data Science

Master's thesis

Predicted Motion Pressure - Using Machine Learning Algorithms to Metricize Pressure Created by Defensive Linemen in the NFL, as well as to Predict Their Motion to Evaluate the Players Performance and to Increase the Players Safety

Christopher Patzanovsky

Thesis presented in fulfillment of the requirements for the degree of Master of Statistics and Data Science, specialization Data Science

Confidential

SUPERVISOR :
Prof. dr. Dirk VALKENBORG

Contents

1 Abstract

Data analytics has been a major part in American football to allow for evaluation of both player and team performances, with a significant amount of data being readily available. Here, a more in-depth analysis in the area of pass rushing and pass blocking will be done by using K-Nearest Neighbor (KNN) machine learning models to find a more precise metric of determining pressure created by the pass rush. Additionally, KNN would be used to predict the motion of a pass rusher, which would then be used to predict pressure created in that new location, allowing for the creation of an entirely new metric, by comparing the true pressure with the predicted one. An extensive analysis with these new metrics would then be conducted, giving American football players and coaches new insights into player and team performances, which can be used to improve the quality of the game, by the means of performance, while at the same time creating new ways of focusing on the safety of the Quarterback. Finally, a dashboard was build on an HTML-based webpage, visualizing the new metrics for a better understanding of its concepts.

2 Introduction

2.1 American Football and Data Science

American football has been the most popular sport in the United States since 1972, but while not claiming the top spot outside the United States, the sport has been slowly expanding over the last several years.[1] In 2007, the National Football League (NFL), the largest American football league in the world, played their first game on international ground in London, and has since then, as of 2023, played 44 games outside of the United States in the United Kingdom, Mexico, and Germany, showing a rising interest in the sport around the world. An estimated 200 million people have watched the 2022 Super Bowl, seeing the Kansas City Chiefs beat the Philadelphia Eagles 38:35.[2] Approximately 56 million of these views stem from outside of the United States, showing an increase of 7% compared to 2021.[3] Due to the fact that American football is deeply rooted in the United States' culture, US Americans typically grow up with the sport and get familiar with it from a very early age. When this is not the case (like for many non-US Americans), an entry into the sport can often be fairly difficult, as, to a layman, the game itself from the outside can look quite confusing and complex. The game flow can appear to be chaotic, as all 22 players on the field are tasked with a certain operation, which means that every player is in motion, throughout the entirety of a play. But, some players move away from the ball, while others run towards it, some players seek for separation from their opponents, while others actively move towards one another. These, and many more small concepts together make up the game of American football. A key factor, though, is the fact that American football (unlike its European counterpart), is very static. It can be compared to tennis, where one point is played after another, and once a point is finished, both players reset and prepare for the next point. The exact same thing happens in American football: once a play is over, it is "called dead" - the ball may not be advanced any further, and no more tackles are allowed to be made. The players then regroup and start a new play. This makes the sport incredibly interesting to both the fan, as well as to data scientists. For the fan, from a tactical standpoint, as the sport mimics the game of chess, where one team reacts to the tactics of the other. For the data scientist, due to the fact that the game is so repetitive, with the same formations and motions being repeated over and over again, data collection and analytics have become a main focus point for the NFL. With this in mind, the NFL, in cooperation with *NFL Next Gen Stats* and *Amazon's Web Services*, started implementing a new technology in 2014, for better and more precise tracking and data collecting.[4] As a result, data analytics has become a vital part in every team's operations as the competitiveness of the sport undoubtably leads to every organization looking for every possible inch of advantage that can be gained over the competition. Naturally, the NFL is constantly looking for new and better ways to analyze the collected data, to improve the game, both from a performance, as well as a safety perspective, as it is not only important for the teams to make the game better, but also safer for all of the players. Under this premise, the "NFL Big Data Bowl" was created in 2019, a yearly competition for data scientists to emerge themselves into a specific area of the game and to look for new and creative ways to improve the sport.

For this thesis, the data provided by the NFL Big Data Bowl 2023 was used to create a new metric to analyze player and team performance, based on machine learning algorithms. But, before going more in depth about the task at hand, the next section will be dedicated to a brief explanation of, and guide for the game of football, to give the reader a better understanding of the sport, as well as familiarize them with specific terminology used in American football.

2.2 Introduction to American Football Terminology, Rules, and Tactics

2.2.1 The Basics

While there are many variations of American football (professional, collegiate, high-school, etc.), for which the rules differ slightly, the core of the game is always the same. To clarify, here, only professional, NFL football will be considered.

The game is played on a field that is 120 yards in length and 53.33 yards in width. The last ten yards on each side of the field are called the **end zones**. The small lines on the field are the yard markers, while the big numbers indicate how many yards one is away from the closest end zone. A representation of an American football field can be seen in Figure 1.

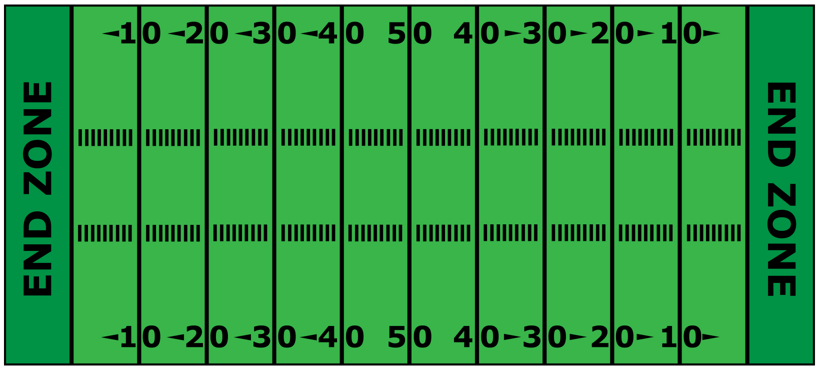


Figure 1: Illustration of an American Football Field

A game consists of four quarters, each of which being 15 minutes long. At any point of the game, there are 22 players on the field, 11 from each team. The team that is in possession of the ball is called the **offense**, the other is called the **defense**. The ultimate goal of the game is to have a player end up in the opponent's end zone, which can be achieved by either running the ball into the end zone, or catching it in it. This achievement will be rewarded with 6 points and is called a **touchdown**. While there are many different, alternative ways to score points (e.g. kicking the ball through the goal posts), those will be unimportant for this thesis.

As mentioned earlier, the game consists of **plays**, which give the offense a chance to advance the ball towards the opponent's end zone. Whenever a team gains **possession** of the ball, that team will get 4 plays (also called **tries** or **downs**) to advance the ball 10 yards. Should the offense succeed in doing so, they will receive another four attempts to again advance 10 yards, until they eventually reach the end zone. As an example: A team gains possession of the ball at their own 35 yard line (so 35 yards away from their own end zone), they now have what is called "1st down and 10". Lets assume that with their first play, they advance the ball 4 yards to their own 39 yard line. It is now "2nd down and 6". With the next play, the team advances the ball 8 yards to go to their own 47 yard line and therefore break the needed 10 yards. They now get the ball at their 47 yard line at it is again 1st down and 10. Should the offense not manage to advance 10 yards in 4 tries, the defense receives possession of the ball and therefore becomes the offense.

Several scenarios can cause a play to be over:
1. The player with the ball was tackled by the defense.
2. The player with the ball has stepped outside of the field.
3. The ball was thrown but not caught and has hit the ground
4. The player reaches the end zone

If the ball is thrown, there are three possible scenarios:
1. The ball is caught by an offensive player, then the pass is called **complete**.
2. The ball is not caught by any player, then the pass is called **incomplete**.
3. The ball is caught by a defensive player, then the pass was **intercepted** and the defense gains possession of the ball and becomes the offense.

If a player is in possession of the ball but it falls out of their hands, it is called a **fumble** and whichever team first picks the ball up will be in possession of the ball.

2.2.2 Player Positions, Formations, and Tactics

American football is a highly tactical game, especially due to the fact that is not fluid. Before every play, both teams are able to regroup and come up with a new strategy to advance the ball/stop the offense from doing so. Every player on the field has a specific role and only if the team works together, execution of the game plan is possible. A typical formation of both the offense and the defense together with the name of each position can be seen in Figure 2. The offense can be separate in the following groups:

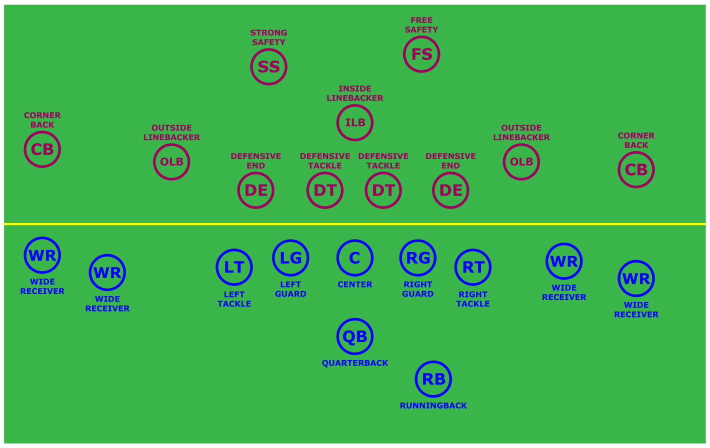


Figure 2: Potential formation of the offense on the bottom and the defense on the top, with the offense lining up with four Wide Receivers and one Running Back and the defense with four Defensive Linemen and three Linebacker

The **Passer/Quarterback (QB)**: In most cases the most important player on the field. The Quarterback can be described as the chess master of the game, as the entire play runs through them. They, together with the coaches, come up with the strategy before the play, give directions to every player on the offense on where to move, and are the ones to decide whether to throw the ball or to run with it. They are tasked with the job to remain calm in a chaotic environment, to choose which player to throw the ball to, and throw a pass precise enough for their teammate to catch, all while being chased by defensive players. A good Quarterback is often key to long-term success for teams, which is why the Quarterback is the best paid position in the NFL.

The **Wide Receivers/Running Backs (WR/RB)**: Often referred to as the “skill players”, they are the players who receive the ball from the Quarterback, either in the form of a throw (for the Wide Receivers) or in the form of a handoff (for the Running Back). Prior to the play, the Quarterback instructs these players where they have to run to in the form of routes. These routes are predetermined, but the Quarterback has to be fully conscious of all the routes of the Wide Receivers and Running Backs, as the chemistry between the players and the timing of the throw has to be just right for the Wide Receivers to even have a chance to catch the ball. How routes can look like for a hypothetical play can be seen in Figure 3.

The **Pass Blocker/Offensive Line**: The **Right Tackle (RT)**, **Right Guard (RG)**, **Center (C)**, **Left Guard (LG)**, and **Left Tackle (LT)** together make up the entirety of the **Offensive Line**. Their entire job is to protect the Quarterback from the defense. Throughout any play, a certain amount of defensive players will try to put pressure on the Quarterback by trying to get to, and tackle them, or at least to allow them as little time as possible to throw the ball and therefore not allow for the Wide Receivers to get to the positions where they are meant to be at. The Offensive Line acts as a shield around the Quarterback and tries to block those defensive players from getting through. For right-handed Quarterbacks, the Left Tackle is usually the most important position in the Offensive Line, as they are the player who protect the Quarterback’s blindside. For a right-handed Quarterback, when they prepare to throw, their left leg will be in front, with their upper body facing to the right. With the players wearing helmets, their view is already narrowed, meaning that they are very limited in their views of the left side of the field. If a defensive player is running towards them from the right side, they are typically able to recognize that threat and react to it, but with their left side being "blind", a tackle to the Quarterback can usually have worse consequences, as they are not able to brace their body for impact. This is why the Left Tackle is that important in doing as good of a job as possible from avoid the defense from breaking through to the Quarterback.

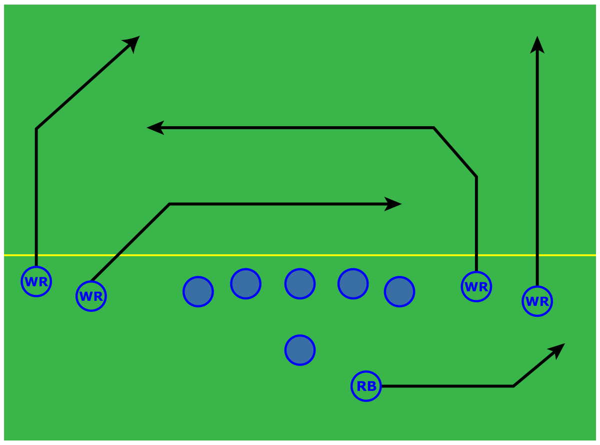


Figure 3: An example of the routes the WRs and the RB could run for a hypothetical pass play. The exact location of a turning point is determined prior to the play and known to both the WR/RB and the QB.

There are countless opportunities for the offense to structure a play. As mentioned before, they can choose between passing and running the ball. They may choose to engage more players than just the typical five Offensive Lineman in the pass block to give the Quarterback more time, or they may choose to engage more Wide Receivers to increase the chances of a completion, while accepting the fact that they will have less time to throw.

Similarly, the defense has many different options to line up themselves and react to the formation of the offense. Generally speaking, the defense can be split into two groups: the **Pass Rush** and the **Coverage**. The Pass Rushers are the direct opponents to the Offensive Line, therefore are the players trying to break through the Offensive Line to get to the Quarterback. The Pass Rush is often referred to as the **Defensive Line** and consists mainly of the **Defensive End (DE)** and the **Defensive Tackle (DT)**, but these positions have further subcategories. The Coverage on the other hand, which mostly consists of the **Cornerback/Defensive Back (CB)** and the **Strong** and **Free Safeties (SS/FS)**, are tasked with not allowing for the Wide Receivers to catch the ball or tackle them if they do catch it. In an in-between position are the **Outside** and **Inside Linebacker (OLB/ILB)**, these are players who can be engaged either in the pass rush, or drop back and help out in the Coverage. The way a defense lines up before a play is called a **formation**, which can take on many different forms. The defense may choose to play **man-to-man coverage**, meaning that every defender chooses one offensive player to be assigned to, or they may play **zone coverage**, where the defenders have a specific zone that they are tasked with defending. These coverages are further categorized, for example **Cover 0**,

Cover 1, or **Cover 2**, which can indicate how many Safeties are on the field, or how many Linebackers are engaged in the pass rush. It can be seen how highly complex every play can become, and this is again something the Quarterback has to be aware of, as they are the ones that have to read the formation off of the defense and analyze how many defenders might be potentially rushing and how many will be in coverage, as these factors can create an imbalance very quickly. Naturally, if the defense decides to rush the Quarterback with more players than are available on offense to block, the Defensive Line will be able to break through more easily and take away time for the Quarterback to throw, which is something that could then be used by the Quarterback, as there will be a lack of defenders in Coverage. A visualization of the a possible defensive formation in a simplified form can be seen in Figure 4. Here, the defense is lining up in a formation called "Cover 2 Sink LB Blitz". The defense is using three Defensive Linemen (2 DEs and 1 DT), four Linebacker (2 OLBs and 2 ILBs), two Cornerbacks, and two Safeties. In this case, the three Defensive Linemen, as well as the ILB to the left from the perspective of the Quarterback are rushing the passer, while the OLBs, as well as the right ILB are in pass coverage, having been assigned a zone that they try to defend against Wide Receivers. Similarly, the Safeties are covering a specific zone towards the deeper parts of the field, while the two Cornerbacks are assigned to a certain Wide Receiver that they will cover one-on-one. Again, it would be the job of the Quarterback and the Offensive Line to read this formation and adapt the pass blocking accordingly.

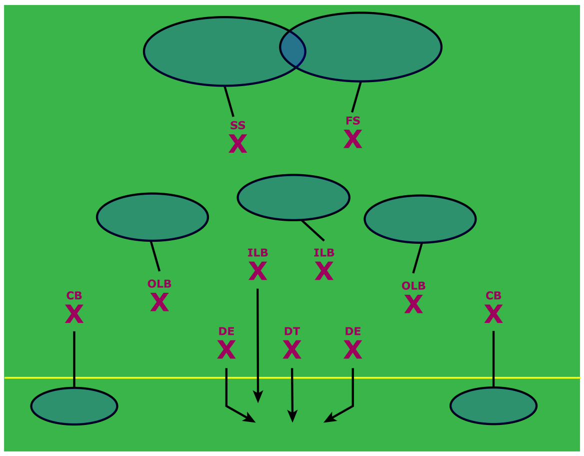


Figure 4: Simplification of a Cover 2 Sink LB Blitz, with one ILB pass rushing, the other being in pass coverage, alongside the two OLB

2.2.3 Additional, Important Terminology in the Context of This Thesis

With the main focus of this thesis being the pass block and pass rush, the possible outcomes for a pass rush have to be understood. If a pass rusher is not able to break through the Offensive Line, the logical consequence is that the player was **blocked**. This block can have resulted from one player (**single-team**), two players (**double-team**), or even more. Naturally, if there are more Offensive Linemen than pass rushers, more than one Offensive Lineman can focus on blocking a pass rusher. This can have strategical consequences, for example if a certain player is known to be an efficient pass rusher, then the Offensive Line intentionally tries to block that player with multiple men, which on the other hand opens up the possibility of one-on-one matchup for other pass rushers.

In the event that a pass rusher is able to break through the Offensive Line, their ultimate goal is to get to the Quarterback. Here, three possible outcomes can occur:

A **sack**: This is the best possible outcome for the pass rusher as it means that they actively tackled the Quarterback to the ground, and therefore ended the play, potentially resulting in a loss of yardage for the offensive team, as the Quarterback typically gets sacked behind the starting line of the play.

A **hit**: Should the pass rusher manage to create contact with the Quarterback that is not quite enough to bring them to the ground, it is recorded as a hit. A hit can still be very effective, as this can result in the Quarterback becoming off-balance, throwing a bad pass, or simply slowing them down, allowing for a different pass rusher to get to them.

A **hurry**: A hurry occurs when the Quarterback is forced to either hurry up with their throw, forcing them to throw the ball more hectically (and therefore more inaccurately) than they intended to, or is forcing the Quarterback to **scramble**, meaning that they have to run away from the threat of the pass rusher and find a new position to throw the ball from or simply try to run with the ball themselves as far as they can. A scramble can be a scripted play by the offense, but does quite naturally come with a certain risk of injuries from a tackle, which a team typically tries to avoid, simply due to the importance of the Quarterback.

A **dropback pass**: This is a specific type of pass in which the Quarterback is designed to move several steps back before throwing the ball. This move has the advantage of creating both distance to the pass rush, as well as allowing the Quarterback to use their momentum to bring their body forwards and generate more power into the throw.

2.3 Area of Focus

For the 2023 NFL Big Data Bowl, pass block and pass rush were chosen to be the main focus for all competitors, meaning that only plays in which the Quarterback dropped back to throw the ball were considered and only corresponding data was provided. For the rest of this thesis, when a pass is mentioned, it is referring to a dropback pass.

3 Research Question

The main goal of this thesis, based on the data provided, was to come up with a new way to analyze the effectiveness of the pass rush of the defense, allowing for a closer evaluation of the performance of both the team, as well as individual players. As mentioned earlier, data analytics already play an important role in the NFL, as player and team performance are a decisive factor in preparations for upcoming games to analyze the opponents, as well as for the own team to find out in which area the team and/or an individual player has to improve in, and therefore structure practices accordingly. But while the importance of data analytics is undeniable, it has to be mentioned that actual evaluation is not equal for every position and player in American football. As an example, the impact of a Wide Receiver on the game can be evaluated via their individual production, as in how many touchdowns they scored, how many yards they ran, or how often they were targeted, while a Quarterback can be evaluated by a calculated Quarterback Rating (QBR), which is based on similar metrics to that of a Wide Receiver.

A problem arises when it comes to the evaluation of the Offensive and Defensive Line, as these players are typically not involved in attempting to catch the ball. Data on whether a pass rusher was able to create a sack, hit, or hurry on the Quarterback in a play is recorded, but the issue with this is the fact that those variables are binary in nature, meaning that on a play, a player either achieved a sack/hit/hurry, or they did not.

With the creation of a more precise metric to analyze the effectiveness of the pass rush, and to define the exact amount of pressure that the Defensive Line acts upon on the offense, new ways to analyze performance will be possible, opening new doors for not only coaches and players, but also giving new insights into the protection of the quarterback, a key point of improvement that is strived for by all teams.

4 Exploratory Data Analysis

4.1 The Data

4.1.1 Overview

The data used for this thesis was provided by the NFL, NFL Next Gen Stats, and Pro Football Focus via the NFL Big Data Bowl 2023, which was hosted on Kaggle, and is publicly available. The data contains information for the first eight weeks of the 2021 NFL season, tracking every outcome of every play, as well as precise telemetry for every player involved on the play. There are 12 CSV files, which can be divided in five different categories: game, play, player, PFF Scouting, and tracking data, all of which will be described in more detail in the following sections.

4.1.2 Game Data

The game data named `games.csv` contains all games played in the first eight weeks of the 2021 NFL season with the names of the teams playing in each game. Each game is given a unique `gameId`, by which games can be identified through the different data sets. A description of each variable in the game data set can be seen in Table 1.

Table 1: Name and description for each variable in the game data

Name	**Description (Domain)**
`gameId`	Game Identifier, unique (numeric)
`season`	Season of the game (numeric)
`week`	Week of the game (numeric)
`gameDate`	Game Date (time, mm/dd/yyyy)
`gameTimeEastern`	Start time of game (time, HH:MM:SS, EST)
`homeTeamAbbr`	Home team three-letter code (text)
`visitorTeamAbbr`	Visiting team three-letter code (text)

4.1.3 Play Data

The play data named `plays.csv` contains play-level information for each play of each game, which includes a description of the play. As an example: (13:33) (Shotgun) T. Brady pass incomplete deep right to C. Godwin. This description contains the time on the game clock, the formation of the offense and (in the event of a pass play), the name of the passer, the intended receiver and where the ball was thrown to. Each play can be identified by a `playId`, as well as the corresponding `gameId`, for which the `playId` will be unique for, meaning that identical `playId` may appear for different games. A description of each variable in the play data set can be seen in Table 2.

Table 2: Name and description for each variable in the play data

Name	**Description (Domain)**
`gameId`	Game Identifier, unique (numeric)
`playId`	Play Identifier, not unique across games (numeric)
`playDescription`	Description of play (text)
`quarter`	Game quarter (numeric)
`down`	Down (numeric)
`yardsToGo`	Distance needed for a first down (numeric)
`possessionTeam`	Team abbr. of team on offense with possession of ball (text)
`defensiveTeam`	Team abbr. of team on defense (text)
`yardLineSide`	3-letter team code corresponding to line-of-scrimmage (text)
`yardLineNumber`	Yard line at line-of-scrimmage (numeric)
`gameClock`	Time on clock of play (MM:SS)
`preSnapHomeScore`	Home score prior to the play (numeric)
`preSnapVisitorScore`	Visiting team score prior to the play (numeric)
`passResult`	Dropback outcome of the play (C: Complete pass, I: Incomplete pass, S: Quarterback sack, IN: Intercepted pass, R: Scramble (text)
`penaltyYards`	Yards gained by offense by penalty (numeric)
`prePenaltyPlayResult`	Net yards gained by the offense, before penalty yardage (numeric)
`playResult`	Net yards gained by the offense, including penalty yardage (numeric)
`foulName[i]`	Name of the i-th penalty committed during the play. i ranges between 1 and 3 (text)
`foulNflId[i]`	`nflId` of the player committing the i-th penalty during the play, with i ranging between 1 and 3 (numeric)
`absoluteYardLineNumber`	Distance from end zone for possession team (numeric)
`offensiveFormation`	Formation used by possession team (text)
`personnelO`	Personnel used by offensive team (text)
`defendersInTheBox`	Number of defenders in close proximity to line-of-scrimmage (numeric)
`personnelD`	Personnel used by defensive team (text)
`dropbackType`	Dropback categorization of quarterback (text)
`pff_playAction`	Indicator for whether offense executes play action fake on the play. Variable provided by PFF (binary)
`pff_passCoverage`	Coverage scheme of defense. Variable provided by PFF (text)
`pff_passCoverageType`	Whether defense's coverage type was man, zone or other. Variable provided by PFF (text)

4.1.4 Player Data

The player data named `players.csv` contains player-level information for each player that has been tracked at least one time throughout the games. Each player can be identified with a unique `nflId`. A description of each variable in the player data set can be seen in Table 3.

4.1.5 PFF Scouting Data

For each game and play, player-level scouting information is provided by Pro Football Focus for all 22 players on the field. This includes information such as which role a certain player has (e.g. pass rush), whether a player created or allowed a sack/hit/hurry on the Quarterback and, if applicable, which player they faced in a matchup. Each player can be identified via the unique `nflId`, the `playId` and the `gameId`. A description of each variable in the PFF Scouting data set can be seen in Table 4. The file is named `pffScoutingData.csv`.

4.1.6 Tracking Data

For each player and the football, in each game and play, specific tracking data was recorded every tenth of a second, which is considered a frame. This tracking data includes the player's and the football's x-coordinates, y-coordinates, speed, acceleration, and orientation for every frame of the play. The tracking data is stored in files named `week[i].csv` with `i` ranging from 1 to 8, corresponding to the week of the game and play. Each player can be uniquely identified via their `nflId`, and corresponding `playId` and `gameId`. A description of each variable in the tracking data set can be seen in Table 5.

Table 3: Name and description for each variable in the player data

Name	Description (Domain)
`nflId`	Player identification number, unique across players (numeric)
`height`	Player height (text)
`weight`	Player weight (text)
`birthDate`	Date of birth (YYY-MM-DD)
`collegeName`	Player college (text)
`officialPosition`	Official player position (text)
`displayName`	Player name (text)

Table 4: Name and description for each variable in the PFF Scouting data

Name	Description (Domain)
`gameId`	Game Identifier, unique (numeric)
`playId`	Play Identifier, not unique across games (numeric)
`nflId`	Player identification number, unique across players (numeric)
`pff_role`	Player's role on this play. Possible values: `Coverage`, `Pass`, `Pass block`, `Pass route`, `Pass rush` (text)
`pff_positionLinedUp`	Position that the player was aligned at the snap of the ball on this play (text)
`pff_hit`	If player is a defensive player, indicator for whether they are credited with recording a hit on this play (binary)
`pff_hurry`	If player is a defensive player, indicator for whether they are credited with recording a hurry on this play (binary)
`pff_sack`	If player is a defensive player, indicator for whether they are credited with recording a sack on this play (binary)
`pff_beatenByDefender`	If player is a blocking offensive player, indicator for whether they are by a defender but was not charged for yielding a hit, hurry or sack (binary)
`pff_hitAllowed`	If player is a blocking offensive player, indicator for whether they are responsible for a hit on the QB (binary)
`pff_hurryAllowed`	If player is a blocking offensive player, indicator for whether they are responsible for a hurry on the QB (binary)
`pff_sackAllowed`	If player is a blocking offensive player, indicator for whether they are responsible for a sack on the QB (binary)
`pff_nflIdBlockedPlayer`	If player is blocking offensive player, the `nflId` of the first defender the offensive player blocked (numeric)
`pff_blockType`	If player is a blocking offensive player, the type of block that the offensive player is executing on the defender (text)
`pff_backFieldBlock`	If player is a blocking offensive player, indicator for whether block occurred in offensive backfield

Table 5: Name and description for each variable in the PFF Scouting data

Name	**Description (Domain)**
`gameId`	Game Identifier, unique (numeric)
`playId`	Play Identifier, not unique across games (numeric)
`nflId`	Player identification number, unique across players (numeric)
`frameId`	Frame identifier for each play, starting at 1 (numeric)
`time`	Time stamp of play (time, yyyy-mm-dd, hh:mm:ss)
`jerseyNumber`	Jersey number of player (numeric)
`club`	Team abbreviation of corresponding player (text)
`playDirection`	Direction that the offense is moving (left or right)
`x`	Player position along the length of the field, 0 - 120 yards (numeric)
`y`	Player position along width of the field, 0 - 53.3 yards (numeric)
`s`	Speed in yards/second (numeric)
`a`	Acceleration in yards/second^2 (numeric)
`dis`	Distance traveled from prior time point, in yards (numeric)
`o`	Player orientation in degrees, 0 - 360 (numeric)
`dir`	Angle of player motion in degrees, 0 - 360 (numeric)
`event`	Tagged play details, including moment of ball snap, pass release, pass catch, tackle, etc (text)

4.2 Data Preprocessing

With 32 teams in the NFL, there are an even 16 possible games per week. Each team receives one “bye week” per year, meaning they will not compete for that weekend. Bye weeks are assigned between week 6 and week 14 and can vary for each team from year to year. In 2021, in week 6, the New York Jets, the Atlanta Falcons, the New Orleans Saints, and the San Francisco 49ers had their bye weeks. For week 7, it was the Buffalo Bills, the Pittsburgh Steelers, the Los Angeles Chargers, the Dallas Cowboys, the Minnesota Vikings, and the Jacksonville Jaguars. And for week 8, the Las Vegas Raiders and the Baltimore Ravens had their bye week. This means that for the first five weeks, 16 games took place per week, 14 in week 6, 13 in week 7 and 15 in week 8. Naturally, this is reflected in the the amount of pass plays that were ran by the offense, as over the first five weeks, the average of pass plays per game was at 1120.8, which dropped to 1004 in week 6, 917 in week 7 and 1032 in week 8. This comes out to approximately 70.14 passing plays per game, or 35.07 per team, on average. The distribution of average pass plays per team can be seen in Figure 5. The Kansas City Chiefs were leading this statistic with 42.375 passing plays on average over the first 8 weeks, while the Green Bay Packers were trailing with 28.25 passing plays on average. As mentioned earlier, the entire focus here is on dropback passes, which are only one type of pass, while the most common at that.[5] Nonetheless, while these are only a portion of all pass plays in a game, the big difference in the amount of plays per team can be simply be explained by the type of offense a team runs, which often correlates to the playing style of the team’s Quarterback. A team with a mobile Quarterback with a strong arm (like Kansas City’s Patrick Mahomes) is prone to build their offense around them, profiting from efficient pass play, while teams with higher focus on run play, or a Quarterback more focused on short passes, would naturally score lower on this list.

Furthermore, one straightforward metric to evaluate Quarterbacks on is the completion percentage, the percentage of how many of their thrown passes were caught by their Wide Receiver. While completion percentage is not a bulletproof metric, as many factors have to be accounted for when analyzing a pass (since a mistake by the Wide Receiver or a good pass rush by the defense, giving the Quarterback too little time and forcing them to throw the ball away, all count against the completion percentage), it still gives a solid first indicator of the efficiency of the Quarterback and the offense as a whole. The distribution of the completion percentage across all 32 teams can be seen in Figure 6.

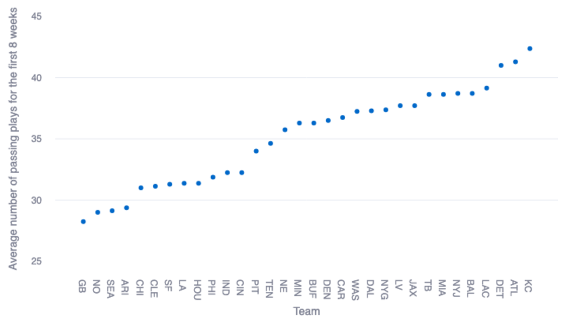


Figure 5: Average number of passing plays for the first 8 weeks per team

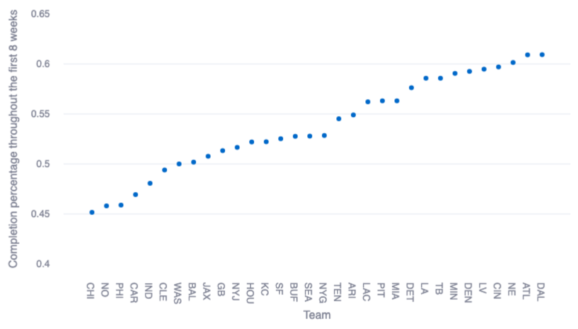


Figure 6: Completion percentage throughout the first 8 weeks per team

Similarly to the average amount of plays per week, the tracking data shows a similar picture, as mentioned earlier, the tracking data collects several metrics for each player and the football, every tenth of a second, for each play and game. This results in 1,091,038 entries, on average, per week, for the first five weeks, and 973,797, 906,292, and 978,949 entries respectively for weeks 6, 7, and 8, coming out to a total of 8,314,178 frames tracked.

Due to the fact that all of the tracking data only includes the `nflId` for each player, but neither the position of the player, nor the role that said player had on that play, adding these two values to the data frame was seen as necessary. All eight tracking CSV files were concatenated into one large dataframe, allowing all upcoming operations to be done at once, as accessing individual weeks would only be necessary for specific tasks, but to ensure that this would be easily possible, a separate column called "weeks" was introduced, representing the corresponding week. With the PFF Scouting data, all necessary data for player position and role on the play could be accessed easily, and while the tracking data describes different aspects of the same play, the quintessence is the exact same - for every play, in both datasets, all players involved will overlap. Therefore, matching the player in both data sets can be done easily with the `gameId`, `playId`, and `nflId` identifier. Since the position and role of a player does not change during a play (if, then only for different plays), the information provided by the PFF Scouting data can be simply added to every frame for that play in the tracking data. Iterating over every entry in the PFF Scouting data, the tracking data can be filtered for the respective identifiers, returning a dataframe that contains every frame of that play for only one player. Role and position can then be added straightforwardly.

Lastly, here it should be added that for all variables that were important to this thesis, no missing values were present.

5 Methodology

5.1 Definition of Pressure

As mentioned, the analysis and evaluation of pass rushers proves to be far more intricate than those of skill players like Quarterbacks and Wide Receivers, as their contribution to the game can be measured directly on the scoreboard. While a direct measurement of a successful pass rush is the occurrence of a sack/hit/hurry on the play, the question arrises whether that is a truly fair metric. As an example, a pass rusher who is being double-teamed by two Offensive Linemen, and is therefore obviously not able to create as much pressure on the Quarterback, will, in this binary metric, be considered to have an unsuccessful play, while in reality, due to the fact that they are attracting an additional Offensive Lineman, they are able to create the opportunity for a different pass rusher to only have one or even no Offensive Linemen blocking them, who is then able to create direct pressure on the Quarterback. So even though the first pass rusher was not able to actively create pressure, their passive contribution on the play may still lead to success on the play.

Due to the fact that information on whether a player created a sack/hit/hurry on a play is available for every pass rusher on every play, for the purpose of this thesis, pressure will be defined as either a sack, a hit, or a hurry having occurred on a play for that pass rusher. While a sack is the ultimate form of pressure that a pass rusher may bring on a play, achieving both a hit on the Quarterback, as well as making the Quarterback hurry up and change plans during the play, are forms of pressure that have to be recognized. In the first eight weeks, the average height for all players who were involved at least once during a pass rush comes out to be just over 6'1" (1.85 m) with an average weight of 252.12 lbs (114.36 kg). So it is easy to imagine the pressure that a pass rusher is putting on the Quarterback if they are able to beat their Offensive Lineman and have a straight path towards the Quarterback. With American football being such a physical sport, the effects of a pass rusher breaking through the Offensive Line are not only present in physical pressure, but also mentally for the Quarterback, as they obviously try to avoid being tackled by a pass rusher, so that no loss of yardage occurs on the play, but also to simply not be tackled, since, even though helmets and padding are being worn, injuries are not uncommon. Injuries in American football are five times more likely to occur during full-contact sessions (which can be games or practice), with 40% of those injuries being strains and sprains.[6] While only 5% of all injures are concussions, the importance of awareness for that injury has to be stressed, as a correlation between head-related injuries and chronic traumatic encephalopathy (CTE) has been found, in addition to mental health related issues, which include dementia, depression, and Parkinson's disease.[7] While helmets have obviously had a huge impact in reducing head-related injures, helmet-to-helmet and helmet-to-ground contact have been shown to be the main causes for concussions.[8]

With this in mind, the strategical (a loss of yardage for the offense), as well as the underlying (mentally for the Quarterback) effects of pressure created on a Quarterback are becoming clear, which is why it is important to be able to do a deeper, more precise analysis of it. Instead of describing pressure as a binary variable for the entirety of a play (as it is now), the goal of this thesis is to define a metric that is able to describe the exact amount of pressure that is created by a pass rusher and the entire team at every frame of the play, to allow for a better analysis. To do so, the K-Nearest Neighbor (KNN) machine learning model will be used. In the second step, KNN will be used to predict the location of a pass rusher, their speed, acceleration, and orientation on the next frame to allow for a comparison of pressures between the predicted location and the real one.

5.2 K-Nearest Neighbor (KNN)

The K-Nearest Neighbor (KNN) algorithm is a supervise, non-parametric learning classifier. It can be used for both classification and for regression problems, both of which will be present in this thesis. With KNN, maximum condition probability is used as a decision rule to predict the classification of a new observation.[9] Simply put, for KNN, the training data is in an n-dimensional space, with n representing the amount of features of the data. A new observation is then put in this space and via proximity to the k-nearest neighbors, the classification of the new observation is then determined. The value for k naturally plays a vital role in the classification process, while other parameters can also be tuned, such as giving weight to the distance to the neighbors. As an example, if k=19, and 9 data points of classification A are relatively close to the new observation, while 10 data points of classification B are relatively far from the new observation, by default, the KNN model would classify the new observation as B, while with a weighted distance, it would be classified as A.

Generally, KNN is based on "majority voting", meaning 50% or more of the k-nearest neighbors need to be of a classification for the new observation to be classified as the same, but an alternative way of parameter tuning would be to change the percentage required for a classification.

An example of a simple KNN with two features and two classes can be seen in Figure 7. Here, three of the five nearest neighbors are of class A, therefore, the KNN also classifies the new observation into class A.

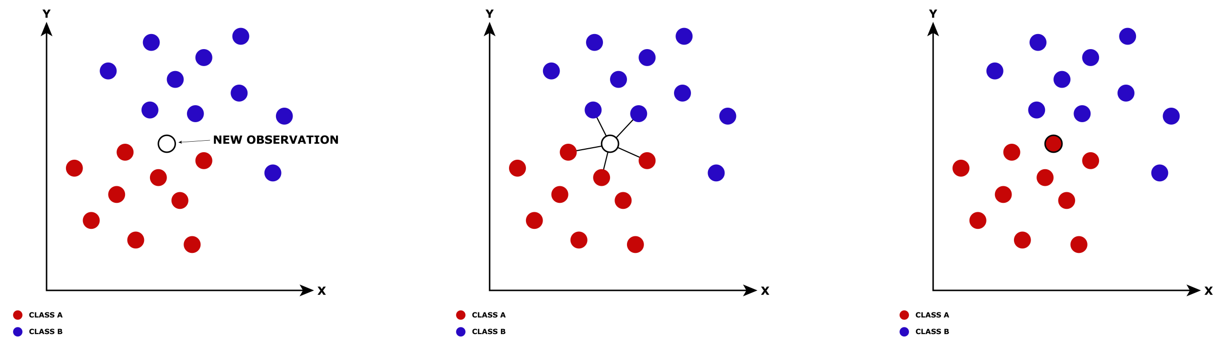


Figure 7: Visualization of a KNN with two classes and k=5

For regression problems, the same concept applies, but instead of a straightforward classification, the average value of the k-nearest neighbors is calculated and used to the determine the value of the new observation.

Another important factor for this thesis was the fact that the probability of a prediction could be extrapolated from a KNN model, which is a key factor for the metric creation and will be discussed in more detail later on. Allowing for all of the tasks at hand that came with the creation and evaluation of a new metric to be brought under the same roof, and be solved with the KNN algorithm, while at the same time keeping full control of parameter tuning, was the main reason why KNN was the algorithm of choice.

5.3 Predicting Pressure of Pass Rusher With KNN

5.3.1 General Concept

The concept and definition of pressure has been established already, but the task at hand now is to find a way of determining the pressure that each player is actually creating for the entirety of the play, frame by frame. For this, the concept of the magnitude of pressure by a player at any given time has to be established first, as there are several factors that have to be accounted for.

Naturally, if an Offensive Linemen is in control of a one-on-one matchup with an pass rusher and is able to block the opponent, the pressure of the pass rusher would be close to zero. A Quarterback can be considered relatively safe from a pass rusher if the blocker is essentially right in between the passer and the pass rusher. The speed and acceleration of the pass rusher would be quite small when only looking at that moment.

Similarly, if a pass rusher has beaten their blocker, or if no blocker was assigned to the pass rusher, the line of the pass rusher towards the Quarterback would be unhindered. The speed and acceleration would consequently increase with every frame and the distance between the passer and the pass rusher would diminish frame by frame. In this instance the pressure would be relatively high, increasing per frame, which could then again be reduced, should a blocker recover and manage to get the pass rusher off of their track.

This is where a machine learning model comes into play. For every play, information is available whether or not a player was able to achieve pressure. In every play, every frame can be looked at as a still image on the play, while at the same time, the outcome of the play for the player is already known. Therefore, a KNN can be used to take the telemetry data for every frame, and classify it into two categories: whether or not a pressure occurred. A new observation would then be put in the identical space, as the same telemetry data would be available, and, based on the k-nearest neighbors, the new frame could be predicted to have ended in a pressure or not. The key advantage here, as mentioned earlier, is the fact that the probability of the classification can be extrapolated per prediction, which can be seen as a perfect fit to metricize pressure. This is the most important step in this thesis and the baseline for everything that follows.

This would mean that, as an example, if for a certain frame, the pass rusher was able to leave their blocker behind and was on a clear path with substantial speed and acceleration, based on the trained data, this frame would, in theory, be very close to other frames of plays that ended in a pressure occurrence, and the model would therefore predict, with a relatively high percentage, for this play to have had a pressure occurred, therefore giving this player, in this frame, a higher magnitude of pressure. With this, from now on the definition of pressure is able to shift from a mere binary metric, to a precise number between 0 and 1.

Additionally, with over 1.5 million recorded frames for pass rushers, the amount of data that can be used for training and testing can be considered substantial. Since the data is already split into weeks, using seven weeks for training, and the eighth for testing, would only be logical, as with that, further (and easier) analysis will be possible.

Data manipulation will be necessary. While there are already many valuable features in the provided data, there is additional telemetry that can be calculated that can be seen as beneficial for the model, as those can have an impact on the accuracy of the model.

It has to be mentioned that, V. Karpick, a finalist of the NFL Big Data Bowl 2023, has also come up with a "Pressure Index", but used a Convolutional Neural Network (CNN) to determine when in a sequence the pass rush was able to create pressure, and did so to evaluate the Offensive Line.[10] Nonetheless, while a different approach, some of the added, calculated features will be similar.

5.3.2 Data Manipulation

Firstly, all computations and data manipulation, as well as the implementation of the machine learning model were done using Python 3.8. All data-related graphs were done with Plotly in Streamlit.

The tracking data provided shows the true direction that a play unfolded into. NFL teams have to switch sides after the first and the third quarter. The sides for the first and third quarter are determined before the game and at halftime via a coin toss, meaning that teams might stay on the same sides after halftime, but nonetheless, both teams play on each side of the field for two quarters. Here, it means that the data will show all teams play both from right to left, as well as from left to right, at every game. For a better performance of the machine learning model, the plays have to be ensured to uniformly go into the same direction. It was chosen for all plays to go from left to right, meaning that the Quarterback would be trying to score a touchdown into the right end zone. Doing so would ensure that pass rush would always come from the right, meaning that the x - and y-coordinates would be of the same domain. Plays which were found to go from right to left, had to be flipped both by the x-axis and by the y-axis, so that it could be ensured that plays that happened towards the right side of the field when looking towards the end zone, would also stay that way. For any frame that was part of a play going from right to left, the x-value would be subtracted from 120, and the y-value would be subtracted from 160/3, corresponding to the length and width of a football field. The final variable that would have to be adapted for this is the orientation, as the player would be turned by 180°. For orientation values smaller than 180, 180 would be added to it, while for values larger than 180, 180 would be subtracted from it, to ensure those values to stay between 0 and 360.

As mentioned earlier, the position on a play as well as the role that the player had on that play (Passer, Pass Route, Pass Coverage, Pass Rush, Pass Bock), where not yet in the tracking data, but could be extrapolated from the PFF Scouting data, adding position and role for every player, at every play, at every frame. The position would be of great value in this data set, as it would simplify the telemetry calculation that would follow.

The following variables per frame are considered for the KNN model to determine the likelihood of pressure occurring:

- speed of the pass rusher
- acceleration of the pass rusher
- orientation of the pass rusher
- x-offset between the pass rusher and the Quarterback

- y-offset between the pass rusher and the Quarterback
- distance between pass rusher and Quarterback
- x-offset to the closest pass blocker
- y-offset to the closest blocker
- distance to the closest blocker
- speed of the closest blocker
- acceleration of the closest blocker
- orientation of the closest blocker
- angle created between the pass rusher, pass blocker and Quarterback, with the Quarterback as the vertex
- x-offset of the pass rusher compared to their last frame
- y-offset of the pass rusher compared to their last frame

Speed, acceleration, and orientation of both the pass rusher, as well as the blocker can be extrapolated right from the data, while for the rest, data manipulation and certain calculations will have to be undertaken. A visualization of these telemetries can be seen in Figure 8.

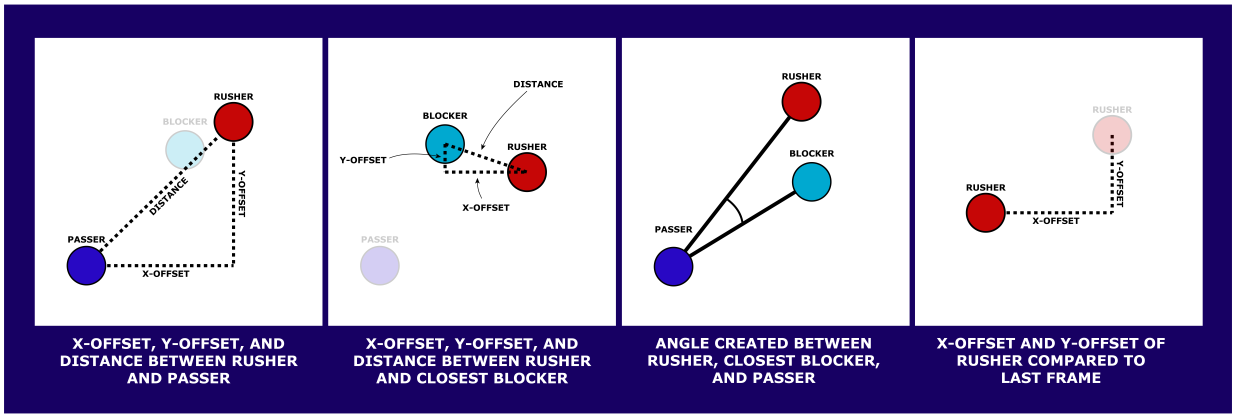

RUSHER
BLOCKER
Y-OFFSET
DISTANCE
PASSER
X-OFFSET
X-OFFSET, Y-OFFSET, AND DISTANCE BETWEEN RUSHER AND PASSER
DISTANCE
BLOCKER
RUSHER
Y-OFFSET
X-OFFSET
PASSER
X-OFFSET, Y-OFFSET, AND DISTANCE BETWEEN RUSHER AND CLOSEST BLOCKER
RUSHER
BLOCKER
PASSER
ANGLE CREATED BETWEEN RUSHER, CLOSEST BLOCKER, AND PASSER
RUSHER
Y-OFFSET
RUSHER
X-OFFSET
X-OFFSET AND Y-OFFSET OF RUSHER COMPARED TO LAST FRAME

Figure 8: Visualization of the calculated metrics between passer, rusher and blocker

With the Quarterback being the main focus of the pass rush, the distance between the pass rusher and the Quarterback is an important factor for the evaluation of pressure. The absolute position on the field, which is the one given, is of no value here, as for the pressure, it is essentially unimportant whether a play is being played at the 20 yard line of the offense, or at the 20 yard line of the defense. There may, of course, be underlying differences in the way pass rush is conducted depending on the position along the play, as different formations and strategies can be applied, but these will, if applicable, show up in the analysis. Nonetheless, the goal of the pass rusher is unchanged: generate as much pressure on the Quarterback as possible. Because of this, centering the play around the Quarterback and taking positions relative to them, allows for a much better metric for the KNN model.

The x - and y-offset between the pass rusher and the Quarterback can be calculated straightforwardly by subtracting their respective x - and y-coordinates from one another. The distance between the pass rusher and the Quarterback can then be calculated using Pythagoras' Theorem. Similarly, generating the x - and y-offset for the pass rusher compared to their last frame can be done by subtracting the corresponding coordinates from those of the last frame from one another. For the first frame of a play, since there is no previous frame, these offsets would automatically be set to 0.

A more challenging situation arises when it comes to the pass blockers, as the number of pass blockers can vary drastically from play to play. As mentioned earlier, for one type of pass play, more players running passing routes may be required, and therefore, only five pass blockers are being used on this play, while a scenario with eight or nine pass blockers can occur close to the opponents end zone, where the offense uses a “fake play”, by pretending to “punch” the ball in with the Running Back (which requires as many pass blockers as possible to help create space for the Running back), but instead throw the ball to an available Wide Receiver, to catch the defense off-guard. Because of this, using the position of all pass blockers in the KNN model becomes impossible, as the number of features would vary from play to play. On the other hand, taking every pass blocker during a play into consideration would not necessarily be of benefit, as blockers are usually being assigned which pass rusher to block, or at least are given a direction into which to block. This means that a Defensive End coming to rush from the left side of the offense will have, for example, the Left Tackle as an assigned blocker, meaning the Right Tackle will have nothing to do with this matchup, as they will be concerned with an entirely different pass rusher, so their position and all their respective telemetry does not influence the aforementioned Defensive End. It therefore means that only taking a selected few pass blockers into consideration is a way to deal with this problem, the question now arises, though, how many pass blockers to consider. As mentioned earlier, a player can be blocked by a multitude of players. One-on-one matchups are the most common form of pass blocking, but double-teaming is not uncommon, either due to a lack of pass rushers or due to intentionally doubling a player. In some instances triple, or even quadruple-teaming is possible, while only nine instances of quadruple-teaming are present in the data. The provided PFF Scouting data contains information on which pass rusher was blocked by each pass blocker. With this, the amount of blockers can be calculated, but this number can obviously be zero as well, as unblocked pass rushers can occur either due to an outnumbering of pass rushers to pass blockers or due to a false assignment, where the two pass blockers block the same pass rusher, leaving a different pass rusher unassigned, who is therefore able to break through to the Quarterback easily. With the number of assigned blockers varying from play to play as well, the PFF Scouting data also does not become of benefit here, and even if an automatic, calculated assignment would be used here, pass rushers will not always be assigned to the blocker who is then credited with a block right from the beginning, as pass rushers may be performing crossing routes amongst each other to make it more difficult for the pass blockers to find the right assignment. So, auto-assigning one pass blocker would mean that the pass rusher and pass blocker could be entirely offset at the start of the play, resulting in false data for the KNN, since the model would, for this player, not be able to recognize the pass blocker that is actually in-between the rusher and the Quarterback, falsely making the magnitude of pressure in this frame relatively high.

As a solution to this, for each frame, only a certain amount of the closest blockers to the rusher could be considered, as this would solve issues regarding rushers switching routes mid-play and having a new assigned blocker, while still generating a logical geometric pattern at an early stage of the play, allowing useful data for the KNN, returning a reasonable pressure magnitude. Additionally, this would allow for the model to have a set number of features, as the number of closest blockers considered would be fixed beforehand.

To generate all the telemetry needed for the KNN, at each frame for a pass rusher, all pass blockers on this play would be taken into account. Implementing this would not be of challenge, as the role of all players on the play was available from the earlier data manipulation, so the data could always be filtered for all pass rushers on the play, in that specific frame. With this subset of the data, the distance to every blocker from a pass rusher could be calculated by subtracting their x - and y-coordinates, and again generating the distance via Pythagoras' Theorem. Additionally, the x - and y-offset and the distance from the pass blocker to their Quarterback can be calculated in the same way that they were for the pass rusher. With this, the angle between the passer, the rusher and the blocker, with the passer at the vertex can be calculated. While computing those telemetries, the speed, acceleration, and orientation of the respective blocker would also be extrapolated and all of those would be put into one array. Once all telemetries were gathered from all pass blockers, the distances to the individual blocker could be compared and ordered from smallest to largest. This allows the dataframe that would be fed into the KNN to have the closest, second closest, etc. blockers, to always be in the proper column and could be validly used as a feature.

The question now arises, at which number to set the amount of closest blockers. A more in-depth description of the parameter tuning of the model, as well as the evaluation of the accuracy will be done later on in this thesis, but it can be noted already that the raw metric of accuracy in this instance is not the *ideal* way of evaluating the model, but when it comes to comparing the models amongst one another, it is giving a valid sense of the model. Due to the fact that four blockers for one pass rusher only occurred in nine different instances throughout all plays, only three models were considered: only the closest blocker, the two closest blockers, and the three closest blockers. The differences between the three models were only incremental, as they peaked at 87.17%, 86.97%, and 87.17%, respectively. The two reasons why the model with only the closest blocker was chosen, was for one to avoid overfitting, as multiple blockers for one pass rusher are commonly not the norm, so in many instances this would just be feeding the model with unnecessary data. Even though the models performed evenly per the accuracy, as mentioned, this is not the best way of evaluating the model. Secondly, in the event that there is indeed more than one blocker assigned for one pass rusher, only having the closest blocker in the model would not make a difference for the amount of pressure being created by the rusher, since beating one blocker would instantly turn the second blocker into the closest, keeping the pressure low, and only once all blockers are beaten the pressure would get higher.

5.3.2 Parameter Tuning

The most important step for a KNN is determining its k-value. Again, here, accuracy is the metric used to compare models within each other. To find an appropriate value for k, the model would be run for all values for k from 1 to 100. This would be done for all three models, with the closest blocker, the two closest blockers, and the three closest blockers. Plotting the k-value against its accuracy for all three models can be seen in Figure 9, with the models having been trained on weeks 1 through 7, and being tested on week 8. For each model, a high increase in accuracy in the lower bounds for k can be seen, which is typical for KNN of this size. All three models would eventually reach a peak and then level out just below the peak. For the model with one blocker, this would happen for k in the 30s, for the model with two blockers for k in the 60s, and for the model with three blockers for k in the 40s.

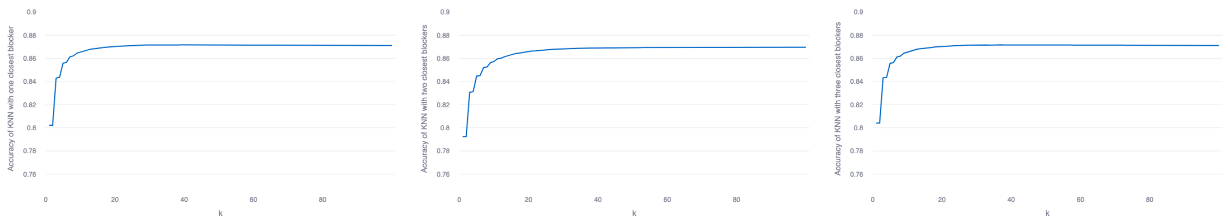


Figure 9: Accuracy of KNN models with only the closest, only the two closets, and only the three closest blockers considered for k up to 100

Weight was added in the form of distance in-between the neighbors. The reason for that being that, for one, without the weight, the classification would merely happen on a pure majority vote, meaning that the probability percentage of the classification for the new observation would always be of the form n/k, with n being the number of the majority classification. By adding the distance as weight, this would calculate the percentage as an average of all the distances, giving the pressure magnitude are more dynamic, and precise number. Secondly, clustering of data is to be expected with this type of data. With a large number of the data points in the beginning of the play looking relatively similar to one another, as well as certain other scenarios within the play, adding weight to the distance would allow for model to account for this and proper predictions closer to similar, already learnt scenarios could be made.

Since the choice was made to only go with the closest blocker, a final value for k had to be established. Since the KNN would be ran a total of eight times, always being trained on seven of the eight weeks, and being tested on the remaining week, the k with the highest accuracy for the model could be established. All models peaked in the high 30s, so a k of 37 was set. This k would be used for every model from here on out to allow for an even and fair evaluation of all models.

5.4 Predicted Motion Pressure (PMP)

5.4.1 General Concept

Having created an entirely new metric that is able to define pressure as a floating value instead of a binary classification, it has opened the door for an entirely new field of analysis of pass rushers. With many outside factors coming into play that generate the pressure metric, an analysis of players based purely on the magnitude of pressure alone would be illogical, which will be explained later on in this thesis. Because of this, an additional metric had to be created to allow for an evaluation of player performance, independent of outside factors (such as position or amount of blockers), while at the same time giving a direct comparison to every other pass rusher. This new metric is called the Predicted Motion Pressure.

The main idea was that, given the provided and calculated data, instead of predicting the amount of pressure from a pass rusher, the location, speed, acceleration, and orientation in the next frame could be predicted by using a KNN, allowing for a comparison between the predicted telemetry data with the actual one. Considering the data at hand, when giving the model the distance to the passer, distance to the closest blocker, speed, acceleration, etc., it is possible to have it predict the x-offset, y-offset, speed, acceleration, and orientation, since instead of giving the KNN the binary variable pressure as an outcome, it would be the x-offset, y-offset, speed, acceleration, and orientation. Naturally, five separate models need to be run. Similarly to the first KNN, the model would put the data of every single frame into a space with all of the provided features. A new observation would then be placed into this space and the k-nearest neighbors could be used to provide a prediction of all five telemetry data points. Logically, here, a regression KNN would have to be used. The model would, instead of comparing the classification of the k-nearest neighbors, take the offset to the k-nearest neighbors and calculate the average out of those, returning the predicted value.

When analyzing this idea, before further evaluation of this model, in theory, the validity of it should be given. When taking any possible frame of a play, based off of all the telemetries in this frame, a model that is taking the k most similar plays and is generating an average value for the next frame, that new value should be a fair representation of what the next location, speed, acceleration, and orientation of the pass rusher should look like, especially considering the amount of data at hand.

With this newly generated information, an entirely new set of telemetry data can be generated, one that is identical in composition to the original one, but instead using the predicted telemetries to generate a new location for the pass rusher. Since this dataframe has the same structure, the same KNN can be used to predict the pressure magnitude for this predicted location, returning the amount of pressure that the “average” player would have achieved in this situation. This can be done frame by frame, allowing for a direct comparison of the real pressure created, and the one for the predicted motion. This entirely new metric is the most valuable asset for player evaluation and analysis.

5.4.2 Data Manipulation

An extensive amount of data manipulation had to be done to ensure that the KNN returns valid predictions, both for the KNN that predicts the location, speed, acceleration, and orientation of the rusher in the next frame, as well as the one that predicts the probability of pressure occurring for the new location. The reason the motion prediction is even possible is due to the knowledge of the required telemetry data, as these values can be used as outputs of the model. While speed, acceleration, and orientation can be used straightforwardly, the issue here arises that the offsets per frame always indicate the offset to the last frame. Predicting this would be nonsensical, as they are already calculated values. Instead, the offsets have to be shifted by one frame backwards, giving the model information on what the x - and y-offsets were in the next frame, the one that is to be evaluated. The offsets from the first frame are not of use (they were arbitrary set to zero anyways). To shift the data, all offsets were put into the list, popping the first entry and appending the list with a new value. The value of this last entry could again be arbitrary, as the last frame for every play would be deleted entirely anyways, since a prediction of the next location is not necessary, since the play is over, and a invalid frame would fill the KNN with faulty data. This now means that every play would be shortened by one frame each for this KNN. Running the KNN as described earlier would return floating numbers for the x-offset and the y-offset, as well as for the speed, acceleration, and orientation. All of these number are rounded to two decimal places, as they are of this form originally.

For the next step, an entirely new set of telemetry data would have to be created that looks identical to the telemetry data calculated or extrapolated earlier. Determining all the needed telemetry is not difficult, given the new offsets determine the new location of the rusher and from this, distance to the passer, distance to the blockers, etc., could be calculated. The issue here is, though, that the predicted offset indicate the location for the next frame, meaning the distances to the passer and blockers would have to be calculated with the telemetry of the next frame. Once this has been established, all telemetries can be calculated or extrapolated in the exact same manner as for the initial pressure prediction KNN, simply by extrapolating the data from the next frame. Speed, acceleration, and orientation of the blockers are given anyways, and the speed, acceleration, and orientation of the rusher were predicted. The procedure of finding the closest pass blocker has to be gone through again as well, though, since the new location of the rusher may find a different pass blocker to be the closest.

Since for the last frame, no offsets, speed, acceleration, and orientation were predicted, the last frame would again have to be taken out of the model, but, one final step would have to be conducted to finalize this model. This KNN now predicts the pressure for the motion occurring on the next frame, since all telemetries are based on the location in the next frame. This means that, to be able to compare the predicted pressure based off of the predicted motion with the predicted pressure with the actual motion, all newly predicted values have to be shifted by one frame forward again. This now leaves an empty value for the first frame, since there is obviously no motion that precedes that of the first frame. To compensate for that, the actual pressure was simply chosen as the starting value. From there on out, comparisons frame-by-frame can be made between the true predicted pressure and the one of the motion that would be predicted for that frame.

5.5 HTML - and JavaScript Based Dashboard

As a final step, an single-page, HTML-based webpage was build, making use of JavaScript, or more specifically, D3. D3 is a JavaScript library, specifically developed for manipulating data, by binding arbitrary data to a Document Object Model (DOM), which allows for the data to be transformed in any way desired, making use of Scalable Vector Graphic (SVG). D3 is a great way of creating not only unique graphs, but also smooth animations with the data. Visualization is a key component in making theoretical concepts more understandable, which in this case is useful to both the data scientist, as well as the American football player or coach. Since precise data of the location of every player on a play, with 10 frames per second, is available, an exact recreation of every play is possible via an animation. The goal here was to build a dashboard that allows the user to visualize any play from a top-down position, while at the same time providing key information on the determined pressure by the KNN model mentioned earlier. Only week 8 was used for this iteration of the dashboard, using different types of transformations of the data to depict and explain how and when pressure was applied by individual player and the entire defense on a play, both by creating animations of the play, as well as graphically. The website can be accessed under http://week8dashboard.x10.mx/. A more in-depth explanation on how to navigate and use the webpage will be given later on, but the user is advised to allow for the data to load when accessing the website, which may take a few seconds.

6 Results and Discussion

6.1 Predicted Pressure Model

6.1.1 Model Accuracy and Validity

Evaluating the accuracy and validity of the KNN model predicting pressure on a play is not a straightforward task. As mentioned, an accuracy of 87.17% was achieved, which, at first glance, appears to be a respectable number. The issues with this metric becomes obvious when looking at the specificity and sensitivity of the model, though. The specificity here represents the amount of frames that the model has correctly predicted to not have been a pressure, compared to the sum of the correctly predicted frames that did not end in a pressure, plus the frames falsely having been predicted to have ended in a pressure, while in actuality not having ended in a pressure. The sensitivity in this case represents the amount of frames correctly having been predicted to have ended in a pressure, divided by the sum of correctly predicted frames that ended in a pressure, plus the frames falsely having been predicted to not have ended in a pressure, while actually having ended in a pressure. While the specificity comes out to an impressive 99.20%, the sensitivity is only at 8.96%. While this can be seen as worrisome at first, an explanation for this can be found quite easily. The issue with the data at hand is that only the overall result of a play is available to the model. This means that, especially for the first few frames of a play where pressure is naturally low (or even zero), as the Quarterback is naturally fully protected by the Offensive Line, and values for speed and acceleration are relatively low, the model classifies these frames into the “no pressure” category, while the play overall might have included pressure, and therefore the prediction would be considered false. As there is no information available on when exactly the pressure has occurred during a play (which is why this metric is being created), this is the only way of dealing with this problem, by having to feed the model with “false” information. This is why KNN is such an ideal choice, though, as it is able to compensate for the fact that some of the data is skewed, which also results in the probability being a fair way of metricizing pressure. The overall prediction of whether or not pressure has occurred per frame can be entirely neglected when it comes to the accuracy of the model. So even if a prediction on a frame is classified “wrongly”, it is the percentage that matters in this case. From a logical perspective, the model returning a low number in the early frames of a play, due to the majority vote of the KNN, is entirely what is trying to be achieved here, even if later on in the play a pressure has occurred, classifying the prediction wrong, and skewing all classification metrics. Therefore, a better way of analyzing the validity of the play has to be found.

With the PFF Scouting data simply classifying whether or not a pressure has occurred on the play, the identical thing can be done for the predictions. For each play, the highest number of pressure predicted will be an indicator on whether or not a play has contained pressure. With the percentage for this KNN having been set at 50% to classify a frame having induced pressure, this means that if only one frame has had a percentage of more than 50% pressure probability on the play, the entire play will be classified to have had pressure created. The accuracy with this method improves slightly to 87.91%, and even though the specificity drops to 89.70%, the sensitivity now has a drastic increase to 73.88%. A deeper analysis now could obviously be done by adjusting the percentage required for the model to classify a frame to have contained a pressure, but these numbers were deemed sufficient for a valid representation of the pressure on the play.

An additional way of analyzing the validity is to look only at the players that were able to achieve a sack on the play, with the sack being the perfect way of determining pressure. A sack ends the play and finds the pass rusher to be right at the Quarterback. This means that, in a valid model, the pressure on these plays would naturally be very high and the model should predict those plays to have contained pressure. For sack plays only, the model has an accuracy of 95.90%, with the average highest pressure on all of those plays coming out to be 84.73%, again giving another form of validation to the model.

As a final, less scientific, but in this case strong way of validating the model, the eye-test comes into play. Animations of each provided play for week 8 can be found in the aforementioned dashboard, along with an animation of the pressure during the play. Alongside actual video footage of any play, the predicted pressure can be compared to what is occurring on the field. It is fairly easy to give an educated estimate on how much pressure a pass rusher is able to create for the duration of a play, as in whether or not the Quarterback is disturbed by their presence, or the pass rusher is clearly being held up, or even blocked. These observations can then be compared to the numbers provided by the model, which appear to give a realistic representation. Figure 10 shows the pressure created by Indianapolis Colts' DeForest Buckner (`nflId` 43296) on play 4334 against the Tennessee Titan (`gameId` 2021103106) in week 8. Buckner's pressure pre-snap is relatively high due to the way he lines up on the play, as this constellation is considered relatively like to have ended in a pressure by the model, and therefore returning a higher percentage number, ergo more pressure. The pressure drops around frame 10, as Buckner is faced head-on by the pass blocker, and this matchup is continuing until around frame 20 to 25 when Buckner is able to beat the pass blocker and is able to create a clear path towards the Quarterback of the Tennessee Titans, Ryan Tannehill. From there on out, the pressure increased significantly, with the model maxing out the pressure at 93.28% at frame 45, achieving the sack, and resulting in an 8 yard loss on the play for the Titans, as well as unfortunately injuring Ryan Tannehill in the process.

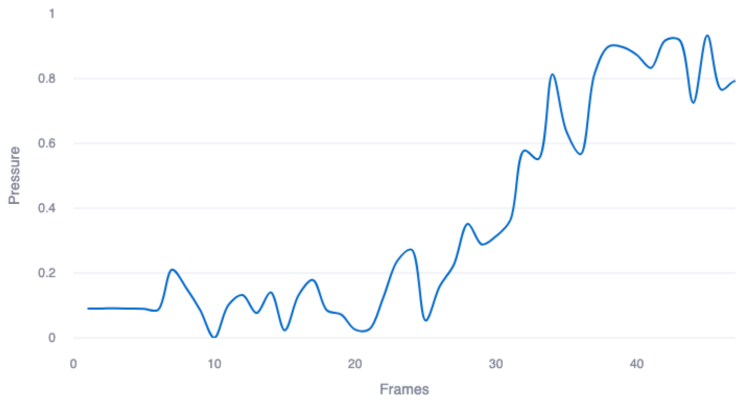


Figure 10: Pressure created by Deforest Buckner on play 4334 against the Tennessee Titans, resulting in a sack

6.1.2 Team Performance Analysis

The advantage of a new metric, especially in sport sciences with the competitiveness in mind, is the fact that it allows for rankings to be created of all sorts, both for players, as well as for the teams. It will be explained further on why this new metric is not entirely ideal for individual performances, but coincidentally allows for major, new ways of analyzing team performances. Initially, instead of looking at the pressure of an individual player on a play, the pressure of the entire pass rush can be investigated. The average pressure of the entire pass rush on every play can be calculated, therefore also allowing for the calculation of the average pressure the pass rush has created amongst all plays the team has been a part of. Similarly, since the opponents of the teams are available for every play, the average pressure that each team has allowed amongst all plays can be calculated as well. From here, the pass rush, as well as the pass block from every team can be ranked accordingly. The average pressure created by the pass rush of each team can be seen in Figure 11, while the average pressure allowed by each team's pass block can be seen in Figure 12. The Carolina Panthers are found to have created the highest amount of pressure with an average of 13.34% per play, while the Atlanta Falcons created the least amount of pressure within the first eight weeks of the 2021 NFL season with 11.70%. The Tampa Bay Buccaneers allowed for the least amount of pressure to be created by their opponents with only 11.45% on average, while the Indianapolis Colts allowed the most pressure with 14.53% on average.

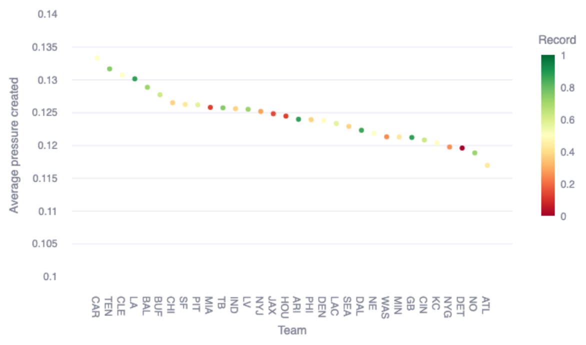


Figure 10: Average pressure created by each team along with their respective win percentage

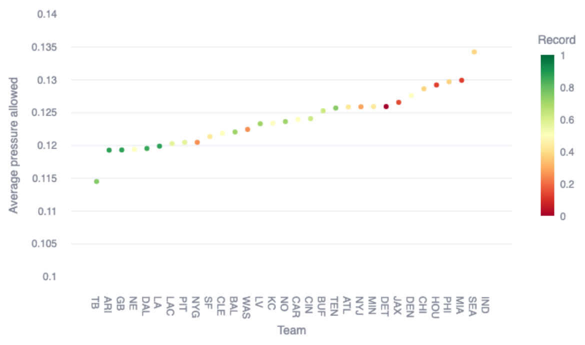


Figure 11: Average pressure allowed by each team along with their respective win percentage

The color of each point represents the winning percentage, with the more green meaning a team is closer to a winning percentage of 1, while the more red meaning a winning percentage closer to 0. Naturally, the more pressure the pass rush creates, the better, while the opposite is the case for the pass block. It has to be noted that, for the entirety of the team-specific analysis, only the frames were accounted for once the ball was snapped to the Quarterback. The duration between the start of the measurements and the actual snapping of the ball can at times be several frames, while at other times the measurements started with the snapping of the ball. While for player-specific analysis, keeping the frames before the ball was snapped can be of benefit, as the motion a pass rusher does prior to the snap can have an impact on the pressure they are creating/will be creating throughout the play, while for the team-based analysis these frames can skew the data, as naturally these frames will include little pressure. By eliminating these frames a more fair analysis can be made.

Success in sport is typically measured by wins, and maximizing every aspect of the game to increase the chances of success will always be of highest priority of any team. With this in mind, it is logical to further analyze the direct impact that both the pass rush and the pass block have on the success of the team, as in whether or not the amount of pressure created/allowed has any correlation to the team's winningness.
The win percentage after the first eight games can be analyzed, but it has to be mentioned that this can be a relatively small sample size in terms of games played, being only about half-way into the season. An example for this are the Kansas City Chiefs: after week 8, their win-loss record was 4-4, putting them at a winning percentage of 50%, while eventually going on a run and finishing the season with a 12-5 record, putting their winning percentage at 70.59%, making the playoffs and just barely losing out on going the Super Bowl, losing to the Cincinnati Bengals 24:27 in overtime in the AFC Championship game. While, of course, many variables can change throughout the course of a season, for the purpose of this analysis it will be assumed that the average pressure created and allowed over the course of the first eight weeks is a fair representation of that throughout the season. The distribution of pressure created and pressure allowed compared to the winning percentage of a team after the first eight weeks can be seen in Figure 12 and 13. The distribution of pressure created and pressure allowed versus the winning percentage at the end of the regular season can be seen in Figure 14 and 15.

Doing a straightforward Pearson correlation, the linear relationship between pressure created/ allowed per team and the respective win percentage can be established. For the average pressure created by the pass rush versus the win percentage after eight weeks, the Pearson correlation coefficient was found to be 0.2160, with a p-value of 0.2351. For the pressure created by the pass rush versus the winning percentage after all 17 weeks, the Pearson correlation coefficient was found to be 0.1013, with a p-value of 0.5814. For the pressure allowed by the pass rush versus the winning percentage after eight weeks, the Pearson correlation coefficient comes out to be -0.4947, with a p-value of 0.0044, while after all 17 weeks the Pearson correlation coefficient dropped to -0.350, with a p-value of 0.049. It can therefore be seen that for the pressure allowed by the pass block, a slight negative correlation (logically, as the less pressure allowed, the better) with the winning percentage could be determined, both being considered significant due to their p-value at an alpha level of 0.05.

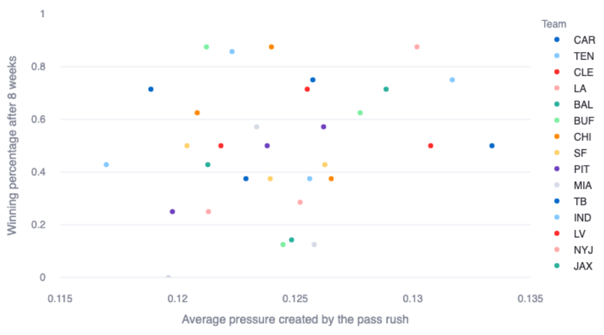


Figure 12: Average pressure created by each team versus the team's winning percentage after 8 weeks

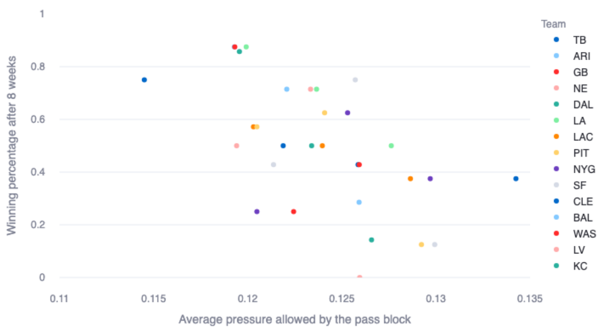


Figure 13: Average pressure allowed by each team versus the team's winning percentage after 8 weeks

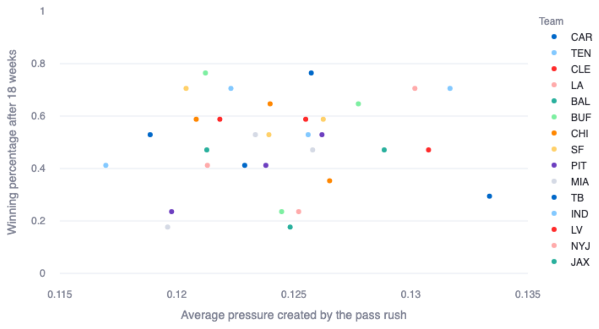


Figure 14: Average pressure created by each team versus the team's winning percentage after 18 weeks

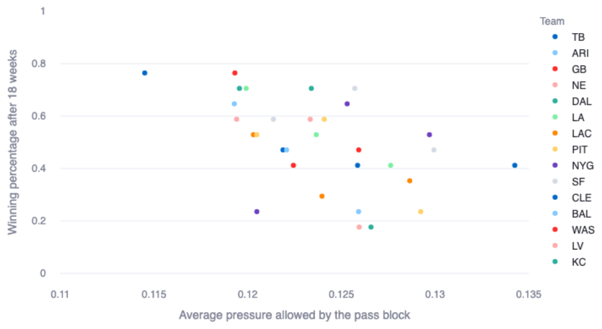


Figure 15: Average pressure allowed by each team versus the team's winning percentage after 18 weeks

The following additional ranking were created and can be found in the appendix:

1. Average pressure created by the pass rush per team (11.1.1)
2. Average pressure allowed by the pass block per team (11.1.2)
3. Average pressure created by each position (11.1.3)
4. Average pressure created by each formation (11.1.4)
5. Average pressure created by the pass rush per team for the NLT (11.1.5)
6. Average pressure created by the pass rush per team for Cover-2 (11.1.6)
7. Average pressure created by each position for the Green Bay Packers (11.1.7)
8. Average pressure created by each formation for the Los Angeles Rams (11.1.8)
9. Average pressure allowed by the pass block per team for the LLB (11.1.9)
10. Average pressure allowed by the pass block per team for Cover-3 (11.1.10)
11. Average pressure allowed by each position for the New York Giants (11.1.11)
12. Average pressure allowed by each formation for the New York Jets (11.1.12)

For rankings 5 to 12, only one team/position/formation was chosen to be shown here, just to give a general sense of how these rankings look like. Creating the same rankings for other teams/ positions/formations can be done straightforwardly with the Python code provided, also in the appendix. Teams/positions/formations will be towards the top of the respective ranking, the higher the created pressure, or the lower the allowed pressure for each category.

From a statistician's point of view, this new metric has opened an entirely new field of analysis that could be further investigated, as for every play it is now known how much pressure each position created, how many blockers that position had, which formation the defense was in, and obviously, who the opponent was. It would therefore, from a pure statistical view, be of relevance to find the correlations within these categories, and determine the ones who have a significant impact on the amount of pressure that is being created. But considering that this is the field of sport science, it is believed that this would not necessarily be the most beneficial step to take. The reason for that being is, while data analytics play such a vital role in the world of sports, a purely statistical approach would mean a generalization, which is not always helpful to teams and their performance, which is what should be of highest priority. Sports, especially American football, is a field of fluidity and spontaneity, where not everything can be seen as entirely binary. Momentary and situational impact is a difficult metric to calculate, as too many factors come into play. A perfect example for this is the analysis done here comparing the pressure created/allowed to the winning percentage. Following a naive approach, pass rush could be deemed as entirely insignificant to the success of a team, and could therefore lead to the assumption that teams should not put their efforts and resources into it, while in reality, the impact that specific roles can have on the game is not always as clear. A good pass rush may be able to bring pressure in decisive situations, creating a turnover in just the right moment. The “bend don’t break mentality” is a philosophy followed by many defensive coordinators, in which a defense may allow the offense to progress the ball, but eventually not allow for a touchdown to happen, therefore tiring the offense out, as they would spend long and grueling time on the field, only to come up empty handed.

An example to showcase this is the amount of pressure created by each position, which can be seen in Figure 16. The NRT (Nose Right Tackle, a type of Defensive Tackle) position is found to created, on average, the least amount of pressure. But when looking at the distribution of pressure created by each team for every position (Figure 17), the NRT of the Green Bay Packers is found to create the most amount of pressure amongst all NRTs from different teams. Now, when looking at the amount of pressure created by each position from the Green Bay Packers, the NRT ranks 7th out of 18, the highest out of any of their Tackle positions (the full ranking for the pressure created by each position from the Green Bay Packers can be found in the appendix in section 11.1.7). So even though the NRT is overall the position creating the least amount of pressure, to the Green Bay Packers, this position has turned out to have made a siignificant impact. So even when doing any form regression model, an interaction term between the team and the position may turn out to be of significance, this would still be a generalization of the whole matter, and not be of benefit to the team, as they do not necessarily have any interest in finding general correlations between certain factors and the pressure created, but rather are interested in their own data, their performance, and how that compares to the rest of the league. The correlations and causations have to be done on a team-specific level, with the correlations being established by the team’s data analyst, and the causations being investigated by the coaches, as they are the ones with the highest knowledge of the game. From a data scientist’s perspective, on a league-wide level, providing the metric, and allowing for all types of rankings to be created, can be considered a completed task.

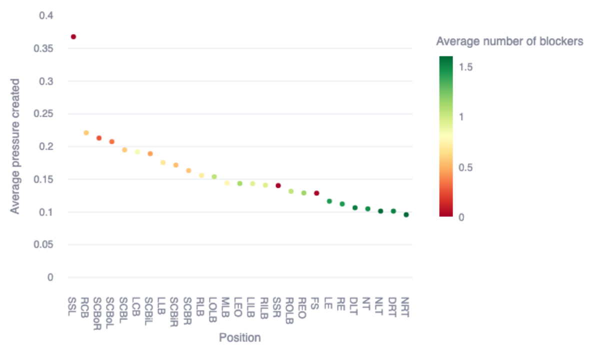


Figure 16: Average pressure created per position

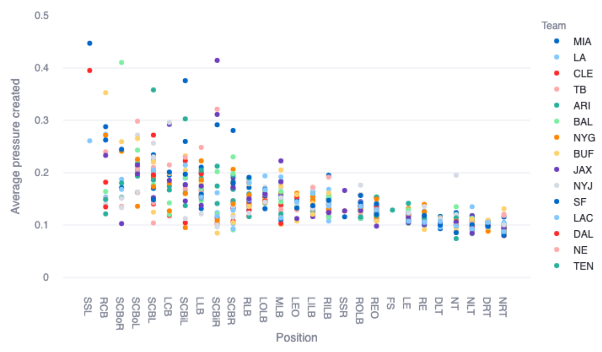


Figure 17: Average pressure created per team per position

The aforementioned example of the Green Bay Packers would be exactly how this metric and these created rankings are intended to be used on a team-specific level. On the one hand, teams have now received a new, more precise metric to evaluate the pass rush, allowing for teams to analyze both their own defense, by not only seeing which positions create the most pressure within a team, but also how their own positions compare to the rest of the league, and the offense, identically seeing which positions from the opponents were able to create the most pressure over the previous weeks, or which formation was the most successful against the team, and are therefore able to adjust their game plans going forward. But also opponent-specific analysis to prepare for upcoming games have now received a new metric to be used, as teams are able to analyze the positions and formations that have created the most pressure against or by the next opponent, allowing for strategies to be adapted to the opponent with far more accuracy than previously possible.

The opportunities for analysis within just one team have exponentially increased with this new metric to evaluate pressure, and it is up to each team on how to implement it. Interaction terms within one team may open the door to entirely new discoveries within the tactics and strategies of the game.

6.1.3 Ineffectiveness of the Model for Player-Level Analyses

An analysis of the players' performance would be the logical next step to take, naturally, as it was the individual players for whom the pressure was calculated. Unfortunately, this analysis is not quite as straightforward as one might initially believe for it to be. Calculating the pressure players were able to create over all of their plays and putting all players into one ranking based on this statistic would be highly unbalanced and quite simply unfair, as it would be like comparing a Formula 1 car with a rally car based on the same metrics. Similar to how each car is build for a specific purpose (the Formula 1 car being built specifically for circuit racing on paved tracks, being optimized for high straight-line - and corning speeds, while the rally cars is built for off-road racing on different types of surfaces), each position in the NFL has a different task that is linked to specific scenarios and different outcomes. Different positions require different body types and shapes, as a Wide Receiver ideally has a figure for fast acceleration, while reaching high speeds to create separation from Cornerbacks and Safeties, while Offensive Linemen need to be as powerful as possible to withstand the pressure of the pass rush, with no need to sprint for extended periods of time. This is why players are typically bound to one position, only switching sides, or at most switch to a position that is as similar as possible, as their physique would simply put them in a disadvantage at an entirely different position. This was already seen when comparing the average amounts of pressure that each position was able to create during the pass rush. It is due to the nature of the game that a Linebacker, when attempting the pass rush, will not be faced with as many blockers (if any), compared to a Defensive End who is opposing a pass blocker right from the start of the play. This is exactly the point of the Linebacker, as with engaging a Linebacker in the pass rush, there is a certain risk-and-reward that comes with it: the additional pressure created by the Linebacker due to the outnumbering of rushers to blockers, will give the Quarterback less time to throw the ball, while leaving more space open in the secondary part of the defense for a Wide Receiver to catch the ball. This can nicely be seen in Figure 18, showing the average pressure created by every player, regardless of their position with respect to the average amount of blockers that each player faced. Obviously, the more blockers a player has to face, the less pressure they will be able to create, but is exactly this what makes a player analysis, even when looking at each position individually, difficult. Even for the same position, even though one player might have created a higher average amount of pressure, the average amount of blockers may have been smaller as well, compared to a different player, making the comparison unfair. The average may also be skewed due to plays in which a pass rusher was left unassigned, and therefore created a high amount of pressure throughout the play.

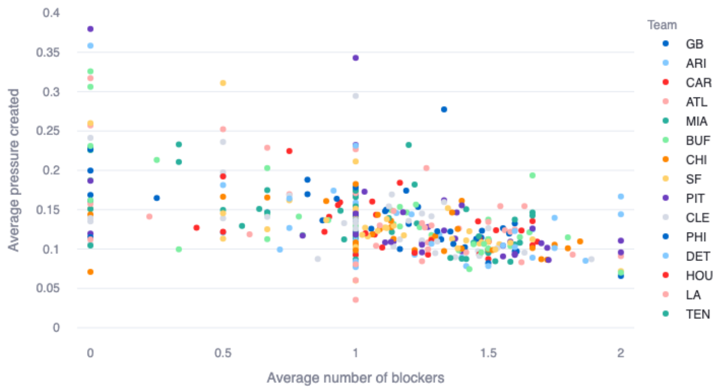


Figure 18: Average number of blockers vs average amount of pressure created for every pass rusher

This does not mean that this metric is not useful for analysis, it just shows that, again, a generalization is difficult to do, due to the fact of the vast numbers of factors that play into account in a competitive setting. Alternatively, this metric should be used as a tool for players and coaches for play-specific analyses of players. Individual plays can be analyzed to understand what a player did well, or where they could have improved. Questions like why a certain play did not create as much pressure can be answered far more easily now that a metric for every frame of the play is available, instead of only having an overall result of the play. Identically, a team like the Tennessee Titans would naturally further analyze play 4334 in their game against the Indianapolis Colts that was mentioned earlier, in which DeForest Buckner sacked Ryan Tannehill and injured him on the play, to understand why Buckner was able to break through, while now having a more precise metric to evaluate the pressure received frame-by-frame, especially considering the gravity of that play.

This does of course now also reward players who were not credited with a binary form of pressure on a play by Pro Football Focus, while in actuality having created a fair amount of pressure percentage-wise, further changing the perception and understanding of pressure.

Nonetheless, being able to actually compare pass rushers with one another should still be the goal to be able to properly evaluate a player's performance to the rest of the league. And while a direct comparison was shown to be a difficult task, it is believed that the Predicted Motion Pressure was not only able to solve this issue, but also allow for a comparison of every pass rusher, regardless of position.

6.2 Predicted Motion Pressure Model

6.2.1 Advantages of the Predicted Motion Pressure Model

It has already been established that American football is a situational, but fast and reactionary sport. It is this characteristic that can be used to create not only a metric that is a more precise way of measuring pressure (as it was done in the previous sections), but rather an entirely new metric, the Predicted Motion Pressure (PMP). The reason PMP is able to function, is due to the fact that the difference in the pressure being evaluated from the predicted motion to that of the actual motion is *only* due to the motion of the pass rusher. Every other variable can be taken over from the actual occurrences on the field, which means that the position of the pass blocker and the passer can be entirely accepted in the model, giving the model a strong form of validity. This is because of the reactionary nature of pass rush and pass block. Similarly to chess, in which the black pieces have to react to the moves being done by the white pieces (at least in the first stages of the game), the pass blocker is the one reacting to the moves being made by the pass rusher. The pass rusher is the one that has to be able to get past the pass blocker, while it is the pass blocker who has to analyze the moves of the pass rusher and react accordingly. If a pass rusher moves towards the right, so will the pass blocker, to not allow the rusher to get past. This concept is the most fundamental foundation of the PMP.

Additionally, the frequency of measurements being made at 10 per second, allow for a fairly precise evaluation. With plays commonly being only a few seconds long, the moves being made have to be split-second decisions. The more time would pass between measurements, the larger the differentials in speed, acceleration, orientation, and position, leading to less accurate predictions, and therefore less powerful evaluations. From a data manipulation point of view, predicting the motion several frames into the future would be entirely possible, while tedious. The issue with this would be that, due to the vast number of factors influencing players' motions, the entropy of the data on the play would increase with every play, as the predictions would move further and further away from the actualities. As an example, if a pass rusher is predicted to make a move towards the right side of the field, while in actuality moving towards the left, the pass blocker will, with some delay caused by having to react to the motion, obviously also move to the left. Now, if the predictions of the motion would be used to predict the following frames, the player would be likely to continue going further right, especially since the pass blocker would move further to the left, as the real motion is being used there, encouraging the model to even further move the pass rusher away from the blocker, resulting in entirely faulty predictions that are unusable for any form of evaluation. If something similar would be desirable to do, the motions of the pass blocker would also have to be predicted, which would lead to less valuable comparisons between the predicted and actual motion of the pass rusher.

Being able to keep all variables besides the motion of the pass rusher to be identical, allows for a straightforward analysis of the efficiency of a pass rusher. The difference in pressure created for every single frame essentially displays how good or bad the move of the pass rusher was, compared to an average of all other pass rushers throughout the first eight weeks who were in a situation as similar as possible to the one of the pass rusher being evaluated. This is, again, why KNN is such a fine choice here, not only because of the predictions that it is able to make, but also because the concept behind it gives validity to the evaluation.

This delta in pressure is now an entirely new metric that pass rushers can be evaluated by, both over the duration of the play, and for situation-specific instances, in which a certain move on a specific play can be analyzed by the coach or player to determine whether a player could have done a better move, or if the one made was actually above average and being the reason for success on the play, allowing for entirely new insights into the motions, effectiveness, and efficiency of pass rushers.

An additional advantage point of this new metric is that it is entirely independent of the role of a player, as well as the outcome of the play. It is completely non-detrimental whether a pass rushers had two blockers, creating close to no pressure on the play, or was able to achieve a sack on the play, creating maximum amount of pressure. It is the efficiency and the impact of every move that is being compared, so even if the pass rusher did have two blockers on a play, some of the moves that were made could have created more pressure than would have been predicted, but due to the fact that there were two blockers, it was simply a suboptimal situation for the pass rusher to get out of. On the other hand, even though a player achieved a sack on the play, there may have still been situations in which a better move could have been made, even though naturally, if the play ended in a pressure, it can be assumed that the moves made were efficient anyways.

It becomes obvious that this metric does not evaluate the ability of a player to create pressure on the play, but rather uses the earlier created metric of pressure to evaluate a player's decision-making abilities. According to the National Strength and Conditioning Association, anticipation appears to be a trainable quality, so giving players a metric that allows them to evaluate their decision-making abilities could see further improvements in the players' performances.[11]

Lastly, unlike in the pressure metric, the amount of times a player engaged in the pass rush is not detrimental to the evaluation process. When just looking at the pressure, the Cornerbacks and Safeties did achieve an above average amount of pressure, which has two reasons: for one, naturally, these positions being engaged in the pass rush is usually done as an element of surprise, risking the open Wide Receiver, while at the same time having a high probability of not being assigned to a pass blocker, creating a high amount of pressure from the get-go, therefore achieving a higher average. But it is exactly the rare usage of these positions in the pass rush that can lead to a higher average, as they have a much smaller sample size, skewing the average in their favor, creating an imbalance in the data. For the PMP, this is not as much of a problem. Of course, if a player engages more frequently in the pass rush, their average will eventually level out to a true value, but nonetheless, a smaller sample size, or even just one engagement in the pass rush can be used to determine efficiency to a certain extend.

6.2.2 Player Performance and Efficiency Analysis

An illustration of the PMP in comparison to the actual pressure created can be seen in Figure 19. Here, Harold Landry's (`nflId` 46110) pressure and PMP in the Tennessee Titan's game against the Indianapolis Colts (`gameId` 2021103106) in week 8 on play 225 can be seen. The actual pressure is colored blue, the PMP red. Due to the fact that the play was relatively long, with about 7 second, and Landry not being able to create much pressure during the early parts of the play, he was still able to generate pressure on the Colt's Quarterback Carson Wentz towards the end of the play, resulting in Wentz throwing the ball away and not completing the pass, bringing Landry's pressure to a solid 17.39%. Simultaneously, the delta between the actual pressure and the PMP on the play for Landry was at a impressive 3.1508, ranking this as the 160th best play amongst the 36362 available plays, meaning that Landry did not only do an obviously good job by creating pressure, but was also able to generate much more pressure than would have been predicted for him to achieve.

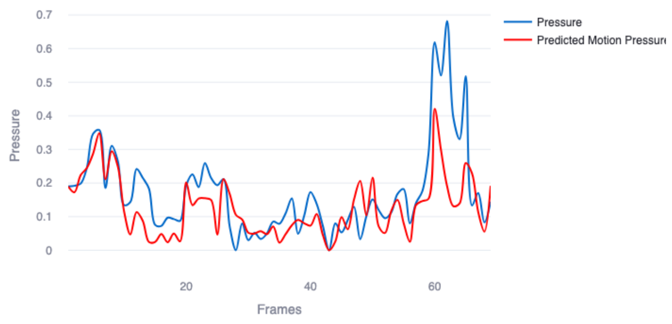


Figure 19: Harold Landry's pressure and PMP on play 225 in the game TEN vs IND

Naturally, the higher the delta, the better. For all plays, the delta per play averages out to 0.1929. Rounding the delta per play to one decimal place and looking at the amount of occurrences, it can be seen that the distribution is normal. This distribution can be seen in Figure 20. The highest delta per play within the first eight weeks was achieved by Arizona Cardinals' Tanner Vallejo (`nflId` 45008) on play 2855 against the Houston Texans (gameId 2021102408), with a delta of 6.9974, while the lowest delta was achieved by Denico Autry from the Tennessee Titans on play 742 against the Buffalo Bills (`gameId` 2021101800), with a delta of -4.5858. The 20 highest deltas can be found in the appendix, section 11.2.1, while the Python code to create the full list can be found in section 11.3.4. The presence of players in this whom are known for their pass rushing abilities like Aaron Donald and Josh Allen (twice in the top 20) should not come as a surprise, but rather as a confirmation of the validity of the PMP.

Looking at Figure 21, the distribution between the average pressure created and the pressure to PMP delta per play can be seen. A significant correlation at an alpha level of 0.05 could be found, with a Pearson Correlation coefficient of 0.4904. While this correlation is logical, since a high pressure on a play is likely to have originated from a good motion by the pass rusher, it is of course not guaranteed, stressing the validity of the PMP as a separate analytical metric that can be used for individual evaluations. Again, it is not necessarily the outcome of the play that should be analyzed when looking at the delta with the PMP, but rather the efficiency of the player, so the two metrics should be considered independent from one another.

The overall efficiency and decision-making abilities of a pass rusher can be evaluated by computing the average of the PMP delta over all pass rushes conducted by a player. The 20 highest average deltas can be found in section 11.2.2, with Grant Delpit (`nflId` 52452) topping that list with an average delta 2.4524.

While the PMP is a metric based on an average on the motions of all the pass rushers, the delta that is being created here comes with a highly personal characteristic. While the pressure that a player creates in a frame is completely dependent on the overall situation, the difference in pressure and PMP is result of the decisions of the pass rusher, and that alone. Players with higher delta values have only come to that due to the well made decisions. These decisions can, and should, of course be further analyzed by the teams, coaches, and players. There is no once-in-for-all answer to what a player has to do to do achieve a higher amount of pressure with their next motion, as every play is unique and too many factors would have to be accounted for. Rather, this new metric can be used to study individual situations to recognize patterns in the player's behavior, both good and bad, again to improve the performance. A generalization for teams or positions is again of no use, as it is the individual that needs to be analyzed. The rankings created allow for all the players to compare themselves to the rest of the league, while especially giving the opportunity to study the higher-ranked players.

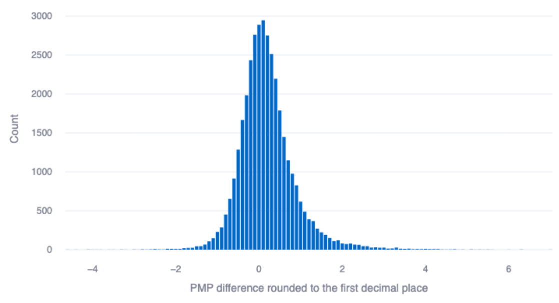


Figure 20: Distribution of the PMP delta

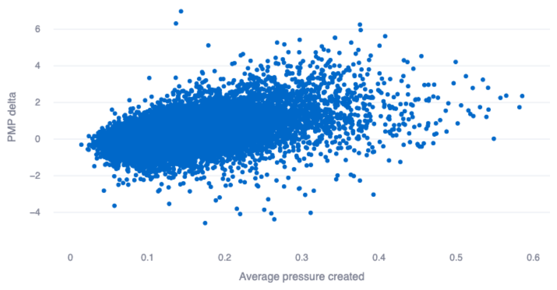


Figure 21: Average pressure vs PMP delta per play

6.3 Deployment in the Form of a Dashboard

The visualization aspect of this thesis has always been of highest priority. On the one hand, this is due to the intricacy of the sport, the game itself is highly tactical and chaotic, with all the established factors having an influence on pressure created by the pass rush. On the other hand, describing the motions of an American football player and being able to fully grasp the concept of the created metrics to the fullest extend is difficult to achieve on static paper, which this thesis is. Because of that, creating a dashboard that allows the user to have animations at hand to show not only the entirety of the play, but also the pressure created by the entire offense and individual players, was seen as a beneficial step to take under the premise of this thesis. While Python is an incredible tool for data analytics and data visualization to a certain extend, the animations in mind would put Python to its limits. With the knowledge gained from the Data Visualization course taken for this Master's trajectory, shifting to HTML and JavaScript appeared to be the ideal solution. Especially with D3 being optimized for visualization by using SVG objects, animations can not only be run smoothly, but also in high quality. With this in mind, a dashboard was created using the predictions for pressure made on week 8 of the data, allowing for the user to choose any game and play played during this week and is able to visually analyze the play and the pressure. Two screenshots from the dashboard can be seen in the appendix in section 11.5.

Firstly, with the two dropdown windows the user is able to select which game and play they would like to see. After the selection, the first animation window will show the entirety of the game frame by frame, keeping 200 ms between each frame, meaning half of the speed of the movements in real time. This was chosen to give more time to the user to take as much in as possible. Still, the animation is able to create a smooth recreation of the play, also because of the precise data provided by the NFL. The proper abbreviations of the teams will be written in the respective end zones, as well as useful information of the play, such as a description of the play, score in the game, and offensive and defensive personnel, will be provided to the right of the animation. The color of each team is equal to the actual jersey color that each team wore during the game.

In the second window, the same play is animated, also at half the speed, this time focusing on the pressure created by the pass rush. The opacity of the color of the pass rushers will increase, the more pressure an individual pass rusher is able to create. To the right, a line graph will animate the total pressure created by the pass rush frame by frame, in synchronization with the animations of the play. Alongside, information on the maximum and average pressure created can be found.

Lastly, the user is able to select one of the pass rushers from the selected play. Again, an animation of the play is shown in the third window, this time highlighting the chosen pass rusher. To the right, the first line graph shows an animation of the pressure created by the pass rusher per frame with a blue line, while the second line graph shows the same animation, but in addition also the Predicted Motion Pressure with a red line, to be able to compare the two. Additionally, information on the maximum and average pressure created by the pass rusher, as well as the delta between the actual created pressure and the PMP over the course of the entire play is shown.

Every animation will automatically restart once the play is over. At any point a different play can be chosen to be viewed without interference of the previous play.

7 Ethical Thinking, Societal Relevance, and Stakeholder Awareness

7.1 Stakeholder Awareness

The National Football League is hosting some of the largest franchises in the world. According to Forbes, the Dallas Cowboys are the most valuable amongst the 32 teams in the league with an evaluation of 5.7 billion US Dollars.[12] It's become quite clear that the NFL has evolved from being merely a league that is governing a sport, to a high-profile business. The success of a team can have huge impact on the financial evaluation of the team with teams handling income numbers in the hundreds of millions. Bringing data analytics and the creation of new metrics into the game always comes with the main goal in mind: to improve the game in any shape or form. But while the impact of these analyses can be quite obvious for players and coaches, the list can be extended when looking at the bigger picture. The owners of the club have a huge interest in finding ways to improve performance, as this is directly correlated to a financial gain. The analysis done in this thesis was done purely from the point of view of a data scientist, it is up to the team to do individual analysis based on their needs, as this thesis merely provides the new metric along with a general analysis. But a deeper dive by the teams may uncover newly found tactics, which can make the game from an entertainment standpoint better or worse, which therefore affects the millions of fans all around the globe. The aspect of player safety will be touched upon in the next section, but the health of the player is something that not only affects the players themselves, but also the coaches, and, not to forget, the team physicians. It can be seen that the NFL is an intricate operation, where many nodes of the web are connected with one another. So one small change in the structure can have a huge impact on many people, and while no predictions want to be made on the impact that this new metrics will make on the entire NFL, the awareness for it has to be ever present. On top of the stakeholder list here will always be the player, though. The game itself should always remain the main focus point, and players, as athletes would look to use any information provided to improve their own game. Then again, for the players, this is also a profession, often as the main source of income. Improvements on the side of the player can be difference in millions of Dollars in the next contract. It is clear that, even just for one stakeholder, the impact of change can reach many different layers.

7.2 Societal Relevance

7.2.1 Player Safety in Regards to Players, Coaches, and Fans

As mentioned earlier, injuries are not an uncommon occurrence in the NFL, mainly due to the fact that American football is a full-contact sport. The NFL is regarded as the best American football league in the world, with the rosters of the teams being filled with professional, high-performance athletes, so staying as injury-free as possible is always a main aspect of their routines. Reducing injuries as much as possible should always be high on the agenda for every sport, while the reasoning behind it can often differ. For one, especially with the presence of head injuries (clearly a main topic in American football), long-term effects and complications can be a byproduct. Depression, dementia or even Parkinson's disease are serious issues, and the risks of any of these occurring need to be reduced as much as possible. At the end of the day, while it is making money to these players, this is only a sport, for which it is not worth putting so much on the line. With this in mind, the safety aspect of this thesis is obviously a massive topic, as the pass rush and pass block are exactly what determine how safe the Quarterback is, as these are the plays that often are the reason for more serious injuries for Quarterbacks. The Offensive Line is able to gain significant new insights into the workings of a defense from the new metrics generated, and are therefore able to adjust accordingly and better protect their Quarterback. It was seen that a better performing Offensive Line correlates with a better performing team, much of which can be accredited directly to the Quarterback, who is given more time behind the Offensive Line to throw the ball.

Secondly, the Quarterback is obviously immediately affected by the safety provided to them, as not only their performance can be in danger, but also their well-being. With the risk of injuries increasing with a weaker Offensive Line, a lot is at stake for the Quarterback, as a long-term injury can cost the player extensive amounts of income. Time off of the field will set any player (regardless of the sport) back in regards to their performance and they will need time to get back to their previous level.

But this immediately also affects the coach, as they are the ones that have to work with what they are given. And if the Quarterback is injured, they are forced to play the (possibly weaker) backup Quarterback, which can change the entire setup and tactics of the offense.

Lastly, the safety and injury-risk minimization will always have an enormous impact on the fans watching the game. From a fan perspective, the best possible American football should be played, which automatically implies the best players playing, in their best possible form. The fan is rooting for their team to win, but the probability of the team winning is obviously higher, in the better condition the team is.

A prime example for this are the 2022 San Francisco 49ers. A staggering four of their Quarterbacks were injured during the play either by active or passive impact of the defense, leaving them without an option during the NFC Championship Game in the playoffs. Their run for a title was massively affected by all these injuries. It can be seen as quite unfortunate when a team loses a game not because of bad performance, but simply because of not being able to play their best player, only because of injury. From the fan perspective this is a disappointment, but obviously also from that of the competitor.

No claims are being made that any of the work done in this thesis could make an immediate impact by drastically reducing injury numbers due to better safety for the Quarterback, but even an ever so slightly improvement that can be gained in that aspect would be considered a big success.

7.2.2 Performance Analysis to Improve the Game

As mentioned already, apart from the health-science related aspect of safety, data analysis done in any sport are often linked to performance improvements. Here, it is easy to wander off into the philosophical questions, as in what the societal aspect of any sport really is, but it can be argued that it is in fact larger than one might initially think. Similarly to what was brought up about the impact player safety has on the millions of fans, the same can be said about performance analysis in general. While it was noted that new discoveries do not necessarily lead to the game itself being better, as a newly found tactic, revealed by the metrics, might turn the entirety of the game into a defensive bout, resulting in less points being scores, meaning less exciting games. But even then, the game would again start to evolve into an entirely different, better performing direction. Especially for American football, with the ever lasting fight between offense and defense, new strategies from one side will be caught out by the other, and game plans will be adapted accordingly. New metrics for players and coaches to learn from will continue to push the limits that athletes can go to into new heights. Athletes, especially on that level, are competitors at heart, it may often be difficult to understand for non-professional athletes, with what intensity they work, and with how much determination they want to win. Improvements in any area will always make the game better from an athletic standpoint, and even if only that, this is automatically increasing the entertainment for the people around the sport.

7.3 Player Performance Evaluation and Ethics

There is one major point that has to be brought up when discussing the evaluation, not of a team, but rather of individual players. A team will always look to improve as a unit, while for the individual it will remain a fight against oneself. The major flaw that comes with creating any form of metric in sports, is that automatically the players will be ranked by it. Logically, some players perform better than others, and ending at the bottom of a list can have massive impacts on a player. As a data scientist, rankings are purely made to point specific observations out, and not to criticize any player, positively or negatively. Exactly because of that, it is recommended to not take the rankings as criticism, but rather as a tool for the individual players to see where they can improve on. This goes for any player, from first to last on the list. Perfection is difficult to reach, even for the best player in any category; improvements, even if incremental, can always be made. It is understood that the entire operation of the NFL is business in itself, and therefore the financial aspect will always play a role, even when it comes to the evaluation of players, but an individual metric should never be the once-in-for-all, decisive factor on whether or not to cut or promote a player. Especially for this metric, it was shown that even a successful play that ended in a sack can have a lower PMP value, and should therefore only be used as a tool for improvement.

8 Conclusion

Manipulating the data provided by the NFL that focuses on pass rush and pass block, new telemetries were created to analyze a pass rusher's x-offset, y-offset, and distance to the passer, and to the blocker, as well as the angle created between the three players for every frame on a play, with each frame representing a tenth of a second. Along with information on the speed, acceleration, and orientation of the pass rusher and pass blocker, as well as information on whether or not the play has contained a sack, a hit, or a hurry, a K-Nearest Neighbor machine learning algorithm was able to be used to predict the likelihood of pressure having occurred on a play, based on the created and provided telemetry data, for each frame. This probability percentage was then defined as a new metric to far more precisely measure pressure occurring on a play, as it allows for an evaluation of pressure at any moment of the play. This has created numerous possibilities for not only players, but also for teams to better understand when, for a pass rusher, they were able to apply pressure to the offense by doing play-specific analysis, as well as which positions and formations are performing to what standard for each team. It has also opened the door for opponent-specific analysis to improve the creation of the game plan. Generalizing analysis were shown to not be of use here, while rather team-specific analysis could create entirely new ways of perceiving the game. The different categories that were established were ranked, both league-wide, as well as team-specific. Overall, the Carolina Panthers were found to have created the highest amount of pressure over the course of the first eight weeks of the 2021 NFL season, while the Tampa Bay Buccaneers allowed for the least amount of pressure to be created by their opponents.

Furthermore, an additional metric was created by using the same KNN to predict the motion (position, speed, acceleration, and orientation) of a pass rusher, and therefore allowing for the comparison of the actual pressure created by the pass rusher to that of the predicted motion. Due to the ways that KNN works, the predicted motion could be considered the average move amongst all pass rushers who were found to be in a similar position, with similar telemetries, and therefore the difference in pressure between the reality and prediction could be seen as how good the move that the pass rusher made compared to the average. This metric was shown to be an ideal metric to compare not only pass rushers from the same position, but all pass rushers, as this metric is not influenced by the position, number of blockers, or even the total amount of pressure created, as it evaluates the decision making of a pass rusher and how intuitive the player moves. So even though a player might have created a pressure on the play, their Predicted Motion Pressure may have been fairly low, pointing towards the fact that they could have done even better moves, while a player being double-teamed for the entirety of the play, and never getting the chance to get to the Quarterback, might have still made solid decisions on the play.

Finally, a dashboard was created based on HTML and JavaScript's D3 library to animate not only the plays, but also the pressure created by each player on the play, as well as every PMP. This allows for a better understanding and analysis of when players did or did not create pressure on a play and where good or bad decisions were being made. Especially in regards to a dynamic, and occasionally chaotic sport such as American football, visualizing was deemed the most important factor.

9 Future Research

9.1 Finding the Optimal Move

The opportunities when it comes to data analytics, even only in this part of American football, are endless. This can be seen alone from the vast differences in directions that the finalists of the NFL Big Data Bowl 2023 decided to go in. For the project, there was one clear point that jumped out in regards to future research: finding the optimal move of the pass rusher, as in the best physically possible move that would have created the highest magnitude of pressure. The predicted move, with all of its telemetries, is essentially the "average move" that the pass rushers have made in a similar situation, and a comparison between the actual move and the predicted one turned out to be a solid metric to evaluate a pass rusher's performances, regardless of the situation they found themselves in. Nonetheless, finding the optimal move that a pass rusher could make in a certain situation, with the same metrics that were used for all the KNNs in this thesis, would open the door for entirely new aspects of the game, especially from a tactical side of view. The informations gained from seeing the best possible move could be used to deeply analyze the current understanding of what the best move actually is. It could uncover new techniques or tactics for teams to implement and improve the pass rusher, while at the same time giving the Offensive Line new strategies to prepare for. From an evaluation standpoint, seeing how good the actual move made was compared to the best one would create an entirely new metric, with three values now being able to be compared to one another.

In regards to this thesis, the reason the implementation was not completed, was due to the "physically possible" aspect of the move. When sticking with KNN, it would be entirely possible to brute force every single location around the pass rusher, with all possible values for speed, acceleration and orientation, to determine the best one. From a pure Physics standpoint, when only analyzing the telemetries of the pass rusher, establishing humanly possible bounds for those is not entirely possible. Without a blocker, naturally the highest amount of pressure a rusher could bring with their next move would be to get as close a possible to the Quarterback in the next frame. When being, lets assume, several yards away from the Quarterback, though, it is (unfortunately) not humanly possible for the passer to suddenly be right by the Quarterback. Finding the maximum amount of acceleration change a passer could have within a tenth of a second would automatically lead to the maximum amount of velocity change that could be made in that time frame (as acceleration is simply the change of velocity over a certain period of time), while identically being able to calculate the maximum x - and y-distances that could be travelled with that velocity in that time (again, since velocity is the change in distance over a certain delta of time). Something similar could be done for the maximum orientation change that is possible in a tenth of a second, and these bounds could even be adapted to the player specifically, as the delta of the values may be larger or smaller for certain athletes. With this, all possible scenarios could be tested and the pressure could be determined via the KNN, finding the telemetries with the highest magnitude of pressure, which would then be the best move. The issue here is the fact that this does not account for any blockers at all. As soon as a blocker is within that path, the brute force method would start to induce chaos, as there would be a high number of variables that first would have to be understood, and then be accounted for. Firstly, as mentioned, the best possible move a rusher could make would, in most cases, be the move as close to the Quarterback as possible, that is humanly possible. But, as soon as a blocker is within this path, the entire equation falls apart, as it is obviously entirely physically impossible for the rusher to suddenly move through the blocker in the tenth of a second, if they were in front of them a moment ago. The question now arises, what the best move would indeed be when a blocker is in the way. This would be starting an entirely new analysis of player motion. Potentially, given the speed of a rusher, it would be to face the blocker head on and try to overpower them, while on the other hand, maybe an agility move to get around the blocker would be the best move in that case. This analysis would be a massive undertaking and would be an entire project in itself, for which the time resources of this thesis were simply not enough.

But, should this be attempted, sticking to KNN was found to not necessarily be the best approach. A very fitting method for this kind of analysis would be to use a deep learning model in the form of reinforcement learning, or more precisely Q-Learning. Simplified, a Q-Learning model analyses the state that it is currently in, looks at all the possible actions it could take, and, by a defined reward that it would receive from taking the action, would be able to establish the best possible action that it could take. This is fitting for the problem at hand, as the state would be clearly defined, and the reward would come in the form of amount of pressure received, with more obviously being equal to a higher reward. The only issue here are the possible actions that can be taken from a state, as this again leads back to the humanly possible states that a rusher could be in, but if that problem can be tackled and solved, implementing the Q-Learning model would be the best option to go with.

9.2 Dashboard Improvements

The dashboard that was shown for the purpose of this thesis could be considered the foundation of an intricate new way for coaches to analyze the way American football is played. The main goal of the dashboard was to be clear and straightforward, so that both data scientists, as well as players and coaches could benefit from it. And while the dashboard in itself already provides a fair amount of information, the extensions that could be made for it are endless. For team specific analyses to be implemented into it would be the first point, while more in-depth player data could be beneficial as well, such as being able to pull up multiple plays of a player to compare at once. Film study is a vital point in the operation of an American football team, and it is believed that the dashboard could be used in combination with that. Especially when focusing on one team specifically, the dashboard could be adapted to the needs of the team, prioritizing those aspects of the game that the team would deem important. With access to actual video footage of a play, the dashboard could be fed with those to pull up the play besides the animation, to be able to have an *actual* visualization of the play. Overall, the foundation has been laid for a far bigger project.

10 Resources

1 Norman, J. (2018, January 4). *Football Still Americans' Favorite Sport to Watch*. Gallup. https://news.gallup.com/poll/224864/football-americans-favorite-sport-watch.aspx

2 NFL. (2023, February 27). *Super Bowl LVII total viewing audience estimated at 200 million*. NFL. https://www.nfl.com/news/super-bowl-lvii-total-viewing-audience-estimated-at-200-million

3 Marca. (2023, March 4). *Super Bowl sets new ratings record outside the US: Which countries are watching NFL the most?*. Marca. https://www.marca.com/en/nfl/2023/03/04/6403a874e2704e19348b45c0.html

4 NFL. (2023). *NFL Next Gen Stats*. NFL Operations. https://operations.nfl.com/gameday/technology/nfl-next-gen-stats/

5 Wilkinson, S. (1993). *Football: Winning Offense*. Sports Illustrated.

6 Saal, J.A. (1991). *Common American Football Injuries*. Sports Med 12, 132–147. https://doi.org/10.2165/00007256-199112020-00005

7 Maiese, K. (2008). *The Merck Manual Home - Health Handbook*. Merck Publishing

8 Belson, K. (2014, January 30). *Concussions Show Decline Of 13 Percent, N.F.L. Says.* The New York Times. https://www.nytimes.com/2014/01/31/sports/football/nfl-reports-concussions-dropped-13-percent-in-2013.html

9 James, G., Witten, D., Hastie, T., Tibshirani, R. (2021). *An Introduction to Statistical Learning: With Applications in R.* (2nd ed. pp. 39-42). Springer

10 Karpick, V. (2023). *Completions Added Through Suppression of Pressure*. NFL Big Data Bowl 2023. https://www.kaggle.com/code/vincentkarpick/completions-added-through-suppression-of-pressure

11 National Strength and Conditioning Association. (2017). *Factors Determining Quickness: Anticipation*. NSCA. https://www.nsca.com/education/articles/kinetic-select/factors-determining-quickness-anticipation/

12 Ozanian, M., Badenhause, K. (2020, September 10). *The NFL's Most valuable Teams 2020: How Much Is Your Favorite Team Worth?*. Forbes. https://www.forbes.com/sites/mikeozanian/2020/09/10/the-nfls-most-valuable-teams-2020-how-much-is-your-favorite-team-worth/

11 Appendix

11.1 Pressure Metric Based Rankings

11.1.1 Average Pressure Created by the Pass Rush per Team

Number	Team	Average Pressure Created
1	CAR	13.36%
2	TEN	13.17%
3	CLE	13.08%
4	LA	13.01%
5	BAL	12.89%
6	BUF	12.78%
7	CHI	12.65%
8	SF	12.63%
9	PIT	12.62%
10	MIA	12.58%
11	TB	12.57%
12	IND	12.56%
13	LV	12.55%
14	NYJ	12.52%
15	JAX	12.48%
16	HOU	12.45%
17	ARI	12.40%
18	PHI	12.39%
19	DEN	12.38%
20	LAC	12.34%
21	SEA	12.29%
22	DAL	12.23%
23	NE	12.18%
24	WAS	12.13%
25	MIN	12.13%
26	GB	12.12%
27	CIN	12.08%
28	KC	12.04%
29	NYG	11.98%
30	DET	11.96%
31	NO	11.89%
32	ATL	11.70%

11.1.2 Average Pressure Allowed by the Pass Block per Team

Number	Team	Average Pressure Allowed
1	TB	11.45%
2	ARI	11.93%
3	GB	11.93%
4	NE	11.94%
5	DAL	11.96%
6	LA	11.99%
7	LAC	12.03%
8	PIT	12.05%
9	NYG	12.05%
10	SF	12.14%
11	CLE	12.19%
12	BAL	12.21%
13	WAS	12.24%
14	LV	12.33%
15	KC	12.34%
16	NO	12.37%
17	CAR	12.40%
18	CIN	12.41%
19	BUF	12.53%
20	TEN	12.57%
21	ATL	12.59%
22	NYJ	12.59%
23	MIN	12.59%
24	DET	12.59%
25	JAX	12.66%
26	DEN	12.76%
27	CHI	12.86%
28	HOU	12.92%
29	PHI	11.97%
30	MIA	11.99%
31	SEA	13.42%
32	IND	14.53%

11.1.3 Average Pressure Created by Each Position

Position	**Position**	**Average Pressure Created**
1	SSL	33.77%
2	RCB	22.11%
3	SCBoR	21.28%
4	SCBoL	20.74%
5	SCBL	19.50%
6	LCB	19.19%
7	SCBiL	18.91%
8	SCBR	16.33%
9	RLB	15.59%
10	LOLB	15.39%
11	MLB	14.43%
12	LEO	14.34%
13	LILB	14.33%
14	RILB	14.11%
15	SSR	14.04%
16	ROLB	13.17%
17	REO	12.90%
18	FS	12.88%
19	LE	11.64%
20	RE	11.22%
21	DLT	10.64%
22	NT	10.49%
23	NLT	10.13%
24	DRT	10.12%
25	NRT	9.59%

11.1.4 Average Pressure Created by Each Coverage

Number	**Position**	**Average Pressure Created**
1	Prevent	13.75%
2	Cover-0	13.21%
3	Bracket	13.10%
4	2-Man	12.96%
5	Cover-1	12.44%
6	Red Zone	13.39%
7	Cover-3	12.39%
8	Cover-2	12.37%
9	Quarters	12.27%
10	Cover-6	12.06%
11	Miscellaneous	12.01%
12	Goal Line	11.58%

11.1.5 Average Pressure Created by the Pass Rush per Team for the NRT

Number	**Team**	**Average Pressure Created**
1	GB	13.11%
2	TB	12.06%
3	SEA	11.83%
4	MIA	11.54%
5	LA	10.49%
6	DET	10.27%
7	ARI	10.15%
8	HOU	10.04%
9	TEN	9.85%
10	NO	9.84%
11	CAR	9.71%
12	NE	9.65%
13	NYG	9.61%
14	IND	9.60%
15	PIT	9.67%
16	CIN	9.57%
17	SF	9.55%
18	JAX	9.54%
19	DEN	9.53%
20	DAL	9.29%
21	BAL	9.18%
22	WAS	9.06%
23	BUF	8.88%
24	MIN	8.76%
25	CHI	8.73%
26	LV	8.56%
27	ATL	8.40%
28	LAC	8.34%
29	PHI	8.32%
30	NYJ	8.14%
31	KC	8.03%
32	CLE	7.98%

11.1.6 Average Pressure Created by the Pass Rush per Team for Cover-2

Number	Team	Average Pressure Created
1	CAR	14.64%
2	LA	13.56%
3	BAL	13.50%
4	MIN	13.34%
5	JAX	13.07%
6	ARI	13.00%
7	HOU	12.92%
8	IND	12.89%
9	NYJ	12.86%
10	WAS	12.82%
11	NO	12.75%
12	TEN	12.75%
13	PIT	12.73%
14	CHI	12.62%
15	PHI	12.50%
16	KC	12.35%
17	NYG	12.33%
18	BUF	12.31%
19	NE	12.16%
20	CLE	12.08%
21	DAL	11.96%
22	SEA	11.95%
23	TB	11.86%
24	ATL	11.77%
25	DET	11.60%
26	MIA	11.37%
27	LAC	11.26%
28	SF	11.21%
29	CIN	11.18%
30	LV	10.96%
31	GB	10.55%
32	DEN	9.40%

11.1.7 Average Pressure Created by Each Position for the Green Bay Packers

Position	Position	Average Pressure Created
1	SCBoR	25.91%
2	LCB	19.83%
3	LLB	17.79%
4	RILB	14.91%
5	LOLB	14.87%
6	LILB	14.18%
7	RLB	13.56%
8	NRT	13.11%
9	ROLB	12.80%
10	SCBL	12.44%
11	NLT	10.98%
12	LE	10.79%
13	LEO	10.77%
14	RE	10.77%
15	REO	10.72%
16	DLT	10.39%
17	MLB	10.13%
18	DRT	9.53%
19	NT	9.39%

11.1.8 Average Pressure Created by Each Coverage for the Los Angeles Rams

Number	**Position**	**Average Pressure Created**
1	Cover-1	14.01%
2	Cover-0	13.58%
3	Cover-2	13.56%
4	Cover-3	13.15%
5	Prevent	13.08%
6	Red Zone	12.80%
7	Quarters	12.63%
8	Cover-6	12.53%
9	2-Man	11.63%

11.1.9 Average Pressure Allowed by the Pass Block per Team against the LLB

Number	Team	Average Pressure Allowed
1	NE	13.68%
2	ARI	14.07%
3	HOU	14.20%
4	BUF	14.71%
5	SF	14.99%
6	JAX	15.08%
7	MIN	15.34%
8	MIA	15.63%
9	GB	15.76%
10	NYJ	15.95%
11	IND	16.03%
12	DET	16.05%
13	SEA	16.11%
14	LAC	16.35%
15	WAS	16.60%
16	CLE	16.70%
17	DAL	16.77%
18	TB	17.04%
19	LA	17.51%
20	CAR	18.06%
21	BAL	18.14%
22	NYG	18.40%
23	CHI	18.43%
24	DEN	18.55%
25	PIT	19.56%
26	ATL	20.07%
27	LV	20.37%
28	TEN	21.66%
29	PHI	22.35%
30	CIN	22.48%
31	NO	26.98%
32	KC	27.42%

11.1.10 Average Pressure Allowed by the Pass Block per Team against Cover-3

Number	Team	Average Pressure Allowed
1	DAL	11.37%
2	LAC	11.62%
3	TB	11.74%
4	NE	11.79%
5	GB	11.81%
6	PIT	11.82%
7	LA	11.83%
8	SF	11.93%
9	NYG	11.96%
10	ARI	11.97%
11	CIN	11.99%
12	WAS	12.06%
13	CLE	12.10%
14	BAL	12.11%
15	ATL	12.19%
16	MIN	12.24%
17	LV	12.27%
18	TEN	12.28%
19	NYJ	12.31%
20	NO	12.33%
21	HOU	12.36%
22	BUF	12.37%
23	KC	12.45%
24	JAX	12.50%
25	CAR	12.76%
26	DET	12.78%
27	CHI	13.09%
28	SEA	13.09%
29	MIA	13.12%
30	DEN	13.24%
31	PHI	13.40%
32	IND	14.69%

11.1.11 Average Pressure Allowed against Each Position for the New York Giants

Position	Position	Average Pressure Allowed
1	SCBoL	8.36%
2	NRT	8.56%
3	SCBR	8.74%
4	DLT	9.36%
5	DRT	9.57%
6	NLT	10.37%
7	NT	10.43%
8	RE	10.99%
9	LE	11.17%
10	LILB	11.41%
11	REO	11.60%
12	SCBiL	11.65%
13	ROLB	12.49%
14	RILB	14.03%
15	LEO	14.32%
16	RLB	15.40%
17	LOLB	15.46%
18	SCBoR	17.32%
19	LLB	18.40%
20	MLB	18.79%
21	SCBL	20.28%
2	SCBiR	22.81%

11.1.12 Average Pressure Allowed against Each Coverage for the New York Jets

Number	Position	Average Pressure Allowed
1	Red Zone	11.39%
2	Cover-0	12.20%
3	Cover-3	12.31%
4	Cover-2	12.55%
5	Cover-6	12.67%
6	Cover-1	12.84%
7	Quarters	12.98%
8	2-Man	15.35%
9	Bracket	17.22%

11.2 PMP Based Rankings

11.2.1 Top 20 Highest PMP Delta on One Play

Number	Player	nflId	Team	PMP Delta on One Play
1	Tanner Vallejo	45008	ARI	6.9774
2	Chris Jones	43326	KC	6.3135
3	Carlos Dunlap	35493	SEA	6.2516
4	Josh Allen	47790	JAX	5.9540
5	Aaron Donald	41239	LA	5.6189
6	Matt Judon	43435	NE	5.5378
7	Azeez Ojulari	53479	NYG	5.5329
8	John Franklin-Meyers	46204	NYJ	5.4213
9	Charles Harris	44834	DET	5.2702
10	Josh Allen	47790	JAX	5.2690
11	B.J. Hill	46138	CIN	5.1722
12	Efe Obada	42331	BUF	5.1161
13	Jabrill Peppers	44837	NYG	5.0989
14	Leonard Williams	42349	NYG	5.0646
15	Everson Griffen	35539	MIN	4.8936
16	Demario Davis	38607	NO	4.8716
17	Kyler Fackrell	43377	LAC	4.8623
18	Harold Landry	46110	TEN	4.7474
19	Logan Wilson	52473	CIN	4.7408
20	Myles Bryan	52991	NE	4.7333

11.2.2 Top 20 Highest Average PMP Delta

Number	Player	nflId	Team	Average PMP Delta
1	Grant Delpit	52452	CLE	2.4534
2	Kevin King	44845	GB	2.4184
3	George Odum	46349	IND	2.3689
4	Tashaun Gipson	38868	CHI	2.3554
5	Charvarius Ward	46757	KC	2.3291
6	Myles Bryan	52991	NE	2.3132
7	Tanner Vallejo	45008	ARI	2.1690
8	Donte Deayon	43761	LA	2.1390
9	Avonte Maddox	46194	PHI	2.0817
10	Ryan Neal	46711	SEA	1.9981
11	Tre'Davious White	44839	BUF	1.9162
12	Caden Sterns	53581	DEN	1.8749
13	Elijah Lee	45045	CLE	1.7442
14	Grant Stuard	53688	TB	1.7355
15	M.J. Steward	46122	CLE	1.8090
16	Daryl Worley	43366	DET	1.5591
17	Jamal Dean	47877	TB	1.5519
18	Markus Bailey	52623	CIN	1.5346
19	Kristian Fulton	52469	TEN	1.4951
20	Micah Kiser	46216	DEN	1.3408

11.3 Python Code

11.3.1 Disclaimer

Due to the fact that millions of data points have to be processed at various stages of the code, some individual computations took several hours to compile. Letting the code run from top to bottom would most likely result in a computation time of over a day, which is why the code has be structured in the following way: for any computation that requires a changing of the data frame, or the creation of a new one, a new function would be built that loads the data frame that is to be manipulated, does the computation, and then creates an entirely new CSV file. This helps to keep track of where in the process one is, but if its desired to imitate the steps, the size of the files will come out to several gigabytes. The in-between created files can obviously be deleted once it is all completed. The Python code will be separated into four sections: the data preprocessing with the exploratory data analysis, the both machine learning models with all the required data manipulation, the data analysis, and finally the Streamlit code for all visualizations that are found in this thesis.

11.3.2 Data Preprocessing

```python
import pandas as pd

# Add the week to each recorded frame and concatenate the weeks
def add_weeks_to_data():
    # Load data
    week1 = pd.read_csv('week1.csv')
    week2 = pd.read_csv('week2.csv')
    week3 = pd.read_csv('week3.csv')
    week4 = pd.read_csv('week4.csv')
    week5 = pd.read_csv('week5.csv')
    week6 = pd.read_csv('week6.csv')
    week7 = pd.read_csv('week7.csv')
    week8 = pd.read_csv('week8.csv')

    # Add number of weeks
    week1['week'] = 1
    week2['week'] = 2
    week3['week'] = 3
    week4['week'] = 4
    week5['week'] = 5
    week6['week'] = 6
    week7['week'] = 7
    week8['week'] = 8

    # Concat all weeks
    frames = [week1, week2, week3, week4, week5, week6, week7, week8]

    # Create new df
    all_weeks = pd.concat(frames, ignore_index=True)
    all_weeks = all_weeks.drop(all_weeks.columns[0], axis=1)

    # Create new csv
    all_weeks.to_csv('all_weeks_concat.csv')

# Adding the position of each player for each frame to the data set
def add_position_to_week():
    # Load the required csv files
    week = pd.read_csv('all_weeks_concat.csv')
    pff_data = pd.read_csv('pffScoutingData.csv')

    # For each player and each frame in the data set, find the corresponding position via the PFF Scouting data
    for i in range(len(week)):
        if week['team'][i] != 'football':
            df = pff_data.loc[
                (pff_data['gameId'] == week['gameId'][i]) & (pff_data['playId'] == week['playId'][i]) & (
                        pff_data['nflId'] == week['nflId'][i])]
            df = df.reset_index(drop=True)
            week.at[i, 'position'] = df['pff_positionLinedUp'][0]   # Add the position in a new column

    # Create a new CSV file
    week.to_csv('all_weeks_with_positions.csv')

# Flip all plays with play direction going left, so that all plays will have the same play direction.
# X-axis and y-axis each have to be reversed and the orientation has to be flipped by 180 degrees.
def flip_plays():
    # Load the data
    weeks = pd.read_csv('all_weeks_with_positions.csv')

    length = 120        # Define the length of the field
    width = 160 / 3     # Define the width of the field

    # For each frame, if the play direction is left, flip the play
    for i in range(len(weeks)):
        if weeks['playDirection'][i] == 'left':
            weeks.at[i, 'x'] = length - weeks['x'][i]   # Reversing the x-axis
            weeks.at[i, 'y'] = width - weeks['y'][i]    # Reversing the y-axis

            # If the orientation is more than 180 degrees, 180 will be subtracted from it, otherwise 180 will be added
            if weeks['o'][i] >= 180:
                weeks.at[i, 'o'] = weeks['o'][i] - 180
            else:
                weeks.at[i, 'o'] = 180 + weeks['o'][i]

            weeks.at[i, 'playDirection'] = 'right'  # Change the label for play direction

    # Create a new CSV file
    # If a function had to be redone, the CSV file would have a version number in it, this can obviously be changed
    weeks.to_csv('all_weeks_going_right2.csv')


# Adding the role to every frame for every player, this will be useful to determine passer, pass rush and pass block
# later.
def add_role():
    # Load the data
    pff_data = pd.read_csv('pffScoutingData.csv')
    weeks = pd.read_csv('all_weeks_going_right2.csv')

    weeks['role'] = ""  # Add a new column

    # Same procedure as with the position, just with the role this time
    for i in range(len(weeks)):
        if weeks['team'][i] != 'football':
            df = pff_data.loc[
                (pff_data['gameId'] == weeks['gameId'][i]) & (pff_data['playId'] == weeks['playId'][i]) & (
                        pff_data['nflId'] == weeks['nflId'][i])]
            df = df.reset_index(drop=True)
            weeks.at[i, 'role'] = df['pff_role'][0]     # Overwrite the role with the correct one

    # Create a new CSV file
    weeks.to_csv('weeks_final2.csv')

# The preprocessing in the previous step turned out to create floats for some of the int-type data, like the nflId
# as well as turning the x, y and o data into floats with several trailing zeros.
# It was also noticed that the newly created data frames would have leading empty columns, therefore the first three
# would be dropped here, but it will be considered from here on out.
def cleaning_all_weeks():
    weeks = pd.read_csv('weeks_final2.csv')     # Load the data
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks['nflId'] = weeks['nflId'].fillna(0)
    weeks['nflId'] = weeks['nflId'].astype(int)
    weeks['jerseyNumber'] = weeks['jerseyNumber'].fillna(0)
    weeks['jerseyNumber'] = weeks['jerseyNumber'].astype(int)
    weeks['x'] = weeks['x'].round(2)
    weeks['y'] = weeks['y'].round(2)
    weeks['o'] = weeks['o'].round(2)

    # Create new CSV file
    weeks.to_csv('weeks_final2.1.csv')

# Filter the data frame to only contain the passer, the pass rusher and the pass blocker, as this will be needed later
# on.
def only_passer_rusher_blocker():
    # Load the data
    weeks = pd.read_csv('weeks_final2.1.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks_reduced = weeks.loc[(weeks['role'] == 'Pass') | (weeks['role'] == 'Pass Block') | (weeks['role'] == 'Pass Rush')]
    weeks_reduced = weeks_reduced.reset_index(drop=True)

    # Create a new CSV file
    weeks_reduced.to_csv('weeks_only_passer_blocker_rusher.csv')


# Adding whether a sack, hit, or hurry has occurred on a play for a pass rusher, defining the occurrence of either one
# as a pressure and adding the binary event to every frame for that play for that pass rusher.
def add_sack_hit_hurry_pressure():
    # Loading the data
    weeks = pd.read_csv('weeks_only_passer_blocker_rusher.csv')
    pff_data = pd.read_csv('pffScoutingData.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    # Creating new columns
    weeks['pressure'] = 0
    weeks['sack'] = 0
    weeks['hit'] = 0
    weeks['hurry'] = 0

    # Iterating over every frame. If via the PFF Scouting data a pressure has occurred on the play, add that information
    # to the data frame.
    for i in range(len(weeks)):
        df = pff_data.loc[
            (pff_data['gameId'] == weeks['gameId'][i]) & (pff_data['playId'] == weeks['playId'][i]) & (
                    pff_data['nflId'] == weeks['nflId'][i])]
        df = df.reset_index(drop=True)
        if (df['pff_hit'][0] == 1) | (df['pff_hurry'][0] == 1) | (df['pff_sack'][0] == 1):
            weeks.at[i, 'pressure'] = 1
        if df['pff_hit'][0] == 1:
            weeks.at[i, 'hit'] = 1
        if df['pff_hurry'][0] == 1:
            weeks.at[i, 'hurry'] = 1
        if df['pff_sack'][0] == 1:
            weeks.at[i, 'sack'] = 1

    # Create a new CSV file
    weeks.to_csv('weeks_with_sack_hit_hurry_pressure.csv')


# Add the pass result of the play to the data frame for potential further analysis later on
def add_pass_result():
    # Load the data
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks['pass_result'] = ''  # Add a new column

    # Iterate over every frame
    for i in range(len(weeks)):
        # Find the corresponding play via the plays data and retrieve the pass result from there
        df = plays.loc[
            (plays['gameId'] == weeks['gameId'][i]) & (plays['playId'] == weeks['playId'][i])]
        df = df.reset_index(drop=True)
        weeks.at[i, 'pass_result'] = df['passResult'][0]  # Overwrite the pass result

    # Create a new CSV file
    weeks.to_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')

# Lastly, some EDA that was mentioned in the thesis

# Finding the total frames tracked for each week
def find_avg_number_of_plays_per_game():
    # Load all weeks
    week1 = pd.read_csv('week1.csv')
    week2 = pd.read_csv('week2.csv')
    week3 = pd.read_csv('week3.csv')
    week4 = pd.read_csv('week4.csv')
    week5 = pd.read_csv('week5.csv')
    week6 = pd.read_csv('week6.csv')
    week7 = pd.read_csv('week7.csv')
    week8 = pd.read_csv('week8.csv')

    # Average amount for non-bye weeks and the all frames for all weeks
    avg_tracking = (len(week1) + len(week2) + len(week3) + len(week4) + len(week5)) / 5
    total_tracking = len(week1) + len(week2) + len(week3) + len(week4) + len(week5) + len(week6) + len(week7) + len(
        week8)

    # Print to display
    print(len(week1))
    print(len(week2))
    print(len(week3))
    print(len(week4))
    print(len(week5))
    print(len(week6))
    print(len(week7))
    print(len(week8))
    print(avg_tracking)
    print(total_tracking)
```

```python
# Finding the average amount of plays per game
def find_avg_number_of_plays_per_game():
    # Loading the data
    games = pd.read_csv('games.csv')
    plays = pd.read_csv('plays.csv')

    # Filtering the games data to the those of each week
    games_week1 = games.loc[(games['week'] == 1)]
    games_week1 = games_week1.reset_index(drop=True)

    games_week2 = games.loc[(games['week'] == 2)]
    games_week2 = games_week2.reset_index(drop=True)

    games_week3 = games.loc[(games['week'] == 3)]
    games_week3 = games_week3.reset_index(drop=True)

    games_week4 = games.loc[(games['week'] == 4)]
    games_week4 = games_week4.reset_index(drop=True)

    games_week5 = games.loc[(games['week'] == 5)]
    games_week5 = games_week5.reset_index(drop=True)

    games_week6 = games.loc[(games['week'] == 6)]
    games_week6 = games_week6.reset_index(drop=True)

    games_week7 = games.loc[(games['week'] == 7)]
    games_week7 = games_week7.reset_index(drop=True)

    games_week8 = games.loc[(games['week'] == 8)]
    games_week8 = games_week8.reset_index(drop=True)

    # Create a list for all games of the respective week and filter the plays data frame to only the plays of the
    # respective week.
    week1_games_list = games_week1['gameId'].unique()
    plays_week1 = plays[plays['gameId'].isin(week1_games_list)]
    plays_week1 = plays_week1.reset_index(drop=True)

    week2_games_list = games_week2['gameId'].unique()
    plays_week2 = plays[plays['gameId'].isin(week2_games_list)]
    plays_week2 = plays_week2.reset_index(drop=True)

    week3_games_list = games_week3['gameId'].unique()
    plays_week3 = plays[plays['gameId'].isin(week3_games_list)]
    plays_week3 = plays_week3.reset_index(drop=True)

    week4_games_list = games_week4['gameId'].unique()
    plays_week4 = plays[plays['gameId'].isin(week4_games_list)]
    plays_week4 = plays_week4.reset_index(drop=True)

    week5_games_list = games_week5['gameId'].unique()
    plays_week5 = plays[plays['gameId'].isin(week5_games_list)]
    plays_week5 = plays_week5.reset_index(drop=True)

    week6_games_list = games_week6['gameId'].unique()
    plays_week6 = plays[plays['gameId'].isin(week6_games_list)]
    plays_week6 = plays_week6.reset_index(drop=True)

    week7_games_list = games_week7['gameId'].unique()
    plays_week7 = plays[plays['gameId'].isin(week7_games_list)]
    plays_week7 = plays_week7.reset_index(drop=True)

    week8_games_list = games_week8['gameId'].unique()
    plays_week8 = plays[plays['gameId'].isin(week8_games_list)]
    plays_week8 = plays_week8.reset_index(drop=True)

    # Display the amount of plays per week
    print(len(plays_week1))
    print(len(plays_week2))
    print(len(plays_week3))
    print(len(plays_week4))
    print(len(plays_week5))
    print(len(plays_week6))
    print(len(plays_week7))
    print(len(plays_week8))
    # Calculate the average amount of plays per game
    plays_per_game = len(plays) / len(games)

    # Display the average
    print(plays_per_game)

# Finding the highest amount of blockers a pass rusher has faced
def find_highest_number_of_blockers():
    # Load the data
    plays = pd.read_csv('plays.csv')
    pff_data = pd.read_csv('pffScoutingData.csv')

    max_blockers = 0    # Set max number of blocker to 0

    # Iterate over all plays
    for i in range(len(plays)):
        # Filter the PFF Scouting data to be only one play each
        one_play = pff_data.loc[(pff_data['gameId'] == plays['gameId'][i]) & (pff_data['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)

        # Filter the one-play data to only show the pass rushers
        only_rushers = one_play.loc[(one_play['pff_role'] == 'Pass Rush')]
        only_rushers = only_rushers.reset_index(drop=True)
        rusher_id = only_rushers['nflId'].unique()  # Get the nflId for each pass rusher on the play

        # Iterate over all pass rushers on the play and find how many pass blockers were credited with blocking this
        # pass rusher. If the blocker count is a new max, overwrite the previous max
        for j in rusher_id:
            blocker_count = (one_play['pff_nflIdBlockedPlayer'] == j).sum()
            if blocker_count > max_blockers:
                max_blockers = blocker_count

    # Display the maximum amount of blockers
    print(max_blockers)
```

11.3.3 Machine Learning Models

```python
import pandas as pd
import math
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.neighbors import KNeighborsRegressor


# Calculating the required telemetry for every pass rusher for the three closest blockers as described in the thesis
def add_telemetry_rushers():
    # Load the data
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    # Add the required, described columns
    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
    weeks['y_to_blocker'] = 0
    weeks['distance_to_blocker'] = 0
    weeks['blocker_o'] = 0
    weeks['angle'] = 0
    weeks['x_to_second_blocker'] = 0
    weeks['y_to_second_blocker'] = 0
    weeks['distance_to_second_blocker'] = 0
    weeks['second_blocker_o'] = 0
    weeks['second_angle'] = 0
    weeks['x_to_third_blocker'] = 0
    weeks['y_to_third_blocker'] = 0
    weeks['distance_to_third_blocker'] = 0
    weeks['third_blocker_o'] = 0
    weeks['third_angle'] = 0
    weeks['x_offset'] = 0
    weeks['y_offset'] = 0

    # Create an empty data frame only responsible for the telemetry data
    telemetry_df = pd.DataFrame(columns=weeks.columns)

    # Iterate over all plays
    for k in range(len(plays)):
        # Filter the data to the current play
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
        one_play = one_play.reset_index(drop=True)
        # Iterate over every frame of the current play
        for i in range(len(one_play)):
            # If the current frame is that of a pass rusher, calculate all required telemetry for that frame
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i])]
                df_passer = df_passer.reset_index(drop=True)
                # All telemetries in regards to the passer
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                # Create a data frame of all pass blockers on this frame
                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                # Iterate over every found pass blocker and calculate the required telemetries to the pass rusher and
                # the angle between pass rusher, pass blocker and passer.
                # Append every telemetry to a list. Append the list to a bigger list that will hold the telemetries to
                # each pass blocker. This will allow to determine the closest blockers and allow for a quick indexing to
                # the right column.
```

```
for j in range(len(df_blocker)):
    x_to_blocker = one_play['x'][i] - df_blocker['x'][j]
    individual_blocker.append(x_to_blocker)

    y_to_blocker = one_play['y'][i] - df_blocker['y'][j]
    individual_blocker.append(y_to_blocker)

    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
    individual_blocker.append(distance_to_blocker)

    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

    # Vector calculation to find the angle between passer, pass blocker, and pass rusher
    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
        i] * y_passer_to_blocker
    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
    cos_theta = min(1, max(-1, cos_theta))
    theta = math.acos(cos_theta)
    theta_degrees = math.degrees(theta)
    individual_blocker.append(theta_degrees)
    individual_blocker.append(df_blocker['o'][j])

    all_blockers.append(individual_blocker)     # Append list to bigger list
    individual_blocker = []                     # Reset the individual blocker list

# Findin the smallest distance smallest distance
min_val = float('inf')
closest_blocker = None
for sub_arr in all_blockers:
    if sub_arr[2] < min_val:
        min_val = sub_arr[2]
        closest_blocker = sub_arr

# Indexing the telemetries to the columns of the closest blocker
one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
one_play.at[i, 'angle'] = closest_blocker[3]
one_play.at[i, 'blocker_o'] = closest_blocker[4]

# Removing the closest blocker and finding the second closest blocker
all_blockers.remove(closest_blocker)
min_val = float('inf')
closest_blocker = None
for sub_arr in all_blockers:
    if sub_arr[2] < min_val:
        min_val = sub_arr[2]
        closest_blocker = sub_arr

# Indexing the telemetries to the columns of the second closest blocker
one_play.at[i, 'x_to_second_blocker'] = closest_blocker[0]
one_play.at[i, 'y_to_second_blocker'] = closest_blocker[1]
one_play.at[i, 'distance_to_second_blocker'] = closest_blocker[2]
one_play.at[i, 'second_angle'] = closest_blocker[3]
one_play.at[i, 'second_blocker_o'] = closest_blocker[4]

# Removing the closest blocker and finding the third closest blocker
all_blockers.remove(closest_blocker)
min_val = float('inf')
closest_blocker = None
for sub_arr in all_blockers:
    if sub_arr[2] < min_val:
        min_val = sub_arr[2]
        closest_blocker = sub_arr

# Indexing the telemetries to the columns of the third closest blocker
one_play.at[i, 'x_to_third_blocker'] = closest_blocker[0]
one_play.at[i, 'y_to_third_blocker'] = closest_blocker[1]
one_play.at[i, 'distance_to_third_blocker'] = closest_blocker[2]
one_play.at[i, 'third_angle'] = closest_blocker[3]
one_play.at[i, 'third_blocker_o'] = closest_blocker[4]
```

```
                # Adding the offset to the previous frame. Setting it to zero in case of the first frame
                if one_play['frameId'][i] == 1:
                    one_play.at[i, 'x_offset'] = 0
                    one_play.at[i, 'y_offset'] = 0
                else:
                    df_rusher = one_play.loc[
                        (one_play['nflId'] == one_play['nflId'][i]) & (
                                one_play['frameId'] == one_play['frameId'][i] - 1)]
                    df_rusher = df_rusher.reset_index(drop=True)
                    one_play.at[i, 'x_offset'] = df_rusher['x'][0] - one_play['x'][i]
                    one_play.at[i, 'y_offset'] = df_rusher['y'][0] - one_play['y'][i]

        # Adding the newly found telemetries to the telemetry data frame
        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    # Deleting all non-pass rushers from the list, as those will not be important for the prediction
    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    # Creating a new CSV file
    only_rushers.to_csv('weeks_only_rushers_with_telemetry_three_blockers.csv')

# Finding the optimal k for one blocker with the telemetries created for k from 1 to 100, predicting the occurrence of
# a pressure on the play.
def find_optimal_k_one_blocker():
    # Load the data
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_three_blockers.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    # Round the telemetries to two decimal places, as this is the precision of all other telemetries
    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    # Train the model on the first seven weeks
    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                    weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)

    # Test the model on the eighth week
    weeks_test = weeks.loc[(weeks['week'] == 8)]
    weeks_test = weeks_test.reset_index(drop=True)

    # Defining the features (the created telemetries, plus speed, acceleartion, and orientation)
    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a']

    # Creating the train and test datasets
    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    # Running the KNN for all k from 1 to 100 with weights for the distances to the closest neighbors
    for i in range(1, 101):
        clf = KNeighborsClassifier(n_neighbors=i, weights='distance')

        # Train the KNN
        clf = clf.fit(X_train, y_train)

        # Predict the response for the test dataset
        y_pred = clf.predict(X_test)

        # Display the accuracy to find the best one
        print("Accuracy with one blocker:", metrics.accuracy_score(y_test, y_pred), "for k being:", i)
```

```python
# Repeat the process from the previous function, but with the two closest blockers
def find_optimal_k_two_blockers():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_three_blockers.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['x_to_second_blocker'] = weeks['x_to_second_blocker'].round(2)
    weeks['y_to_second_blocker'] = weeks['y_to_second_blocker'].round(2)
    weeks['distance_to_second_blocker'] = weeks['distance_to_second_blocker'].round(2)
    weeks['second_blocker_o'] = weeks['second_blocker_o'].round(2)
    weeks['second_angle'] = weeks['second_angle'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                    weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 8)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a',
                'x_to_second_blocker', 'y_to_second_blocker', 'distance_to_second_blocker', 'second_blocker_o',
                'second_angle']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    for i in range(1, 101):
        clf = KNeighborsClassifier(n_neighbors=i, weights='distance')

        clf = clf.fit(X_train, y_train)

        y_pred = clf.predict(X_test)

        print("Accuracy with two blockers:", metrics.accuracy_score(y_test, y_pred), "for k being:", i)

# Repeat the process from the previous function, but with the three closest blockers
def find_optimal_k_three_blockers():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_three_blockers.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['x_to_second_blocker'] = weeks['x_to_second_blocker'].round(2)
    weeks['y_to_second_blocker'] = weeks['y_to_second_blocker'].round(2)
    weeks['distance_to_second_blocker'] = weeks['distance_to_second_blocker'].round(2)
    weeks['second_blocker_o'] = weeks['second_blocker_o'].round(2)
    weeks['second_angle'] = weeks['second_angle'].round(2)
    weeks['x_to_third_blocker'] = weeks['x_to_third_blocker'].round(2)
    weeks['y_to_third_blocker'] = weeks['y_to_third_blocker'].round(2)
    weeks['distance_to_third_blocker'] = weeks['distance_to_third_blocker'].round(2)
    weeks['third_blocker_o'] = weeks['third_blocker_o'].round(2)
    weeks['third_angle'] = weeks['third_angle'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 8)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a',
                'x_to_second_blocker', 'y_to_second_blocker', 'distance_to_second_blocker', 'second_blocker_o',
                'second_angle', 'x_to_third_blocker', 'y_to_third_blocker', 'distance_to_third_blocker',
                'third_blocker_o', 'third_angle']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    for i in range(1, 101):
        clf = KNeighborsClassifier(n_neighbors=i, weights='distance')

        clf = clf.fit(X_train, y_train)

        y_pred = clf.predict(X_test)

        print("Accuracy with three blockers:", metrics.accuracy_score(y_test, y_pred), "for k being:", i)


# With the accuracies being close to one another and the decision being made to advance with only the closest blocker,
# the telemetries were recalculated to also include the speed and acceleration of the blocker.
# Same process as before, but this time only taking the closest blocker into consideration
def add_telemetry_rusher_for_one_blocker_full():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
    weeks['y_to_blocker'] = 0
    weeks['distance_to_blocker'] = 0
    weeks['blocker_o'] = 0
    weeks['angle'] = 0
    weeks['blocker_s'] = 0
    weeks['blocker_a'] = 0
    weeks['x_offset'] = 0
    weeks['y_offset'] = 0

    telemetry_df = pd.DataFrame(columns=weeks.columns)

    for k in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
        one_play = one_play.reset_index(drop=True)
        for i in range(len(one_play)):
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i])]
                df_passer = df_passer.reset_index(drop=True)
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                for j in range(len(df_blocker)):
                    x_to_blocker = one_play['x'][i] - df_blocker['x'][j]
                    individual_blocker.append(x_to_blocker)

                    y_to_blocker = one_play['y'][i] - df_blocker['y'][j]
                    individual_blocker.append(y_to_blocker)

                    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                    individual_blocker.append(distance_to_blocker)

                    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                        i] * y_passer_to_blocker
                    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                    cos_theta = min(1, max(-1, cos_theta))
                    theta = math.acos(cos_theta)
                    theta_degrees = math.degrees(theta)
                    individual_blocker.append(theta_degrees)
                    individual_blocker.append(df_blocker['o'][j])
                    individual_blocker.append(df_blocker['s'][j])
                    individual_blocker.append(df_blocker['a'][j])
                    all_blockers.append(individual_blocker)
                    individual_blocker = []

                min_val = float('inf')
                closest_blocker = None
                for sub_arr in all_blockers:
                    if sub_arr[2] < min_val:
                        min_val = sub_arr[2]
                        closest_blocker = sub_arr

                one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                one_play.at[i, 'angle'] = closest_blocker[3]
                one_play.at[i, 'blocker_o'] = closest_blocker[4]
                one_play.at[i, 'blocker_s'] = closest_blocker[5]
                one_play.at[i, 'blocker_a'] = closest_blocker[6]

                if one_play['frameId'][i] == 1:
                    one_play.at[i, 'x_offset'] = 0
                    one_play.at[i, 'y_offset'] = 0
                else:
                    df_rusher = one_play.loc[
                        (one_play['nflId'] == one_play['nflId'][i]) & (
                                one_play['frameId'] == one_play['frameId'][i] - 1)]
                    df_rusher = df_rusher.reset_index(drop=True)
                    one_play.at[i, 'x_offset'] = df_rusher['x'][0] - one_play['x'][i]
                    one_play.at[i, 'y_offset'] = df_rusher['y'][0] - one_play['y'][i]

        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')

# Again determining the best k for the KNN, this time with all telemetries. The same function can be done by testing for
# each week individually and taking the average k out of all 8 models.
# Same process as before.
def find_optimal_k_one_blocker_full():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)
    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 8)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    for i in range(1, 101):
        clf = KNeighborsClassifier(n_neighbors=i, weights='distance')

        clf = clf.fit(X_train, y_train)

        y_pred = clf.predict(X_test)

        print("Accuracy with one blocker:", metrics.accuracy_score(y_test, y_pred), "for k being:", i)


# With the ideal k having been found, run the model with that k=37 and al telemetries, now also extracting the
# probability of each prediction, which will be defined as the amount of pressure created by the pass rusher.
# Run the model eight times, always training on seven weeks and predicting on the chosen eighth.
def predict_pressure_week8():
    # Load the data
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    # Round all telemetries to two decimal places
    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    # Splitting the data into train and test data. Training on weeks 1-7 and testing on week 8 here.
    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 8)]
    weeks_test = weeks_test.reset_index(drop=True)

    # Defining the features
    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)  # Prediction of occurrence of pressure on the play
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]  # Probability of occurrence of pressure
    # Add predictions to the data frame
    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    # Creating a new CSV file
    weeks_test.to_csv('week8_predicted_pressure_final.csv')


# Identical to the week 8 predictions
def predict_pressure_week7():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 7)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week7_predicted_pressure_final.csv')

# Identical to the week 8 predictions
def predict_pressure_week6():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 6)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week6_predicted_pressure_final.csv')

# Identical to the week 8 predictions
def predict_pressure_week5():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 5)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week5_predicted_pressure_final.csv')

# Identical to the week 8 predictions
def predict_pressure_week4():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 4)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week4_predicted_pressure_final.csv')

# Identical to the week 8 predictions
def predict_pressure_week3():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 4) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 3)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week3_predicted_pressure_final.csv')

# Identical to the week 8 predictions
def predict_pressure_week2():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 3) | (weeks['week'] == 4) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 2)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week2_predicted_pressure_final.csv')

# Identical to the week 8 predictions
def predict_pressure_week1():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 1)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week1_predicted_pressure_final.csv')

# Concatenating all the predictions into one data frame
def concat_all_weeks():
    week1 = pd.read_csv('week1_predicted_pressure_final.csv')
    week2 = pd.read_csv('week2_predicted_pressure_final.csv')
    week3 = pd.read_csv('week3_predicted_pressure_final.csv')
    week4 = pd.read_csv('week4_predicted_pressure_final.csv')
    week5 = pd.read_csv('week5_predicted_pressure_final.csv')
    week6 = pd.read_csv('week6_predicted_pressure_final.csv')
    week7 = pd.read_csv('week7_predicted_pressure_final.csv')
    week8 = pd.read_csv('week8_predicted_pressure_final.csv')

    frames = [week1, week2, week3, week4, week5, week6, week7, week8]

    all_weeks = pd.concat(frames, ignore_index=True)
    all_weeks = all_weeks.drop(all_weeks.columns[0], axis=1)

    all_weeks.to_csv('weeks_predicted_pressure.csv')

# Adding the opponent of every team per game for every frame for further analysis later on
def add_opponent():
    # Load the data
    weeks = pd.read_csv('weeks_predicted_pressure.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    games = pd.read_csv('games.csv')
    weeks['opponent'] = ""  # Create a new column

    # Iterate over every frame
    for i in range(len(weeks)):
        # Find the game on the frame via the games dataset
        game = games.loc[(games['gameId'] == weeks['gameId'][i])]
        game = game.reset_index(drop=True)
        # If the team on the frame is the home team, add the visiting team as the opponent, else, vice versa
        if game['homeTeamAbbr'][0] == weeks['team'][i]:
            weeks.at[i, 'opponent'] = game['visitorTeamAbbr'][0]
        else:
            weeks.at[i, 'opponent'] = game['homeTeamAbbr'][0]

    # Creatin a new CSV file
    weeks.to_csv('weeks_predicted_pressure_final.csv')
```

```python
# This concludes the first KNN model. Next, the PMP will be determined.

# As described in the thesis, to be able to predict the motion in the next frame of a pass rusher, the x-offset and
# y-offset, speed, acceleration, and orientation from the next frame have to be retrieved and replaced with the current
# data.
def get_xysao_for_all():
    # Load the data
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    # Round all telemetries to two decimal places
    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    # The to-be-predicted telemetries are put into a list. Append an arbitrary value to the list and pop the first
    # entry. This moves every value up by one. The last frame is not important here as that one does not need to be
    # predicted.
    x_offset_list = weeks['x_offset'].tolist()
    x_offset_list.append(0)
    x_offset_list.pop(0)
    y_offset_list = weeks['y_offset'].tolist()
    y_offset_list.append(0)
    y_offset_list.pop(0)
    speed_list = weeks['s'].tolist()
    speed_list.append(0)
    speed_list.pop(0)
    acc_list = weeks['a'].tolist()
    acc_list.append(0)
    acc_list.pop(0)
    o_list = weeks['o'].tolist()
    o_list.append(0)
    o_list.pop(0)

    # Append the lists to new columns
    weeks['x_offset_next'] = x_offset_list
    weeks['y_offset_next'] = y_offset_list
    weeks['s_next'] = speed_list
    weeks['a_next'] = acc_list
    weeks['o_next'] = o_list

    # Iterate over every frame to find the last frame of every play, which can be deleted. This needs to be done so that
    # those frames do not interfere with the predictions later on.
    for i in range(len(weeks)-1):
        if weeks['frameId'][i] > weeks['frameId'][i+1]:
                weeks = weeks.drop(i)

    weeks = weeks.reset_index(drop=True)
    weeks = weeks.drop(weeks.index[-1])
    weeks = weeks.reset_index(drop=True)

    # Creating a new CSV file
    weeks.to_csv('all_weeks_with_next_xysao.csv')

# With the to-be-predicted telemetries now in the right spot, a KNN model can be run over the data identically to the
# earlier KNN, this time predicting the x-offset, y-offset, speed, acceleration, and orientation. For now, only
# predictions for the eighth week will be made, as those will be the focus of the dashboard, the other weeks will follow
# later.
def predict_xysao():
    # Load the data
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    # Split the datasets into training and testing data
    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 8)]
    weeks_test = weeks_test.reset_index(drop=True)

    # Define the features
    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next'] # x-offset as outcome variable
    X_test1 = weeks_test[features]

    # Run the identical model to before to predict the x-offset
    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)

    weeks_test['x_offset_predicted'] = y_predicted_x_offset     # Adding the predicted x-offset

    # Change the outcome variable to the y-offset
    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    # Run the identical model to before to predict the y-offset
    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset     # Adding the predicted y-offset

    # Change the outcome variable to the speed
    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    # Run the identical model to before to predict the speed
    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next     # Adding the predicted speed

    # Change the outcome variable to the acceleration
    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    # Run the identical model to before to predict the acceleration
    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next      # Adding the predicted acceleration

    # Run the identical model to before to predict the orientation
    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)
```

```
    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next     # Adding the predicted orientation

    # Creating a new CSV file with all the predicted telemetries
    weeks_test.to_csv('week8_predicted_xysao_final.csv')

# Creating the telemetries with the predicted motion of the pass rusher. As described in the thesis, the data has to
# again be manipulated, as the predicted telemetries will have to be matched with the next frame, as that is what is
# intended to be predicted. With there being one frame less per play, since the last one was deleted, there need to be
# no worries about list indices to be out of bounds, as the next frame can always be taken. The calculations are then
# identical to those made earlier.
def create_telemetry_week8_double_ml():
    # Load the data
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.loc[(weeks['week'] == 8)]
    week8_predicted_xy = pd.read_csv('week8_predicted_xysao.csv')

    # New columns for telemetries
    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
    weeks['y_to_blocker'] = 0
    weeks['distance_to_blocker'] = 0
    weeks['blocker_o'] = 0
    weeks['angle'] = 0
    weeks['blocker_s'] = 0
    weeks['blocker_a'] = 0
    weeks['x_offset'] = 0
    weeks['y_offset'] = 0
    weeks['speed'] = 0
    weeks['acceleration'] = 0
    weeks['orientation'] = 0

    # New data frame for telemetries
    telemetry_df = pd.DataFrame(columns=weeks.columns)

    # Iterating over all plays
    for k in range(len(plays)):
        # filter for the tracking data of the current play
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
        one_play = one_play.reset_index(drop=True)

        last_frame = one_play['frameId'].max()  # finding the amount of frames in the play

        # Iterate over every frame from the current play
        for i in range(len(one_play)):
            # Drop the last frame, again, as this frame will not need to be predicted
            if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
                one_play = one_play.drop(i)
            else:
                # If the current frame is that of a pass rusher, calculate the required telemetries identically to
                # earlier, just with the data from the pass blocker and passer of the next frame.
                if one_play['role'][i] == 'Pass Rush':
                    df_passer = one_play.loc[
                        (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                    df_passer = df_passer.reset_index(drop=True)
                    df_rusher_predictor = week8_predicted_xy.loc[(week8_predicted_xy['gameId'] == one_play['gameId'][i])
                                                             & (week8_predicted_xy['playId'] == one_play['playId'][i])
                                                               & (week8_predicted_xy['nflId'] == one_play['nflId'][i])
                                                               & (week8_predicted_xy['frameId'] == one_play['frameId'][i])]
                    df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                    one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                    df_rusher_predictor['x_offset_predicted'][0]
                    one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                    df_rusher_predictor['y_offset_predicted'][0]
                    one_play.at[i, 'distance_to_passer'] = round(
                        math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                    df_blocker = one_play.loc[
                        (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                    df_blocker = df_blocker.reset_index(drop=True)
                    all_blockers = []
                    individual_blocker = []

                    # Calculate the required telemetries for all blockers, appending them to a list. The data here is
                    # already those of the next frame, so the same code as earlier can be used.
                    for j in range(len(df_blocker)):
                        x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                        individual_blocker.append(x_to_blocker)

                        y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                        individual_blocker.append(y_to_blocker)

                        distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                        individual_blocker.append(distance_to_blocker)

                        x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                        y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                        distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                        dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                            i] * y_passer_to_blocker
                        cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                        cos_theta = min(1, max(-1, cos_theta))
                        theta = math.acos(cos_theta)
                        theta_degrees = math.degrees(theta)
                        individual_blocker.append(theta_degrees)
                        individual_blocker.append(df_blocker['o'][j])
                        individual_blocker.append(df_blocker['s'][j])
                        individual_blocker.append(df_blocker['a'][j])
                        all_blockers.append(individual_blocker)
                        individual_blocker = []

                    # Finding the smallest distance, only taking the closest blocker here
                    min_val = float('inf')
                    closest_blocker = None
                    for sub_arr in all_blockers:
                        if sub_arr[2] < min_val:
                            min_val = sub_arr[2]
                            closest_blocker = sub_arr

                    one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                    one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                    one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                    one_play.at[i, 'angle'] = closest_blocker[3]
                    one_play.at[i, 'blocker_o'] = closest_blocker[4]
                    one_play.at[i, 'blocker_s'] = closest_blocker[5]
                    one_play.at[i, 'blocker_a'] = closest_blocker[6]

                    # Add the offset to the last frame, which is straightforward, as it is the predicted one
                    one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                    one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                    one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                    one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                    one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]

        # Add telemetry to telemetry data frame
        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    # Filter for only the pass rushers, as passer and pass blockers are no longer required
    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    # Creating a new CSV file
    only_rushers.to_csv('weeks8_predicted_telemetry_rushers2.csv')

# Predicting the pressure for the predicted motion is identical to the earlier predictions, with the only difference
# being the fact that the model is trained over weeks 1-7, but obviously tested on the predicted telemetry data set.
def predict_pressure_with_predicted_telemetry():
    # Load the data
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    # Filtering for the training dataset and rounding all telemetries to two decimal places
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 1) | (weeks_train['week'] == 2) | (weeks_train['week'] == 3) | (weeks_train['week'] == 4) | (
                weeks_train['week'] == 5) | (weeks_train['week'] == 6) | (weeks_train['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    # Using the predicted motion telemetry dataset as the test dataset and rounding all telemetries to two decimal
    # places.
    weeks_test = pd.read_csv('weeks8_predicted_telemetry_rushers2.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']

    # Defining the features
    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    # Running the model, again with k=37
    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    # Adding the predicted occurrence, and, most importantly, the probability of a pressure occurring for the predicted
    # motion telemetry
    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    # Creating a new CSV file
    weeks_test.to_csv('week8_double_ml_pressure_final.csv')

# Finalizing the prediction by adding the PMP to the predicted pressure data frame
def finalize_prediction():
    # Loading the data
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week8_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week8 = pd.read_csv('week8_double_ml_pressure_final.csv')
    week8 = week8.drop(week8.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    # Iterating over every play
    for i in range(len(plays)):
        # Filtering for the current play from the predicted pressure data frame
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        # Filtering for the current play from the PMP data frame
        one_play_predicted = week8.loc[(week8['gameId'] == plays['gameId'][i]) & (week8['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()   # Find all unique players (pass rushers)

        # Iterate over every player, filter the play data by the current player
        for j in players:
            # The predicted pressure from the first KNN is needed here to get the actual pressure on the first frame,
            # as that will be used for the PMP in the first frame. The other PMP have to be moved one frame back,
            # naturally, as they predict the pressure of the next frame.
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            # Turn all PMP for this play into a list
            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            # Append the full PMP list with the actual pressure occurrence on the first frame
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            # Append the entries of the smaller list to the bigger one, this will result in the entries of the PMP to
            # now be the same size as the actual pressure occurrence predicitons
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            # Repeat the process, this time with the pressure probabilities for the predicted motion
            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    # Add the list to the column. The actual pressure and the PMP will now fit the right frames.
    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    # Creatin a new CSV file
    weeks.to_csv('week8_pmp_final.csv')

# This function will be used to fill the data frame with all frames with the predicted pressure and the PMP values for
# the pass rusher. The values for all non-pass rushers will be set to 1 as this will help later on with the
# visualization. Here, the data frame with all tracking data will be filtered for only week 8, the other weeks will be
# done later on.
def fill_dataframe():
    # Load the data
    predicted_weeks = pd.read_csv('week8_pmp_final.csv')
    predicted_weeks = predicted_weeks.drop(predicted_weeks.columns[0], axis=1)
    weeks = pd.read_csv('weeks_final2.1.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week8 = weeks.loc[(weeks['week'] == 8)]     # Filter for week 8
    week8 = week8.reset_index(drop=True)
    week8['predicted_pressure'] = 1     # Set pressure occurrence to 1
    week8['predicted_motion_pressure'] = 1      # Set pressure probability to 1

    # Iterate over every frame to add the appropriate predictions for the frames with pass rushers.
    for i in range(len(week8)):
        print(i)
        if week8['role'][i] == 'Pass Rush':
            player = predicted_weeks.loc[(predicted_weeks['gameId'] == week8['gameId'][i]) &
                                         (predicted_weeks['playId'] == week8['playId'][i]) &
                                         (predicted_weeks['nflId'] == week8['nflId'][i]) &
                                         (predicted_weeks['frameId'] == week8['frameId'][i])]
            player = player.reset_index(drop=True)
            week8.at[i, 'predicted_pressure'] = player['predicted_pressure_probability'][0]
            week8.at[i, 'predicted_motion_pressure'] = player['predicted_motion_predicted_pressure_probability_outcome'][0]

    # Creating a new CSV file
    week8.to_csv('week8_all_data_filled.csv')
```

```python
# Adding the game name to every frame, this is done for the dashboard later, so that the actual names of the game can
# be used and not the gameId.
def add_game_name():
    # Load the data
    week8 = pd.read_csv('week8_all_data_filled.csv')
    week8 = week8.drop(week8.columns[0], axis=1)

    week8['game_name'] = ''     # Create new column

    # Iterate over every frame and add the appropriate game name
    for i in range(len(week8)):
        if week8['gameId'][i] == 2021102800:
            week8.at[i, 'game_name'] = 'Arizona Cardinals vs Green Bay Packers'
        if week8['gameId'][i] == 2021103100:
            week8.at[i, 'game_name'] = 'Atlanta Falcons vs Carolina Panthers'
        if week8['gameId'][i] == 2021103103:
            week8.at[i, 'game_name'] = 'Cleveland Browns vs Pittsburgh Steelers'
        if week8['gameId'][i] == 2021103102:
            week8.at[i, 'game_name'] = 'Chicago Bears vs San Francisco 49ers'
        if week8['gameId'][i] == 2021103101:
            week8.at[i, 'game_name'] = 'Buffalo Bills vs Miami Dolphins'
        if week8['gameId'][i] == 2021103104:
            week8.at[i, 'game_name'] = 'Detroit Lions vs Philadelphia Eagles'
        if week8['gameId'][i] == 2021103105:
            week8.at[i, 'game_name'] = 'Houston Texans vs Los Angeles Rams'
        if week8['gameId'][i] == 2021103106:
            week8.at[i, 'game_name'] = 'Indianapolis Colts vs Tennessee Titans'
        if week8['gameId'][i] == 2021103107:
            week8.at[i, 'game_name'] = 'New York Jets vs Cincinnati Bengals'
        if week8['gameId'][i] == 2021103109:
            week8.at[i, 'game_name'] = 'Seattle Seahawks vs Jacksonville Jaguars'
        if week8['gameId'][i] == 2021103110:
            week8.at[i, 'game_name'] = 'Denver Broncos vs Washington Football Team'
        if week8['gameId'][i] == 2021103111:
            week8.at[i, 'game_name'] = 'New Orleans Saints vs Tampa Bay Buccaneers'
        if week8['gameId'][i] == 2021103112:
            week8.at[i, 'game_name'] = 'Minnesota Vikings vs Dallas Cowboys'
        if week8['gameId'][i] == 2021110100:
            week8.at[i, 'game_name'] = 'Kansas City Chiefs vs New York Giants'
        if week8['gameId'][i] == 2021103108:
            week8.at[i, 'game_name'] = 'Los Angeles Chargers vs New England Patriots'

    # Creating new CSV file
    week8.to_csv('week8_all_data_and_names_final.csv')

# Get the opponent for each frame in week 8, again for the dashboard later on
def get_opponent():
    # Load the data
    week8 = pd.read_csv('week8_all_data_and_names_final.csv')
    week8 = week8.drop(week8.columns[0], axis=1)

    week8['opponent'] = ''  # Create new column

    # Iterate over every frame and add the approapriate opponent
    for i in range(len(week8)):
        if week8['team'][i] == 'ATL':
            week8.at[i, 'opponent'] = 'CAR'
        if week8['team'][i] == 'CAR':
            week8.at[i, 'opponent'] = 'ATL'
        if week8['team'][i] == 'ARI':
            week8.at[i, 'opponent'] = 'GB'
        if week8['team'][i] == 'GB':
            week8.at[i, 'opponent'] = 'ARI'
        if week8['team'][i] == 'DET':
            week8.at[i, 'opponent'] = 'PHI'
        if week8['team'][i] == 'PHI':
            week8.at[i, 'opponent'] = 'DET'
        if week8['team'][i] == 'SEA':
            week8.at[i, 'opponent'] = 'JAX'
        if week8['team'][i] == 'JAX':
            week8.at[i, 'opponent'] = 'SEA'
        if week8['team'][i] == 'CLE':
            week8.at[i, 'opponent'] = 'PIT'
        if week8['team'][i] == 'PIT':
            week8.at[i, 'opponent'] = 'CLE'
        if week8['team'][i] == 'IND':
            week8.at[i, 'opponent'] = 'TEN'
        if week8['team'][i] == 'TEN':
            week8.at[i, 'opponent'] = 'IND'
        if week8['team'][i] == 'NYJ':
            week8.at[i, 'opponent'] = 'CIN'
        if week8['team'][i] == 'CIN':
            week8.at[i, 'opponent'] = 'NYJ'
        if week8['team'][i] == 'HOU':
            week8.at[i, 'opponent'] = 'LA'
        if week8['team'][i] == 'LA':
            week8.at[i, 'opponent'] = 'HOU'
        if week8['team'][i] == 'NO':
            week8.at[i, 'opponent'] = 'TB'
        if week8['team'][i] == 'TB':
            week8.at[i, 'opponent'] = 'NO'
        if week8['team'][i] == 'DEN':
            week8.at[i, 'opponent'] = 'WAS'
        if week8['team'][i] == 'WAS':
            week8.at[i, 'opponent'] = 'DEN'
        if week8['team'][i] == 'LAC':
            week8.at[i, 'opponent'] = 'NE'
        if week8['team'][i] == 'NE':
            week8.at[i, 'opponent'] = 'LAC'
        if week8['team'][i] == 'BUF':
            week8.at[i, 'opponent'] = 'MIA'
        if week8['team'][i] == 'MIA':
            week8.at[i, 'opponent'] = 'BUF'
        if week8['team'][i] == 'CHI':
            week8.at[i, 'opponent'] = 'SF'
        if week8['team'][i] == 'SF':
            week8.at[i, 'opponent'] = 'CHI'
        if week8['team'][i] == 'MIN':
            week8.at[i, 'opponent'] = 'DAL'
        if week8['team'][i] == 'DAL':
            week8.at[i, 'opponent'] = 'MIN'
        if week8['team'][i] == 'KC':
            week8.at[i, 'opponent'] = 'NYG'
        if week8['team'][i] == 'NYG':
            week8.at[i, 'opponent'] = 'KC'
    # Creating a new CSV file
    week8.to_csv('week8_all_data_and_names_and_opponents_final.csv')

# Also for the dashboard, add the player name to every frame
def add_player_name():
    # Load the data
    week8 = pd.read_csv('week8_all_data_and_names_and_opponents_final.csv')
    week8 = week8.drop(week8.columns[0], axis=1)
    player = pd.read_csv('players.csv')
    player_selected = player[['nflId', 'displayName']]
    merged_df = pd.merge(week8, player_selected, on='nflId', how='left')    # Merge data frames by nflId

    # Final data frame for the individual predicted pressures and PMP for the dashboard
    merged_df.to_csv('week8_all_data_and_names_and_opponents_and_game_names_final.csv')

# Calculating the pressure created by the entire defense on every frame. Also for the dashboard.
def get_total_pressure_by_defense():
    # Load the data
    rushers_total_pressure = pd.DataFrame(columns=['gameId', 'playId', 'frameId', 'total_pressure'])
    week8 = pd.read_csv('week8_all_data_and_names_and_opponents_and_game_names_final.csv')
    week8 = week8.drop(week8.columns[0], axis=1)
    games = week8['gameId'].unique().tolist()
    # Iterate over every game in week 8
    for i in games:
        game_df = week8.loc[(week8['gameId'] == i)]
        game_df = game_df.reset_index(drop=True)
        plays = game_df['playId'].unique().tolist() # Find the plays for each game
        # Iterate over every play for that game
        for j in plays:
            play_df = game_df.loc[(game_df['playId'] == j)]
            play_df = play_df.reset_index(drop=True)
            play_length = play_df['frameId'].max()
            rushers_df = play_df.loc[(play_df['role'] == 'Pass Rush')]  # Filter for the pass rushers on that play
            rushers_df = rushers_df.reset_index(drop=True)

            # For every frame, take the sum of the predicted pressure of every pass rusher and add the new row to the
            # new dataframe
            for k in range(play_length):
                rushers_per_frame = rushers_df.loc[(rushers_df['frameId'] == k+1)]
                rushers_per_frame = rushers_per_frame.reset_index(drop=True)
                total_pressure = rushers_per_frame['predicted_pressure'].sum()
                new_row = {'gameId': i, 'playId': j, 'frameId': k+1, 'total_pressure': total_pressure}
                rushers_total_pressure = rushers_total_pressure.append(new_row, ignore_index=True)

    # Creating a new CSV file
    rushers_total_pressure.to_csv('week8_total_pressure_defense_final.csv')

# As some of the column types got switched from int to float here, this had to be reversed
def convert_total_pressure_df():
    week8 = pd.read_csv('week8_total_pressure_defense_final.csv')
    week8 = week8.drop(week8.columns[0], axis=1)
    week8['gameId'] = week8['gameId'].astype(int)
    week8['playId'] = week8['playId'].astype(int)
    week8['frameId'] = week8['frameId'].astype(int)
    week8.to_csv('week8_total_pressure_defense_final.csv')

# Similarly to above, adding the game names to this data frame
def add_game_name_to_total_pressure_df():
    week8 = pd.read_csv('week8_total_pressure_defense_final.csv')
    week8 = week8.drop(week8.columns[0], axis=1)

    for i in range(len(week8)):
        print(i)
        if week8['gameId'][i] == 2021102800:
            week8.at[i, 'game_name'] = 'Arizona Cardinals vs Green Bay Packers'
        if week8['gameId'][i] == 2021103100:
            week8.at[i, 'game_name'] = 'Atlanta Falcons vs Carolina Panthers'
        if week8['gameId'][i] == 2021103103:
            week8.at[i, 'game_name'] = 'Cleveland Browns vs Pittsburgh Steelers'
        if week8['gameId'][i] == 2021103102:
            week8.at[i, 'game_name'] = 'Chicago Bears vs San Francisco 49ers'
        if week8['gameId'][i] == 2021103101:
            week8.at[i, 'game_name'] = 'Buffalo Bills vs Miami Dolphins'
        if week8['gameId'][i] == 2021103104:
            week8.at[i, 'game_name'] = 'Detroit Lions vs Philadelphia Eagles'
        if week8['gameId'][i] == 2021103105:
            week8.at[i, 'game_name'] = 'Houston Texans vs Los Angeles Rams'
        if week8['gameId'][i] == 2021103106:
            week8.at[i, 'game_name'] = 'Indianapolis Colts vs Tennessee Titans'
        if week8['gameId'][i] == 2021103107:
            week8.at[i, 'game_name'] = 'New York Jets vs Cincinnati Bengals'
        if week8['gameId'][i] == 2021103109:
            week8.at[i, 'game_name'] = 'Seattle Seahawks vs Jacksonville Jaguars'
        if week8['gameId'][i] == 2021103110:
            week8.at[i, 'game_name'] = 'Denver Broncos vs Washington Football Team'
        if week8['gameId'][i] == 2021103111:
            week8.at[i, 'game_name'] = 'New Orleans Saints vs Tampa Bay Buccaneers'
        if week8['gameId'][i] == 2021103112:
            week8.at[i, 'game_name'] = 'Minnesota Vikings vs Dallas Cowboys'
        if week8['gameId'][i] == 2021110100:
            week8.at[i, 'game_name'] = 'Kansas City Chiefs vs New York Giants'
        if week8['gameId'][i] == 2021103108:
            week8.at[i, 'game_name'] = 'Los Angeles Chargers vs New England Patriots'

    # Final data frame for the pressure by the entire defense for the dashboard
    week8.to_csv('week8_total_pressure_defense_final.csv')


# Now, the same predictions as for week 8 have to be made for all other weeks. The code is straightforward and the same,
# only swapping out the appropriate week.
# Week 7:
def predict_xysao_week7():
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 7)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next']
    X_test1 = weeks_test[features]

    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)

    weeks_test['x_offset_predicted'] = y_predicted_x_offset

    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset

    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next

    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next

    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)

    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next

    weeks_test.to_csv('week7_predicted_xysao_final.csv')
```

```python
def create_telemetry_week7_double_ml():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.loc[(weeks['week'] == 7)]
    week7_predicted_xy = pd.read_csv('week7_predicted_xysao_final.csv')

    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
    weeks['y_to_blocker'] = 0
    weeks['distance_to_blocker'] = 0
    weeks['blocker_o'] = 0
    weeks['angle'] = 0
    weeks['blocker_s'] = 0
    weeks['blocker_a'] = 0
    weeks['x_offset'] = 0
    weeks['y_offset'] = 0
    weeks['speed'] = 0
    weeks['acceleration'] = 0
    weeks['orientation'] = 0

    telemetry_df = pd.DataFrame(columns=weeks.columns)

    for k in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
        one_play = one_play.reset_index(drop=True)
        last_frame = one_play['frameId'].max()
        for i in range(len(one_play)):
            if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
                one_play = one_play.drop(i)
            else:
                if one_play['role'][i] == 'Pass Rush':
                    df_passer = one_play.loc[
                        (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                    df_passer = df_passer.reset_index(drop=True)
                    df_rusher_predictor = week7_predicted_xy.loc[(week7_predicted_xy['gameId'] == one_play['gameId'][i])
                                                             & (week7_predicted_xy['playId'] == one_play['playId'][i])
                                                                & (week7_predicted_xy['nflId'] == one_play['nflId'][i])
                                                                & (week7_predicted_xy['frameId'] == one_play['frameId'][i])]
                    df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                    one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                  df_rusher_predictor['x_offset_predicted'][0]
                    one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                  df_rusher_predictor['y_offset_predicted'][0]
                    one_play.at[i, 'distance_to_passer'] = round(
                        math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                    df_blocker = one_play.loc[
                        (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                    df_blocker = df_blocker.reset_index(drop=True)
                    all_blockers = []
                    individual_blocker = []

                    for j in range(len(df_blocker)):
                        x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                        individual_blocker.append(x_to_blocker)

                        y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                        individual_blocker.append(y_to_blocker)

                        distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                        individual_blocker.append(distance_to_blocker)

                        x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                        y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                        distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                        dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                            i] * y_passer_to_blocker
                        cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                        cos_theta = min(1, max(-1, cos_theta))
                        theta = math.acos(cos_theta)
                        theta_degrees = math.degrees(theta)
                        individual_blocker.append(theta_degrees)
                        individual_blocker.append(df_blocker['o'][j])
                        individual_blocker.append(df_blocker['s'][j])
                        individual_blocker.append(df_blocker['a'][j])

                        all_blockers.append(individual_blocker)
                        individual_blocker = []


                    min_val = float('inf')
                    closest_blocker = None

                    for sub_arr in all_blockers:
                        if sub_arr[2] < min_val:
                            min_val = sub_arr[2]
                            closest_blocker = sub_arr

                    one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                    one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                    one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                    one_play.at[i, 'angle'] = closest_blocker[3]
                    one_play.at[i, 'blocker_o'] = closest_blocker[4]
                    one_play.at[i, 'blocker_s'] = closest_blocker[5]
                    one_play.at[i, 'blocker_a'] = closest_blocker[6]

                    one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                    one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                    one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                    one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                    one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]


        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks7_predicted_telemetry_rushers.csv')


def predict_pressure_with_predicted_telemetry_week7():
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 1) | (weeks_train['week'] == 2) | (weeks_train['week'] == 3) | (weeks_train['week'] == 4) | (
                weeks_train['week'] == 5) | (weeks_train['week'] == 6) | (weeks_train['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    weeks_test = pd.read_csv('weeks7_predicted_telemetry_rushers.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week7_double_ml_pressure_final.csv')


def finalize_prediction_week7():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week7_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week7 = pd.read_csv('week7_double_ml_pressure_final.csv')
    week7 = week7.drop(week7.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    for i in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        one_play_predicted = week7.loc[(week7['gameId'] == plays['gameId'][i]) & (week7['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()
        for j in players:
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    weeks.to_csv('week7_pmp_final.csv')


# Week 6:
def predict_xy_week6():
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 6)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next']
    X_test1 = weeks_test[features]

    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)

    weeks_test['x_offset_predicted'] = y_predicted_x_offset

    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset

    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next

    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next

    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)

    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next

    weeks_test.to_csv('week6_predicted_xysao_final.csv')


def create_telemetry_week6_double_ml():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
```

```python
weeks = weeks.drop(weeks.columns[0], axis=1)
weeks = weeks.loc[(weeks['week'] == 6)]
week6_predicted_xy = pd.read_csv('week6_predicted_xysao_final.csv')

weeks['x_to_passer'] = 0
weeks['y_to_passer'] = 0
weeks['distance_to_passer'] = 0
weeks['x_to_blocker'] = 0
weeks['y_to_blocker'] = 0
weeks['distance_to_blocker'] = 0
weeks['blocker_o'] = 0
weeks['angle'] = 0
weeks['blocker_s'] = 0
weeks['blocker_a'] = 0
weeks['x_offset'] = 0
weeks['y_offset'] = 0
weeks['speed'] = 0
weeks['acceleration'] = 0
weeks['orientation'] = 0

telemetry_df = pd.DataFrame(columns=weeks.columns)

for k in range(len(plays)):
    one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
    one_play = one_play.reset_index(drop=True)
    last_frame = one_play['frameId'].max()
    for i in range(len(one_play)):
        if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
            one_play = one_play.drop(i)
        else:
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_passer = df_passer.reset_index(drop=True)
                df_rusher_predictor = week6_predicted_xy.loc[(week6_predicted_xy['gameId'] == one_play['gameId'][i])
                                                             & (week6_predicted_xy['playId'] == one_play['playId'][i])
                                                                & (week6_predicted_xy['nflId'] == one_play['nflId'][i])
                                                                & (week6_predicted_xy['frameId'] == one_play['frameId'][i])]
                df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                 df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                 df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                for j in range(len(df_blocker)):
                    x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                    individual_blocker.append(x_to_blocker)

                    y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                    individual_blocker.append(y_to_blocker)

                    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                    individual_blocker.append(distance_to_blocker)

                    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                        i] * y_passer_to_blocker
                    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                    cos_theta = min(1, max(-1, cos_theta))
                    theta = math.acos(cos_theta)
                    theta_degrees = math.degrees(theta)
                    individual_blocker.append(theta_degrees)
                    individual_blocker.append(df_blocker['o'][j])
                    individual_blocker.append(df_blocker['s'][j])
                    individual_blocker.append(df_blocker['a'][j])

                    all_blockers.append(individual_blocker)
                    individual_blocker = []

                min_val = float('inf')
                closest_blocker = None

                for sub_arr in all_blockers:
                    if sub_arr[2] < min_val:
                        min_val = sub_arr[2]
                        closest_blocker = sub_arr

                one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                one_play.at[i, 'angle'] = closest_blocker[3]
                one_play.at[i, 'blocker_o'] = closest_blocker[4]
                one_play.at[i, 'blocker_s'] = closest_blocker[5]
                one_play.at[i, 'blocker_a'] = closest_blocker[6]

                one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]


        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks6_predicted_telemetry_rushers.csv')


def predict_pressure_with_predicted_telemetry_week6():
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 1) | (weeks_train['week'] == 2) | (weeks_train['week'] == 3) | (weeks_train['week'] == 4) | (
                weeks_train['week'] == 5) | (weeks_train['week'] == 7) | (weeks_train['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    weeks_test = pd.read_csv('weeks6_predicted_telemetry_rushers.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
```

```python
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week6_double_ml_pressure_final.csv')


def finalize_prediction_week6():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week6_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week6 = pd.read_csv('week6_double_ml_pressure_final.csv')
    week6 = week6.drop(week6.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    for i in range(len(plays)):
        print(i)
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        one_play_predicted = week6.loc[(week6['gameId'] == plays['gameId'][i]) & (week6['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()
        for j in players:
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    weeks.to_csv('week6_pmp_final.csv')


# Week 5:
def predict_xy_week5():
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 5)]
    weeks_test = weeks_test.reset_index(drop=True)
    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next']
    X_test1 = weeks_test[features]

    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)

    weeks_test['x_offset_predicted'] = y_predicted_x_offset

    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset

    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next

    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next

    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)

    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next

    weeks_test.to_csv('week5_predicted_xysao_final.csv')


def create_telemetry_week5_double_ml():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.loc[(weeks['week'] == 5)]
    week5_predicted_xy = pd.read_csv('week5_predicted_xysao_final.csv')
```

```
weeks['x_to_passer'] = 0
weeks['y_to_passer'] = 0
weeks['distance_to_passer'] = 0
weeks['x_to_blocker'] = 0
weeks['y_to_blocker'] = 0
weeks['distance_to_blocker'] = 0
weeks['blocker_o'] = 0
weeks['angle'] = 0
weeks['blocker_s'] = 0
weeks['blocker_a'] = 0
weeks['x_offset'] = 0
weeks['y_offset'] = 0
weeks['speed'] = 0
weeks['acceleration'] = 0
weeks['orientation'] = 0

telemetry_df = pd.DataFrame(columns=weeks.columns)

for k in range(len(plays)):
    one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
    one_play = one_play.reset_index(drop=True)
    last_frame = one_play['frameId'].max()
    for i in range(len(one_play)):
        if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
            one_play = one_play.drop(i)
        else:
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_passer = df_passer.reset_index(drop=True)
                df_rusher_predictor = week5_predicted_xy.loc[(week5_predicted_xy['gameId'] == one_play['gameId'][i])
                                                             & (week5_predicted_xy['playId'] == one_play['playId'][i])
                                                                & (week5_predicted_xy['nflId'] == one_play['nflId'][i])
                                                                & (week5_predicted_xy['frameId'] == one_play['frameId'][i])]
                df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                  df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                  df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                for j in range(len(df_blocker)):
                    x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                    individual_blocker.append(x_to_blocker)

                    y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                    individual_blocker.append(y_to_blocker)

                    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                    individual_blocker.append(distance_to_blocker)

                    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                        i] * y_passer_to_blocker
                    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                    cos_theta = min(1, max(-1, cos_theta))
                    theta = math.acos(cos_theta)
                    theta_degrees = math.degrees(theta)
                    individual_blocker.append(theta_degrees)
                    individual_blocker.append(df_blocker['o'][j])
                    individual_blocker.append(df_blocker['s'][j])
                    individual_blocker.append(df_blocker['a'][j])
```

```
                        all_blockers.append(individual_blocker)
                        individual_blocker = []

                    min_val = float('inf')
                    closest_blocker = None

                    for sub_arr in all_blockers:
                        if sub_arr[2] < min_val:
                            min_val = sub_arr[2]
                            closest_blocker = sub_arr

                    one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                    one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                    one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                    one_play.at[i, 'angle'] = closest_blocker[3]
                    one_play.at[i, 'blocker_o'] = closest_blocker[4]
                    one_play.at[i, 'blocker_s'] = closest_blocker[5]
                    one_play.at[i, 'blocker_a'] = closest_blocker[6]

                    one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                    one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                    one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                    one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                    one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]

        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks5_predicted_telemetry_rushers.csv')


def predict_pressure_with_predicted_telemetry_week5():
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 1) | (weeks_train['week'] == 2) | (weeks_train['week'] == 3) | (weeks_train['week'] == 4) | (
                weeks_train['week'] == 6) | (weeks_train['week'] == 7) | (weeks_train['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    weeks_test = pd.read_csv('weeks5_predicted_telemetry_rushers.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']
    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week5_double_ml_pressure_final.csv')


def finalize_prediction_week5():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week5_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week5 = pd.read_csv('week5_double_ml_pressure_final.csv')
    week5 = week5.drop(week5.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    for i in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        one_play_predicted = week5.loc[(week5['gameId'] == plays['gameId'][i]) & (week5['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()
        for j in players:
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    weeks.to_csv('week5_pmp_final.csv')


# Week 4:
def predict_xy_week4():
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 4)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']
    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next']
    X_test1 = weeks_test[features]

    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)

    weeks_test['x_offset_predicted'] = y_predicted_x_offset

    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset

    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next

    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next

    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)

    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next

    weeks_test.to_csv('week4_predicted_xysao_final.csv')


def create_telemetry_week4_double_ml():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.loc[(weeks['week'] == 4)]
    week4_predicted_xy = pd.read_csv('week4_predicted_xysao_final.csv')

    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
```

```
weeks['y_to_blocker'] = 0
weeks['distance_to_blocker'] = 0
weeks['blocker_o'] = 0
weeks['angle'] = 0
weeks['blocker_s'] = 0
weeks['blocker_a'] = 0
weeks['x_offset'] = 0
weeks['y_offset'] = 0
weeks['speed'] = 0
weeks['acceleration'] = 0
weeks['orientation'] = 0

telemetry_df = pd.DataFrame(columns=weeks.columns)

for k in range(len(plays)):
    one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
    one_play = one_play.reset_index(drop=True)
    last_frame = one_play['frameId'].max()
    for i in range(len(one_play)):
        if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
            one_play = one_play.drop(i)
        else:
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_passer = df_passer.reset_index(drop=True)
                df_rusher_predictor = week4_predicted_xy.loc[(week4_predicted_xy['gameId'] == one_play['gameId'][i])
                                                             & (week4_predicted_xy['playId'] == one_play['playId'][i])
                                                               & (week4_predicted_xy['nflId'] == one_play['nflId'][i])
                                                               & (week4_predicted_xy['frameId'] == one_play['frameId'][i])]
                df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                  df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                  df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                for j in range(len(df_blocker)):
                    x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                    individual_blocker.append(x_to_blocker)

                    y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                    individual_blocker.append(y_to_blocker)

                    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                    individual_blocker.append(distance_to_blocker)

                    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                        i] * y_passer_to_blocker
                    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                    cos_theta = min(1, max(-1, cos_theta))
                    theta = math.acos(cos_theta)
                    theta_degrees = math.degrees(theta)
                    individual_blocker.append(theta_degrees)
                    individual_blocker.append(df_blocker['o'][j])
                    individual_blocker.append(df_blocker['s'][j])
                    individual_blocker.append(df_blocker['a'][j])

                    all_blockers.append(individual_blocker)
                    individual_blocker = []

                min_val = float('inf')
                closest_blocker = None
                for sub_arr in all_blockers:
                    if sub_arr[2] < min_val:
                        min_val = sub_arr[2]
                        closest_blocker = sub_arr

                one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                one_play.at[i, 'angle'] = closest_blocker[3]
                one_play.at[i, 'blocker_o'] = closest_blocker[4]
                one_play.at[i, 'blocker_s'] = closest_blocker[5]
                one_play.at[i, 'blocker_a'] = closest_blocker[6]

                one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]

        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks4_predicted_telemetry_rushers.csv')


def predict_pressure_with_predicted_telemetry_week4():
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 1) | (weeks_train['week'] == 2) | (weeks_train['week'] == 3) | (weeks_train['week'] == 5) | (
                weeks_train['week'] == 6) | (weeks_train['week'] == 7) | (weeks_train['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    weeks_test = pd.read_csv('weeks4_predicted_telemetry_rushers.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week4_double_ml_pressure_final.csv')


def finalize_prediction_week4():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week4_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week4 = pd.read_csv('week4_double_ml_pressure_final.csv')
    week4 = week4.drop(week4.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    for i in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        one_play_predicted = week4.loc[(week4['gameId'] == plays['gameId'][i]) & (week4['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()
        for j in players:
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    weeks.to_csv('week4_pmp_final.csv')


# Week 3:
def predict_xy_week3():
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 4) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 3)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next']
    X_test1 = weeks_test[features]

    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')
    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)

    weeks_test['x_offset_predicted'] = y_predicted_x_offset

    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset

    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next

    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next

    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)

    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next

    weeks_test.to_csv('week3_predicted_xysao_final.csv')


def create_telemetry_week3_double_ml():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.loc[(weeks['week'] == 3)]
    week3_predicted_xy = pd.read_csv('week3_predicted_xysao_final.csv')

    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
    weeks['y_to_blocker'] = 0
    weeks['distance_to_blocker'] = 0
    weeks['blocker_o'] = 0
    weeks['angle'] = 0
    weeks['blocker_s'] = 0
```

```
weeks['blocker_a'] = 0
weeks['x_offset'] = 0
weeks['y_offset'] = 0
weeks['speed'] = 0
weeks['acceleration'] = 0
weeks['orientation'] = 0

telemetry_df = pd.DataFrame(columns=weeks.columns)

for k in range(len(plays)):
    one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
    one_play = one_play.reset_index(drop=True)
    last_frame = one_play['frameId'].max()
    for i in range(len(one_play)):
        if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
            one_play = one_play.drop(i)
        else:
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_passer = df_passer.reset_index(drop=True)
                df_rusher_predictor = week3_predicted_xy.loc[(week3_predicted_xy['gameId'] == one_play['gameId'][i])
                                                             & (week3_predicted_xy['playId'] == one_play['playId'][i])
                                                               & (week3_predicted_xy['nflId'] == one_play['nflId'][i])
                                                               & (week3_predicted_xy['frameId'] == one_play['frameId'][i])]
                df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                                 df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                                df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                for j in range(len(df_blocker)):
                    x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                    individual_blocker.append(x_to_blocker)

                    y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                    individual_blocker.append(y_to_blocker)

                    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                    individual_blocker.append(distance_to_blocker)

                    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                        i] * y_passer_to_blocker
                    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                    cos_theta = min(1, max(-1, cos_theta))
                    theta = math.acos(cos_theta)
                    theta_degrees = math.degrees(theta)
                    individual_blocker.append(theta_degrees)
                    individual_blocker.append(df_blocker['o'][j])
                    individual_blocker.append(df_blocker['s'][j])
                    individual_blocker.append(df_blocker['a'][j])

                    all_blockers.append(individual_blocker)
                    individual_blocker = []

                min_val = float('inf')
                closest_blocker = None

                for sub_arr in all_blockers:
                    if sub_arr[2] < min_val:
                        min_val = sub_arr[2]
                        closest_blocker = sub_arr
                one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                one_play.at[i, 'angle'] = closest_blocker[3]
                one_play.at[i, 'blocker_o'] = closest_blocker[4]
                one_play.at[i, 'blocker_s'] = closest_blocker[5]
                one_play.at[i, 'blocker_a'] = closest_blocker[6]

                one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]

        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks3_predicted_telemetry_rushers.csv')


def predict_pressure_with_predicted_telemetry_week3():
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 1) | (weeks_train['week'] == 2) | (weeks_train['week'] == 4) | (weeks_train['week'] == 5) | (
                weeks_train['week'] == 6) | (weeks_train['week'] == 7) | (weeks_train['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    weeks_test = pd.read_csv('weeks3_predicted_telemetry_rushers.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')
    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week3_double_ml_pressure_final.csv')


def finalize_prediction_week3():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week3_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week3 = pd.read_csv('week3_double_ml_pressure_final.csv')
    week3 = week3.drop(week3.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    for i in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        one_play_predicted = week3.loc[(week3['gameId'] == plays['gameId'][i]) & (week3['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()
        for j in players:
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    weeks.to_csv('week3_pmp_final.csv')


# Week 2:
def predict_xy_week2():
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 3) | (weeks['week'] == 4) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 2)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next']
    X_test1 = weeks_test[features]

    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)
    weeks_test['x_offset_predicted'] = y_predicted_x_offset

    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset

    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next

    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next

    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)

    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next

    weeks_test.to_csv('week2_predicted_xysao_final.csv')


def create_telemetry_week2_double_ml():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.loc[(weeks['week'] == 2)]
    week2_predicted_xy = pd.read_csv('week2_predicted_xysao_final.csv')

    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
    weeks['y_to_blocker'] = 0
    weeks['distance_to_blocker'] = 0
    weeks['blocker_o'] = 0
    weeks['angle'] = 0
    weeks['blocker_s'] = 0
    weeks['blocker_a'] = 0
    weeks['x_offset'] = 0
    weeks['y_offset'] = 0
    weeks['speed'] = 0
    weeks['acceleration'] = 0
```

```python
weeks['orientation'] = 0

telemetry_df = pd.DataFrame(columns=weeks.columns)

for k in range(len(plays)):
    one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
    one_play = one_play.reset_index(drop=True)
    last_frame = one_play['frameId'].max()
    for i in range(len(one_play)):
        if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
            one_play = one_play.drop(i)
        else:
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_passer = df_passer.reset_index(drop=True)
                df_rusher_predictor = week2_predicted_xy.loc[(week2_predicted_xy['gameId'] == one_play['gameId'][i])
                                                           & (week2_predicted_xy['playId'] == one_play['playId'][i])
                                                             & (week2_predicted_xy['nflId'] == one_play['nflId'][i])
                                                             & (week2_predicted_xy['frameId'] == one_play['frameId'][i])]
                df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                     df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                     df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                for j in range(len(df_blocker)):
                    x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                    individual_blocker.append(x_to_blocker)

                    y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                    individual_blocker.append(y_to_blocker)

                    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                    individual_blocker.append(distance_to_blocker)

                    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                        i] * y_passer_to_blocker
                    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                    cos_theta = min(1, max(-1, cos_theta))
                    theta = math.acos(cos_theta)
                    theta_degrees = math.degrees(theta)
                    individual_blocker.append(theta_degrees)
                    individual_blocker.append(df_blocker['o'][j])
                    individual_blocker.append(df_blocker['s'][j])
                    individual_blocker.append(df_blocker['a'][j])

                    all_blockers.append(individual_blocker)
                    individual_blocker = []

                min_val = float('inf')
                closest_blocker = None

                for sub_arr in all_blockers:
                    if sub_arr[2] < min_val:
                        min_val = sub_arr[2]
                        closest_blocker = sub_arr

                one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                one_play.at[i, 'angle'] = closest_blocker[3]
                one_play.at[i, 'blocker_o'] = closest_blocker[4]
                one_play.at[i, 'blocker_s'] = closest_blocker[5]
                one_play.at[i, 'blocker_a'] = closest_blocker[6]

                # add offset
                one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]

        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks2_predicted_telemetry_rushers.csv')


def predict_pressure_with_predicted_telemetry_week2():
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 1) | (weeks_train['week'] == 3) | (weeks_train['week'] == 4) | (weeks_train['week'] == 5) | (
                weeks_train['week'] == 6) | (weeks_train['week'] == 7) | (weeks_train['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    weeks_test = pd.read_csv('weeks2_predicted_telemetry_rushers.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]
    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability

    weeks_test.to_csv('week2_double_ml_pressure_final.csv')


def finalize_prediction_week2():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week2_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week2 = pd.read_csv('week2_double_ml_pressure_final.csv')
    week2 = week2.drop(week2.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    for i in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        one_play_predicted = week2.loc[(week2['gameId'] == plays['gameId'][i]) & (week2['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()
        for j in players:
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    weeks.to_csv('week2_pmp_final.csv')


# Week 1:
def predict_xy_week1():
    weeks = pd.read_csv('all_weeks_with_next_xysao.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks_train = weeks.loc[
        (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (weeks['week'] == 5) | (
                weeks['week'] == 6) | (weeks['week'] == 7) | (weeks['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 1)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train1 = weeks_train[features]
    y_train1 = weeks_train['x_offset_next']
    X_test1 = weeks_test[features]

    clf1 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf1 = clf1.fit(X_train1, y_train1)

    y_predicted_x_offset = clf1.predict(X_test1)

    weeks_test['x_offset_predicted'] = y_predicted_x_offset

    X_train2 = weeks_train[features]
    y_train2 = weeks_train['y_offset_next']
    X_test2 = weeks_test[features]

    clf2 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf2 = clf2.fit(X_train2, y_train2)

    y_predicted_y_offset = clf2.predict(X_test2)

    weeks_test['y_offset_predicted'] = y_predicted_y_offset

    X_train3 = weeks_train[features]
    y_train3 = weeks_train['s_next']
    X_test3 = weeks_test[features]

    clf3 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf3 = clf3.fit(X_train3, y_train3)

    y_predicted_s_next = clf3.predict(X_test3)

    weeks_test['s_next_predicted'] = y_predicted_s_next

    X_train4 = weeks_train[features]
    y_train4 = weeks_train['a_next']
    X_test4 = weeks_test[features]

    clf4 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf4 = clf4.fit(X_train4, y_train4)

    y_predicted_a_next = clf4.predict(X_test4)

    weeks_test['a_next_predicted'] = y_predicted_a_next

    X_train5 = weeks_train[features]
    y_train5 = weeks_train['o_next']
    X_test5 = weeks_test[features]

    clf5 = KNeighborsRegressor(n_neighbors=37, weights='distance')

    clf5 = clf5.fit(X_train5, y_train5)

    y_predicted_o_next = clf5.predict(X_test5)

    weeks_test['o_next_predicted'] = y_predicted_o_next

    weeks_test.to_csv('week1_predicted_xysao_final.csv')


def create_telemetry_week1_double_ml():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('weeks_with_sack_hit_hurry_pressure_and_play_result.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    weeks = weeks.loc[(weeks['week'] == 1)]
    week1_predicted_xy = pd.read_csv('week1_predicted_xysao_final.csv')

    weeks['x_to_passer'] = 0
    weeks['y_to_passer'] = 0
    weeks['distance_to_passer'] = 0
    weeks['x_to_blocker'] = 0
    weeks['y_to_blocker'] = 0
    weeks['distance_to_blocker'] = 0
    weeks['blocker_o'] = 0
    weeks['angle'] = 0
    weeks['blocker_s'] = 0
    weeks['blocker_a'] = 0
    weeks['x_offset'] = 0
    weeks['y_offset'] = 0
    weeks['speed'] = 0
    weeks['acceleration'] = 0
    weeks['orientation'] = 0

    telemetry_df = pd.DataFrame(columns=weeks.columns)
```

```
for k in range(len(plays)):
    one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][k]) & (weeks['playId'] == plays['playId'][k])]
    one_play = one_play.reset_index(drop=True)
    last_frame = one_play['frameId'].max()
    for i in range(len(one_play)):
        if (one_play['frameId'][i] == last_frame) & (one_play['role'][i] == 'Pass Rush'):
            one_play = one_play.drop(i)
        else:
            if one_play['role'][i] == 'Pass Rush':
                df_passer = one_play.loc[
                    (one_play['role'] == 'Pass') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_passer = df_passer.reset_index(drop=True)
                df_rusher_predictor = week1_predicted_xy.loc[(week1_predicted_xy['gameId'] == one_play['gameId'][i])
                                                     & (week1_predicted_xy['playId'] == one_play['playId'][i])
                                                        & (week1_predicted_xy['nflId'] == one_play['nflId'][i])
                                                        & (week1_predicted_xy['frameId'] == one_play['frameId'][i])]
                df_rusher_predictor = df_rusher_predictor.reset_index(drop=True)
                one_play.at[i, 'x_to_passer'] = df_passer['x'][0] - one_play['x'][i] -
                                                 df_rusher_predictor['x_offset_predicted'][0]
                one_play.at[i, 'y_to_passer'] = df_passer['y'][0] - one_play['y'][i] -
                                                 df_rusher_predictor['y_offset_predicted'][0]
                one_play.at[i, 'distance_to_passer'] = round(
                    math.sqrt((one_play['y_to_passer'][i] ** 2) + (one_play['x_to_passer'][i]) ** 2), 2)

                df_blocker = one_play.loc[
                    (one_play['role'] == 'Pass Block') & (one_play['frameId'] == one_play['frameId'][i+1])]
                df_blocker = df_blocker.reset_index(drop=True)
                all_blockers = []
                individual_blocker = []

                for j in range(len(df_blocker)):
                    x_to_blocker = one_play['x'][i] - df_blocker['x'][j] - df_rusher_predictor['x_offset_predicted'][0]
                    individual_blocker.append(x_to_blocker)

                    y_to_blocker = one_play['y'][i] - df_blocker['y'][j] - df_rusher_predictor['y_offset_predicted'][0]
                    individual_blocker.append(y_to_blocker)

                    distance_to_blocker = math.sqrt((x_to_blocker ** 2) + (y_to_blocker ** 2))
                    individual_blocker.append(distance_to_blocker)

                    x_passer_to_blocker = df_passer['x'][0] - df_blocker['x'][j]
                    y_passer_to_blocker = df_passer['y'][0] - df_blocker['y'][j]
                    distance_passer_to_blocker = round(math.sqrt(x_passer_to_blocker ** 2 + y_passer_to_blocker ** 2))

                    dot_product = one_play['x_to_passer'][i] * x_passer_to_blocker + one_play['y_to_passer'][
                        i] * y_passer_to_blocker
                    cos_theta = dot_product / (distance_passer_to_blocker * one_play['distance_to_passer'][i])
                    cos_theta = min(1, max(-1, cos_theta))
                    theta = math.acos(cos_theta)
                    theta_degrees = math.degrees(theta)
                    individual_blocker.append(theta_degrees)
                    individual_blocker.append(df_blocker['o'][j])
                    individual_blocker.append(df_blocker['s'][j])
                    individual_blocker.append(df_blocker['a'][j])

                    all_blockers.append(individual_blocker)
                    individual_blocker = []

                min_val = float('inf')
                closest_blocker = None

                for sub_arr in all_blockers:
                    if sub_arr[2] < min_val:
                        min_val = sub_arr[2]
                        closest_blocker = sub_arr

                one_play.at[i, 'x_to_blocker'] = closest_blocker[0]
                one_play.at[i, 'y_to_blocker'] = closest_blocker[1]
                one_play.at[i, 'distance_to_blocker'] = closest_blocker[2]
                one_play.at[i, 'angle'] = closest_blocker[3]
                one_play.at[i, 'blocker_o'] = closest_blocker[4]
                one_play.at[i, 'blocker_s'] = closest_blocker[5]
                one_play.at[i, 'blocker_a'] = closest_blocker[6]
```

```
                    one_play.at[i, 'x_offset'] = df_rusher_predictor['x_offset_predicted'][0]
                    one_play.at[i, 'y_offset'] = df_rusher_predictor['y_offset_predicted'][0]
                    one_play.at[i, 'speed'] = df_rusher_predictor['s_next_predicted'][0]
                    one_play.at[i, 'acceleration'] = df_rusher_predictor['a_next_predicted'][0]
                    one_play.at[i, 'orientation'] = df_rusher_predictor['o_next_predicted'][0]


        telemetry_df = pd.concat([telemetry_df, one_play], ignore_index=True)

    only_rushers = telemetry_df.loc[(telemetry_df['role'] == 'Pass Rush')]
    only_rushers = only_rushers.reset_index(drop=True)

    only_rushers.to_csv('weeks1_predicted_telemetry_rushers.csv')


def predict_pressure_with_predicted_telemetry_week1():
    weeks_train = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')
    weeks_train = weeks_train.drop(weeks_train.columns[0], axis=1)
    weeks_train = weeks_train.loc[
        (weeks_train['week'] == 2) | (weeks_train['week'] == 3) | (weeks_train['week'] == 4) | (weeks_train['week'] == 5) | (
                weeks_train['week'] == 6) | (weeks_train['week'] == 7) | (weeks_train['week'] == 8)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_train['x_to_passer'] = weeks_train['x_to_passer'].round(2)
    weeks_train['y_to_passer'] = weeks_train['y_to_passer'].round(2)
    weeks_train['distance_to_passer'] = weeks_train['distance_to_passer'].round(2)
    weeks_train['x_to_blocker'] = weeks_train['x_to_blocker'].round(2)
    weeks_train['y_to_blocker'] = weeks_train['y_to_blocker'].round(2)
    weeks_train['distance_to_blocker'] = weeks_train['distance_to_blocker'].round(2)
    weeks_train['blocker_o'] = weeks_train['blocker_o'].round(2)
    weeks_train['angle'] = weeks_train['angle'].round(2)
    weeks_train['blocker_s'] = weeks_train['blocker_s'].round(2)
    weeks_train['blocker_a'] = weeks_train['blocker_a'].round(2)
    weeks_train['x_offset'] = weeks_train['x_offset'].round(2)
    weeks_train['y_offset'] = weeks_train['y_offset'].round(2)

    weeks_test = pd.read_csv('weeks1_predicted_telemetry_rushers.csv')
    weeks_test = weeks_test.drop(weeks_test.columns[0], axis=1)
    weeks_test['x_to_passer'] = weeks_test['x_to_passer'].round(2)
    weeks_test['y_to_passer'] = weeks_test['y_to_passer'].round(2)
    weeks_test['distance_to_passer'] = weeks_test['distance_to_passer'].round(2)
    weeks_test['x_to_blocker'] = weeks_test['x_to_blocker'].round(2)
    weeks_test['y_to_blocker'] = weeks_test['y_to_blocker'].round(2)
    weeks_test['distance_to_blocker'] = weeks_test['distance_to_blocker'].round(2)
    weeks_test['blocker_o'] = weeks_test['blocker_o'].round(2)
    weeks_test['angle'] = weeks_test['angle'].round(2)
    weeks_test['blocker_s'] = weeks_test['blocker_s'].round(2)
    weeks_test['blocker_a'] = weeks_test['blocker_a'].round(2)
    weeks_test['x_offset'] = weeks_test['x_offset'].round(2)
    weeks_test['y_offset'] = weeks_test['y_offset'].round(2)
    weeks_test['speed'] = weeks_test['speed'].round(2)
    weeks_test['acceleration'] = weeks_test['acceleration'].round(2)
    weeks_test['orientation'] = weeks_test['orientation'].round(2)
    weeks_test['s'] = weeks_test['speed']
    weeks_test['a'] = weeks_test['acceleration']
    weeks_test['o'] = weeks_test['orientation']

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)
    y_predicted_pressure_probability = clf.predict_proba(X_test)[:, 1]

    weeks_test['predicted_pressure'] = y_predicted_pressure
    weeks_test['predicted_pressure_probability'] = y_predicted_pressure_probability
    weeks_test.to_csv('week1_double_ml_pressure_final.csv')


def finalize_prediction_week1():
    plays = pd.read_csv('plays.csv')
    weeks = pd.read_csv('week1_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    week1 = pd.read_csv('week1_double_ml_pressure_final.csv')
    week1 = week1.drop(week1.columns[0], axis=1)
    full_pressure_list = []
    full_pressure_probability_list = []

    for i in range(len(plays)):
        one_play = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        one_play_predicted = week1.loc[(week1['gameId'] == plays['gameId'][i]) & (week1['playId'] == plays['playId'][i])]
        one_play_predicted = one_play_predicted.reset_index(drop=True)
        players = one_play['nflId'].unique().tolist()
        for j in players:
            one_play_player = one_play.loc[(one_play['nflId'] == j)]
            one_play_player = one_play_player.reset_index(drop=True)
            one_play_predicted_player = one_play_predicted.loc[(one_play_predicted['nflId'] == j)]
            one_play_predicted_player = one_play_predicted_player.reset_index(drop=True)

            indiv_player_pressure = one_play_predicted_player['predicted_pressure'].tolist()
            full_pressure_list.append(one_play_player['predicted_pressure'][0])
            for k in range(len(indiv_player_pressure)):
                full_pressure_list.append(indiv_player_pressure[k])

            indiv_player_pressure_probability = one_play_predicted_player['predicted_pressure_probability'].tolist()
            full_pressure_probability_list.append(one_play_player['predicted_pressure_probability'][0])
            for l in range(len(indiv_player_pressure_probability)):
                full_pressure_probability_list.append(indiv_player_pressure_probability[l])

    weeks['predicted_motion_predicted_pressure_outcome'] = full_pressure_list
    weeks['predicted_motion_predicted_pressure_probability_outcome'] = full_pressure_probability_list

    weeks.to_csv('week1_pmp_final.csv')


# Lastly, the accurary and validity of the model have to be analyzed.
# First, determine sensitivity and specificity. The model is implemented identically to before.
def sensitivity_specificity():
    weeks = pd.read_csv('weeks_only_rushers_with_telemetry_one_blocker_full.csv')

    weeks = weeks.drop(weeks.columns[0], axis=1)

    weeks['x_to_passer'] = weeks['x_to_passer'].round(2)
    weeks['y_to_passer'] = weeks['y_to_passer'].round(2)
    weeks['distance_to_passer'] = weeks['distance_to_passer'].round(2)
    weeks['x_to_blocker'] = weeks['x_to_blocker'].round(2)
    weeks['y_to_blocker'] = weeks['y_to_blocker'].round(2)
    weeks['distance_to_blocker'] = weeks['distance_to_blocker'].round(2)
    weeks['blocker_o'] = weeks['blocker_o'].round(2)
    weeks['angle'] = weeks['angle'].round(2)
    weeks['blocker_s'] = weeks['blocker_s'].round(2)
    weeks['blocker_a'] = weeks['blocker_a'].round(2)
    weeks['x_offset'] = weeks['x_offset'].round(2)
    weeks['y_offset'] = weeks['y_offset'].round(2)

    weeks_train = weeks.loc[
        (weeks['week'] == 1) | (weeks['week'] == 2) | (weeks['week'] == 3) | (weeks['week'] == 4) | (
                weeks['week'] == 5) | (weeks['week'] == 6) | (weeks['week'] == 7)]
    weeks_train = weeks_train.reset_index(drop=True)
    weeks_test = weeks.loc[(weeks['week'] == 8)]
    weeks_test = weeks_test.reset_index(drop=True)

    features = ['x_to_passer', 'y_to_passer', 'distance_to_passer', 'x_to_blocker', 'y_to_blocker',
                'distance_to_blocker', 'blocker_o', 'angle', 'x_offset', 'y_offset', 'o', 's', 'a', 'blocker_s',
                'blocker_a']

    X_train = weeks_train[features]
    y_train = weeks_train['pressure']
    X_test = weeks_test[features]
    y_test = weeks_test['pressure']

    clf = KNeighborsClassifier(n_neighbors=37, weights='distance')

    clf = clf.fit(X_train, y_train)

    y_predicted_pressure = clf.predict(X_test)

    # Get true positives, true negative, false positives, and false negatives since this is a binary classification
    tn, fp, fn, tp = confusion_matrix(y_test, y_predicted_pressure).ravel()
    specificity = tn / (tn+fp)
    sensitivity = tp / (tp + fn)

    # Display both
    print(specificity)
    print(sensitivity)


# Determine the predicitons per play for one pass rusher. If one frame predicts a pressure, the entire play will be
# classified as a pressure having occurred
def per_play_predictions():
    players = pd.read_csv('rushers_per_play_final.csv')
    players = players.drop(players.columns[0], axis=1)
    players['predicted_result'] = 0
    weeks = pd.read_csv('weeks_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    for i in range(len(players)):
        print(i)
        one_player = weeks.loc[(weeks['gameId'] == players['gameId'][i]) & (weeks['playId'] == players['playId'][i]) &
(weeks['nflId'] == players['nflId'][i])]
        one_player = one_player.reset_index(drop=True)
        # This will be 1 if any frame had a probability of larger than 50%
        predicted_result = one_player['predicted_pressure'].max()
        players.at[i, 'predicted_result'] = predicted_result

    players.to_csv('rushers_per_play_accuracy.csv')

# Specificity, sensitivity, and accuracy with the per-play predicitons
def play_analysis():
    players = pd.read_csv('rushers_per_play_accuracy.csv')
    players = players.drop(players.columns[0], axis=1)
    true_pressure = []
    predicted_pressure = []
    # For each play, if a pressure has occurred, that will be the true pressure, which will be compared to the predicted
    # one.
    for i in range(len(players)):
        if (players['rusher_sack'][i] == 1) | (players['rusher_hit'][i] == 1) | (players['rusher_hurry'][i] == 1):
            true_pressure.append(1)
        else:
            true_pressure.append(0)
        if players['predicted_result'][i] == 1:
            predicted_pressure.append(1)
        else:
            predicted_pressure.append(0)

    #  Get true positives, true negative, false positives, and false negatives since this is a binary classification
    tn, fp, fn, tp = confusion_matrix(true_pressure, predicted_pressure).ravel()

    # Calculate specificity, sensitivity, and accuracy
    specificity = tn / (tn + fp)
    sensitivity = tp / (tp + fn)
    accuracy = (tp + tn) / (tp + tn + fp + fn)

    # Display all values
    print(specificity)
    print(sensitivity)
    print(accuracy)

# Get the accuracy of sack plays having been correctly predicted, as this is the ultimate pressure that can be created
def sack_result():
    players = pd.read_csv('rushers_per_play_accuracy.csv')
    players = players.drop(players.columns[0], axis=1)
    true_sack = []
    predicted_pressure = []
```

```python
sack_percentage_list = []
for i in range(len(players)):
    # If a sack occurs on the play, add it to the true sack list. If it is also predicted, add a 1 to the prediction
    # list, 0 otherwise.
    if players['rusher_sack'][i] == 1:
        true_sack.append(1)
        sack_percentage_list.append(players['highest_pressure_on_play'][i])
        if players['predicted_result'][i] == 1:
            predicted_pressure.append(1)
        else:
            predicted_pressure.append(0)

#  Get true positives, true negative, false positives, and false negatives since this is a binary classification
tn, fp, fn, tp = confusion_matrix(true_sack, predicted_pressure).ravel()
# Calculate specificity, sensitivity, and accuracy
specificity = tn / (tn + fp)
sensitivity = tp / (tp + fn)
accuracy = (tp + tn) / (tp + tn + fp + fn)
# Caclulate the average pressure occuring on sack plays
average_pressure = sum(sack_percentage_list) / len(sack_percentage_list)
# Display all
print(average_pressure)
print(specificity)
print(sensitivity)
print(accuracy)
```

11.3.4 Data Analysis

```python
import pandas as pd
pd.set_option('display.max_columns', None)
import scipy.stats as stats

# Analyse each pass rusher per play by getting average pressure, highest pressure, amount of blockers, length of play
# and additional stats.
def rushers_per_play():
    # Load the data
    weeks = pd.read_csv('weeks_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    plays = pd.read_csv('plays.csv')
    players = pd.read_csv('players.csv')
    pff_data = pd.read_csv('pffScoutingData.csv')

    # Define the new data frame with the required stats
    column_names = ['gameId', 'playId', 'nflId', 'player_name', 'team', 'opponent', 'position', 'total_pressure',
                    'avg_pressure',
                    'highest_pressure_on_play', 'blockers', 'len_of_play', 'rusher_sack', 'rusher_hit', 'rusher_hurry',
                    'play_result']

    rushers = pd.DataFrame(columns=column_names)

    # Iterate over every play
    for i in range(len(plays)):
        # Filter the tracking data of the pass rushers only the current play
        plays_df = weeks.loc[(weeks['gameId'] == plays['gameId'][i]) & (weeks['playId'] == plays['playId'][i])]
        plays_df = plays_df.reset_index(drop=True)
        player_list = plays_df['nflId'].unique()    # Get the list of all pass rushers on the play
        for k in player_list:
            # Filter for only the current player
            player_df = plays_df.loc[(plays_df['nflId'] == k)]
            player_df = player_df.reset_index(drop=True)
            # Take only the frames either starting with the ball snap or every play if no event is specified
            ball_snap_list = player_df['event'].tolist()
            if ('ball_snap' in ball_snap_list) & (ball_snap_list[0] != 'ball_snap'):
                row_index = ball_snap_list.index('ball_snap')
                player_df = player_df.drop(player_df.index[:row_index])
                player_df = player_df.reset_index(drop=True)

            # Take over the required static data such as game, play, name, team, opponent and position
            gameId = player_df['gameId'][0]
            playId = player_df['playId'][0]
            nflId = player_df['nflId'][0]

            indiv_player_df = players.loc[(players['nflId'] == k)]
            indiv_player_df = indiv_player_df.reset_index(drop=True)
            name = indiv_player_df['displayName'][0]

            team = player_df['team'][0]
            opponent = player_df['opponent'][0]
            position = player_df['position'][0]

            total_pressure = player_df['predicted_pressure_probability'].sum()  # Total amount of pressure on play
            avg_pressure = total_pressure / len(player_df)  # Average amount of pressure created on play
            highest_pressure_on_play = player_df['predicted_pressure_probability'].max()    # Max amount of pressure

            pff_play = pff_data.loc[(pff_data['gameId'] == plays['gameId'][i]) & (pff_data['playId'] == plays['playId'][i])]
            pff_play = pff_play.reset_index(drop=True)
            blockers = pff_play['pff_nflIdBlockedPlayer'].eq(k).sum()   # Get the amount of blockers the player faced

            len_of_play = len(player_df)    # Length of the play

            # Did player create a pressure on the play according to PFF
            pff_rusher = pff_play.loc[(pff_play['nflId'] == k)]
            pff_rusher = pff_rusher.reset_index(drop=True)
            rusher_sack = pff_rusher['pff_sack'][0]
            rusher_hit = pff_rusher['pff_hit'][0]
            rusher_hurry = pff_rusher['pff_hurry'][0]

            # Get the pass result on the play, if it's a penalty, skip this play
            one_play = plays.loc[(plays['gameId'] == plays['gameId'][i]) & (plays['playId'] == plays['playId'][i])]
            one_play = one_play.reset_index(drop=True)
            play_result = one_play['passResult'][0]

            if play_result != 'R':
                new_row = {'gameId': gameId, 'playId': playId, 'nflId': nflId, 'player_name': name, 'team': team,
                           'opponent': opponent, 'position': position, 'total_pressure': total_pressure,
                           'avg_pressure': avg_pressure,
                           'highest_pressure_on_play': highest_pressure_on_play, 'blockers': blockers,
                           'len_of_play': len_of_play, 'rusher_sack': rusher_sack, 'rusher_hit': rusher_hit,
                           'rusher_hurry': rusher_hurry, 'play_result': play_result}

                rushers = rushers.append(new_row, ignore_index=True)

    # Creating new CSV file
    rushers.to_csv('rushers_per_play_final.csv')


# Create the average pressure created for each individual pass rusher
def full_player_analysis():
    # Load the data
    rushers_per_play = pd.read_csv('rushers_per_play_final.csv')
    rushers = rushers_per_play['nflId'].unique().tolist()  # Get all unique pass rushers

    # Create new data frame
    column_names = ['nflId', 'player_name', 'team', 'position', 'total_pressure', 'avg_pressure',
                    'total_blockers', 'avg_blockers', 'pressures_created', 'total_plays']
    rushers_avg = pd.DataFrame(columns=column_names)

    # Iterate over every pass rusher
    for i in rushers:
        rusher = rushers_per_play.loc[(rushers_per_play['nflId'] == i)]
        rusher = rusher.reset_index(drop=True)
        # Get all the wanted information
        nflId = rusher['nflId'][0]
        player_name = rusher['player_name'][0]
        team = rusher['team'][0]
        position = rusher['position'][0]
        total_pressure = rusher['avg_pressure'].sum()
        avg_pressure = total_pressure / len(rusher)  # Compute the average amount of pressure created
        total_blockers = rusher['blockers'].sum()
        avg_blockers = total_blockers / len(rusher)
        pressures_created = 0
        for j in range(len(rusher)):
            if (rusher['rusher_sack'][j] == 1.0) | (rusher['rusher_hit'][j] == 1.0) | (
                    rusher['rusher_hurry'][j] == 1.0):
                pressures_created += 1
        total_plays = len(rusher)

        new_row = {'nflId': nflId, 'player_name': player_name, 'team': team,
                   'position': position, 'total_pressure': total_pressure, 'avg_pressure': avg_pressure,
                   'total_blockers': total_blockers, 'avg_blockers': avg_blockers,
                   'pressures_created': pressures_created, 'total_plays': total_plays}
        rushers_avg = rushers_avg.append(new_row, ignore_index=True)  # Append all data to new data frame

    # Creating new CSV file
    rushers_avg.to_csv('rushers_avg_final.csv')


# Finding the average pressure per defense per play
def team_pressure_per_play():
    # Load the data
    rushers_per_play = pd.read_csv('rushers_per_play_final.csv')
    plays = pd.read_csv('plays.csv')
    pff = pd.read_csv('pffScoutingData.csv')
    # Create new data frame
    column_names = ['gameId', 'playId', 'team', 'opponent', 'number_of_rushers', 'number_of_blockers' 'total_pressure',
                    'avg_pressure',
                    'play_result', 'len_of_play', 'coverage', 'coverage_type']
    team_per_play = pd.DataFrame(columns=column_names)

    # Iterate over every play
    for i in range(len(plays)):
        # Find the number of pass rushers on a play
        number_of_rusher_df = pff.loc[(pff['gameId'] == plays['gameId'][i]) &
                                      (pff['playId'] == plays['playId'][i]) & (pff['pff_role'] == 'Pass Rush')]
        number_of_rusher_df = number_of_rusher_df.reset_index(drop=True)
        number_of_rushers = len(number_of_rusher_df['pff_role'])

        # This is done to avoid game 2021101700 play 2372 where there were no rushers
        if (plays['passResult'][i] != 'R') & (number_of_rushers != 0):
            rusher = rushers_per_play.loc[(rushers_per_play['gameId'] == plays['gameId'][i]) &
                                          (rushers_per_play['playId'] == plays['playId'][i])]
            rusher = rusher.reset_index(drop=True)
            # Get all required data
            blockers_on_play = pff.loc[(pff['gameId'] == plays['gameId'][i]) &
                                       (pff['playId'] == plays['playId'][i]) & (pff['pff_role'] == 'Pass Block')]
            blockers_on_play = blockers_on_play.reset_index(drop=True)

            gameId = rusher['gameId'][0]
            playId = rusher['playId'][0]
            team = rusher['team'][0]
            opponent = rusher['opponent'][0]
            number_of_rushers = len(rusher)
            number_of_blockers = len(blockers_on_play)
            total_pressure = rusher['total_pressure'].sum()
            avg_pressure = rusher['avg_pressure'].sum() / number_of_rushers  # Calculate average pressure on play
            play_result = rusher['play_result'][0]
            len_of_play = rusher['len_of_play'][0]
            coverage = plays['pff_passCoverage'][i]
            coverage_type = plays['pff_passCoverageType'][i]

            # Append data to new data frame
            new_row = {'gameId': gameId, 'playId': playId, 'team': team,
                       'opponent': opponent, 'number_of_rushers': number_of_rushers,
                       'number_of_blockers': number_of_blockers,
                       'total_pressure': total_pressure, 'avg_pressure': avg_pressure,
                       'play_result': play_result, 'len_of_play': len_of_play, 'coverage': coverage,
                       'coverage_type': coverage_type}
            team_per_play = team_per_play.append(new_row, ignore_index=True)

    # Creating new CSV file
    team_per_play.to_csv('team_per_play.csv')


# Data frame created to plot team's average pressure created per team with the winning percentage after eight weeks
def team_vs_pressure_8_weeks():
    # Load data
    teams_per_play = pd.read_csv('team_per_play.csv')
    teams = teams_per_play['team'].unique().tolist()
    # Create new data frame
    column_names = ['team', 'avg_pressure', 'avg_blockers', 'avg_rushers', 'total_plays', 'avg_length', 'record']
    teams_avg = pd.DataFrame(columns=column_names)
    # Iterate over every team
    for i in teams:
        team = teams_per_play.loc[(teams_per_play['team'] == i)]
        team = team.reset_index(drop=True)
        # Get important data like average pressure, blockers, rushers per play
        team_name = team['team'][0]
        avg_pressure = team['avg_pressure'].sum() / len(team)
        avg_blockers = team['number_of_blockers'].sum() / len(team)
        avg_rushers = team['number_of_rushers'].sum() / len(team)
        total_play = len(team)
        avg_length = team['len_of_play'].sum() / len(team)
        # Add team record after eight weeks
        if team_name == "BUF":
            record = 5 / 8
        if team_name == "NE":
            record = 4 / 8
        if team_name == "NYJ":
            record = 2 / 7
        if team_name == "MIA":
            record = 1 / 8
        if team_name == "BAL":
            record = 5 / 7
        if team_name == "CIN":
            record = 5 / 8
        if team_name == "PIT":
            record = 4 / 7
        if team_name == "CLE":
            record = 4 / 8
        if team_name == "TEN":
            record = 6 / 8
        if team_name == "LAC":
            record = 4 / 7
        if team_name == "DEN":
            record = 4 / 8
        if team_name == "KC":
            record = 4 / 8
        if team_name == "DAL":
            record = 6 / 7
        if team_name == "PHI":
            record = 3 / 8
        if team_name == "NYG":
            record = 2 / 8
        if team_name == "WAS":
            record = 2 / 8
        if team_name == "GB":
            record = 7 / 8
        if team_name == "MIN":
            record = 3 / 7
        if team_name == "CHI":
            record = 3 / 8
        if team_name == "DET":
            record = 0
        if team_name == "TB":
            record = 6 / 8
        if team_name == "NO":
            record = 5 / 7
        if team_name == "CAR":
            record = 4 / 8
        if team_name == "ATL":
            record = 3 / 7
        if team_name == "ARI":
            record = 7 / 8
        if team_name == "LA":
            record = 7 / 8
        if team_name == "SF":
            record = 3 / 7
        if team_name == "SEA":
            record = 3 / 8
        if team_name == "LV":
            record = 5 / 7
        if team_name == "IND":
            record = 3 / 8
        if team_name == "JAX":
            record = 1 / 7
        if team_name == "HOU":
            record = 1 / 8

        # Append data to new data frame
        new_row = {'team': team_name, 'avg_pressure': avg_pressure, 'avg_blockers': avg_blockers,
                   'avg_rushers': avg_rushers, 'total_plays': total_play, 'avg_length': avg_length, 'record': record}
        teams_avg = teams_avg.append(new_row, ignore_index=True)

    # Creating new CSV file
    teams_avg.to_csv('teams_avg_final.csv')


# Identical code to the previous function, but replace team with opponent to see the pressure that each offense faced/
# How much pressure they allowed
def opponent_pressure_8_weeks():
    teams_per_play = pd.read_csv('team_per_play.csv')
    teams = teams_per_play['opponent'].unique().tolist()

    column_names = ['team', 'avg_pressure', 'avg_blockers', 'avg_rushers', 'total_plays', 'avg_length', 'record']
    teams_avg = pd.DataFrame(columns=column_names)

    for i in teams:
        team = teams_per_play.loc[(teams_per_play['opponent'] == i)]
        team = team.reset_index(drop=True)
```

```
team_name = team['opponent'][0]
avg_pressure = team['avg_pressure'].sum() / len(team)
avg_blockers = team['number_of_blockers'].sum() / len(team)
avg_rushers = team['number_of_rushers'].sum() / len(team)
total_play = len(team)
avg_length = team['len_of_play'].sum() / len(team)
if team_name == "BUF":
    record = 5 / 8
if team_name == "NE":
    record = 4 / 8
if team_name == "NYJ":
    record = 2 / 7
if team_name == "MIA":
    record = 1 / 8
if team_name == "BAL":
    record = 5 / 7
if team_name == "CIN":
    record = 5 / 8
if team_name == "PIT":
    record = 4 / 7
if team_name == "CLE":
    record = 4 / 8
if team_name == "TEN":
    record = 6 / 8
if team_name == "LAC":
    record = 4 / 7
if team_name == "DEN":
    record = 4 / 8
if team_name == "KC":
    record = 4 / 8
if team_name == "DAL":
    record = 6 / 7
if team_name == "PHI":
    record = 3 / 8
if team_name == "NYG":
    record = 2 / 8
if team_name == "WAS":
    record = 2 / 8
if team_name == "GB":
    record = 7 / 8
if team_name == "MIN":
    record = 3 / 7
if team_name == "CHI":
    record = 3 / 8
if team_name == "DET":
    record = 0
if team_name == "TB":
    record = 6 / 8
if team_name == "NO":
    record = 5 / 7
if team_name == "CAR":
    record = 4 / 8
if team_name == "ATL":
    record = 3 / 7
if team_name == "ARI":
    record = 7 / 8
if team_name == "LA":
    record = 7 / 8
if team_name == "SF":
    record = 3 / 7
if team_name == "SEA":
    record = 3 / 8
if team_name == "LV":
    record = 5 / 7
if team_name == "IND":
    record = 3 / 8
if team_name == "JAX":
    record = 1 / 7
if team_name == "HOU":
    record = 1 / 8

new_row = {'team': team_name, 'avg_pressure': avg_pressure, 'avg_blockers': avg_blockers,
           'avg_rushers': avg_rushers, 'total_plays': total_play, 'avg_length': avg_length, 'record': record}
teams_avg = teams_avg.append(new_row, ignore_index=True)
```

```
    teams_avg.to_csv('opponent_avg_final.csv')


# Identical code above for average pressure created per team, but this time with record after all weeks
def team_vs_pressure_all_weeks():
    teams_per_play = pd.read_csv('team_per_play.csv')
    teams = teams_per_play['team'].unique().tolist()

    column_names = ['team', 'avg_pressure', 'avg_blockers', 'avg_rushers', 'total_plays', 'avg_length', 'record', 'final_record']
    teams_avg = pd.DataFrame(columns=column_names)

    for i in teams:
        team = teams_per_play.loc[(teams_per_play['team'] == i)]
        team = team.reset_index(drop=True)
        team_name =  team['team'][0]
        avg_pressure = team['avg_pressure'].sum() / len(team)
        avg_blockers = team['number_of_blockers'].sum() / len(team)
        avg_rushers = team['number_of_rushers'].sum() / len(team)
        total_play = len(team)
        avg_length = team['len_of_play'].sum() / len(team)
        if team_name == "BUF":
            record = 5/8
            final_record = 11/17
        if team_name == "NE":
            record = 4/8
            final_record = 10 / 17
        if team_name == "NYJ":
            record = 2/7
            final_record = 4 / 17
        if team_name == "MIA":
            record = 1/8
            final_record = 8 / 17
        if team_name == "BAL":
            record = 5/7
            final_record = 8 / 17
        if team_name == "CIN":
            record = 5/8
            final_record = 10 / 17
        if team_name == "PIT":
            record = 4/7
            final_record = 9 / 17
        if team_name == "CLE":
            record = 4/8
            final_record = 8 / 17
        if team_name == "TEN":
            record = 6/8
            final_record = 12 / 17
        if team_name == "LAC":
            record = 4/7
            final_record = 9 / 17
        if team_name == "DEN":
            record = 4/8
            final_record = 7 / 17
        if team_name == "KC":
            record = 4/8
            final_record = 12 / 17
        if team_name == "DAL":
            record = 6/7
            final_record = 12 / 17
        if team_name == "PHI":
            record = 3/8
            final_record = 9 / 17
        if team_name == "NYG":
            record = 2/8
            final_record = 4 / 17
        if team_name == "WAS":
            record = 2/8
            final_record = 7 / 17
        if team_name == "GB":
            record = 7/8
            final_record = 13 / 17
        if team_name == "MIN":
            record = 3/7
            final_record = 8 / 17
        if team_name == "CHI":
            record = 3/8
            final_record = 6 / 17
        if team_name == "DET":
            record = 0
            final_record = 3 / 17
        if team_name == "TB":
            record = 6/8
            final_record = 13 / 17
        if team_name == "NO":
            record = 5/7
            final_record = 9 / 17
        if team_name == "CAR":
            record = 4/8
            final_record = 5 / 17
        if team_name == "ATL":
            record = 3/7
            final_record = 7 / 17
        if team_name == "ARI":
            record = 7/8
            final_record = 11 / 17
        if team_name == "LA":
            record = 7/8
            final_record = 12 / 17
        if team_name == "SF":
            record = 3/7
            final_record = 10 / 17
        if team_name == "SEA":
            record = 3/8
            final_record = 7 / 17
        if team_name == "LV":
            record = 5/7
            final_record = 10 / 17
        if team_name == "IND":
            record = 3/8
            final_record = 9 / 17
        if team_name == "JAX":
            record = 1/7
            final_record = 3 / 17
        if team_name == "HOU":
            record = 1/8
            final_record = 4 / 17

        new_row = {'team': team_name, 'avg_pressure': avg_pressure, 'avg_blockers': avg_blockers,
        'avg_rushers': avg_rushers, 'total_plays': total_play, 'avg_length': avg_length, 'record': record,
'final_record': final_record}
        teams_avg = teams_avg.append(new_row, ignore_index=True)

    teams_avg.to_csv('teams_avg_final.csv')


# Identical code above for average pressure allowed per team, but this time with record after all weeks
def opponent_pressure():
    teams_per_play = pd.read_csv('team_per_play.csv')
    teams = teams_per_play['opponent'].unique().tolist()

    column_names = ['team', 'avg_pressure', 'avg_blockers', 'avg_rushers', 'total_plays', 'avg_length', 'record', 'final_record']
    teams_avg = pd.DataFrame(columns=column_names)

    for i in teams:
        team = teams_per_play.loc[(teams_per_play['opponent'] == i)]
        team = team.reset_index(drop=True)
        team_name = team['opponent'][0]
        avg_pressure = team['avg_pressure'].sum() / len(team)
        avg_blockers = team['number_of_blockers'].sum() / len(team)
        avg_rushers = team['number_of_rushers'].sum() / len(team)
        total_play = len(team)
        avg_length = team['len_of_play'].sum() / len(team)
        if team_name == "BUF":
            record = 5 / 8
            final_record = 11 / 17
        if team_name == "NE":
            record = 4 / 8
            final_record = 10 / 17
```

```
if team_name == "NYJ":
    record = 2 / 7
    final_record = 4 / 17
if team_name == "MIA":
    record = 1 / 8
    final_record = 8 / 17
if team_name == "BAL":
    record = 5 / 7
    final_record = 8 / 17
if team_name == "CIN":
    record = 5 / 8
    final_record = 10 / 17
if team_name == "PIT":
    record = 4 / 7
    final_record = 9 / 17
if team_name == "CLE":
    record = 4 / 8
    final_record = 8 / 17
if team_name == "TEN":
    record = 6 / 8
    final_record = 12 / 17
if team_name == "LAC":
    record = 4 / 7
    final_record = 9 / 17
if team_name == "DEN":
    record = 4 / 8
    final_record = 7 / 17
if team_name == "KC":
    record = 4 / 8
    final_record = 12 / 17
if team_name == "DAL":
    record = 6 / 7
    final_record = 12 / 17
if team_name == "PHI":
    record = 3 / 8
    final_record = 9 / 17
if team_name == "NYG":
    record = 2 / 8
    final_record = 4 / 17
if team_name == "WAS":
    record = 2 / 8
    final_record = 7 / 17
if team_name == "GB":
    record = 7 / 8
    final_record = 13 / 17
if team_name == "MIN":
    record = 3 / 7
    final_record = 8 / 17
if team_name == "CHI":
    record = 3 / 8
    final_record = 6 / 17
if team_name == "DET":
    record = 0
    final_record = 3 / 17
if team_name == "TB":
    record = 6 / 8
    final_record = 13 / 17
if team_name == "NO":
    record = 5 / 7
    final_record = 9 / 17
if team_name == "CAR":
    record = 4 / 8
    final_record = 5 / 17
if team_name == "ATL":
    record = 3 / 7
    final_record = 7 / 17
if team_name == "ARI":
    record = 7 / 8
    final_record = 11 / 17
if team_name == "LA":
    record = 7 / 8
    final_record = 12 / 17
if team_name == "SF":
    record = 3 / 7
```

```
            final_record = 10 / 17
        if team_name == "SEA":
            record = 3 / 8
            final_record = 7 / 17
        if team_name == "LV":
            record = 5 / 7
            final_record = 10 / 17
        if team_name == "IND":
            record = 3 / 8
            final_record = 9 / 17
        if team_name == "JAX":
            record = 1 / 7
            final_record = 3 / 17
        if team_name == "HOU":
            record = 1 / 8
            final_record = 4 / 17

        new_row = {'team': team_name, 'avg_pressure': avg_pressure, 'avg_blockers': avg_blockers,
'avg_rushers': avg_rushers, 'total_plays': total_play, 'avg_length': avg_length, 'record':
record,'final_record': final_record}
        teams_avg = teams_avg.append(new_row, ignore_index=True)

    teams_avg.to_csv('opponent_avg_final.csv')

# Pearson Correlation for the pass rush vs winning percentage after eight weeks
def defense_winning_eight_weeks():
    pressure_defense = pd.read_csv('teams_avg_final.csv')
    pressure_defense = pressure_defense.drop(pressure_defense.columns[0], axis=1)

    corr, p_value = stats.pearsonr(pressure_defense['avg_pressure'], pressure_defense['record'])

    print("Pearson correlation coefficient:", corr)
    print("p-value:", p_value)

# Pearson Correlation for the pass rush vs winning percentage after all weeks
def defense_winning_all_weeks():
    pressure_defense = pd.read_csv('teams_avg_final.csv')
    pressure_defense = pressure_defense.drop(pressure_defense.columns[0], axis=1)

    corr, p_value = stats.pearsonr(pressure_defense['avg_pressure'], pressure_defense['final_record'])

    print("Pearson correlation coefficient:", corr)
    print("p-value:", p_value)

# Pearson Correlation for the pass block vs winning percentage after eight weeks
def offense_winning_eight_weeks():
    pressure_defense = pd.read_csv('opponent_avg_final.csv')
    pressure_defense = pressure_defense.drop(pressure_defense.columns[0], axis=1)

    corr, p_value = stats.pearsonr(pressure_defense['avg_pressure'], pressure_defense['record'])

    print("Pearson correlation coefficient:", corr)
    print("p-value:", p_value)

# Pearson Correlation for the pass block vs winning percentage after eight weeks
def offense_winning_all_weeks():
    pressure_defense = pd.read_csv('opponent_avg_final.csv')
    pressure_defense = pressure_defense.drop(pressure_defense.columns[0], axis=1)

    corr, p_value = stats.pearsonr(pressure_defense['avg_pressure'], pressure_defense['final_record'])

    print("Pearson correlation coefficient:", corr)
    print("p-value:", p_value)


# Code to create a CSV file to showcase the average pressure created by position for each team, which will be used
# for visualization
def position_per_team_pressure():
    # Load data
    rushers = pd.read_csv('rushers_per_play_final.csv')
    rushers = rushers.drop(rushers.columns[0], axis=1)
    # Load the average pressure by team and sort it, iterating over every unique team in that list will allow for the
    # final data frame here to be in order of the average pressure allowed by each team, which will be easier for the
    # visualization later on
    teams_avg = pd.read_csv('teams_avg_final.csv')
    teams_avg = teams_avg.drop(teams_avg.columns[0], axis=1)
    teams_avg = teams_avg.sort_values(by='avg_pressure', ascending=False)
    team_names = teams_avg['team'].unique()
    # Create new data frame
    column_names = ['team', 'position', 'avg_pressure', 'number_of_plays']
    teams_and_positions = pd.DataFrame(columns=column_names)
    # Iterate over every team
    for i in team_names:
        team = rushers.loc[(rushers['team'] == i)]
        team = team.reset_index(drop=True)
        positions_per_team = team['position'].unique()  # Take each unique position
        for j in positions_per_team:
            # Get the average pressure created by each position
            position = team.loc[(team['position'] == j)]
            position = position.reset_index(drop=True)
            team_name = i
            position_name = j
            avg_pressure = position['avg_pressure'].mean()
            number_of_plays = len(position)
            # Add data to new data frame
            new_row = {'team': team_name, 'position': position_name, 'avg_pressure': avg_pressure, 'number_of_plays':
number_of_plays}
            teams_and_positions = teams_and_positions.append(new_row, ignore_index=True)

    teams_and_positions.to_csv('positions_per_team_avg.csv')

# Get the average pressure created for each team per position. This is the identical code as before, switching out the
# position and the team. The resulting data frame is identical, just in a different order, but this was done for easier
# handling later on in the visualization
def teams_per_position_pressure():
    rushers = pd.read_csv('rushers_per_play_final.csv')
    rushers = rushers.drop(rushers.columns[0], axis=1)
    position_avg = pd.read_csv('position_avg_final.csv')
    position_avg = position_avg.drop(position_avg.columns[0], axis=1)
    position_avg = position_avg.sort_values(by='avg_pressure', ascending=False)
    position_avg = position_avg.reset_index(drop=True)
    position_names = position_avg['position'].unique()
    column_names = ['position', 'team', 'avg_pressure', 'number_of_plays']
    teams_and_positions = pd.DataFrame(columns=column_names)

    for i in position_names:
        position = rushers.loc[(rushers['position'] == i)]
        position = position.reset_index(drop=True)
        teams_per_position = position['team'].unique()
        for j in teams_per_position:
            team = position.loc[(position['team'] == j)]
            team = team.reset_index(drop=True)
            team_name = j
            position_name = i
            avg_pressure = team['avg_pressure'].mean()
            number_of_plays = len(team)

            new_row = {'position': position_name, 'team': team_name, 'avg_pressure': avg_pressure,
                       'number_of_plays': number_of_plays}
            teams_and_positions = teams_and_positions.append(new_row, ignore_index=True)

    teams_and_positions.to_csv('teams_per_positions_avg.csv')


# Average pressure allowed by each team per formation/coverage.
# Identical code to before, just changing position with formation.
def teams_per_formation_pressure():
    team_per_play = pd.read_csv('team_per_play.csv')
    team_per_play = team_per_play.drop(team_per_play.columns[0], axis=1)
    column_names = ['formation', 'team', 'avg_pressure']
    teams_and_formation = pd.DataFrame(columns=column_names)
    formations = team_per_play['coverage'].unique()
    for i in formations:
        formation = team_per_play.loc[(team_per_play['coverage'] == i)]
        formation = formation.reset_index(drop=True)
        teams = formation['team'].unique()
        for j in teams:
            team = formation.loc[(formation['team'] == j)]
            team = team.reset_index(drop=True)
            formation_name = i
```

```python
            team_name = j
            avg_pressure = team['avg_pressure'].mean()

            new_row = {'formation': formation_name, 'team': team_name, 'avg_pressure': avg_pressure}

            teams_and_formation = teams_and_formation.append(new_row, ignore_index=True)

    teams_and_formation.to_csv('teams_and_formation.csv')


# Average pressure allowed by each team per position. Identical code to before, but changing out the team with the
# opponent, returning in the average amount of pressure that was allowed, instead of created.
def opponents_per_position_pressure():
    rushers = pd.read_csv('rushers_per_play_final.csv')
    rushers = rushers.drop(rushers.columns[0], axis=1)
    position_avg = pd.read_csv('position_avg_final.csv')
    position_avg = position_avg.drop(position_avg.columns[0], axis=1)
    position_avg = position_avg.sort_values(by='avg_pressure', ascending=False)
    position_avg = position_avg.reset_index(drop=True)
    position_names = position_avg['position'].unique()
    column_names = ['position', 'team', 'avg_pressure', 'number_of_plays']
    teams_and_positions = pd.DataFrame(columns=column_names)

    for i in position_names:
        position = rushers.loc[(rushers['position'] == i)]
        position = position.reset_index(drop=True)
        teams_per_position = position['opponent'].unique()
        for j in teams_per_position:
            team = position.loc[(position['opponent'] == j)]
            team = team.reset_index(drop=True)
            team_name = j
            position_name = i
            avg_pressure = team['avg_pressure'].mean()
            number_of_plays = len(team)

            new_row = {'position': position_name, 'team': team_name, 'avg_pressure': avg_pressure,
                       'number_of_plays': number_of_plays}
            teams_and_positions = teams_and_positions.append(new_row, ignore_index=True)

    teams_and_positions.to_csv('opponents_per_positions_avg.csv')


# Identical code to above, changing position with formation/coverage to return the average pressure allowed by each team
# per coverage.
def opponents_per_formation_pressure():
    team_per_play = pd.read_csv('team_per_play.csv')
    team_per_play = team_per_play.drop(team_per_play.columns[0], axis=1)
    column_names = ['formation', 'team', 'avg_pressure']
    teams_and_formation = pd.DataFrame(columns=column_names)
    formations = team_per_play['coverage'].unique()
    for i in formations:
        formation = team_per_play.loc[(team_per_play['coverage'] == i)]
        formation = formation.reset_index(drop=True)
        teams = formation['opponent'].unique()
        for j in teams:
            team = formation.loc[(formation['opponent'] == j)]
            team = team.reset_index(drop=True)
            formation_name = i
            team_name = j
            avg_pressure = team['avg_pressure'].mean()

            new_row = {'formation': formation_name, 'team': team_name, 'avg_pressure': avg_pressure}

            teams_and_formation = teams_and_formation.append(new_row, ignore_index=True)

    teams_and_formation.to_csv('opponents_and_formation.csv')

# Get the rankings for pressure created by each team's NRT
def NRT_all_teams():
    tpp = pd.read_csv('teams_per_positions_avg.csv')
    tpp = tpp.drop(tpp.columns[0], axis=1)
    nrt = tpp.loc[(tpp['position'] == 'NRT')]
    nrt = nrt.sort_values(by='avg_pressure', ascending=False)
    nrt = nrt.reset_index(drop=True)
    nrt.to_csv('nrt.csv')

# Rank the average pressure created by all positions of the Green Bay Packers
def GB_all_positions():
    ppt = pd.read_csv('positions_per_team_avg.csv')
    ppt = ppt.drop(ppt.columns[0], axis=1)
    gb = ppt.loc[(ppt['team'] == 'GB')]
    gb = gb.sort_values(by='avg_pressure', ascending=False)
    gb = gb.reset_index(drop=True)
    gb.to_csv('gb.csv')

# Rank the average pressure created in Cover-2 for all teams
def cover2_all_teams():
    formations = pd.read_csv('teams_and_formation.csv')
    formations = formations.drop(formations.columns[0], axis=1)
    cover2 = formations.loc[(formations['formation'] == 'Cover-2')]
    cover2 = cover2.sort_values(by='avg_pressure', ascending=False)
    cover2 = cover2.reset_index(drop=True)
    cover2.to_csv('cover2.csv')

# Rank the average pressure created by all coverages for the Los Angeles Rams
def LA_all_formations():
    formations = pd.read_csv('teams_and_formation.csv')
    formations = formations.drop(formations.columns[0], axis=1)
    la = formations.loc[(formations['team'] == 'LA')]
    la = la.sort_values(by='avg_pressure', ascending=False)
    la = la.reset_index(drop=True)
    la.to_csv('la.csv')

# Rank the average pressure allowed against the LLB by all teams
def LLB_all_opponents():
    tpp = pd.read_csv('opponents_per_positions_avg.csv')
    tpp = tpp.drop(tpp.columns[0], axis=1)
    llb = tpp.loc[(tpp['position'] == 'LLB')]
    llb = llb.sort_values(by='avg_pressure', ascending=True)
    llb = llb.reset_index(drop=True)
    llb.to_csv('llb.csv')

# Rank the average pressure allowed against the Cover-3 formation against all opponents
def cover3_all_opponents():
    formations = pd.read_csv('opponents_and_formation.csv')
    formations = formations.drop(formations.columns[0], axis=1)
    cover3 = formations.loc[(formations['formation'] == 'Cover-3')]
    cover3 = cover3.sort_values(by='avg_pressure', ascending=True)
    cover3 = cover3.reset_index(drop=True)
    cover3.to_csv('cover3.csv')

# Rank the average pressure allowed against all positions by the New York Giants
def NYG_all_positions():
    ppt = pd.read_csv('opponents_per_positions_avg.csv')
    ppt = ppt.drop(ppt.columns[0], axis=1)
    nyg = ppt.loc[(ppt['team'] == 'NYG')]
    nyg = nyg.sort_values(by='avg_pressure', ascending=True)
    nyg = nyg.reset_index(drop=True)
    nyg.to_csv('nyg.csv')

# Rank the average pressure allowed against all coverages by the New York Jets
def NYJ_all_formations():
    formations = pd.read_csv('opponents_and_formation.csv')
    formations = formations.drop(formations.columns[0], axis=1)
    nyj = formations.loc[(formations['team'] == 'NYJ')]
    nyj = nyj.sort_values(by='avg_pressure', ascending=True)
    nyj = nyj.reset_index(drop=True)
    nyj.to_csv('nyj.csv')

# Show the average pressure allowed by each position. Identical code to the average pressure per team.
def positions_vs_pressure():
    teams_per_play = pd.read_csv('rushers_per_play_final.csv')
    positions = teams_per_play['position'].unique().tolist()

    column_names = ['position', 'avg_pressure', 'avg_blockers', 'total_plays']
    position_avg = pd.DataFrame(columns=column_names)

    for i in positions:
```

```python
        position = teams_per_play.loc[(teams_per_play['position'] == i)]
        position = position.reset_index(drop=True)
        position_name = position['position'][0]
        avg_pressure = position['avg_pressure'].sum() / len(position)
        avg_blockers = position['blockers'].sum() / len(position)
        total_plays = len(position)

        new_row = {'position': position_name, 'avg_pressure': avg_pressure, 'avg_blockers': avg_blockers,
                   'total_plays': total_plays}
        position_avg = position_avg.append(new_row, ignore_index=True)

    position_avg = position_avg.sort_values(by='avg_pressure', ascending=True)
    position_avg.to_csv('position_avg_final.csv')

# Average pressure created by each coverage type. Identical code as above.
def coverage_vs_pressure():
    teams_per_play = pd.read_csv('team_per_play.csv')
    coverages = teams_per_play['coverage'].unique().tolist()

    column_names = ['coverage', 'avg_pressure', 'avg_blockers', 'total_plays']
    coverage_avg = pd.DataFrame(columns=column_names)

    for i in coverages:
        coverage = teams_per_play.loc[(teams_per_play['coverage'] == i)]
        coverage = coverage.reset_index(drop=True)
        coverage_name = coverage['coverage'][0]
        avg_pressure = coverage['avg_pressure'].sum() / len(coverage)
        avg_blockers = coverage['number_of_blockers'].sum() / len(coverage)
        total_plays = len(coverage)

        new_row = {'coverage': coverage_name, 'avg_pressure': avg_pressure, 'avg_blockers': avg_blockers,
                   'total_plays': total_plays}
        coverage_avg = coverage_avg.append(new_row, ignore_index=True)

    coverage_avg = coverage_avg.sort_values(by='avg_pressure', ascending=True)
    coverage_avg.to_csv('coverage_avg_final.csv')

# Filter the data frame only for DeForest Buckner and his sack against the Tennessee Titans, which will be used for
# visualization later
def buckner_sack_play():
    weeks = pd.read_csv('weeks_predicted_pressure_final.csv')
    weeks = weeks.drop(weeks.columns[0], axis=1)
    buckner = weeks.loc[(weeks['gameId'] == 2021103106) & (weeks['playId'] == 4334) & (weeks['nflId'] == 43296)]
    buckner = buckner.reset_index(drop=True)
    buckner.to_csv('buckner_sack_game_2021103106_play_4334.csv')


# Lastly, analyses on the PMP have to be done.

# Concatenate all PMP predictions into one data frame
def concat_all_pmp():
    week1 = pd.read_csv('week1_pmp_final.csv')
    week2 = pd.read_csv('week2_pmp_final.csv')
    week3 = pd.read_csv('week3_pmp_final.csv')
    week4 = pd.read_csv('week4_pmp_final.csv')
    week5 = pd.read_csv('week5_pmp_final.csv')
    week6 = pd.read_csv('week6_pmp_final.csv')
    week7 = pd.read_csv('week7_pmp_final.csv')
    week8 = pd.read_csv('week8_pmp_final.csv')

    frames = [week1, week2, week3, week4, week5, week6, week7, week8]

    all_weeks = pd.concat(frames, ignore_index=True)
    all_weeks = all_weeks.drop(all_weeks.columns[0], axis=1)

    all_weeks.to_csv('pmp_all_weeks.csv')

# Finding the difference between PMP and actual pressure for every frame
def find_delta_every_frame():
    pmp = pd.read_csv('pmp_all_weeks.csv')
    pmp = pmp.drop(pmp.columns[0], axis=1)
    pmp['pmp_delta'] = pmp['predicted_pressure_probability'] - pmp['predicted_motion_predicted_pressure_probability_outcome']
    pmp.to_csv('pmp_with_delta_every_play.csv')
```

```
# Find the difference between PMP and actual pressure for the overall play per player
def pmp_per_player_per_play():
    pmp = pd.read_csv('pmp_with_delta_every_play.csv')
    plays = pd.read_csv('plays.csv')
    pmp_per_player_per_play = pd.DataFrame(columns=['gameId', 'playId', 'nflId', 'team', 'position', 'avg_pressure',
'pmp_delta'])

    for i in range(len(plays)):
        print(i)
        one_play = pmp.loc[(pmp['gameId'] == plays['gameId'][i]) & (pmp['playId'] == plays['playId'][i])]
        one_play = one_play.reset_index(drop=True)
        players = one_play['nflId'].unique()
        for j in players:
            one_player = one_play.loc[(one_play['nflId'] == j)]
            one_player = one_player.reset_index(drop=True)
            gameId = one_player['gameId'][0]
            playId = one_player['playId'][0]
            nflId = one_player['nflId'][0]
            team = one_player['team'][0]
            position = one_player['position'][0]
            avg_pressure = one_player['predicted_pressure_probability'].mean()
            pmp_delta = one_player['pmp_delta'].sum()

            new_row = {'gameId': gameId, 'playId': playId, 'nflId': nflId, 'team': team,
                       'position': position, 'avg_pressure': avg_pressure,'pmp_delta': pmp_delta}
            pmp_per_player_per_play = pmp_per_player_per_play.append(new_row, ignore_index=True)

    pmp_per_player_per_play.to_csv('pmp_per_player_per_play.csv')

# Find the average pressure to PMP delta across all pass rushers
def find_average_delta():
    pmp = pd.read_csv('pmp_per_player_per_play.csv')
    pmp = pmp.drop(pmp.columns[0], axis=1)
    print(pmp['pmp_delta'].mean())

# Pearson Correlation between delta on a play and average pressure created on a play
def pressure_pmp_correlations():
    pressure_pmp = pd.read_csv('pmp_per_player_per_play.csv')
    pressure_pmp = pressure_pmp.drop(pressure_pmp.columns[0], axis=1)

    corr, p_value = stats.pearsonr(pressure_pmp['avg_pressure'], pressure_pmp['pmp_delta'])

    print("Pearson correlation coefficient:", corr)
    print("p-value:", p_value)

# Create new dataset for each pass rusher and their average PMP delta
def average_pmp_per_player():
    pmp = pd.read_csv('pmp_per_player_per_play.csv')
    pmp = pmp.drop(pmp.columns[0], axis=1)
    all_players = pd.read_csv('players.csv')
    players = pmp['nflId'].unique()
    pmp_per_player = pd.DataFrame(
        columns=['nflId', 'name', 'pmp_delta'])
    for i in players:
        player = pmp.loc[(pmp['nflId'] == i)]
        player = player.reset_index(drop=True)
        nflId = player['nflId'][0]
        one_player = all_players.loc[(all_players['nflId'] == i)]
        one_player = one_player.reset_index(drop=True)
        name = one_player['displayName'][0]
        pmp_delta = player['pmp_delta'].mean()
        new_row = {'nflId': i, 'name': name, 'pmp_delta': pmp_delta}
        pmp_per_player = pmp_per_player.append(new_row, ignore_index=True)

    pmp_per_player.to_csv('pmp_per_player.csv')
```

11.3.5 Streamlit Visualizations

```python
import altair as alt
import streamlit as st
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go

# Running this code will allow the user to open a Streamlit site locally that shows all the graphs more in-depth than
# in the thesis, as hovering over the data points is possible.

alt.themes.enable("streamlit")

st.set_page_config(layout="wide", page_title="Master Thesis Visualizations - Christopher Patzanovsky",
                   page_icon=":statue_of_liberty:")

st.write("""
# Master Thesis - Analysis of the NFL
### Christopher Patzanovsky
""")

# Read all the required data frames
df = pd.read_csv('rushers_avg.csv')
df = df.drop(df.columns[0], axis=1)

df2 = pd.read_csv('teams_avg_final.csv')
df2 = df2.drop(df2.columns[0], axis=1)
sorted_df2 = df2.sort_values(by='avg_pressure', ascending=False)

df3 = pd.read_csv('opponent_avg_final.csv')
df3 = df3.drop(df3.columns[0], axis=1)
sorted_df3 = df3.sort_values(by='avg_pressure', ascending=True)

df4 = pd.read_csv('position_avg_final.csv')
df4 = df4.drop(df4.columns[0], axis=1)
df4 = df4.sort_values(by='avg_pressure', ascending=False)

df6 = pd.read_csv('teams_completions_percentages.csv')
df6 = df6.drop(df6.columns[0], axis=1)
df6_avg_plays = df6.sort_values(by='avg_plays', ascending=True)
df6_comp_perc = df6.sort_values(by='completion_percentage', ascending=True)

df7 = pd.read_csv('buckner_sack_game_2021103106_play_4334.csv')
df7 = df7.drop(df7.columns[0], axis=1)

df8 = pd.read_csv('positions_per_team_avg.csv')
df9 = pd.read_csv('teams_per_positions_avg.csv')

df10 = pd.read_csv('week8_pmp_final.csv')
df10 = df10.drop(df10.columns[0], axis=1)

landry = df10.loc[(df10['gameId'] == 2021103106) & (df10['playId'] == 225) & (df10['nflId'] == 46110)]
landry = landry.reset_index(drop=True)

df11 = pd.read_csv('pmp_per_player_per_play.csv')
df11 = df11.drop(df11.columns[0], axis=1)

df11['delta_rounded'] = df11['pmp_delta'].round(1)
df11g = df11.groupby('delta_rounded').count().reset_index()


# Distribution of all players and their average pressure created by the amount of blockers they faced
fig = px.scatter(df, x='avg_blockers', y='avg_pressure', color='team',
                     range_x=[-0.05, 2.05],range_y=[0, 0.4],
                     hover_data=['player_name', 'team', 'position'], labels={'avg_blockers': 'Average number of blockers',
'avg_pressure': 'Average pressure created', 'team': 'Team'})

st.write(fig)

# Defining the color scale for the winning respective percentage
colorscale = [[0, 'red'], [0.5, 'red'], [0.5, 'green'], [1, 'green']]
```

```
# Average pressure created by each team with winning record
fig3 = px.scatter(sorted_df2, x='team', y='avg_pressure',
                     range_y=[0.10, 0.14], color='record',
                  color_continuous_scale='RdYlGn',
                  range_color=(0, 1), labels={'team': 'Team', 'avg_pressure': 'Average pressure created', 'record': 'Record'})
st.write(fig3)

# Average pressure allowed by each team with respective winning record
fig4 = px.scatter(sorted_df3, x='team', y='avg_pressure',
                     range_y=[0.10, 0.14], color='record',
                  color_continuous_scale='RdYlGn',
                  range_color=(0, 1), labels={'team': 'Team', 'avg_pressure': 'Average pressure allowed', 'record': 'Record'})
st.write(fig4)

# Average pressure created per position with the respective average number of blockers faced
fig5 = px.scatter(df4, x='position', y='avg_pressure',
                     range_y=[0, 0.4], color='avg_blockers',
                  color_continuous_scale='RdYlGn',
                  range_color=(0, 1.60801144492132), labels={'position': 'Position', 'avg_pressure': 'Average pressure created',
'avg_blockers': 'Average number of blockers'}, category_orders={'position': df4['position'].unique()})

st.write(fig5)

# Average number of plassing plays for the first 8 weeks per team
fig6 = px.scatter(df6_avg_plays, x='team', y='avg_plays',
weeks'})             range_y=[25, 45], labels={'team': 'Team', 'avg_plays': 'Average number of passing plays for the first 8
st.write(fig6)

# Completion percentage for each team for the first 8 weeks
fig7 = px.scatter(df6_comp_perc, x='team', y='completion_percentage',
first 8 weeks'})     range_y=[0.4, 0.65], labels={'team': 'Team', 'completion_percentage': 'Completion percentage throughout the
st.write(fig7)

# Create a data frame with the accuracy of each k from 1 to 100 for all three modeled KNNs
k_list = []
for i in range(100):
    k_list.append(i+1)

data = {'k': k_list,
        'one_blocker': [0.8023052607935839, 0.8023052607935839, 0.8429267170303333, 0.8437046329825407, 0.8558445044620602,
0.856644333539682, 0.8613885251918768, 0.8621609628627307, 0.8645166238447675, 0.8654260185494607, 0.8665490662269433,
0.8672009817080185, 0.8679953325042867, 0.8683568990736226, 0.8688170747073228, 0.86905811908688, 0.8695292512832874,
0.8697319476933696, 0.8701044708254126, 0.870241427859252, 0.8704934288015164, 0.8706413423980629, 0.8708221256827308,
0.8708221256827308, 0.8710741266249952, 0.8711836922520667, 0.8714302149129775, 0.8714411714756847, 0.8715616936654633,
0.8716219547603525, 0.8714959542892203, 0.8715726502281704, 0.8715671719468169, 0.871616476478999, 0.8715507371027561,
0.871720563824717, 0.8716329113230598, 0.8716055199162919, 0.8716603027298276, 0.8716712592925348, 0.871578128509524,
0.8716274330417062, 0.8716109981976454, 0.871654824448474, 0.8716493461671204, 0.8716603027298276, 0.8716931724179491,
0.8716329113230598, 0.8716767375738883, 0.8715616936654633, 0.8715836067908775, 0.8715836067908775, 0.871578128509524,
0.8715616936654633, 0.8715452588214025, 0.8715178674146347, 0.8715671719468169, 0.871501432570574, 0.8715069108519276,
0.8714685628824526, 0.8713699538180882, 0.871501432570574, 0.8714137800689168, 0.8714630846010989, 0.8714466497570382,
0.8714576063197453, 0.8714521280383918, 0.8714356931943311, 0.871397345224856, 0.8714028235062097, 0.8713809103807953,
0.8714083017875632, 0.8713918669435025, 0.8713316058486131, 0.8713535189740275, 0.8714083017875632, 0.8714028235062097,
0.871320649285906, 0.871320649285906, 0.8712329967842488, 0.8712001270961274, 0.8712549099096631, 0.8712384750656024,
0.871243953346956, 0.8712549099096631, 0.8712384750656024, 0.8712384750656024, 0.871243953346956, 0.8711891705334203,
0.8712056053774809, 0.8711891705334203, 0.8712220402215417, 0.8711508225639453, 0.8712110836588346, 0.871139866001238,
0.8711891705334203, 0.8711508225639453, 0.8711508225639453, 0.8711343877198845, 0.8711234311571774],

        'two_blocker': [0.7924936588893332, 0.7924936588893332, 0.8306991930491566, 0.8311867600896247, 0.8446304625312947,
0.8451508992598842, 0.8519823161077906, 0.8525411008058552, 0.8562389407195174, 0.8573072055834644, 0.8596299968773796,
0.860062781104312, 0.8613775686291697, 0.8622157456762665, 0.8633387933537491, 0.8640016653975315, 0.8644782758752924,
0.8650041908852355, 0.8654424533935214, 0.8658478462136858, 0.866412109193104, 0.8664395005998718, 0.8667462843556719,
0.8669051545149256, 0.8672995907723828, 0.8673653301486258, 0.867787157812851, 0.8679350714093974, 0.8679788976602261,
0.8681815940703083, 0.8683568990736226, 0.8684335950125727, 0.8685048126701691, 0.8686582045480692, 0.8688061181446157,
0.8689540317411621, 0.8688444661140907, 0.8688992489276264, 0.8689266403343943, 0.8690362059614658, 0.8690252493987586,
0.8690909887750015, 0.8690800322122944, 0.869019771117405, 0.869019771117405, 0.8689485534598086, 0.8690909887750015,
0.8690909887750015, 0.869162206432598, 0.8691731629953051, 0.8692443806529015, 0.869277250341023, 0.8693429897172659,
0.8693484679986194, 0.8694415987816302, 0.8694635119070445, 0.8695182947205803, 0.8694744684697516, 0.8695566426900553,
0.8695895123781767, 0.8695456861273482, 0.8695949906595303, 0.8696388169103589, 0.8696552517544196, 0.8696552517544196,
0.8695895123781767, 0.8697100345679554, 0.8696278603476517, 0.8696278603476517, 0.8696771648798339, 0.8696223820662982,
0.8696716865984804, 0.8696607300357732, 0.8696442951917125, 0.8696333386290053, 0.8696771648798339, 0.8696333386290053,
0.8696004689408838, 0.8697100345679554, 0.869649773473066, 0.8696607300357732, 0.8696442951917125, 0.8696552517544196,
0.869649773473066, 0.8696333386290053, 0.869649773473066, 0.8696388169103589, 0.8696716865984804, 0.8696552517544196,
0.8696662083171267, 0.8696442951917125, 0.8696826431611875, 0.8696935997238946, 0.8696990780052481, 0.8697483825374304,
0.8696935997238946, 0.8697209911306625, 0.8697374259747233, 0.8697209911306625, 0.8696826431611875],

        'three_blocker': [0.8040309194199595, 0.8040309194199595, 0.8432389790674869, 0.8434416754775692, 0.8556472863333315,
0.8564197240041854, 0.8612296550326232, 0.8620294841102449, 0.8644563627498781, 0.8654095837053999, 0.8666202838845397,
0.8672283731147864, 0.8680994198500046, 0.8683568990736226, 0.8687951615819085, 0.8691786412766587, 0.8694142073748624,
0.8698962961339769, 0.8701044708254126, 0.8703674283303842, 0.8705208202082843, 0.8707509080251343, 0.8707344731810737,
0.8709207347470952, 0.8710303003741666, 0.8712932578791381, 0.8713863886621489, 0.8714356931943311, 0.8715123891332811,
0.8714521280383918, 0.871578128509524, 0.8714192583502703, 0.8716219547603525, 0.8714411714756847, 0.8716055199162919,
0.8716876941365954, 0.8717041289806562, 0.8715671719468169, 0.8715836067908775, 0.8715726502281704, 0.8716055199162919,
0.8716822158552419, 0.8716712592925348, 0.8717041289806562, 0.8717041289806562, 0.8716986506993026, 0.8716931724179491,
0.871654824448474, 0.8716931724179491, 0.8716931724179491, 0.8716767375738883, 0.8715890850722311, 0.8715452588214025,
0.871501432570574, 0.8715288239773419, 0.8715671719468169, 0.8715562153841097, 0.8714685628824526, 0.8714904760078668,
0.8714795194451597, 0.8714083017875632, 0.8714247366316239, 0.8713754320994418, 0.8714247366316239, 0.8714192583502703,
0.8714137800689168, 0.8714137800689168, 0.8713480406926739, 0.8714466497570382, 0.8713644755367346, 0.8713863886621489,
0.871397345224856, 0.8713480406926739, 0.8713261275672596, 0.8713480406926739, 0.8713316058486131, 0.8713151710045525,
0.871282301316431, 0.8712549099096631, 0.8712384750656024, 0.8711672574080059, 0.8711782139707132, 0.8711453442825917,
0.871139866001238, 0.8711453442825917, 0.8711015180317631, 0.8711453442825917, 0.8711015180317631, 0.8711727356893595,
0.8711234311571774, 0.8710960397504095, 0.8711069963131166, 0.8711015180317631, 0.8711069963131166, 0.8711015180317631,
0.8710686483436416, 0.8710193438114595, 0.871052213499581, 0.8710686483436416, 0.8711015180317631]}

print(len(data['three_blocker']))

# Plot all three graphs for eahc KNN
fig8 = px.line(data, x='k', y='one_blocker',
blocker'})           range_x=[0, 101], range_y=[0.75, 0.9], labels={'k': 'k', 'one_blocker': 'Accuracy of KNN with one closest
st.write(fig8)

fig9 = px.line(data, x='k', y='two_blocker',
blockers'})          range_x=[0, 101], range_y=[0.75, 0.9], labels={'k': 'k', 'two_blocker': 'Accuracy of KNN with two closest
st.write(fig9)

fig10 = px.line(data, x='k', y='three_blocker',
closest blockers'})  range_x=[0, 101], range_y=[0.75, 0.9], labels={'k': 'k', 'three_blocker': 'Accuracy of KNN with three
st.write(fig10)

# Average pressure created vs the win percentage eight weeks
fig11 = px.scatter(sorted_df2, x='avg_pressure', y='record',
                     range_x=[0.115, 0.135], range_y=[0, 1], color='team',  labels={'avg_pressure': 'Average pressure created by
the pass rush', 'record': 'Winning percentage after 8 weeks', 'team': 'Team'})

st.write(fig11)

# Average pressure allowed vs the win percentage eight weeks
fig12 = px.scatter(sorted_df3, x='avg_pressure', y='record',
                     range_x=[0.11, 0.135], range_y=[0, 1], color='team', labels={'avg_pressure': 'Average pressure allowed by
the pass block', 'record': 'Winning percentage after 8 weeks', 'team': 'Team'})

st.write(fig12)

# Average pressure created vs the win percentage all weeks
fig13 = px.scatter(sorted_df2, x='avg_pressure', y='final_record',
                     range_x=[0.115, 0.135], range_y=[0, 1], color='team', labels={'avg_pressure': 'Average pressure created by
the pass rush', 'final_record': 'Winning percentage after 18 weeks', 'team': 'Team'})

st.write(fig13)

# Average pressure allowed vs the win percentage all weeks
fig14 = px.scatter(sorted_df3, x='avg_pressure', y='final_record',
                     range_x=[0.11, 0.135], range_y=[0, 1], color='team', labels={'avg_pressure': 'Average pressure allowed by
the pass block', 'final_record': 'Winning percentage after 18 weeks', 'team': 'Team'})
st.write(fig14)

# Predicted pressure over the course of one play for DeForest Buckner
fig15 = px.line(df7, x='frameId', y='predicted_pressure_probability',
line_shape='spline') range_x=[0, 47], range_y=[0, 1], labels={'frameId': 'Frames', 'predicted_pressure_probability': 'Pressure'},
st.write(fig15)

# Average pressure created by each position per team
fig16 = px.scatter(df8, x='team', y='avg_pressure', range_y=[0, 0.5], color='position', labels={'team': 'Team', 'avg_pressure':
'Average pressure created', 'position': 'position'})

st.write(fig16)

# Average pressure created by each team per position
fig17 = px.scatter(df9, x='position', y='avg_pressure', range_y=[0, 0.5], color='team', labels={'position': 'Position',
'avg_pressure': 'Average pressure created', 'team': 'Team'}, category_orders={'position': df9['position'].unique()})

st.write(fig17)

# Predicted pressure and PMP for one play of Harold Landry
fig18 = go.Figure()
fig18.add_trace(go.Scatter(x=landry['frameId'], y=landry['predicted_pressure_probability'],
                    mode='lines',
                    name='Pressure', line_shape='spline'))
```

```
fig18.add_trace(go.Scatter(x=landry['frameId'], y=landry['predicted_motion_predicted_pressure_probability_outcome'],
                    mode='lines', line_color='red',
                    name='Predicted Motion Pressure', line_shape='spline'))
fig18.update_layout(xaxis_title='Frames',
                   yaxis_title='Pressure')
st.write(fig18)

# Bar plot to show the distribution of the PMP delta across all pass rushers
fig19 = px.bar(df11g,
             x='delta_rounded', y='pmp_delta', labels={'delta_rounded': 'PMP difference rounded to the first decimal place',
'pmp_delta': 'Count'},
             barmode='stack')
st.write(fig19)

# Scatter plot to the distribution of average pressure created per play vs the PMP delta on that play
fig20 = px.scatter(df11,
             x='avg_pressure', y='pmp_delta',labels={'avg_pressure': 'Average pressure created', 'pmp_delta': 'PMP delta'})

st.write(fig20)
```

11.4 HTML - and JavaScript Code

```
<!DOCTYPE html>
<html>
    <head>
        <script src="https://d3js.org/d3.v7.min.js"></script>
        <style>
            /* Add your custom CSS styles here */
            .container {
                display:flex;
            }
            #graph-container-legend {
                width: 720px;
                height: 50px;
            }
            #graph-container {
                width: 720px;
                height: 320px;
                border: 5px solid black;
            }
            #text-container {
                width: 1500px;
                height: 320px;
            }
            #graph-container2-legend {
                width: 720px;
                height: 50px;
            }
            #graph-container2 {
                width: 720px;
                height: 320px;
                border: 5px solid black;
            }
            #graph-container4 {
                width: 320px;
                height: 320px;
                margin: 20px;
            }
            #text-container2 {
                width: 720px;
                height: 320px;
            }
            #graph-container3-legend {
                width: 1700px;
                height: 50px;
            }
            #graph-container3 {
                width: 720px;
                height: 320px;
                border: 5px solid black;
            }
            #graph-container5 {
                width: 320px;
                height: 320px;
                margin: 20px;
            }
            #graph-container6 {
                width: 320px;
                height: 320px;
                margin: 20px;
            }
            #text-container3 {
                width: 720px;
                height: 320px;
            }
            .dot {
                stroke: black;
                stroke-opacity: 1;
                fill-opacity: ß;
            }
        </style>
    </head>
    <body>
        <h1>Visualization of Machine-Learning-Predicted Pressure and Predicted Motion Pressure (PMP) for all Games in Week 8 of the 2021 NFL Season</h1>
        <p>Please allow up to 30 seconds for the data to load.</p>
        <p>This webpage was build entirely with HTML and JavaScript's D3 library, visualizing all games of week 8 of the 2021 NFL Season, with a main focus on the pass rush and pass block.</p>
        <p>Below, first the game, and then the desired play can be chosen.</p>
        <p>The first section will show a top-down animation of the play at 0.5 times the speed. The colors of the teams represent the colors of the jerseys worn for that game.</p>
         <p>The second section will show a top-down animation of the play at 0.5 times the speed, with the pass rushers pressure being indicated by the change of color of the individual circle. The more red, the higher the
pressure. The overall pressure throught the play is shown in the graph to the right of the animation.</p>
         <p>In the last section one of the pass rushers involved on the play can be chosen. The animation will focus on the selected player. To the right of the animation the graphs for the pressure, as well as the PMP can
be seen.</p>
        <p>An in-depth explanation of the generation of the predicted pressure and the PMP can be found at the bottom of this page. Should there be a delay between the graph and the animation, please refresh the page.</p>
        <select id="game-dropdown">
            <option value="">Select a game</option>
        </select>
        <select id="play-dropdown">
            <option value="">Select a play</option>
        </select>
        <h3>Full visualization of the play</h3>
        <div id="graph-container-legend"></div>
        <div class="container">
            <div id="graph-container"></div>
            <div id="text-container"></div>
        </div>
        <hr>
        <h3>Visualization of pressure on the play</h3>
        <div id="graph-container2-legend"></div>
        <div class="container">
            <div id="graph-container2"></div>
            <div id="graph-container4"></div>
            <div id="text-container2"></div>
        </div>
        <hr>
        <h3>Individual Player Analysis</h3>
        <select id="player-dropdown">
            <option value="">Select a player</option>
        </select>
        <div id="graph-container3-legend"></div>
        <div class="container">
            <div id="graph-container3"></div>
            <div id="graph-container6"></div>
            <div id="graph-container5"></div>
            <div id="text-container3"></div>
        </div>
        <hr>
        <h3>How the predicted pressure was created</h3>
           <p>Using a KNN machine learning model, telemetry data from every frame, of every play, for every player was evaluated on the probability on whether or not a sack, hit, or hurry would be the outcome of the
play.<br>This telemetry includes speed, acceleration, orientation of the rusher and blocker, as well as distances and angles to the passer and blocker from the rusher.<br>Due to the fact that whether or not a play would
end in a sack/hit/hurry was available for every play, every frame could be fed with the information of the end of the play, allowing for the KNN to predict the outcome of the play as the outcome variable.</p>
        <h3>How the Predicted Motion Pressure (PMP) was created</h3>
         <p>Similarly to the KNN above, the telemetry between the rusher, blocker and passer could be used to predict the next x - and y-coordinates, speed, acceleration, and orientation of a rusher.<br>New telemetry of the
rusher with that new position could then be used in the with the original KNN to predict the pressure in that new position, allwoing for a comparison between the actual pressure and the predicted one.</p>

        <script>
```

```
// Creating the layout of the first animation.
const svg = d3.select("#graph-container").append("svg").attr("width", 720).attr("height", 320)
const rect1 = svg.append("rect").attr("x",0).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
const rect2 = svg.append("rect").attr("x",60).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect3 = svg.append("rect").attr("x",120).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect4 = svg.append("rect").attr("x",180).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect5 = svg.append("rect").attr("x",240).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect6 = svg.append("rect").attr("x",300).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect7 = svg.append("rect").attr("x",360).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect8 = svg.append("rect").attr("x",420).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect9 = svg.append("rect").attr("x",480).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect10 = svg.append("rect").attr("x",540).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect11 = svg.append("rect").attr("x",600).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect12 = svg.append("rect").attr("x",660).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
const text1 = svg.append("text").attr("x",65).attr("y",30).attr("fill","white").text("G")
const text2 = svg.append("text").attr("x",65).attr("y",290).attr("fill","white").text("G")
const text3 = svg.append("text").attr("x",113).attr("y",30).attr("fill","white").text("10")
const text4 = svg.append("text").attr("x",113).attr("y",290).attr("fill","white").text("10")
const text5 = svg.append("text").attr("x",172).attr("y",30).attr("fill","white").text("20")
const text6 = svg.append("text").attr("x",172).attr("y",290).attr("fill","white").text("20")
const text7 = svg.append("text").attr("x",232).attr("y",30).attr("fill","white").text("30")
const text8 = svg.append("text").attr("x",232).attr("y",290).attr("fill","white").text("30")
const text9 = svg.append("text").attr("x",292).attr("y",30).attr("fill","white").text("40")
const text10 = svg.append("text").attr("x",292).attr("y",290).attr("fill","white").text("40")
const text11 = svg.append("text").attr("x",352).attr("y",30).attr("fill","white").text("50")
const text12 = svg.append("text").attr("x",352).attr("y",290).attr("fill","white").text("50")
const text13 = svg.append("text").attr("x",412).attr("y",30).attr("fill","white").text("40")
const text14 = svg.append("text").attr("x",412).attr("y",290).attr("fill","white").text("40")
const text15 = svg.append("text").attr("x",472).attr("y",30).attr("fill","white").text("30")
const text16 = svg.append("text").attr("x",472).attr("y",290).attr("fill","white").text("30")
const text17 = svg.append("text").attr("x",532).attr("y",30).attr("fill","white").text("20")
const text18 = svg.append("text").attr("x",532).attr("y",290).attr("fill","white").text("20")
const text19 = svg.append("text").attr("x",593).attr("y",30).attr("fill","white").text("10")
const text20 = svg.append("text").attr("x",593).attr("y",290).attr("fill","white").text("10")
const text21 = svg.append("text").attr("x",645).attr("y",30).attr("fill","white").text("G")
const text22 = svg.append("text").attr("x",645).attr("y",290).attr("fill","white").text("G")

// Creating the layout of the second animation.
const svg_2 = d3.select("#graph-container2").append("svg").attr("width", 720).attr("height", 320)
const rect1_2 = svg_2.append("rect").attr("x",0).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect2_2 = svg_2.append("rect").attr("x",60).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect3_2 = svg_2.append("rect").attr("x",120).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect4_2 = svg_2.append("rect").attr("x",180).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect5_2 = svg_2.append("rect").attr("x",240).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect6_2 = svg_2.append("rect").attr("x",300).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect7_2 = svg_2.append("rect").attr("x",360).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect8_2 = svg_2.append("rect").attr("x",420).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect9_2 = svg_2.append("rect").attr("x",480).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect10_2 = svg_2.append("rect").attr("x",540).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect11_2 = svg_2.append("rect").attr("x",600).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const rect12_2 = svg_2.append("rect").attr("x",660).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
const text1_2 = svg_2.append("text").attr("x",65).attr("y",30).attr("fill","black").text("G")
const text2_2 = svg_2.append("text").attr("x",65).attr("y",290).attr("fill","black").text("G")
const text3_2 = svg_2.append("text").attr("x",113).attr("y",30).attr("fill","black").text("10")
const text4_2 = svg_2.append("text").attr("x",113).attr("y",290).attr("fill","black").text("10")
const text5_2 = svg_2.append("text").attr("x",172).attr("y",30).attr("fill","black").text("20")
const text6_2 = svg_2.append("text").attr("x",172).attr("y",290).attr("fill","black").text("20")
const text7_2 = svg_2.append("text").attr("x",232).attr("y",30).attr("fill","black").text("30")
const text8_2 = svg_2.append("text").attr("x",232).attr("y",290).attr("fill","black").text("30")
const text9_2 = svg_2.append("text").attr("x",292).attr("y",30).attr("fill","black").text("40")
const text10_2 = svg_2.append("text").attr("x",292).attr("y",290).attr("fill","black").text("40")
const text11_2 = svg_2.append("text").attr("x",352).attr("y",30).attr("fill","black").text("50")
const text12_2 = svg_2.append("text").attr("x",352).attr("y",290).attr("fill","black").text("50")
const text13_2 = svg_2.append("text").attr("x",412).attr("y",30).attr("fill","black").text("40")
const text14_2 = svg_2.append("text").attr("x",412).attr("y",290).attr("fill","black").text("40")
const text15_2 = svg_2.append("text").attr("x",472).attr("y",30).attr("fill","black").text("30")
const text16_2 = svg_2.append("text").attr("x",472).attr("y",290).attr("fill","black").text("30")
const text17_2 = svg_2.append("text").attr("x",532).attr("y",30).attr("fill","black").text("20")
const text18_2 = svg_2.append("text").attr("x",532).attr("y",290).attr("fill","black").text("20")
const text19_2 = svg_2.append("text").attr("x",593).attr("y",30).attr("fill","black").text("10")
const text20_2 = svg_2.append("text").attr("x",593).attr("y",290).attr("fill","black").text("10")
const text21_2 = svg_2.append("text").attr("x",645).attr("y",30).attr("fill","black").text("G")
const text22_2 = svg_2.append("text").attr("x",645).attr("y",290).attr("fill","black").text("G")

// Creating the layout of the third animation.
const svg3 = d3.select("#graph-container3").append("svg").attr("width", 720).attr("height", 320)
const rect1_3 = svg3.append("rect").attr("x",0).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
const rect2_3 = svg3.append("rect").attr("x",60).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect3_3 = svg3.append("rect").attr("x",120).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect4_3 = svg3.append("rect").attr("x",180).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect5_3 = svg3.append("rect").attr("x",240).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect6_3 = svg3.append("rect").attr("x",300).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect7_3 = svg3.append("rect").attr("x",360).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect8_3 = svg3.append("rect").attr("x",420).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect9_3 = svg3.append("rect").attr("x",480).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect10_3 = svg3.append("rect").attr("x",540).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect11_3 = svg3.append("rect").attr("x",600).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
const rect12_3 = svg3.append("rect").attr("x",660).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
const text1_3 = svg3.append("text").attr("x",65).attr("y",30).attr("fill","white").text("G")
const text2_3 = svg3.append("text").attr("x",65).attr("y",290).attr("fill","white").text("G")
const text3_3 = svg3.append("text").attr("x",113).attr("y",30).attr("fill","white").text("10")
const text4_3 = svg3.append("text").attr("x",113).attr("y",290).attr("fill","white").text("10")
const text5_3 = svg3.append("text").attr("x",172).attr("y",30).attr("fill","white").text("20")
const text6_3 = svg3.append("text").attr("x",172).attr("y",290).attr("fill","white").text("20")
const text7_3 = svg3.append("text").attr("x",232).attr("y",30).attr("fill","white").text("30")
const text8_3 = svg3.append("text").attr("x",232).attr("y",290).attr("fill","white").text("30")
const text9_3 = svg3.append("text").attr("x",292).attr("y",30).attr("fill","white").text("40")
const text10_3 = svg3.append("text").attr("x",292).attr("y",290).attr("fill","white").text("40")
const text11_3 = svg3.append("text").attr("x",352).attr("y",30).attr("fill","white").text("50")
const text12_3 = svg3.append("text").attr("x",352).attr("y",290).attr("fill","white").text("50")
const text13_3 = svg3.append("text").attr("x",412).attr("y",30).attr("fill","white").text("40")
const text14_3 = svg3.append("text").attr("x",412).attr("y",290).attr("fill","white").text("40")
const text15_3 = svg3.append("text").attr("x",472).attr("y",30).attr("fill","white").text("30")
const text16_3 = svg3.append("text").attr("x",472).attr("y",290).attr("fill","white").text("30")
const text17_3 = svg3.append("text").attr("x",532).attr("y",30).attr("fill","white").text("20")
const text18_3 = svg3.append("text").attr("x",532).attr("y",290).attr("fill","white").text("20")
const text19_3 = svg3.append("text").attr("x",593).attr("y",30).attr("fill","white").text("10")
const text20_3 = svg3.append("text").attr("x",593).attr("y",290).attr("fill","white").text("10")
const text21_3 = svg3.append("text").attr("x",645).attr("y",30).attr("fill","white").text("G")
const text22_3 = svg3.append("text").attr("x",645).attr("y",290).attr("fill","white").text("G")

//  Loading in all necessary CSV files
d3.csv("/data/week8_all_data_and_names_and_opponents_and_game_names_final.csv").then(function(data) {
    d3.csv("/data/week8_total_pressure_defense_final.csv").then(function(data2) {
        d3.csv("/data/games_week8_html.csv").then(function(data_games) {
            d3.csv("/data/plays_with_names_html.csv").then(function(data_plays) {

                // Fill the game dropdown window with all the possile games from the data.
                //Based on the selection made, the play dropdown window is filled with the plays from that game.
                var games = Array.from(new Set(data.map(function(d) { return d.game_name; })))

                var gameDropdown = d3.select("#game-dropdown")
                gameDropdown.selectAll("option").remove();
                var defaultOption = gameDropdown.append("option")
                    .text("Select a game")
                    .attr("value", "");
                var gameOptions = gameDropdown.selectAll("option")
                    .data([""].concat(games))
                    .enter()
                    .append("option")
                    .text(function(d) { return d; })
                    .attr("value", function(d) { return d; })

                defaultOption.attr("selected", true)

                gameDropdown.on("change", function() {
                    var selectedGame = this.value
                    var selectedGamePlays = [...new Set(data.filter(function(d) { return d.game_name === selectedGame; })
                            .map(function(d) { return d.playId; }))];

                    var playDropdown = d3.select("#play-dropdown")
                    playDropdown.selectAll("option").remove()

                    playDropdown.append("option")
                        .text("Select a play")
                        .attr("value", "")

                    playDropdown.selectAll("option")
                        .data([""].concat(selectedGamePlays))
                        .enter()
                        .append("option")
                        .text(function(d) { return d; })
                        .attr("value", function(d) { return d; })
                })


                d3.select("#play-dropdown").on("change", function() {

                    //  Once the play is selected, reset all containers. This is for when a new play is selected.
                    d3.select('#graph-container-legend').html('')
                    d3.select('#graph-container').html('')
                    d3.select('#text-container').html('')
                    d3.select('#graph-container2-legend').html('')
                    d3.select('#graph-container2').html('')
                    d3.select('#graph-container3-legend').html('')
                    d3.select('#graph-container3').html('')
                    d3.select('#graph-container4').html('')
                    d3.select('#graph-container5').html('')
                    d3.select('#graph-container6').html('')
                    d3.select('#text-container2').html('')

                    // Specifying height and width of the containers.
                    const width = 720;
                    const width_text = 1500
                    const height = 320;

                    const width_legend = 720
                    const height_legend = 50

                    const width_graph = 320
                    const height_graph = 320

                    // Recreate the layouts of the animations.
                    const svg_legend = d3
                        .select("#graph-container-legend")
                        .append("svg")
                        .attr("width", width_legend)
                        .attr("height", height_legend)

                    const svg = d3
                        .select("#graph-container")
                        .append("svg")
                        .attr("width", width)
                        .attr("height", height)

                    const svg_text = d3
                        .select("#text-container")
                        .append("svg")
                        .attr("width", width_text)
                        .attr("height", height)

                    const svg_legend2 = d3
                        .select("#graph-container2-legend")
                        .append("svg")
                        .attr("width", width_legend)
                        .attr("height", height_legend)

                    const svg_2 = d3
                        .select("#graph-container2")
                        .append("svg")
                        .attr("width", width)
                        .attr("height", height)

                    const svg_text2 = d3
                        .select("#text-container2")
                        .append("svg")
                        .attr("width", width)
                        .attr("height", height)

                    const rect1 = svg.append("rect").attr("x",0).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
                    const rect2 = svg.append("rect").attr("x",60).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect3 = svg.append("rect").attr("x",120).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect4 = svg.append("rect").attr("x",180).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect5 = svg.append("rect").attr("x",240).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect6 = svg.append("rect").attr("x",300).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect7 = svg.append("rect").attr("x",360).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect8 = svg.append("rect").attr("x",420).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect9 = svg.append("rect").attr("x",480).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect10 = svg.append("rect").attr("x",540).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect11 = svg.append("rect").attr("x",600).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect12 = svg.append("rect").attr("x",660).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
                    const text1 = svg.append("text").attr("x",65).attr("y",30).attr("fill","white").text("G")
                    const text2 = svg.append("text").attr("x",65).attr("y",290).attr("fill","white").text("G")
                    const text3 = svg.append("text").attr("x",113).attr("y",30).attr("fill","white").text("10")
                    const text4 = svg.append("text").attr("x",113).attr("y",290).attr("fill","white").text("10")
                    const text5 = svg.append("text").attr("x",172).attr("y",30).attr("fill","white").text("20")
                    const text6 = svg.append("text").attr("x",172).attr("y",290).attr("fill","white").text("20")
                    const text7 = svg.append("text").attr("x",232).attr("y",30).attr("fill","white").text("30")
                    const text8 = svg.append("text").attr("x",232).attr("y",290).attr("fill","white").text("30")
                    const text9 = svg.append("text").attr("x",292).attr("y",30).attr("fill","white").text("40")
                    const text10 = svg.append("text").attr("x",292).attr("y",290).attr("fill","white").text("40")
                    const text11 = svg.append("text").attr("x",352).attr("y",30).attr("fill","white").text("50")
                    const text12 = svg.append("text").attr("x",352).attr("y",290).attr("fill","white").text("50")
                    const text13 = svg.append("text").attr("x",412).attr("y",30).attr("fill","white").text("40")
                    const text14 = svg.append("text").attr("x",412).attr("y",290).attr("fill","white").text("40")
                    const text15 = svg.append("text").attr("x",472).attr("y",30).attr("fill","white").text("30")
                    const text16 = svg.append("text").attr("x",472).attr("y",290).attr("fill","white").text("30")
                    const text17 = svg.append("text").attr("x",532).attr("y",30).attr("fill","white").text("20")
                    const text18 = svg.append("text").attr("x",532).attr("y",290).attr("fill","white").text("20")
                    const text19 = svg.append("text").attr("x",593).attr("y",30).attr("fill","white").text("10")
                    const text20 = svg.append("text").attr("x",593).attr("y",290).attr("fill","white").text("10")
                    const text21 = svg.append("text").attr("x",645).attr("y",30).attr("fill","white").text("G")
                    const text22 = svg.append("text").attr("x",645).attr("y",290).attr("fill","white").text("G")

                    const rect1_2 = svg_2.append("rect").attr("x",0).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect2_2 = svg_2.append("rect").attr("x",60).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect3_2 = svg_2.append("rect").attr("x",120).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect4_2 = svg_2.append("rect").attr("x",180).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect5_2 = svg_2.append("rect").attr("x",240).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect6_2 = svg_2.append("rect").attr("x",300).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect7_2 = svg_2.append("rect").attr("x",360).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect8_2 = svg_2.append("rect").attr("x",420).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect9_2 = svg_2.append("rect").attr("x",480).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect10_2 = svg_2.append("rect").attr("x",540).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect11_2 = svg_2.append("rect").attr("x",600).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const rect12_2 = svg_2.append("rect").attr("x",660).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "white").attr("stroke", "black")
                    const text1_2 = svg_2.append("text").attr("x",65).attr("y",30).attr("fill","black").text("G")
                    const text2_2 = svg_2.append("text").attr("x",65).attr("y",290).attr("fill","black").text("G")
                    const text3_2 = svg_2.append("text").attr("x",113).attr("y",30).attr("fill","black").text("10")
                    const text4_2 = svg_2.append("text").attr("x",113).attr("y",290).attr("fill","black").text("10")
                    const text5_2 = svg_2.append("text").attr("x",172).attr("y",30).attr("fill","black").text("20")
                    const text6_2 = svg_2.append("text").attr("x",172).attr("y",290).attr("fill","black").text("20")
                    const text7_2 = svg_2.append("text").attr("x",232).attr("y",30).attr("fill","black").text("30")
                    const text8_2 = svg_2.append("text").attr("x",232).attr("y",290).attr("fill","black").text("30")
                    const text9_2 = svg_2.append("text").attr("x",292).attr("y",30).attr("fill","black").text("40")
                    const text10_2 = svg_2.append("text").attr("x",292).attr("y",290).attr("fill","black").text("40")
                    const text11_2 = svg_2.append("text").attr("x",352).attr("y",30).attr("fill","black").text("50")
                    const text12_2 = svg_2.append("text").attr("x",352).attr("y",290).attr("fill","black").text("50")
                    const text13_2 = svg_2.append("text").attr("x",412).attr("y",30).attr("fill","black").text("40")
                    const text14_2 = svg_2.append("text").attr("x",412).attr("y",290).attr("fill","black").text("40")
                    const text15_2 = svg_2.append("text").attr("x",472).attr("y",30).attr("fill","black").text("30")
                    const text16_2 = svg_2.append("text").attr("x",472).attr("y",290).attr("fill","black").text("30")
                    const text17_2 = svg_2.append("text").attr("x",532).attr("y",30).attr("fill","black").text("20")
                    const text18_2 = svg_2.append("text").attr("x",532).attr("y",290).attr("fill","black").text("20")
                    const text19_2 = svg_2.append("text").attr("x",593).attr("y",30).attr("fill","black").text("10")
                    const text20_2 = svg_2.append("text").attr("x",593).attr("y",290).attr("fill","black").text("10")
                    const text21_2 = svg_2.append("text").attr("x",645).attr("y",30).attr("fill","black").text("G")
                    const text22_2 = svg_2.append("text").attr("x",645).attr("y",290).attr("fill","black").text("G")

                    const svg3 = d3.select("#graph-container3").append("svg").attr("width", 720).attr("height", 320)
                    const rect1_3 = svg3.append("rect").attr("x",0).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
                    const rect2_3 = svg3.append("rect").attr("x",60).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect3_3 = svg3.append("rect").attr("x",120).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect4_3 = svg3.append("rect").attr("x",180).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect5_3 = svg3.append("rect").attr("x",240).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect6_3 = svg3.append("rect").attr("x",300).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect7_3 = svg3.append("rect").attr("x",360).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect8_3 = svg3.append("rect").attr("x",420).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect9_3 = svg3.append("rect").attr("x",480).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect10_3 = svg3.append("rect").attr("x",540).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect11_3 = svg3.append("rect").attr("x",600).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                    const rect12_3 = svg3.append("rect").attr("x",660).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
                    const text1_3 = svg3.append("text").attr("x",65).attr("y",30).attr("fill","white").text("G")
                    const text2_3 = svg3.append("text").attr("x",65).attr("y",290).attr("fill","white").text("G")
                    const text3_3 = svg3.append("text").attr("x",113).attr("y",30).attr("fill","white").text("10")
                    const text4_3 = svg3.append("text").attr("x",113).attr("y",290).attr("fill","white").text("10")
                    const text5_3 = svg3.append("text").attr("x",172).attr("y",30).attr("fill","white").text("20")
                    const text6_3 = svg3.append("text").attr("x",172).attr("y",290).attr("fill","white").text("20")
                    const text7_3 = svg3.append("text").attr("x",232).attr("y",30).attr("fill","white").text("30")
                    const text8_3 = svg3.append("text").attr("x",232).attr("y",290).attr("fill","white").text("30")
                    const text9_3 = svg3.append("text").attr("x",292).attr("y",30).attr("fill","white").text("40")
                    const text10_3 = svg3.append("text").attr("x",292).attr("y",290).attr("fill","white").text("40")
                    const text11_3 = svg3.append("text").attr("x",352).attr("y",30).attr("fill","white").text("50")
                    const text12_3 = svg3.append("text").attr("x",352).attr("y",290).attr("fill","white").text("50")
                    const text13_3 = svg3.append("text").attr("x",412).attr("y",30).attr("fill","white").text("40")
                    const text14_3 = svg3.append("text").attr("x",412).attr("y",290).attr("fill","white").text("40")
                    const text15_3 = svg3.append("text").attr("x",472).attr("y",30).attr("fill","white").text("30")
                    const text16_3 = svg3.append("text").attr("x",472).attr("y",290).attr("fill","white").text("30")
                    const text17_3 = svg3.append("text").attr("x",532).attr("y",30).attr("fill","white").text("20")
                    const text18_3 = svg3.append("text").attr("x",532).attr("y",290).attr("fill","white").text("20")
                    const text19_3 = svg3.append("text").attr("x",593).attr("y",30).attr("fill","white").text("10")
                    const text20_3 = svg3.append("text").attr("x",593).attr("y",290).attr("fill","white").text("10")
                    const text21_3 = svg3.append("text").attr("x",645).attr("y",30).attr("fill","white").text("G")
                    const text22_3 = svg3.append("text").attr("x",645).attr("y",290).attr("fill","white").text("G")


                    // Defining the xScale and the yScale for the animation of the play.
                    // Scaling linear to fit the height and the width of the play.
                    const xScale = d3.scaleLinear()
                        .domain([0, 120])
                        .range([0, width]);

                    const yScale = d3.scaleLinear()
                        .domain([0, 160/3])
                        .range([0, height]);

                    // Defining the duration and delay of the animation, making it at 0.5 times the actual speed, since on frame represents 1/10th of a second.
                    const duration = 200
                    const delay = 200

                    let frameIndex = 0  // frameIndex to keep track of the frame during the animation

                    // Matching the offensive team with the jersey they wore during the actual game for the first animation
                    function getColor(d) {
                        if (d.team === "ATL") {return "black"}
                        if (d.team === "CAR") {return "white"}
                        if (d.team === "ARI") {return "red"}
                        if (d.team === "GB") {return "green"}
                        if (d.team === "DET") {return "blue"}
                        if (d.team === "PHI") {return "white"}
                        if (d.team === "SEA") {return "navy"}
                        if (d.team === "JAX") {return "white"}
                        if (d.team === "CLE") {return "orange"}
                        if (d.team === "PIT") {return "white"}
                        if (d.team === "IND") {return "blue"}
                        if (d.team === "TEN") {return "white"}
                        if (d.team === "NYJ") {return "black"}
                        if (d.team === "CIN") {return "white"}
                        if (d.team === "HOU") {return "black"}
                        if (d.team === "LA") {return "white"}
                        if (d.team === "NO") {return "white"}
                        if (d.team === "TB") {return "red"}
                        if (d.team === "DEN") {return "orange"}
                        if (d.team === "WAS") {return "white"}
                        if (d.team === "LAC") {return "navy"}
                        if (d.team === "NE") {return "white"}
                        if (d.team === "BUF") {return "white"}
                        if (d.team === "MIA") {return "turquoise"}
                        if (d.team === "CHI") {return "navy"}
                        if (d.team === "SF") {return "white"}
                        if (d.team === "MIN") {return "purple"}
                        if (d.team === "DAL") {return "white"}
                        if (d.team === "KC") {return "red"}
                        if (d.team === "NYG") {return "white"}
                        if (d.team === "football") {return "brown"}
                    }

                    // Matching the defensive team with the jersey they wore during the actual game for the first animation
                    function getOpponent(d) {
                        if (d.opponent === "ATL") {return "black"}
                        if (d.opponent === "CAR") {return "white"}
                        if (d.opponent === "ARI") {return "red"}
                        if (d.opponent === "GB") {return "green"}
                        if (d.opponent === "DET") {return "blue"}
                        if (d.opponent === "PHI") {return "white"}
                        if (d.opponent === "SEA") {return "navy"}
                        if (d.opponent === "JAX") {return "white"}
                        if (d.opponent === "CLE") {return "orange"}
                        if (d.opponent === "PIT") {return "white"}
                        if (d.opponent === "IND") {return "blue"}
                        if (d.opponent === "TEN") {return "white"}
                        if (d.opponent === "NYJ") {return "black"}
                        if (d.opponent === "CIN") {return "white"}
                        if (d.opponent === "HOU") {return "black"}
                        if (d.opponent === "LA") {return "white"}
                        if (d.opponent === "NO") {return "white"}
                        if (d.opponent === "TB") {return "red"}
                        if (d.opponent === "DEN") {return "orange"}
                        if (d.opponent === "WAS") {return "white"}
                        if (d.opponent === "LAC") {return "navy"}
                        if (d.opponent === "NE") {return "white"}
                        if (d.opponent === "BUF") {return "white"}
                        if (d.opponent === "MIA") {return "turquoise"}
                        if (d.opponent === "CHI") {return "navy"}
                        if (d.opponent === "SF") {return "white"}
                        if (d.opponent === "MIN") {return "purple"}
                        if (d.opponent === "DAL") {return "white"}
                        if (d.opponent === "KC") {return "red"}
                        if (d.opponent === "NYG") {return "white"}
                    }

                    // Function to get match the specific roles of the player to a color for the second animation
                    function getColor2(d) {
                        if (d.role === "Pass Rush") {return "red"}
                        if (d.role === "Coverage") {return "white"}
                        if (d.role === "Pass") {return "blue"}
                        if (d.role === "Pass Route") {return "lightgreen"}
                        if (d.role === "Pass Block") {return "green"}
                        else {return "black"}
                    }

                    var selectedGame = d3.select("#game-dropdown").property("value")    //  finding the selected game
                    var selectedPlay = this.value                                       //  finding the selected play


                    // Filtering the data by selected game and play
                    var playData = data.filter(function(d) {
                        return d.game_name === selectedGame && d.playId === selectedPlay
                    })

                    // Filtering for the first index of the play data to get the information on the home and away team
                    var firstIndexData = playData.filter(function(d) {
                        return d.frameId == 1
                    })

                    // Filtering the data of the first index only for the Quarterback
                    var firstIndexDataQB = firstIndexData.filter(function(d) {
                        return d.role === "Pass"
                    })

                    // Filtering the second data frame by selected game and play
                    var playData2 = data2.filter(function(d) {
                        return d.game_name === selectedGame && d.playId === selectedPlay
                    })

                    // This was used later on as a simpler tool of finding the names of the team
                    var teams = data_games.filter(function(d) {
                        return d.game_name === selectedGame
                    })

                    // This filtered the original plays.csv data to determine the score on the play
                    var teams_plays = data_plays.filter(function(d) {
                        return d.game_name === selectedGame && d.playId === selectedPlay
                    })

                    // Determinig the maximum frames to help set the x-axis, the maximum pressure on all plays to help set the y-axis,
                    // and the maximum and average pressure on the selected play by the entire team.
                    maxFrames = d3.max(playData2, function(d) { return parseInt(d.frameId); })
                    maxPressure = d3.max(data2, function(d) { return parseFloat(d.total_pressure); })
                    maxPressure_on_play = d3.max(playData2, function(d) { return parseFloat(d.total_pressure); })
                    meanPressure_on_play = d3.mean(playData2, function(d) { return parseFloat(d.total_pressure); })

                    // Setting up the first graph
                    var xScale_graph = d3.scaleLinear()
                                            .domain([1, maxFrames])
                                            .range([40, width_graph])

                    var yScale_graph = d3.scaleLinear()
                                            .domain([0, maxPressure])
                                            .range([height_graph-40, 0])

                    var svg_graph = d3.select('#graph-container4')
                                        .append("svg")
                                        .attr("width", width_graph)
                                        .attr("height", height_graph)

                    var path = svg_graph.append("path")
                                            .attr("class", "line")
                                            .style("fill", "none")
                                            .style("stroke", "steelblue")
                                            .style("stroke-width", "2")

                    var xAxis = d3.axisBottom(xScale_graph)

                    svg_graph.append("g")
                                .attr("class", "x-axis")
                                .attr("transform", "translate(0," + (height_graph - 40) + ")")
                                .call(xAxis)

                    var yAxis = d3.axisLeft(yScale_graph).ticks(maxPressure)

                    svg_graph.append("g")
                        .attr("class", "y-axis")
                        .attr("transform", "translate(40,0)")
                        .call(yAxis)

                    svg_graph.append("text")
                            .attr("class", "y-axis-label")
                            .attr("transform", "rotate(-90)")
                            .attr("x", -height_graph / 2 + 20)
                            .attr("y", 15)
                            .style("text-anchor", "middle")
                            .text("Pressure");

                    svg_graph.append("text")
                            .attr("class", "x-axis-label")
                            .attr("x", width_graph / 2 + 20)
                            .attr("y", height_graph)
                            .style("text-anchor", "middle")
                            .text("Frame");

                    var currentIndex = 0    // keeping track of the current index

                    var line = d3.line()
                                    .x(function(d) { return xScale_graph(d.frameId); })
                                    .y(function(d) { return yScale_graph(d.total_pressure); })
                                    .curve(d3.curveCardinal)

                    // Creating a line for the pressure of the graph, revealing it frame by frame and smooting it out
                    // The graph will run parallel to the second animation for better visualization.
                    // They will eventually run out of sync, the page just has to be refreshed then.
                    function updateLine() {
                        var currentFrameData = playData2.slice(0, currentIndex + 1)
                        var holdPeriodDuration = delay*2.5

                        path.datum(currentFrameData)
                            .attr("d", line
                            )
                        currentIndex++

                        if (currentIndex >= playData2.length) {
                            setTimeout(function(){
                                currentIndex = 0
                                path.attr("d", null)
                                updateLine()
                            }, holdPeriodDuration)
                        }
                        else {
                            setTimeout(updateLine, delay)
                        }
                    }

                    setTimeout(updateLine, delay)

                    // Filling the container next to the first animation with useful information, such a score on the play, personnel by the offense and defense and the coverage type.
                    const team_text0 = svg_text.append("text").data(teams_plays).attr("x",35).attr("y",30).attr("fill","black").text(d => d.playDescription)
                    const team_text1 = svg_text.append("text").attr("x",35).attr("y",60).attr("fill","black").text("Current Score: ")
                    const team_text2 = svg_text.append("text").data(teams).attr("x",135).attr("y",60).attr("fill","black").text(d => d.homeTeamAbbr)
                    const team_text3 = svg_text.append("text").data(teams_plays).attr("x",175).attr("y",60).attr("fill","black").text(d => d.preSnapHomeScore)
                    const team_text4 = svg_text.append("text").attr("x",200).attr("y",60).attr("fill","black").text("-")
                    const team_text5 = svg_text.append("text").data(teams_plays).attr("x",215).attr("y",60).attr("fill","black").text(d => d.preSnapVisitorScore)
                    const team_text6 = svg_text.append("text").data(teams).attr("x",240).attr("y",60).attr("fill","black").text(d => d.visitorTeamAbbr)
                    const team_text7 = svg_text.append("text").attr("x",35).attr("y",90).attr("fill","black").text("Offensive Personnel: ")
                    const team_text8 = svg_text.append("text").data(teams_plays).attr("x",180).attr("y",90).attr("fill","black").text(d => d.personnelO)
                    const team_text9 = svg_text.append("text").attr("x",35).attr("y",120).attr("fill","black").text("Defensive Personnel: ")
                    const team_text10 = svg_text.append("text").data(teams_plays).attr("x",180).attr("y",120).attr("fill","black").text(d => d.personnelD)
                    const team_text11 = svg_text.append("text").attr("x",35).attr("y",150).attr("fill","black").text("Defensive Coverage: ")
                    const team_text12 = svg_text.append("text").data(teams_plays).attr("x",180).attr("y",150).attr("fill","black").text(d => d.pff_passCoverage)

                    // Filling the container next to the second animation and first pressure graph with information on the maximum and average pressure of the defense.
                    const pressure_text0 = svg_text2.append("text").attr("x",35).attr("y",240).attr("fill","black").text("Maximum pressure by the defense: ")
                    const pressure_text1 = svg_text2.append("text").attr("x",285).attr("y",240).attr("fill","black").text(maxPressure_on_play)
                    const pressure_text2 = svg_text2.append("text").attr("x",35).attr("y",270).attr("fill","black").text("Average pressure by the defense: ")
                    const pressure_text3 = svg_text2.append("text").attr("x",285).attr("y",270).attr("fill","black").text(meanPressure_on_play)

                    // Adding the team names to the end zone of the first graph
                        const endzone_text = svg.append("text").data(firstIndexDataQB).attr("x",650).attr("y",175).attr("fill","white").text(d => d.opponent).attr("transform","rotate(90, 690, 160)").attr("font-
                    size", "40px")

                       const endzone_text2 = svg.append("text").data(firstIndexDataQB).attr("x",0).attr("y",175).attr("fill","white").text(d => d.team).attr("transform","rotate(270, 30, 160)").attr("font-size",

                    "40px")


                    // Adding a legend to the first animation with the appropriate team names. Removing all circles beforehand in case of a different play being loaded.
                    svg_legend.selectAll("circle").remove();
                    const circle_svg_legend = svg_legend.append("circle").data(firstIndexDataQB).attr("cx",25).attr("cy",25).attr("r",5).attr("fill",getColor).attr("stroke", "black").attr("stroke-width", 1)
                    const off_team_svg_legend = svg_legend.append("text").data(firstIndexDataQB).attr("x",35).attr("y",30).attr("fill","black").text(d => d.team)
                      const circle_opponent_svg_legend = svg_legend.append("circle").data(firstIndexDataQB).attr("cx",85).attr("cy",25).attr("r",5).attr("fill",getOpponent).attr("stroke", "black").attr("stroke-
                    width", 1)

                    const def_team_svg_legend = svg_legend.append("text").data(firstIndexDataQB).attr("x",95).attr("y",30).attr("fill","black").text(d => d.opponent)
                         const circle_football_svg_legend = svg_legend.append("circle").data(firstIndexDataQB).attr("cx",145).attr("cy",25).attr("r",5).attr("fill","brown").attr("stroke", "black").attr("stroke-
                    width", 1)

                    const football_svg_legend = svg_legend.append("text").data(firstIndexDataQB).attr("x",155).attr("y",30).attr("fill","black").text("Football")

                    // Adding a legend to the second animation with the appropriate role names. Removing all circles beforehand in case of a different play being loaded.
                    svg_legend2.selectAll("circle").remove();
                    const circle_qb = svg_legend2.append("circle").attr("cx",25).attr("cy",25).attr("r",5).attr("fill","blue").attr("stroke", "black").attr("stroke-width", 1)
                    const qb_text = svg_legend2.append("text").attr("x",35).attr("y",30).attr("fill","black").text("QB")
                    const circle_receiver = svg_legend2.append("circle").attr("cx",85).attr("cy",25).attr("r",5).attr("fill","lightgreen").attr("stroke", "black").attr("stroke-width", 1)
                    const receiver_text = svg_legend2.append("text").attr("x",95).attr("y",30).attr("fill","black").text("WR")
                    const circle_ol = svg_legend2.append("circle").attr("cx",145).attr("cy",25).attr("r",5).attr("fill","green").attr("stroke", "black").attr("stroke-width", 1)
                    const ol_text = svg_legend2.append("text").attr("x",155).attr("y",30).attr("fill","black").text("OL")
                    const circle_cov = svg_legend2.append("circle").attr("cx",205).attr("cy",25).attr("r",5).attr("fill","white").attr("stroke", "black").attr("stroke-width", 1)
                    const cov_text = svg_legend2.append("text").attr("x",215).attr("y",30).attr("fill","black").text("Cov")
                    const circle_rush = svg_legend2.append("circle").attr("cx",265).attr("cy",25).attr("r",5).attr("fill","red").attr("stroke", "black").attr("stroke-width", 1)
                    const rush_text = svg_legend2.append("text").attr("x",275).attr("y",30).attr("fill","black").text("Pass Rush (opacity changes based on pressure created by player)")

                    // Function for both the first, and the second animation.
                    // Animates the play from the top-down view frame by frame. Circles have to be deleted after every frame, if the frame index reaches its maximum value, the animation resets.
                    function animateFrames() {
                        const frameData = playData.filter(d => parseInt(d.frameId) === frameIndex)

                        svg.selectAll("circle").remove();

                        svg.selectAll("circle")
                            .data(frameData)
                            .enter()
                            .append("circle")
                            .attr("r", 5)
                            .attr("cx", d => xScale(parseFloat(d.x)))
                            .attr("cy", d => yScale(parseFloat(d.y)))
                            .attr("fill", getColor)
                            .attr("stroke", "black")
                            .attr("stroke-width", 1)
                            .transition()
                            .duration(duration)
                            .attrTween("cx", function (d) {
                                const interpolateX = d3.interpolate(this.getAttribute("cx"), xScale(parseFloat(d.x)))
                                return function (t) {
                                    return interpolateX(t)
                                }
                            })
                            .attrTween("cy", function (d) {
                                const interpolateY = d3.interpolate(this.getAttribute("cy"), yScale(parseFloat(d.y)));
                                return function (t) {
                                    return interpolateY(t)
                                }
                            })

                        svg_2.selectAll("circle").remove();

                        const dots = svg_2.selectAll("circle")
                            .data(frameData)
                            .enter()
                            .append("circle")
                            .attr("r", 5)
                            .attr("cx", d => xScale(parseFloat(d.x)))
                            .attr("cy", d => yScale(parseFloat(d.y)))
                            .attr("fill", getColor2)
                            .attr("stroke", "black")
                            .attr("stroke-opacity", 1)
                            .attr("stroke-width", 1)
                            .style("fill-opacity", d => d.predicted_pressure)

                            .transition()
                            .duration(duration)
                            .attrTween("cx", function (d) {
                                const interpolateX = d3.interpolate(this.getAttribute("cx"), xScale(parseFloat(d.x)))
                                return function (t) {
                                    return interpolateX(t)
                                }
                            })
                            .attrTween("cy", function (d) {
                                const interpolateY = d3.interpolate(this.getAttribute("cy"), yScale(parseFloat(d.y)));
                                return function (t) {
                                    return interpolateY(t)
                                }
                            })

                        frameIndex++

                        if (frameIndex > d3.max(playData, d => parseInt(d.frameId))) {
                            frameIndex = 0
                        }

                        setTimeout(animateFrames, delay);
                    }


                    setTimeout(animateFrames, delay)

                    // Creating the drop down window for the player selection and filling it with all the pass rushers on the play.
                    var selectedGame = this.value
                    var selectedPlayPlayers = [...new Set(playData.filter(function(d) { return d.playId === selectedPlay && d.role === "Pass Rush"; })
                            .map(function(d) { return d.displayName; }))];

                    var playerDropdown = d3.select("#player-dropdown")
                    playerDropdown.selectAll("option").remove()

                    playerDropdown.append("option")
                        .text("Select a Pass Rusher")
                        .attr("value", "")

                    playerDropdown.selectAll("option")
                        .data([""].concat(selectedPlayPlayers))
                        .enter()
                        .append("option")
                        .text(function(d) { return d; })
                        .attr("value", function(d) { return d; })

                    // Once the player is selected, same procedure as for the first two animations: delete the contents of the containers in case of a new play being selected and define all necessary variables.
                    d3.select("#player-dropdown").on("change", function() {
                        d3.select('#graph-container3-legend').html('')
                        d3.select('#graph-container3').html('')
                        d3.select('#graph-container5').html('')
                        d3.select('#graph-container6').html('')
                        d3.select('#text-container3').html('')

                        const width = 720;
                        const height = 320;

                        const width_legend = 1700
                        const height_legend = 50

                        const width_graph = 320
                        const height_grpah = 320

                        const svg_legend3 = d3
                            .select("#graph-container3-legend")
                            .append("svg")
                            .attr("width", width_legend)
                            .attr("height", height_legend)

                        const svg3 = d3
                            .select("#graph-container3")
                            .append("svg")
                            .attr("width", width)
                            .attr("height", height)

                        const svg_text3 = d3
                            .select("#text-container3")
                            .append("svg")
                            .attr("width", width)
                            .attr("height", height)

                        const rect1 = svg3.append("rect").attr("x",0).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
                        const rect2 = svg3.append("rect").attr("x",60).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect3 = svg3.append("rect").attr("x",120).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect4 = svg3.append("rect").attr("x",180).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect5 = svg3.append("rect").attr("x",240).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect6 = svg3.append("rect").attr("x",300).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect7 = svg3.append("rect").attr("x",360).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect8 = svg3.append("rect").attr("x",420).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect9 = svg3.append("rect").attr("x",480).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect10 = svg3.append("rect").attr("x",540).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect11 = svg3.append("rect").attr("x",600).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "green").attr("stroke", "black")
                        const rect12 = svg3.append("rect").attr("x",660).attr("y", 0).attr("width", 60).attr("height", 320).attr("fill", "lightgreen").attr("stroke", "black")
                        const text1 = svg3.append("text").attr("x",65).attr("y",30).attr("fill","white").text("G")
                        const text2 = svg3.append("text").attr("x",65).attr("y",290).attr("fill","white").text("G")
                        const text3 = svg3.append("text").attr("x",113).attr("y",30).attr("fill","white").text("10")
                        const text4 = svg3.append("text").attr("x",113).attr("y",290).attr("fill","white").text("10")
                        const text5 = svg3.append("text").attr("x",172).attr("y",30).attr("fill","white").text("20")
                        const text6 = svg3.append("text").attr("x",172).attr("y",290).attr("fill","white").text("20")
                        const text7 = svg3.append("text").attr("x",232).attr("y",30).attr("fill","white").text("30")
                        const text8 = svg3.append("text").attr("x",232).attr("y",290).attr("fill","white").text("30")
                        const text9 = svg3.append("text").attr("x",292).attr("y",30).attr("fill","white").text("40")
                        const text10 = svg3.append("text").attr("x",292).attr("y",290).attr("fill","white").text("40")
                        const text11 = svg3.append("text").attr("x",352).attr("y",30).attr("fill","white").text("50")
                        const text12 = svg3.append("text").attr("x",352).attr("y",290).attr("fill","white").text("50")
                        const text13 = svg3.append("text").attr("x",412).attr("y",30).attr("fill","white").text("40")
                        const text14 = svg3.append("text").attr("x",412).attr("y",290).attr("fill","white").text("40")
                        const text15 = svg3.append("text").attr("x",472).attr("y",30).attr("fill","white").text("30")
                        const text16 = svg3.append("text").attr("x",472).attr("y",290).attr("fill","white").text("30")
                        const text17 = svg3.append("text").attr("x",532).attr("y",30).attr("fill","white").text("20")
                        const text18 = svg3.append("text").attr("x",532).attr("y",290).attr("fill","white").text("20")
                        const text19 = svg3.append("text").attr("x",593).attr("y",30).attr("fill","white").text("10")
                        const text20 = svg3.append("text").attr("x",593).attr("y",290).attr("fill","white").text("10")
                        const text21 = svg3.append("text").attr("x",645).attr("y",30).attr("fill","white").text("G")
                        const text22 = svg3.append("text").attr("x",645).attr("y",290).attr("fill","white").text("G")

                        const xScale = d3.scaleLinear()
                            .domain([0, 120])
                            .range([0, width]);

                        const yScale = d3.scaleLinear()
                            .domain([0, 160/3])
                            .range([0, height]);

                        const duration = 200
                        const delay = 200

                        let frameIndex = 0

                        var selectedGame = d3.select("#game-dropdown").property("value")
                        var selectedPlay = d3.select("#play-dropdown").property("value")
                        var selectedPlayer = this.value // finding the selected player
                        var playData = data.filter(function(d) {
                            return d.game_name === selectedGame && d.playId === selectedPlay
                        })

                        var firstIndexData = playData.filter(function(d) {
                            return d.frameId == 1
                        })

                        var firstIndexDataQB = firstIndexData.filter(function(d) {
                            return d.role === "Pass"
                        })

                        // Finding the name of the selected player so it can be used later on.
                        var playerData = playData.filter(function(d) {
                            return d.displayName === selectedPlayer
                        })

                        // Adding the names of the teams to the endzone
                                         const endzone_text = svg3.append("text").data(firstIndexDataQB).attr("x",650).attr("y",175).attr("fill","white").text(d => d.opponent).attr("transform","rotate(90, 690,
                        160)").attr("font-size", "40px")
                            const endzone_text2 = svg3.append("text").data(firstIndexDataQB).attr("x",0).attr("y",175).attr("fill","white").text(d => d.team).attr("transform","rotate(270, 30, 160)").attr("font-
                        size", "40px")

                        // Similar to before, finding the maximum amount of frames for the x-axis
                        // Also finding the maximum and average pressure on the play by the player
                        // Finding the delta between the predicted pressure and the predicted motion pressure by taking the sum of all values of both and subtracting from one another.
                        maxFrames = d3.max(playerData, function(d) { return parseInt(d.frameId); })
                        maxPressure_on_play = d3.max(playerData, function(d) { return parseFloat(d.predicted_pressure); })
                        meanPressure_on_play = d3.mean(playerData, function(d) { return parseFloat(d.predicted_pressure); })
                        sumPressure_on_play = d3.sum(playerData, function(d) { return parseFloat(d.predicted_pressure); })
                        sumPMP_on_play = d3.sum(playerData, function(d) { return parseFloat(d.predicted_motion_pressure); })
                        delta = sumPressure_on_play - sumPMP_on_play

                        // Filling the container to the right of the third animation and the two graphs with information on the maximum and average pressure on the play, as well as the delta between pressure and
                             PMP.
                        const pressure_text0 = svg_text3.append("text").attr("x",35).attr("y",210).attr("fill","black").text("Maximum pressure by the player: ")
                        const pressure_text1 = svg_text3.append("text").attr("x",285).attr("y",210).attr("fill","black").text(maxPressure_on_play)
                        const pressure_text2 = svg_text3.append("text").attr("x",35).attr("y",240).attr("fill","black").text("Average pressure by the player: ")
                        const pressure_text3 = svg_text3.append("text").attr("x",285).attr("y",240).attr("fill","black").text(meanPressure_on_play)
                        const pressure_text4 = svg_text3.append("text").attr("x",35).attr("y",270).attr("fill","black").text("Delta between pressure and PMP: ")
                        const pressure_text5 = svg_text3.append("text").attr("x",285).attr("y",270).attr("fill","black").text(delta)

                        // Creating two animated graphs: one to show the pressure per frame along the play, one to show the pressure and the PMP per frame along the play.
                        // Setup is identical to the first graph above.
                        var xScale_graph = d3.scaleLinear()
                                            .domain([1, maxFrames])
                                            .range([40, width_graph])

                        var yScale_graph = d3.scaleLinear()
                                                .domain([0, 1.05])
                                                .range([height_graph-40, 0])

                        var svg_graph3 = d3.select('#graph-container6')
                                            .append("svg")
                                            .attr("width", width_graph)
                                            .attr("height", height_graph)

                        var path3 = svg_graph3.append("path")
                                                .attr("class", "line3")
                                                .style("fill", "none")
                                                .style("stroke", "steelblue")
                                                .style("stroke-width", "2")

                        var xAxis = d3.axisBottom(xScale_graph)

                        svg_graph3.append("g")
                                    .attr("class", "x-axis")
                                    .attr("transform", "translate(0," + (height_graph - 40) + ")")
                                    .call(xAxis)

                        var yAxis = d3.axisLeft(yScale_graph).ticks(10)

                        svg_graph3.append("g")
                            .attr("class", "y-axis")
                            .attr("transform", "translate(40,0)")
                            .call(yAxis)

                        svg_graph3.append("text")
                                .attr("class", "y-axis-label")
                                .attr("transform", "rotate(-90)")
                                .attr("x", -height_graph / 2 + 20)
                                .attr("y", 15)
                                .style("text-anchor", "middle")
                                .text("Pressure");

                        svg_graph3.append("text")
                                .attr("class", "x-axis-label")
                                .attr("x", width_graph / 2 + 20)
                                .attr("y", height_graph)
                                .style("text-anchor", "middle")
                                .text("Frame");

                        var currentIndex2 = 0

                        var line3 = d3.line()
                                        .x(function(d) { return xScale_graph(d.frameId); })
                                        .y(function(d) { return yScale_graph(d.predicted_pressure); })
                                        .curve(d3.curveCardinal)

                        // Animating both graphs the same way as before.
                        function updateLine2() {
                            var currentFrameData = playerData.slice(0, currentIndex2 + 1)
                            var holdPeriodDuration = delay*2.5

                            path3.datum(currentFrameData)
                                .attr("d", line3
                                )
                                currentIndex2++

                            if (currentIndex2 >= playerData.length) {
                                setTimeout(function(){
                                    currentIndex2 = 0
                                    path3.attr("d", null)
                                    updateLine2()
                                }, holdPeriodDuration)
                            }
                            else {
                                setTimeout(updateLine2, delay)
                            }
                        }

                        setTimeout(updateLine2, delay)


                        var svg_graph = d3.select('#graph-container5')
                                            .append("svg")
                                            .attr("width", width_graph)
                                            .attr("height", height_graph)
```

```
var path = svg_graph.append("path")
                        .attr("class", "line")
                        .style("fill", "none")
                        .style("stroke", "steelblue")
                        .style("stroke-width", "2")

var path2 = svg_graph.append("path")
                        .attr("class", "line2")
                        .style("fill", "none")
                        .style("stroke", "red")
                        .style("stroke-width", "1")


svg_graph.append("g")
            .attr("class", "x-axis")
            .attr("transform", "translate(0," + (height_graph - 40) + ")")
            .call(xAxis)

svg_graph.append("g")
    .attr("class", "y-axis")
    .attr("transform", "translate(40,0)")
    .call(yAxis)

svg_graph.append("text")
        .attr("class", "y-axis-label")
        .attr("transform", "rotate(-90)")
        .attr("x", -height_graph / 2 + 20)
        .attr("y", 15)
        .style("text-anchor", "middle")
        .text("Pressure");

svg_graph.append("text")
        .attr("class", "x-axis-label")
        .attr("x", width_graph / 2 + 20)
        .attr("y", height_graph)
        .style("text-anchor", "middle")
        .text("Frame");

var currentIndex = 0

var line = d3.line()
                .x(function(d) { return xScale_graph(d.frameId); })
                .y(function(d) { return yScale_graph(d.predicted_pressure); })
                .curve(d3.curveCardinal)

var line2 = d3.line()
                .x(function(d) { return xScale_graph(d.frameId); })
                .y(function(d) { return yScale_graph(d.predicted_motion_pressure); })
                .curve(d3.curveCardinal)


function updateLine() {
    var currentFrameData = playerData.slice(0, currentIndex + 1)
    var currentFrameData2 = playerData.slice(0, currentIndex + 1)
    var holdPeriodDuration = delay*2.5

    path.datum(currentFrameData)
        .attr("d", line
        )
    path2.datum(currentFrameData2)
        .attr("d", line2)
    currentIndex++

    if (currentIndex >= playerData.length) {
        setTimeout(function(){
            currentIndex = 0
            path.attr("d", null)
            path2.attr("d", null)
            updateLine()
        }, holdPeriodDuration)
    }
    else {
        setTimeout(updateLine, delay)
    }
}

setTimeout(updateLine, delay)

// Function to match the role of the players to a color for the legend.
function getColor(d) {
    if (d.displayName === selectedPlayer) {return "red"}
    if (d.role === "Pass Rush") {return "white"}
    if (d.role === "Coverage") {return "white"}
    if (d.role === "Pass") {return "black"}
    if (d.role === "Pass Route") {return "black"}
    if (d.role === "Pass Block") {return "black"}
    else {return "blue"}
}

// Adding the legend
svg_legend3.selectAll("circle").remove();
const circle_offense = svg_legend3.append("circle").attr("cx",25).attr("cy",25).attr("r",5).attr("fill","black").attr("stroke", "black").attr("stroke-width", 1)
const offense_text = svg_legend3.append("text").attr("x",35).attr("y",30).attr("fill","black").text("Offense")
const circle_defense = svg_legend3.append("circle").attr("cx",125).attr("cy",25).attr("r",5).attr("fill","white").attr("stroke", "black").attr("stroke-width", 1)
const defense_text = svg_legend3.append("text").attr("x",135).attr("y",30).attr("fill","black").text("Defense")
const circle_football = svg_legend3.append("circle").attr("cx",225).attr("cy",25).attr("r",5).attr("fill","blue").attr("stroke", "black").attr("stroke-width", 1)
const football_text = svg_legend3.append("text").attr("x",235).attr("y",30).attr("fill","black").text("Football")
const circle_player = svg_legend3.append("circle").attr("cx",325).attr("cy",25).attr("r",5).attr("fill","red").attr("stroke", "black").attr("stroke-width", 1)
const player_text = svg_legend3.append("text").attr("x",335).attr("y",30).attr("fill","black").text(selectedPlayer)
const predicted_pressure_text = svg_legend3.append("text").attr("x",800).attr("y",30).attr("fill","black").text("Predicted Pressure:")
const rect_blue = svg_legend3.append("rect").attr("x",930).attr("y", 25).attr("width", 50).attr("height", 2).attr("fill", "steelblue").attr("stroke", "steelblue")
const pmp_text = svg_legend3.append("text").attr("x",1000).attr("y",30).attr("fill","black").text("Predicted Motion Pressure:")
const rect_red = svg_legend3.append("rect").attr("x",1180).attr("y", 25).attr("width", 50).attr("height", 2).attr("fill", "red").attr("stroke", "red")

// Function for the third animation, identical to the first two.
function animateFrames() {
    const frameData = playData.filter(d => parseInt(d.frameId) === frameIndex)
    //console.log(frameIndex)

    svg3.selectAll("circle").remove();

    svg3.selectAll("circle")
        .data(frameData)
        .enter()
        .append("circle")
        .attr("r", 5)
        .attr("cx", d => xScale(parseFloat(d.x)))
        .attr("cy", d => yScale(parseFloat(d.y)))
        .attr("fill", getColor)
        .attr("stroke", "black")
        .attr("stroke-width", 1)
        .transition()
        .duration(duration)
        .attrTween("cx", function (d) {
            const interpolateX = d3.interpolate(this.getAttribute("cx"), xScale(parseFloat(d.x)))
            return function (t) {
                return interpolateX(t)
            }
```

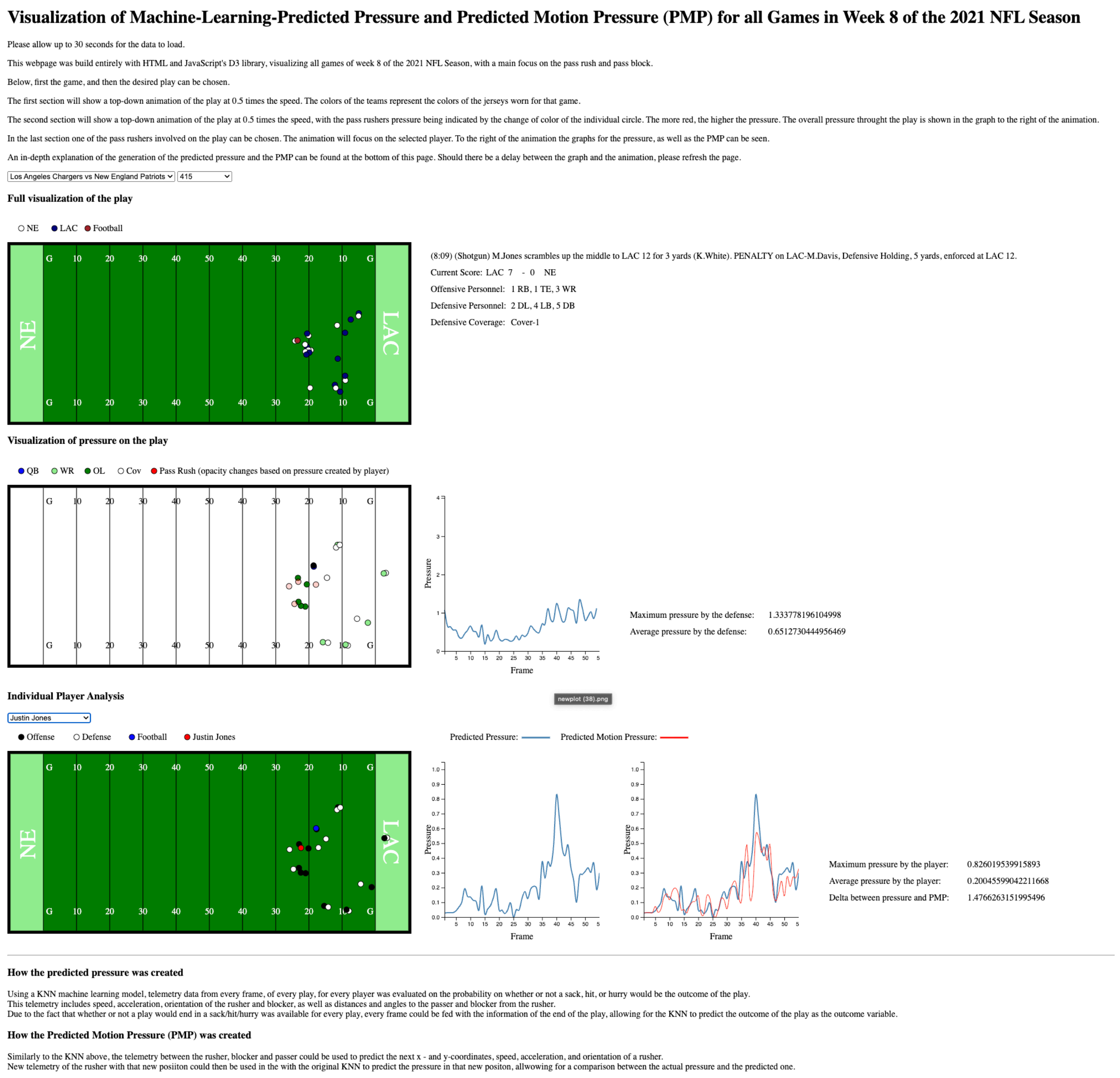
```
                                            })
                                            .attrTween("cy", function (d) {
                                                const interpolateY = d3.interpolate(this.getAttribute("cy"), yScale(parseFloat(d.y)));
                                                return function (t) {
                                                    return interpolateY(t)
                                                }
                                            })
                                        frameIndex++

                                        if (frameIndex > d3.max(playData, d => parseInt(d.frameId))) {
                                            frameIndex = 0
                                        }

                                        setTimeout(animateFrames, delay);
                                    }

                                    setTimeout(animateFrames, delay)
                                })
                })
            })})})})
        </script>
    </body>
<htlm>
```

11.5 Dashboard Screenshots



Visualization of Machine-Learning-Predicted Pressure and Predicted Motion Pressure (PMP) for all Games in Week 8 of the 2021 NFL Season

Please allow up to 30 seconds for the data to load.

This webpage was build entirely with HTML and JavaScript's D3 library, visualizing all games of week 8 of the 2021 NFL Season, with a main focus on the pass rush and pass block.

Below, first the game, and then the desired play can be chosen.

The first section will show a top-down animation of the play at 0.5 times the speed. The colors of the teams represent the colors of the jerseys worn for that game.

The second section will show a top-down animation of the play at 0.5 times the speed, with the pass rushers pressure being indicated by the change of color of the individual circle. The more red, the higher the pressure. The overall pressure throught the play is shown in the graph to the right of the animation.

In the last section one of the pass rushers involved on the play can be chosen. The animation will focus on the selected player. To the right of the animation the graphs for the pressure, as well as the PMP can be seen.

An in-depth explanation of the generation of the predicted pressure and the PMP can be found at the bottom of this page. Should there be a delay between the graph and the animation, please refresh the page.

Los Angeles Chargers vs New England Patriots 415

Full visualization of the play

(8:09) (Shotgun) M.Jones scrambles up the middle to LAC 12 for 3 yards (K.White). PENALTY on LAC-M.Davis, Defensive Holding, 5 yards, enforced at LAC 12.

Current Score: LAC 7 - 0 NE

Offensive Personnel: 1 RB, 1 TE, 3 WR

Defensive Personnel: 2 DL, 4 LB, 5 DB

Defensive Coverage: Cover-1

Visualization of pressure on the play

● QB ● WR ● OL ○ Cov ● Pass Rush (opacity changes based on pressure created by player)
G 10 20 30 40 50 40 30 20 10 G
Pressure
Frame

Maximum pressure by the defense: 1.333778196104998

Average pressure by the defense: 0.6512730444956469

Individual Player Analysis

Justin Jones

● Offense ○ Defense ● Football ● Justin Jones
G 10 20 30 40 50 40 30 20 10 G
NE
LAC
Predicted Pressure:
Predicted Motion Pressure:
Pressure
Frame

Maximum pressure by the player: 0.826019539915893

Average pressure by the player: 0.20045599042211668

Delta between pressure and PMP: 1.4766263151995496

How the predicted pressure was created

Using a KNN machine learning model, telemetry data from every frame, of every play, for every player was evaluated on the probability on whether or not a sack, hit, or hurry would be the outcome of the play.
This telemetry includes speed, acceleration, orientation of the rusher and blocker, as well as distances and angles to the passer and blocker from the rusher.
Due to the fact that whether or not a play would end in a sack/hit/hurry was available for every play, every frame could be fed with the information of the end of the play, allowing for the KNN to predict the outcome of the play as the outcome variable.

How the Predicted Motion Pressure (PMP) was created

Similarly to the KNN above, the telemetry between the rusher, blocker and passer could be used to predict the next x - and y-coordinates, speed, acceleration, and orientation of a rusher.
New telemetry of the rusher with that new posiiton could then be used in the with the original KNN to predict the pressure in that new positon, allwoing for a comparison between the actual pressure and the predicted one.