Problem Chosen	**2020 MCM/ICM Summary Sheet**	**Team Control Number**
D		2021857

Given the data of 38 games from the Huskies soccer team, we define a teamwork function, a function of many variables, that captures a teamwork index per game. We also study both micro and macro elements of the game, using graphs and shortest paths to study favorable positions. We use these findings to formulate strategies and recommendations for the Huskies ownership and coaching staffs.

After addressing assumptions and considerations, we create a teamwork index function that takes in various indicators, including shots, goals scored, player diversity, among others. By assuming a positive correlation between teamwork and goal differentials, we apply a regression to determine the necessary coefficients and dependencies per indicator. We analyze the results in order to educate our recommendations for the coach.

In our analysis, we conduct macro analysis through the use of a graph - we assign players to nodes and assign a shot as a node. We weight the edges based on various factors, including distance from action destination, player roles, and data. From there, we apply a shortest path algorithm to determine the optimal path to a shot from each player.

Micro analysis is done by analyzing chains of actions that lead to a shot. We visualize these chains of actions to see patterns and what kinds of plays lead to shots. With this analysis we can determine what kind of passing structures the team needs to incorporate more of.

Teaming Strategies: Quantifying Teamwork and Using Network Science to Analyze Soccer Strategy

Control Number: 2021857

ICM Weekend A: Problem D

Contents

1 Introduction

1.1 Restatement of the Problem

The success of a soccer team isn't dependent on individual talent - rather it depends on the cohesiveness of the team. The Huskies Coaches have provided data for the past 38 games. They have asked ICM to provide an analysis of this data, with focus on understanding teamwork dynamics.

With this data, ICM has been asked to first create a network that allows one to study formation dynamics. This can be done using the passing data provided, and generating a graph. Both macro and micro considerations must be made in order to study dynamics at a smaller, local scale, versus entire games.

From then, the Huskies have asked the ICM to create a teamwork model. This model should account for a variety of indicators, including shots, goals, tempo, and whatever other indicators the ICM finds necessary.

From this data, conclusions should be drawn to recognize the trends of the team currently. By using the teamwork model, the ICM should provide suggestions to the coaches on how to better their performance in the upcoming seasons.

1.2 Assumptions

More shots taken implies more goals. While every shot on target does not mean the Huskies will score, we assume that, if the Huskies take more shots, they will be more likely to score. We run on the follow up assumption that all shots taken in the data are "reasonable," meaning the player takes a shot if and only if they believe the probability of scoring is nonzero.

Every action taken by a player is intentional, and all occur for the purpose of either setting up a situation to score, or preventing the opposing team from scoring. This assumption allows us to consider chains of actions up to shots and assume that these actions are highly calculated, as to increase the probability of scoring at a later action.

Among the factors of what dictates a 'highly preferred' shot is distance and player. We assume that players have a higher chance when closer to the goal. We can also use the shot distribution, as in Figure 1.

A victory indicates a high level of teamwork. We make this assumption because single players generally do not dominate in soccer.

Single players cannot carry a team. (This will be discussed in more detail later.) While certain player roles can be assumed to perform certain actions better (i.e. Forwards have better shot opportunities than a defender based on positioning) we will assume that individual players cannot carry a team. We make this assumption based on the fact that having one strong shooter, for example, only works if the other players pass to this player.

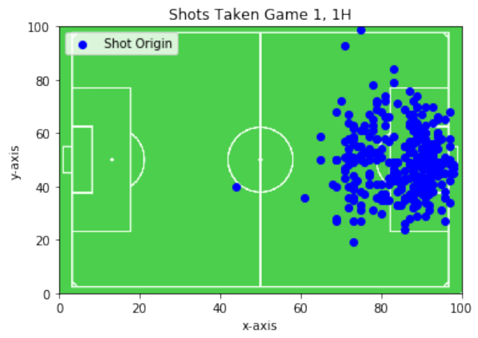

Figure 1: Distribution of Shot Origins: Shows which locations have more shots taken

The closer the ball is to the opponents goal, the more favorable. The location of the ball determines how "dangerous" it is. In essence if the ball is closer to the opponents goal, the Huskies will have better shot opportunities, and thus better goal scoring opportunities.

The opponents defense presses the ball, and they play "hybrid man-zone" defense. This is the most common defensive tactic in modern soccer, where defenders move closer to the player with the ball. We also assume that each opponent player is assigned a Huskies player to guard and a region of the field to control. Basically, we are assuming what is normal pressing and defensive line tactics.

Passing one layer up (i.e. Defender to Mid-fielder) is more favorable than passing multiple layers up or backwards (i.e Defender to Forward). Passing two layers up is more difficult as such passes would have to go a farther distance with high accuracy. At the same time passing two layers up would require the pass to avoid more opponent defenders, as there would be more defenders in the path of the pass.

1.3 Decisions

Offense is more important than defense. Since games cannot be won unless the Huskies score, we decided that our first priority should be to analyze trends in offense, rather than in defense.

Due to the low number of total goals (44), we instead analyzed data that would help maximize the total number of shots taken. The number of shots taken by the Huskies was about one order of magnitude greater. The limited data points we had for number of goals available makes it difficult to arrive to conclusions, as micro-trends may be coincidences. We run under the assumption that more shots implies more goals.

There does not exist enough data on coaches. (More on this point will be covered in later sections.) Using weighted analysis based on opponent difficulty, we found that all coaches are equal in ability. However, we cannot make strong assumptions since every opponent played against, at most, two coaches. Additionally, using win rates on their own are skewed since every coach played less than nine games.

2 Teamwork Model and Performance Indicators

To quantitatively measure teamwork, we define a function of multiple variables (indicators) and assign a numerical value per game. We use this indicator to then analyze which games display the most effective teamwork.

Define T to be the Teamwork Index Function. Such that T is a function of g, the number of goals, s, the number of shots taken, pd, player diversity index, cl, chain length, and t, tempo. The indicators g, and s are given by *matches.csv*. We define equations for indicators pd, cl, and t. We define our function this way such that a high value of T corresponds with a win.

2.1 Teamwork Index Function Indicator definition

2.1.1 pd, Player Diversity

Player diversity aims to measure the distribution of work among the players in a game. We calculate the number of actions each player is involved in per game, and take the standard deviation of the data. We remove the goalie from this data, as the goalie is generally not involved in offensive plays (and often in defensive as well). We then take the reciprocal of the standard deviation, such that pd is larger for games with an even distribution of action. That is:

$$pd = \sqrt{\frac{\sum_i (x_i - \mu)^2}{n}}^{(-1)} \tag{2.1.1}$$

where n is the number of player (minus the goalie(s)), μ is the average number of actions per player in the game, and x_i is the actions for player i.

We take pd to be an important input in T because an even distribution of actions implies that every player on the field played a critical role in the games result.

2.1.2 cl, Chain Length

Chain length aims to measure the amount of cohesive plays between multiple players (i.e. multiple passes and a shot). A chain is defined to be any series of nontrivial actions that is at least one action long. (Substitutions, for example, are a trivial event.) We define cl to be the average length of any chain of actions. That is:

$$cl = \frac{\sum_i c_i}{n_c} \tag{2.1.2}$$

Where c_i is the length of chain i, and n_c is the number of chains.

We take c to be an important input in T because a longer chain of events implies cohesive teamwork and positioning.

2.1.3 t, Tempo

Tempo aims to measure the speed of the game when the home team has control. That is, a higher tempo is measured when the time between actions in a chain of actions is greater.

We define t as follows. First, consider all chains of actions length 2 or greater. In these chains, find τ_i, the average time per action of the chain. Define τ_i as:

$$\tau_i = \frac{a_f - a_i}{n_a} \tag{2.1.3}$$

Where $a_f - a_i$ yields the time interval of the chain, and n_a is the number of actions in the chain. To find t, We sum over all τ_i and take the average by dividing by the number of chains of length 2 or greater. That is:

$$t = \frac{\sum_i \tau_i}{n_c} \tag{2.1.4}$$

Where n_c is the number of chains.

We take t to be an important input in T because a quicker tempo implies more cohesion. That is, high tempo implies high trust between teammates (i.e faster passing).

2.1.4 ag, Aggressiveness

Team aggressiveness aims to measure how aggressive the team is being with the ball. For each game we calculate a "action center of mass." That is the average location of all Huskies player actions in that game. Then we find the distance between this "action center of mass" and our own goal at (0,50). That is:

$$\overline{lx} = \frac{\sum_i (lx_i)}{n} \tag{2.1.5}$$

$$\overline{lx} = \frac{\sum_i (lx_i)}{n} \tag{2.1.6}$$

$$d = \sqrt{\left(\overline{lx}\right)^2 + \left(\overline{ly} - 50\right)^2} \tag{2.1.7}$$

Where n is the number of player (minus the goalie(s)), lx_i is the average x-coordinate for actions by player i, and ly_i is the average y-coordinate for actions by player i.

We take d to be an important input in T because a large dispersion means that the team is covering a larger part of the field together, field control plays a large part in game result.

2.1.5 d, Dispersion

Player dispersion aims to measure how far apart the players are from each other. In dispersion we exclude the goalie when referring to all players. This variable also helps measure how much of the field is being covered by the players. For each game we calculate a "action center of mass." That is the average location of all Huskies player actions in that game. We then calculate the average position of each player in that game: "player center of mass". Now, we find the distance between each "player center of mass" and the "action center of mass", and take the average. That is:

$$d = \frac{\sum_i \sqrt{\left(lx_i - \overline{lx}\right)^2 + \left(ly_i - \overline{ly}\right)^2}}{n} \tag{2.1.8}$$

Where n is the number of player (minus the goalie(s)), $\overline{lx}, \overline{ly}$ are defined as in aggressiveness, lx_i is the average x-coordinate for actions by player i, and ly_i is the average y-coordinate for actions by player i.

We take d to be an important input in T because a large dispersion means that the team is covering a larger part of the field together, field control plays a large part in game result.

2.2 Teamwork Index function considerations

We use the following rankings to determine how much each indicator contributes to the teamwork index: $g, s, t, d, a, cl, pd, ps$, with the first indicator being most important. We choose this ordering because we prioritize winning games and scoring goals: since soccer is a sport, victories should be priorities. Victories occur from scoring goals, and goals occur from taking more shots.

2.3 Defining T

For T, we took inspiration from multi-particle systems in physics as a starting point for our function. (We considered the Fine Structure for hydrogen to examine how smaller corrections change the value of the system, namely through a linear and quadratic correction, and also took influence from hydrogen deviants with multiple electrons.) After editing this model to accommodate for our problem, we define T as:

$$\begin{aligned} T(g, s, t, d, a, cl, pd, ps) = {} & A + B(n_g g) + C(n_s s) + D(n_t t) + E(n_d d) + F(n_a a) \\ & + G1(n_{cl}(cl))^2 + G2(n_{cl}(cl)) \\ & + H1(n_{pd}(pd))^2 + H2(n_{pd}(pd)) \\ & + I1(n_{ps}(ps))^2 + I2(n_{ps}(ps)) \end{aligned} \tag{2.3.1}$$

Where $n_g, n_s, n_t, n_d, n_a, n_{cl}, n_{pd}, n_{ps}$ are normalization factors such that $n_g g$, $n_s s, n_t t$, $n_d d, n_a a, n_{cl} cl, n_{pd} pd, n_{ps} ps \leqslant 1$. Where $B, C, D, E, F, G1, H1, I1, G2$, $H2, I2$ are weights (coefficients in a regression) for each of the normalized variables, and A is a constant (intercept in a regression).

2.4 Regressing T against Victory Margin

We regressed $T(g, s, t, d, a, cl, pd, ps)$ against goal differential, the margin of victory in each game, to see which of these "teamwork" factors play into victories and margin of victory.

We also regressed $T(g, s, \vec{0}, \vec{0}, \vec{0}, \vec{0}, \vec{0}, \vec{0})$ against goal differential to see how goals of a role goals and shots have in victories, and if passing, dispersion, tempo, player diversity, and chain length play a major role. In this second set of regression results we see that our

Dep. Variable:	Goal Diff	**R-squared:**	0.453
Model:	OLS	**Adj. R-squared:**	0.422
Method:	Least Squares	**F-statistic:**	14.50
Date:	Sun, 16 Feb 2020	**Prob (F-statistic):**	2.58e-05
Time:	21:15:16	**Log-Likelihood:**	-14.114
No. Observations:	38	**AIC:**	34.23
Df Residuals:	35	**BIC:**	39.14
Df Model:	2		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	-0.4497	0.163	-2.758	0.009	-0.781	-0.119
g	1.2613	0.235	5.364	0.000	0.784	1.739
s	-0.0195	0.372	-0.053	0.958	-0.774	0.735

Omnibus:	6.302	**Durbin-Watson:**	1.768
Prob(Omnibus):	0.043	**Jarque-Bera (JB):**	5.387
Skew:	-0.914	**Prob(JB):**	0.0677
Kurtosis:	3.247	**Cond. No.**	7.46

Figure 2: Regression Model for $T(g, s, \vec{0}, \vec{0}, \vec{0}, \vec{0}, \vec{0}, \vec{0})$

parameters are useful in determining how strong a team is and if it will win a match since the adjusted R-Squared value is greater. This gives us reason to believe that parameters past goals and shots are necessary for an accurate model. Moreover, by regressing, we are able to understand the relationships between these parameters, teamwork, and victories.

The positive coefficient on goals and aggression imply that higher goals and higher aggression implies a greater goal differential. This is trivially expected. More interestingly, the shot coefficient is negative on both regressions. This implies that players are not conforming into optimal shot opportunities, meaning that high accuracy on shots are preferred.

We notice that the coefficients of pd are determined in such a way that, on the interval [0, 1], high player diversity is favored. This is as expected in our teamwork model.

On the same normalized interval, we see that shorter cl are favored. This would imply that longer chains don't necessarily correlate with higher successes. Again, like shots, this implies that chaining doesn't imply shot opportunities. Thus, compact structures of few players seem optimal. In order to maintain high player diversity, these compact structures should not be limited to the same few players.

Additionally, we interestingly notice that higher passing is not favored. This is a result of our model - we fit our teamwork data for higher goal differentials under the assumption that no players can carry a team. Low passing being favored implies that, while possession is high during the game, passing is not necessarily generating opportunities for scoring. If anything, most passing chains do not result in opportunities, and often lead to more loss of possession.

Dep. Variable:	Goal Diff	**R-squared:**	0.650
Model:	OLS	**Adj. R-squared:**	0.502
Method:	Least Squares	**F-statistic:**	4.391
Date:	Sun, 16 Feb 2020	**Prob (F-statistic):**	0.000935
Time:	21:15:10	**Log-Likelihood:**	-5.6348
No. Observations:	38	**AIC:**	35.27
Df Residuals:	26	**BIC:**	54.92
Df Model:	11		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	0.2161	3.486	0.062	0.951	-6.950	7.382
g	1.1688	0.255	4.583	0.000	0.645	1.693
s	-0.5239	0.505	-1.039	0.309	-1.561	0.513
t	-0.4120	0.566	-0.727	0.474	-1.576	0.752
d	0.7917	0.689	1.150	0.261	-0.624	2.207
a	0.4439	1.758	0.252	0.803	-3.170	4.058
cl2	4.3527	6.554	0.664	0.512	-9.120	17.825
cl	-7.1238	11.070	-0.644	0.526	-29.878	15.631
pd2	-1.1263	5.059	-0.223	0.826	-11.525	9.273
pd	4.2820	7.630	0.561	0.579	-11.401	19.965
ps2	0.2305	4.456	0.052	0.959	-8.929	9.390
ps	-1.4385	7.137	-0.202	0.842	-16.110	13.233

Omnibus:	2.617	**Durbin-Watson:**	1.538
Prob(Omnibus):	0.270	**Jarque-Bera (JB):**	2.386
Skew:	-0.587	**Prob(JB):**	0.303
Kurtosis:	2.643	**Cond. No.**	649.

Figure 3: Regression Model for $T(g, s, t, d, a, cl, pd, ps)$

3 Macro-Analysis of plays

In order to assess teamwork flaws, we take a macro-analysis of all the play data to see what macro-tendencies exist. First, we find the average position that a player goes to over the course of the 38 games. We do so by taking the coordinates of their actions (either active action, or recipient position if a pass) and averaging over the course of all games. We then do a pass and shot analysis. We find the probability that a person takes an action (either a shot, a pass, etc.) by finding the total number of said action and dividing it by the total number of actions that person takes. For example, if a player shoots 10 times and passes 90 times, we say that the probability of passing is 90/100 or 0.9. We more specifically look at passes between players. These values become the raw probabilities of each action. We create a directed graph where every node is a player and "shot" is another node. The raw weights of each edge or these probabilities normalized.

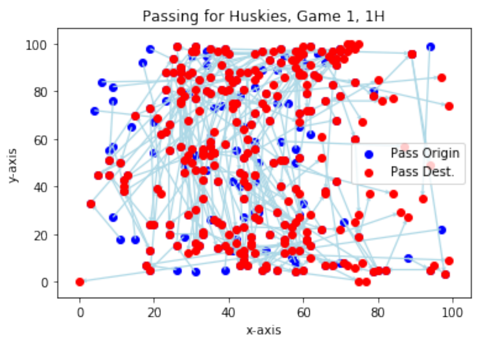

Figure 4: Huskies Passes in 1st Half of Game 1

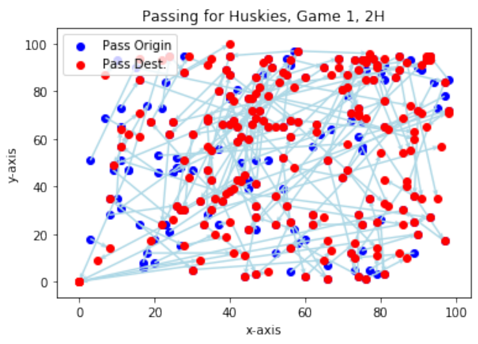

Figure 5: Huskies Passes in 2nd Half of Game 1

3.1 Weight Considerations

As discussed later, we take the negative log to find the weights of our edges on our graph. Thus, we assign fractional weights. We add extra considerations to our raw data as follows. Firstly, based on our assumptions, we weight actions of shorter paths greater. That is, a shot from a player whose average position is closer to a goal is more favorable than that of a player farther from the goal. We do so by first finding the distance squared between the action origin and action destination. Call this value ΔR^2For passing, we assign a *pass_multiplier* to be

$$pass\_multiplier = c_{pm}\frac{1}{(\Delta R^2)^{10}} \tag{3.1.1}$$

Where c_{pm} is a normalization factor, equal to $1/max(pass\_multiplier)$ so that all multipliers or less than 1. We scale this value by a factor of 10 because we assume that passes should be compact.

We assign a similar result for shots:

$$shot\_multiplier = c_{sm}\frac{1}{(\Delta R_g^2)} \tag{3.1.2}$$

Where ΔR_g^2 is the distance from the goal and c_{sm} is a normalization factor. Note that we scale this much less than the *pass_multiplier*. We do so because we prioritize shooting, and would like more shots to occur based on our assumption.

Additionally, we assume that Forwards shoot better than Mid-fielders, who shoot better than Defenders. Goalie's should never shoot, so we remove that edge completely. We accomplish this by adding a *shot_loss* factor. We assign discrete values based on the player type.

$$shot\_loss_{goalie} = 0.0 \tag{3.1.3}$$
$$shot\_loss_{defender} = 0.0001 \tag{3.1.4}$$
$$shot\_loss_{mid-fielder} = 0.1 \tag{3.1.5}$$
$$shot\_loss_{forward} = 1.0 \tag{3.1.6}$$

Notice that these values makes it so that forward's shooting is favorable. We use an exponential drop of from mid-fielder to defender since, based on position, defenders tend to straddle the midfield line when in an offensive formation. Thus, mid-fielders and forwards are the preferred shooters. We do not set the defender *shot_loss* to zero because the probability of a defender shooting should still be greater than that of a goalie.

For passing, we also include a *benefit* factor. We assume that there are certain preferred passing schemes (i.e. goalie passing to defender is favorable over goalie passing to forwards). Most of this data is captured by average distance, but do account for noise, we multiply passing edges by a factor to accommodate. Passing back to the goalie is set to 0 since this is an unfavorable position. Passing to a player one layer above gets a benefit multiplier of 1.0. Passing within a layer also gives a factor of 1.0. Passing forward, skipping a layer (i.e. defender to forward) is dampened by a factor of 10^{-6}. Passing backwards is also dampened.

3.2 Optimal Macro Paths

With these raw weights, we use the following formula to find the actual weights:

$$pass\_weight = -log(raw\_weight * pass\_multiplier * benefit) \tag{3.2.1}$$

$$shot\_weight = -log(raw\_weight * shot\_loss * shot\_multiplier) \tag{3.2.2}$$

Note that, because all values are normalized, the argument within the log functions will be greater than 0 and less than 1. Thus, taking the negative log yields positive numbers. We take the $-log$ of these weights, because we are treating them as pseudo-probabilities. These weights are based off of the passing distribution, so the raw weights in essence create a Markov chain. The weight considerations mean that the edge weights are no longer exact probabilities, still we can treat them as such. This way when we take the shortest path, we get the highest pseudo-probability of taking a good shot.

Player	Chain
F1	[Shot]
F2	[Shot]
F3	[Shot]
F4	[Shot]
F5	[Shot]
F6	[Shot]

Player	Chain
M1	[F4, Shot]
M2	[F4, Shot]
M3	[F4, Shot]
M4	[F4, Shot]
M5	[Shot]
M6	[F4, Shot]
M7	[M1, F4, Shot]
M8	[F4, Shot]
M9	[F4, Shot]
M10	[F4, Shot]
M11	[F4, Shot]
M12	[F4, Shot]
M13	[F4, Shot]

Player	Chain
G1	[D1, M1, F4, Shot]
D1	[M1, F4, Shot]
D2	[M1, F4, Shot]
D3	[M6, F4, Shot]
D4	[M1, F4, Shot]
D5	[M6, F4, Shot]
D6	[M1, F4, Shot]
D7	[M1, F4, Shot]
D8	[M4, F4, Shot]
D9	[M1, F4, Shot]
D10	[M1, F4, Shot]

Table 1: Table of Chains. Left column is the starting player. Right column is the chain. F = Forward, D = Defender, M = Midfielder, G = Goalie

Using these weights, we apply Dijkstra's shortest path algorithm, starting from every player as a destination. The result is what we call the **optimal macro paths** from each player to a goal. We call this macro because we recognize that often players will form an offensive formation and pass back and forth until an opening is found. These paths simply field the big picture setups.

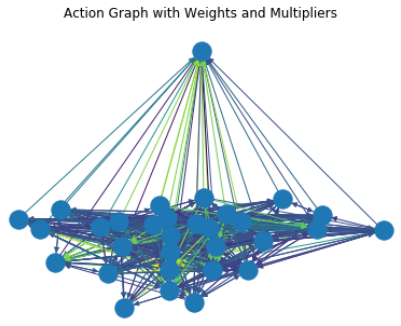

Figure 6: Weighted Directed Graph of all actions. The top node is the node for 'shot' and the remaining are player nodes. Darker colors implies favorable paths.

3.3 Analysis of results

Figure 6 indicates the optimal path to a shot from every player. We notice a few trends. From our weightings of the edges, we notice that Defenders tend to pass to midfielders, mid-fielders tend to pass to forwards, and forwards tend to shoot. On a macro level, this is expected.

We notice that chain wise, Forward 4 is the favorable forward to pass to. Every large chain includes Forward 4, meaning that Forward 4's position is optimal, and in past games, Forward 4 was involved in most plays (whether it be by shooting or by passing and receiving).

We also notice that most passing from defenders generally go to Midfielder 1 or 6, implying that these two midfielders are optimally positioned. We also notice that Midfielder 5 tends to shoot. This means that Midfielder 5 is positioned in a way that shooting is optimal.

We ignore the fact that the goalie generally passes to Defender 1. This simply means that defender 1 tends to be closest to the goalie in proximity.

4 Micro Analysis of Plays

In order to understand individual plays, we take chains of actions by a team and analyze them. This way we can see what passing patterns exist and what passing patterns result in scoring opportunities.

4.1 Passing Structures

Here we see corroboration of conclusions drawn from our teamwork model. Most long passing chains are behind the center line or at midfield. As seen in Figure 7, most long chains of possession and passing are not aggressive enough.The flow of time is in these figures represented by color shift from yellow (start of chain) to blue (end of chain). Instead the team seems to be hanging back and and essentially wasting time. The most forward these long passing chains go are to the left or right of midfield, as in Figure 8. In the chains the ball is never in a dangerous area and there usually are no opportunities to score. Very few passing chains even span midfield, as in Figure 9.

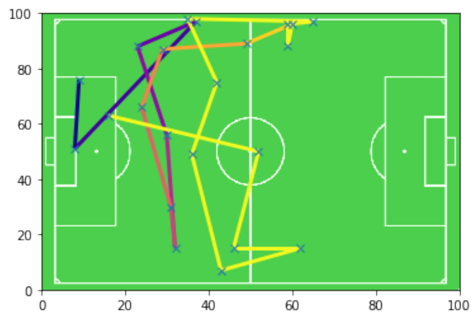

Figure 7: Backpass Chain: Demonstrates a long chain of possession with low amounts of aggression

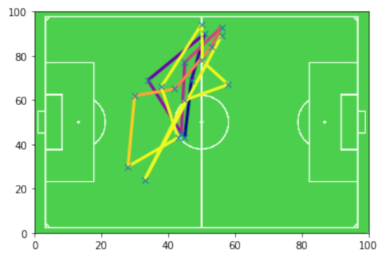

Figure 8: Midpass Chain 1: Demonstrates a pass structure that tends to the edges of the field

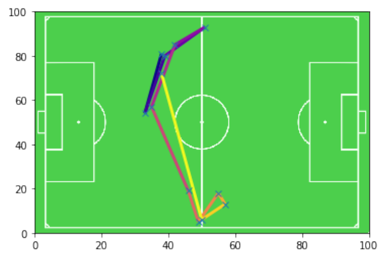

Figure 9: Midpass Chain 2: Demonstrates a pass structure that spans the midfield

This shows that the forwards are hanging back with the defenders and midfielders, only being as aggressive as the back lines. It could also be that the defending lines are unwilling to be more aggressive and are hanging back even when forwards try to move towards goal. With more data on player location, even when not on an action, we would be able to determine why there is a lack of aggressiveness in the team and what kind of gaps the formation lines are leaving.

4.2 Opportunity Creation

Seeing that pure passing structures do not give much insight into what does create scoring opportunities, we filter down from passing chains to action chains that end in a shot. Here we start to notice patterns that create scoring opportunities.

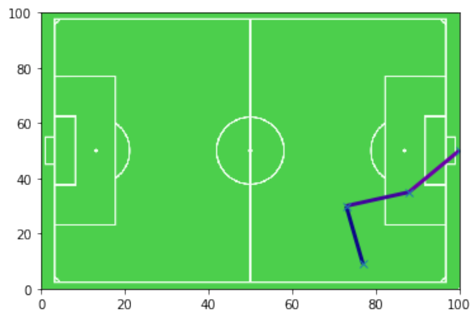

Figure 10: Capitalization on Opponent Mistake ex. 1

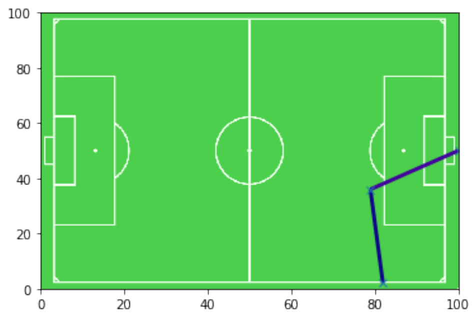

Figure 11: Capitalization on Opponent Mistake ex. 2

We see that the opponent mistakes are the origination of the most dangerous passing chains. In other words when a Huskies player is able to acquire the ball in the opponents half of the field, as in Figure 10 and Figure 11.

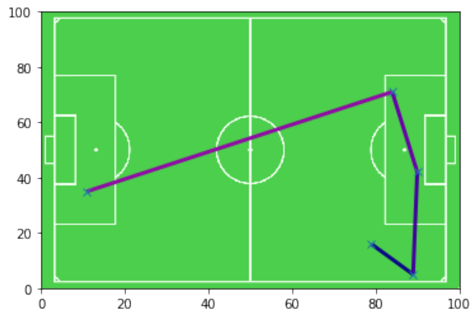

Figure 12: Capitalization on the Counter Attack ex. 1

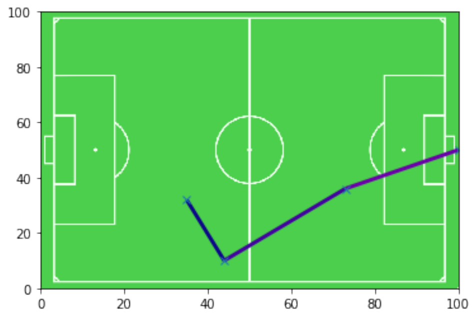

Figure 13: Capitalization on the Counter Attack ex. 2

The Huskies are also able to generate opportunities off of the counterattack, where there is one long action (dribble or pass) from the defending third to the attacking third of the

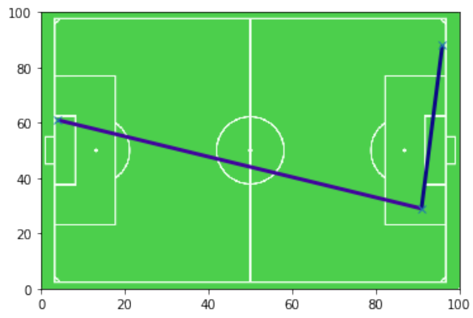

Figure 14: Capitalization on the Counter Attack ex. 3

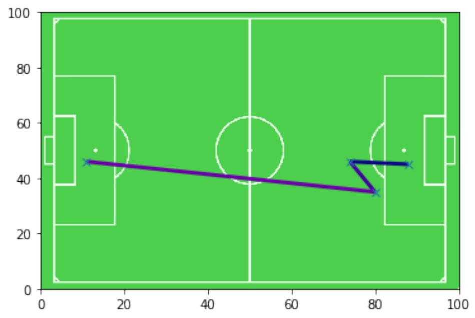

Figure 15: Capitalization on the Counter Attack ex. 4

field, as in Figure 12, Figure 13, Figure 14, Figure 15.

Once the Huskies have control of the ball in the attacking third, strong crossing passes tend to result in the majority of scoring opportunities. Having the midfielders and defenders move up also allows the Huskies to maintain possession in case the ball is cleared or the shot is missed, as in Figure 16. Not having defenders pushing up the field means that play buildup has to restart with the goalie, as in Figure 18. Pulling the midfielders and defenders forward can also help setup a second chance at goal and good plays after a missed shot, as in Figure 17.

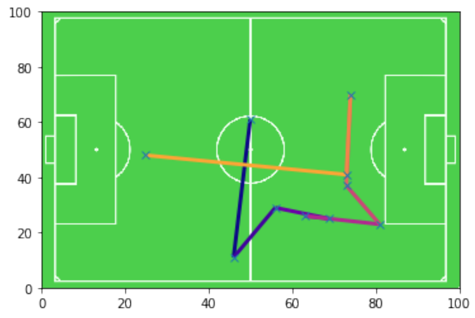

Figure 16: Keeping Possession ex. 1

Unfortunately, the Huskies are not able to develop scoring opportunities from long possession chains Figure 14. A good play structure is one where the passing moves forward and keeps going across the field. This way opponent defenders are drawn away creating open space. In Figure 15 we see an aggressive and accurate passing of the ball downfield, this kind of structure should also be more common. The lack of such plays tells us that the Huskies midfielders are either inaccurate passers or unwilling to pass into the attacking third of the field.

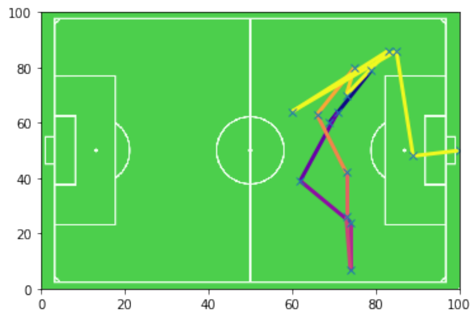

Figure 17: Keeping Possession ex. 2

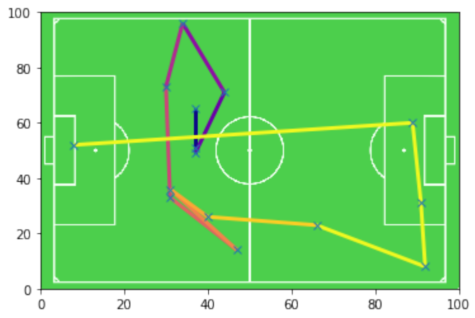

Figure 18: Keeping Possession ex. 3

4.3 Dyadic and Triadic Configurations

We see that with counterattacks and opponent mistakes, dyadic and Triadic configurations result in the better opportunities to score, as in Figure 5 and Figure 7. When it comes to attacking, Dyadic configurations with passes across the width of the field help move the ball forward as described in the above section.

Even more interesting is how Triadic patterns allow the team to keep possession for longer and move the ball further as in Figure 15. In this particular play, one of the corners of the Triadic pattern is swapped with a player farther down the field to keep the ball moving towards the attacking theirs.

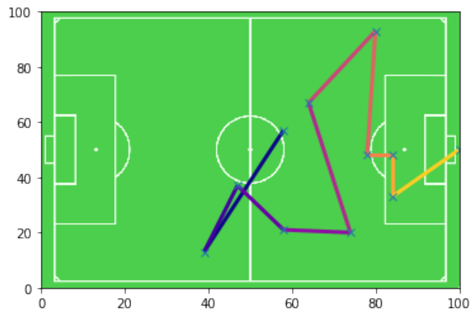

Figure 19: Good Play ex. 1

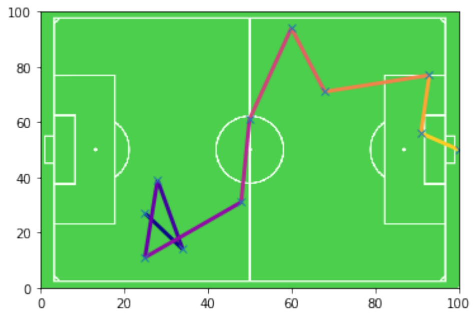

Figure 20: Good Play ex. 2

5 Recommendations for the Huskies' Coaches

5.1 Formation Recommendations

From the chain results of section 4, we notice that certain players are favored. To further improve the assigned teamwork index, a diversity of players on a macro level would be beneficial.

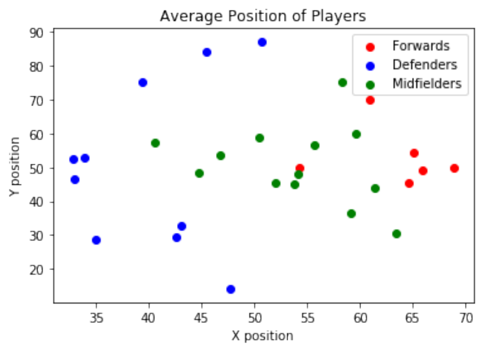

Figure 21: Average position of the players, color coded based on player type (with the goalie omitted)

Figure 16 displays the average position of the players, color coded based on player type (with the goalie omitted). We notice that, the field coverage is quite strong, but there are certain players that are more optimally positioned to be involved in plays. For example, the rightmost red dot is Forward 4, who happens to be involved in a majority of the action chains. One solution to increase diversity is to allow for different players to assume the position Forward 4 usually takes. However, a more practical solution would be to direct the forwards to equalize their average position. That is, the other forwards should prioritize reaching positions closer to the goals. Namely, the leftmost forward on the diagram should push their position closer to the front, and the rightmost forward should tend more toward the left.

A similar result is seen with the midfielders, as they disperse amongst the middle of the field. We don't recommend creating a straight line along $x = 50$, as this would simply be impractical.

5.2 Practice Recommendations

We also noticed that there was a negative correlation between passing and winning games, we see this as an indicator that the Huskies are unable to generate scoring chances off of possession play and passing. Instead it seems like the Huskies generate scoring chances primarily off of counter attacks. This means that the coaching staff should focus practicing on creative passing and play-making off of long possessions.

This recommendation is corroborated by our Micro Analysis of Plays as well. It is very important that the Huskies develop scoring opportunities from possession. The team needs to practice the kinds of play structures shown in Figure 14 and Figure 15. These play structures revolve around moving the ball forward small amounts via passes across the width of the field, such structures open up space near the goal by drawing opponent defenders away from the box. The Huskies midfielders and defenders need to be more aggressive pushing up into the attacking third of the field. They also need to be more aggressive with their passes downfield, as they seem to be too risk-averse.

5.3 Player Decision Recommendations

Given the negative relationship of shooting and teamwork, we would hope that players reassess shooting opportunities and passing opportunities.

We can also use our Macro Analysis of plays, and our shortest path algorithm to recommend players where to go with the ball when they have possession. Going off of Table 1 from Section 4, we could recommend D1 to prefer passing to M2 when he has possession of the ball. In the same way, we could recommend M7 to consider passing to M1 when he has possession of the ball. Thus we would be setting up, on the macro level, the best types of plays.

6 Generalization of Results

While the data and results were geared specifically to soccer, teamwork exists in many places outside of sports. This includes large corporations, government structures, and military operations. Thus, analysis of team dynamics can be extrapolated and applied anywhere.

Indicators like passing, shooting, and goals don't necessarily apply to other team analogs, but there exists indicators that help measure teamwork. Goals and shots can be thought of as accomplishing objectives. More interestingly, we can aim to find analogous indicators for the more complex indicators, such as chain length, player diversity, etc.

In a company setting, where teamwork can again be outlined as a directed graph, chain length can be thought of the number of people on a team per project. This is reasonable since many projects in companies, such as software development, requires the input of front end and back end developers, as well as project managers. Player diversity can be kept the same, since different workers will have different strengths and knowledge bases. However, as in our model, this term should be quadratic since too many people on a project may negatively impact the result.

Of course, the coefficients on the regression will be setting and data dependent, but if enough data is present, one could run the same regression with our model and extrapolate results from there.

7 Bibliography

Buldú, J.M., Busquets, J., Echegoyen, I. et al. (2019). Defining a historic football team: Using Network Science to analyze Guardiola's F.C. Barcelona. Sci Rep, 9, 13602.

Cintia, P., Giannotti, F., Pappalardo, L., Pedreschi, D., Malvaldi, M. (2015). The harsh rule of the goals: Data-driven performance indicators for football teams. 2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA), 1-10, 7344823.

Duch J., Waitzman J.S., Amaral L.A.N. (2010). Quantifying the performance of individual players in a team activity. PLoS ONE, 5: e10937.

Gürsakal, N., Yilmaz, F., Çobanoğlu, H., Çağliyor, S. (2018). Network Motifs in Football. Turkish Journal of Sport and Exercise, 20 (3), 263-272.

8 Appendix: Code

A Data Import and Packages

```
[ ]: import pandas as pd
     import sys
     import numpy as np
     import math as m
     from sklearn import linear_model
     from matplotlib import pyplot as plt
     np.set_printoptions(threshold=sys.maxsize)

     full_events_data = pd.read_csv('fullevents.csv').values
     passing_events_data = pd.read_csv('passingevents.csv').values
     matches_data = pd.read_csv('matches.csv').values
     print(full_events_data.size)
     print("Imports finished")
```

```
[ ]: import networkx as nx #for graphs
     import itertools
     print("Imported")
```

B Pass Scatter Graphs

```
[ ]: #8,9 is the start x,y. 10,11 is the end x,y
     #defining the coordinate arrays
     x_origin_coord = np.array([])
     y_origin_coord = np.array([])
     x_dest_coord = np.array([])
     y_dest_coord = np.array([])


     for row in full_events_data:
       if (row[6] == 'Pass') and (row[0] == 1) and (row[1] ==␣
      ↪'Huskies') and (row[4] == '1H'):
         x_origin_coord = np.append(x_origin_coord, row[8])
         y_origin_coord = np.append(y_origin_coord, row[9])
         x_dest_coord = np.append(x_dest_coord, row[10])
         y_dest_coord = np.append(y_dest_coord, row[11])
```

```
#number of passes Huskies in game 1, Half 1
x_diff = x_dest_coord - x_origin_coord
y_diff = y_dest_coord - y_origin_coord
num_passes_Huskies1_1H = np.size(x_origin_coord)

print("Number of passes for the Huskies in the first game 1H␣
 ↪is: " + str(num_passes_Huskies1_1H))
long_zeros = np.zeros(num_passes_Huskies1_1H)

fig, ax = plt.subplots()
ax.scatter(x_origin_coord, y_origin_coord, color = 'blue',␣
 ↪label='Pass Origin')
#ax.quiver(x_origin_coord[0], y_origin_coord[0], x_diff[0],␣
 ↪y_diff[0])
#ax.arrow(x_origin_coord[0], y_origin_coord[0], x_diff[0],␣
 ↪y_diff[0], head_width=0.1, head_length=0.5, fc='red',␣
 ↪ec='black')
plt.quiver(x_origin_coord, y_origin_coord, x_diff, y_diff,␣
 ↪angles='xy', scale_units='xy', scale=1,␣
 ↪facecolor='lightblue')
ax.scatter(x_dest_coord, y_dest_coord, color = 'red', label =␣
 ↪'Pass Dest.')
plt.title('Passing for Huskies, Game 1, 1H')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()


#defining the coordinate arrays
x_origin_coord = np.array([])
y_origin_coord = np.array([])
x_dest_coord = np.array([])
y_dest_coord = np.array([])


for row in full_events_data:
  if (row[6] == 'Pass') and (row[0] == 1) and (row[1] ==␣
 ↪'Huskies') and (row[4] == '2H'):
    x_origin_coord = np.append(x_origin_coord, row[8])
    y_origin_coord = np.append(y_origin_coord, row[9])
    x_dest_coord = np.append(x_dest_coord, row[10])
    y_dest_coord = np.append(y_dest_coord, row[11])

#number of passes Huskies in game 1, Half 2
x_diff = x_dest_coord - x_origin_coord
y_diff = y_dest_coord - y_origin_coord
num_passes_Huskies1_2H = np.size(x_origin_coord)

print("Number of passes for the Huskies in the first game 2H␣
 ↪is: " + str(num_passes_Huskies1_2H))
long_zeros = np.zeros(num_passes_Huskies1_2H)

fig, ax = plt.subplots()
ax.scatter(x_origin_coord, y_origin_coord, color = 'blue',␣
 ↪label='Pass Origin')
plt.quiver(x_origin_coord, y_origin_coord, x_diff, y_diff,␣
 ↪angles='xy', scale_units='xy', scale=1,␣
 ↪facecolor='lightblue')
ax.scatter(x_dest_coord, y_dest_coord, color = 'red', label =␣
 ↪'Pass Dest.')
plt.title('Passing for Huskies, Game 1, 2H')
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()
```

C Leading Sequence

[ ]:
```
""#### Determining sequences leading up to shots
from matplotlib.collections import LineCollection

#list we will put numpy arrays
action_chain = []

i = 0
action_index = 0
```

```
#while index is okay and we are in GAme 1 TIME 1H
while (i < np.size(full_events_data)-1) and␣
 ↪(full_events_data[i][0] == 1):
  if (full_events_data[i][4] == '2H'):
    if (full_events_data[i][1] == 'Huskies'):
      temp = []
      action_index += 1
      i += 1
      temp.append(full_events_data[i])
      while(full_events_data[i][1] == 'Huskies') and␣
 ↪(full_events_data[i][4] == '2H') and␣
 ↪(full_events_data[i][0] == 1) :
        i += 1
        temp.append(full_events_data[i])
      #once you exit the chain.
      action_chain.append(temp)

    else: #increasing i if we aren't on a 'good' move
        i += 1
  else:
    i += 1


print()
print("PRINTING ACTION CHAIN")
print(action_chain)
for i in range(0, len(action_chain)):
  print("\n","ACTION INDEX: " + str(i))
  t= np.arange(0, len(action_chain[i]))
  this_x=[]
  this_y=[]
  for j in range(0, len(action_chain[i])):
    if action_chain[i][j][-4] != 100:
      this_x.append(action_chain[i][j][-4])
      this_y.append(action_chain[i][j][-3])
    else:
      this_x.append(100)
      this_y.append(50)
    print(action_chain[i][j][2],␣
↪action_chain[i][j][6],action_chain[i][j][7],
          " Start ",␣
↪action_chain[i][j][-4],action_chain[i][j][-3]," End ",
          action_chain[i][j][-2],action_chain[i][j][-1])
  if(len(this_x)>2):
    plt.plot(this_x,this_y, 'x')

    points = np.array([this_x, this_y]).T.reshape(-1, 1, 2)
    segments = np.concatenate([points[:-1], points[1:]],␣
↪axis=1)
    lc = LineCollection(segments, cmap=plt.get_cmap('plasma'),
    norm=plt.Normalize(0, 10))
    lc.set_array(t)
    lc.set_linewidth(3)

    plt.gca().add_collection(lc)
    plt.xlim(0, 100)
    plt.ylim(0, 100)
    plt.show()
```

[ ]:
```
#### Determining sequences leading up to shots
from matplotlib.collections import LineCollection

for game in range(1,39):
  for half in ['1H','2H']:
    #list we will put numpy arrays
    action_chain = []

    i = 0
    action_index = 0
    #while index is okay and we are in GAme 1 TIME 1H
    while (i < 59270) :
      if (full_events_data[i][4] == half)and␣
↪(full_events_data[i][0] == game):
        if (full_events_data[i][1] == 'Huskies'):
          temp = []
          action_index += 1
          i += 1
          temp.append(full_events_data[i])
          while(full_events_data[i][1] == 'Huskies') and␣
↪(full_events_data[i][4] == half) and␣
↪(full_events_data[i][0] == game) :

            i += 1
            temp.append(full_events_data[i])
          #once you exit the chain.
          action_chain.append(temp)

        else: #increasing i if we aren't on a 'good' move
            i += 1
      else:
        i += 1


    print()
    print("PRINTING ACTION CHAIN",game,half)

    for i in range(0, len(action_chain)):
      t= np.arange(0, len(action_chain[i]))
      this_x=[]
      this_y=[]
      this_act=[]
      for j in range(0, len(action_chain[i])):
        if (action_chain[i][j][-4] != 100) and␣
↪(action_chain[i][j][-4] != 0):
          this_x.append(action_chain[i][j][-4])
          this_y.append(action_chain[i][j][-3])
        else:
          this_x.append(100)
          this_y.append(50)
        this_act.append([action_chain[i][j][2],
                         action_chain[i][j][6],
                         action_chain[i][j][7],
                         " Start ",
                         action_chain[i][j][-4],
                         action_chain[i][j][-3],
                         " End ",
                        ␣
 ↪action_chain[i][j][-2],action_chain[i][j][-1]])

      if len(this_x)>2 and "Shot" in [row[2] for row in␣
 ↪this_act]:
        print("\n","ACTION INDEX: " + str(i))
        for actionz in this_act:
          print(actionz)
        points = np.array([this_x, this_y]).T.reshape(-1, 1,␣
 ↪2)
        segments = np.concatenate([points[:-1], points[1:]],␣
 ↪axis=1)
        lc = LineCollection(segments, cmap=plt.
 ↪get_cmap('plasma'),
        norm=plt.Normalize(0, 10))
        lc.set_array(t)
        lc.set_linewidth(3)
        im = plt.imread("Soccer_Field_Transparant.png")
        implot = plt.imshow(im,extent=[0, 100, 0, 100])
        plt.plot(this_x,this_y, 'x')
        plt.gca().add_collection(lc)
        plt.gca().set_aspect(0.66)

        plt.show()
```

```
[ ]: ""#### Determining sequences leading up to shots
from matplotlib.collections import LineCollection

for game in range(1,39):
  for half in ['1H','2H']:
    #list we will put numpy arrays
    action_chain = []

    i = 0
    action_index = 0
    #while index is okay and we are in GAme 1 TIME 1H
    while (i < 59270) :
      #print(i)
      if (full_events_data[i][4] == half)and␣
↪(full_events_data[i][0] == game):
        if (full_events_data[i][1] == 'Huskies'):
          temp = []

          #print()
          #print("Currently on action: " + str(i))
          #print("action index is: " + str(action_index))
          action_index += 1
          #print(full_events_data[i])
          #print("Events in chain: ")
          i += 1
          temp.append(full_events_data[i])
          while(full_events_data[i][1] == 'Huskies') and␣
↪(full_events_data[i][4] == half) and␣
↪(full_events_data[i][0] == game) :
            #print(i)
            i += 1
            temp.append(full_events_data[i])
          #once you exit the chain.
          action_chain.append(temp)

        else: #increasing i if we aren't on a 'good' move
            i += 1
      else:
        i += 1


    print()
    print("PRINTING ACTION CHAIN",game,half)

    for i in range(0, len(action_chain)):
      t= np.arange(0, len(action_chain[i]))
      this_x=[]
      this_y=[]
      this_act=[]
      for j in range(0, len(action_chain[i])):
        if (action_chain[i][j][-4] != 100) and␣
↪(action_chain[i][j][-4] != 0):
```

```
          this_x.append(action_chain[i][j][-4])
          this_y.append(action_chain[i][j][-3])
        else:
          this_x.append(100)
          this_y.append(50)
        this_act.append([action_chain[i][j][2],
                         action_chain[i][j][6],
                         action_chain[i][j][7],
                         " Start ",
                         action_chain[i][j][-4],
                         action_chain[i][j][-3],
                         " End ",
                        ␣
↪action_chain[i][j][-2],action_chain[i][j][-1]])
      #print(this_x,this_y)
      #print([row[2] for row in this_act])
      if len(this_x)>3:
        print("\n","ACTION INDEX: " + str(i))
        for actionz in this_act:
          print(actionz)
        #plt.ylim(0,100)
        #plt.xlim(0,100)
        #plt.show()
        points = np.array([this_x, this_y]).T.reshape(-1, 1,␣
↪2)
        segments = np.concatenate([points[:-1], points[1:]],␣
↪axis=1)
        lc = LineCollection(segments, cmap=plt.
↪get_cmap('plasma'),
        norm=plt.Normalize(0, 10))
        lc.set_array(t)
        lc.set_linewidth(3)
        im = plt.imread("Soccer_Field_Transparant.png")
        implot = plt.imshow(im,extent=[0, 100, 0, 100])
        plt.plot(this_x,this_y, 'x')
        plt.gca().add_collection(lc)
        plt.gca().set_aspect(0.66)
        #ax.add_image(im)
        #plt.xlim(0, 100)
        #plt.ylim(0, 100)
        plt.show()
```

D Teamwork Model

```
[ ]: ""#### Determining sequences leading up to shots
     from matplotlib.collections import LineCollection
     chains={}
     chainlens=[]
     tempo=[]
     for game in range(1,39):
       cls=[]
       tempopg=[]
       halves={}
       for half in ['1H','2H']:
         #list we will put numpy arrays
         action_chain = []

         i = 0
         action_index = 0
         #while index is okay and we are in GAme 1 TIME 1H
         while (i < 59270) :
           #print(i)
           if (full_events_data[i][4] == half)and␣
      ↪(full_events_data[i][0] == game):
             if (full_events_data[i][1] == 'Huskies'):
               temp = []

               #print()
               #print("Currently on action: " + str(i))
               #print("action index is: " + str(action_index))
               action_index += 1
               #print(full_events_data[i])
               #print("Events in chain: ")
               i += 1
               temp.append(full_events_data[i])
               starttime=full_events_data[i][5]
               avgtime=0
```

```
          while(full_events_data[i][1] == 'Huskies') and␣
↪(full_events_data[i][4] == half) and␣
↪(full_events_data[i][0] == game):

            i += 1
            temp.append(full_events_data[i])
            cls.append(len(temp))
            avgtime=(full_events_data[i][5]-starttime)/
↪len(temp)
          if avgtime>0:
            #print(avgtime)
            tempopg.append(avgtime)
          #once you exit the chain.
          action_chain.append(temp)

        else: #increasing i if we aren't on a 'good' move
            i += 1
      else:
        i += 1
    halves[half]=action_chain
  chains[game]=halves
  chainlens.append(np.average(cls))
  tempo.append(np.average(tempopg))

chainlens=np.array(chainlens)
tempo=np.array(tempo)
print(chainlens)
print(tempo)
```

[ ]:
```
tempo=tempo/np.max(tempo)
chain_len=chainlens/np.max(chainlens)
print(chain_len)
print(tempo)
```

[ ]:
```
# current = 1
# game_row = [0]
# for i in range(len(full_events_data)):
#   if full_events_data[i][0] != current:
#     current = full_events_data[i][0]
#     game_row.append(i)
# print(game_row)

num_actions = []
for player in player_name.keys():
  current = []
  n = 0
  r = 1
  for row in full_events_data:
    if row[0] != r:
      current.append(n)
      n = 1
      r += 1
    if row[2] == player:
      n += 1
  current.append(n)
  num_actions.append(current)

print(num_actions)
```

[ ]:
```
goal_diff = []
goals = []
for row in matches_data:
  goals.append(row[3])
  goal_diff.append(row[3] - row[4])
goals=np.array(goals)
goals=goals/np.max(goals)
goal_diff= np.array(goal_diff)
goal_diff= goal_diff/np.max(goal_diff)
print(goal_diff)

r = 1
n = 0
shots_taken = []
for row in full_events_data:
  if row[0] != r:
    shots_taken.append(n)
    n = 1
    r += 1
  if ((row[7] == 'Shot') or (row[7] == 'Free Kick Shot') or␣
 ↪(row[7] == 'Penalty')) and (row[1] == 'Huskies'):
    n += 1
shots_taken.append(n)
print(shots_taken)
shots_taken=np.array(shots_taken)
shots_taken=shots_taken/np.max(shots_taken)
print(shots_taken)
```

```
[ ]: num_actions = num_actions[1:]
     new_action = []
     for i in range(len(num_actions[0])):
       e = []
       for c in range(len(num_actions)):
         e.append(num_actions[c][i])
       new_action.append(e)

     print(new_action)

     standard_dev = [np.std(a) for a in new_action]

     #print(len(standard_dev), standard_dev)

     standard_dev=np.array(standard_dev)

     standard_dev=standard_dev/np.max(standard_dev)
     print(len(standard_dev))
```

```
[ ]:   # define the data/predictors as the pre-set feature names
     df = pd.DataFrame({'g':goals, 's':shots_taken, 'cl2':
      ↪chain_len,'pd2':standard_dev**2,'t':tempo})
     df2= pd.DataFrame({'g':goals, 's':shots_taken})
     # Put the target (housing value -- MEDV) in another DataFrame
     target = pd.DataFrame({'T':goal_diff})
     Xpoints = df
     Xpoints2 = df2
     ypoints = target["T"]
     lm = linear_model.LinearRegression()
     lm2 = linear_model.LinearRegression()
```

```
model = lm.fit(Xpoints,ypoints)
model2= lm2.fit(Xpoints2,ypoints)
predictions = lm.predict(Xpoints)
predictions2= lm2.predict(Xpoints2)
print("Just Goals & Shots:",lm2.
 ↪score(Xpoints2,ypoints),"\n","With TWK:",lm.
 ↪score(Xpoints,ypoints))
```

[ ]:
```
print("T=", lm.coef_, "*",␣
 ↪['goals','shots_taken','chain_len^2','std_dev^2','tempo'])
print("T=",lm2.coef_,"* goals")
```

[ ]:
```
from collections import OrderedDict
player_name = OrderedDict()
for row in full_events_data:
  if row[2] not in player_name:
    if "Huskies" in row[2]:
      player_name[row[2]] = 0
print(player_name)
```

[ ]:
```
r = 1
x = 0
y = 0
game_average = []
player_coord = OrderedDict()
total_player_coord = []
counter = 0
# game number, player, xy
player_counter = [[[0, 0] for i in range(30)] for j in␣
 ↪range(38)]
num_actions = [[0 for i in range(30)] for j in range(38)]
for row in full_events_data:
  counter += 1
  if row[0] != r:
    x = x/counter
    y = y/counter
    game_average.append([x, y])
    x = 0
    y = 0
    r += 1
    counter = 0
  if row[1] == 'Huskies':
    if not np.isnan(row[8]) and not np.isnan(row[9]):
      if row[2] != 'Huskies_G1':
        x += row[8]
        y += row[9]
      else:
        counter -= 1
      #print(r-1, list(player_name.keys()).index(row[2]))
      player_counter[r-1][list(player_name.keys()).
 ↪index(row[2])][0] += row[8]
      player_counter[r-1][list(player_name.keys()).
 ↪index(row[2])][1] += row[9]
      num_actions[r-1][list(player_name.keys()).
 ↪index(row[2])] += 1
game_average.append([x/counter, y/counter])
for i in range(len(player_counter)):
  for j in range(len(player_counter[0])):
    if num_actions[i][j] != 0:
      player_counter[i][j][0] = player_counter[i][j][0]/
 ↪num_actions[i][j]
      player_counter[i][j][1] = player_counter[i][j][1]/
 ↪num_actions[i][j]
for i in range(len(player_counter)):
  for j in range(len(player_counter[0])):
    player_counter[i][j][0] -= game_average[i][0]
    player_counter[i][j][1] -= game_average[i][0]
player_counter = [game[1:] for game in player_counter]
print(game_average)
```

[ ]:
```
# diff between average center of each game and overall average
diff_each_game = [[n[0], n[1] - 50] for n in game_average]
diff_each_game = [(n[0]**2 + n[1]**2)**(1/2) for n in␣
 ↪diff_each_game]
print(diff_each_game)
aggr=np.array(diff_each_game)
aggr=aggr/np.max(aggr)
```

[ ]:
```
# average of distances of each player from center of each game
diff = [[0 for i in range(29)] for j in range(38)]
for i in range(len(player_counter)):
  for j in range(len(player_counter[0])):
    diff_x = player_counter[i][j][0] - game_average[i][0]
    diff_y = player_counter[i][j][1] - game_average[i][1]
    distance = (diff_x**2 + diff_y**2)**1/2
    diff[i][j] = distance
distance_from_center = [np.mean(d) for d in diff]
print(distance_from_center)

distance_from_center=np.array(distance_from_center)
distance_from_center=distance_from_center/np.
 ↪max(distance_from_center)
```

[ ]:
```
num_passes = []
n = 0
r = 1
for row in full_events_data:
  if row[0] != r:
    num_passes.append(n)
    n = 1
    r += 1
  if row[1] == 'Huskies':
    if row[6] == 'Pass':
      n += 1
num_passes.append(n)
print(num_passes)

num_passes=np.array(num_passes)
num_passes=num_passes/np.max(num_passes)
```

[ ]:
```
# define the data/predictors as the pre-set feature names
df = pd.DataFrame({'g':goals, 's':shots_taken, 'cl2':
 ↪chain_len**2,'cl':chain_len,
                   'pd2':standard_dev**2,'pd':
 ↪standard_dev,'p^2':num_passes**2,'p':num_passes, 'con':
 ↪distance_from_center,'t':tempo, 'a': aggr})
df2= pd.DataFrame({'g':goals, 's':shots_taken})
```

```
# Put the target (housing value -- MEDV) in another DataFrame
target = pd.DataFrame({'T':goal_diff})
Xpoints = df
Xpoints2 = df2
ypoints = target["T"]
lm = linear_model.LinearRegression()
lm2 = linear_model.LinearRegression()
model = lm.fit(Xpoints,ypoints)
model2= lm2.fit(Xpoints2,ypoints)
predictions = lm.predict(Xpoints)
predictions2= lm2.predict(Xpoints2)
print("Just Goals & Shots:",lm2.
 ↪score(Xpoints2,ypoints),"\n","With TWK:",lm.
 ↪score(Xpoints,ypoints))
print("T=", str(lm.coef_.round(2)), "\n *",␣
 ↪['goals','shots_taken','chain_len^2','chain_len','std_dev^2','std_dev','pass^2',␣
 ↪'pass','con','tempo', 'aggr'], "+", lm.intercept_.round(5)␣
 ↪)
print("T=",lm2.coef_,"* goals","+",lm2.intercept_)
```

```
[ ]: import statsmodels.formula.api as sm
dfs=  pd.DataFrame({'T':goal_diff,'g':goals, 's':shots_taken,␣
 ↪'cl2':chain_len**2,'cl':chain_len,
                   'pd2':standard_dev**2,'pd':
 ↪standard_dev,'ps2':num_passes**2,'ps':num_passes,
                    'd':distance_from_center,'t':tempo, 'a':␣
 ↪aggr})


dfs2=  pd.DataFrame({'T':goal_diff,'g':goals, 's':
 ↪shots_taken})

result = sm.ols(formula="T ~ g + s + t + d + a + cl2 + cl +␣
 ↪pd2 + pd + ps2 +ps", data=dfs).fit()
result2 = sm.ols(formula="T ~ g + s", data=dfs2).fit()
print(result.params)
print("\n", result2.params)
```

```
[ ]: print(result.summary().as_latex())
```

```
[ ]: print(result2.summary().as_latex())
```

E Goals Study

```
[ ]:
```

```
[ ]: i = 0
     num_events = np.size(full_events_data)


     while (i < 59270):
       if ((full_events_data[i][7] == 'Shot') or␣
      ↪(full_events_data[i][7] == 'Free Kick Shot') or␣
      ↪(full_events_data[i][7] == 'Penalty')) and␣
      ↪(full_events_data[i][1] == 'Huskies') and␣
      ↪(full_events_data[i][0] == 36):
         print()
         print(i)
         print(full_events_data[i][2],full_events_data[i][5],␣
      ↪full_events_data[i][-4], full_events_data[i][-3],␣
      ↪full_events_data[i][-2],␣
      ↪full_events_data[i][-1],full_events_data[i][7])
         print(full_events_data[i+1][2],full_events_data[i+1][5],␣
      ↪full_events_data[i+1][-4], full_events_data[i+1][-3],␣
      ↪full_events_data[i+1][-2],␣
      ↪full_events_data[i+1][-1],full_events_data[i+1][7])
         print(full_events_data[i+2][2],full_events_data[i+2][5],␣
      ↪full_events_data[i+2][-4], full_events_data[i+2][-3],␣
      ↪full_events_data[i+2][-2],␣
      ↪full_events_data[i+2][-1],full_events_data[i+2][7])
         print(full_events_data[i+3][2],full_events_data[i+3][5],␣
      ↪full_events_data[i+3][-4], full_events_data[i+3][-3],␣
      ↪full_events_data[i+3][-2],␣
      ↪full_events_data[i+3][-1],full_events_data[i+3][7])
         print(full_events_data[i+2][5] - full_events_data[i][5])
```

```
    i += 1
```

F Shots Graph

```
[ ]: ######## Shots Graph

     #defining the coordinate arrays
     x_origin_coord = np.array([])
     y_origin_coord = np.array([])
     x_dest_coord = np.array([])
     y_dest_coord = np.array([])

     print(full_events_data[0])

     for row in full_events_data:
       #sort by subaction
       if ((row[7] == 'Shot') or (row[7] == 'Free Kick Shot') or␣
      ↪(row[7] == 'Penalty')) and (row[1] == 'Huskies'):
         x_origin_coord = np.append(x_origin_coord, row[8])
         y_origin_coord = np.append(y_origin_coord, row[9])
         x_dest_coord = np.append(x_dest_coord, row[10])
         y_dest_coord = np.append(y_dest_coord, row[11])

         if(row[10] != 100) and (row[10] != 0):
           print(row)

     num_shots = np.size(x_origin_coord)
     print(num_shots)

     fig, ax = plt.subplots()

     ax.scatter(x_origin_coord, y_origin_coord, color = 'blue',␣
      ↪label='Shot Origin')
     #ax.scatter(x_dest_coord, y_dest_coord, color = 'red', label␣
      ↪= 'Shot Dest.')
     plt.title('Shots Taken Game 1, 1H')
```

```
im = plt.imread("Soccer_Field_Transparant.png")
implot = plt.imshow(im,extent=[0, 100, 0, 100])
plt.gca().set_aspect(0.66)
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.legend()
```

G Graphs and Nodes

[ ]:
```
# number of passes comparison

# first look at match 1
objects = ('Huskies', 'Opponent')
y_pos = np.arange(len(objects))

huskies_counter = 0
enemy_counter = 0
first_match = full_events_data[1:1523]
for p in first_match:
  if p[1] == 'Huskies':
    huskies_counter += 1
  else:
    enemy_counter += 1

plt.bar(y_pos, [huskies_counter, enemy_counter],␣
 ↪align='center', alpha=0.5)
plt.xticks(y_pos, objects)
plt.ylabel('Passes')
plt.title('Number of passes on each team')

plt.show()
```

[ ]:
```

```

[ ]:
```
x_average = []
y_average = []

for player in player_name.keys():
  x_origin_coord = np.array([])
  y_origin_coord = np.array([])
  for row in full_events_data:
    if row[2] == player:
      if not np.isnan(row[8]) and not np.isnan(row[9]):
        x_origin_coord = np.append(x_origin_coord, row[8])
        y_origin_coord = np.append(y_origin_coord, row[9])
  x_average.append(sum(x_origin_coord)/len(x_origin_coord))
  y_average.append(sum(y_origin_coord)/len(x_origin_coord))
print(x_average)
print(y_average)
```

```
[ ]: distance_diff = []
for i in range(len(x_average)):
  temp_diff = []
  for j in range(len(x_average)):
    temp_diff.append(((x_average[i] - x_average[j])**2 +␣
 ↪(y_average[i] - y_average[j])**2)**(1/2))
  distance_diff.append(temp_diff)
print(distance_diff)
```

```
[ ]: for player in player_name.keys():
  x_origin_coord = np.array([])
  y_origin_coord = np.array([])
  for row in full_events_data:
    if row[2] == player:
      x_origin_coord = np.append(x_origin_coord, row[8])
      y_origin_coord = np.append(y_origin_coord, row[9])

  # fig, ax = plt.subplots()

  # ax.scatter(x_origin_coord, y_origin_coord, color =␣
 ↪'blue', label='Pass Origin')
  # plt.title('Player '+player)

  # plt.show()
  plt.clf()
  x = {i:0 for i in range(0, 101, 10)}
  for i in x_origin_coord:
    if not np.isnan(i):
      x[i - i%10] += 1
  plt.bar(range(len(x)), x.values(), align='center')
  plt.xticks(range(len(x)), list(x.keys()))
  plt.xlabel('x-axis')
  plt.ylabel('Count')
  plt.title('Player '+player)
  plt.show()

  plt.clf()
  y = {i:0 for i in range(0, 101, 10)}
  for i in y_origin_coord:
    if not np.isnan(i):
      y[i - i%10] += 1
  plt.bar(range(len(y)), y.values(), align='center')
  plt.xticks(range(len(y)), list(y.keys()))
  plt.xlabel('y-axis')
  plt.ylabel('Count')
  plt.title('Player '+player)
  plt.show()
```

```
[ ]: for player in player_name.keys():
       current_prob = {n:0 for n in player_name.keys()}
       for row in full_events_data:
         if (row[6] == 'Pass') and (row[1] == 'Huskies') and␣
      ↪row[2] == player:
           if row[3] in player_name.keys() and row[3] != row[2]:
             current_prob[row[3]] += 1
       current_prob = {key: value/sum(current_prob.values()) for␣
      ↪key, value in current_prob.items()}
       print(player, current_prob)
       plt.bar(range(len(current_prob)), current_prob.values(),␣
      ↪align='center')
       plt.xticks(range(len(current_prob)), list(current_prob.
      ↪keys()), rotation='vertical')
       plt.xlabel('Player')
       plt.ylabel('Probability of passing to player')
       plt.title('Player '+player)
       plt.show()
```

```
[ ]:
```

```
[ ]: player_pass_list = []
player_name = OrderedDict(player_name)
for player in player_name.keys():
  current_prob = {n:0 for n in player_name.keys()}
  for row in full_events_data:
    if (row[6] == 'Pass') and (row[1] == 'Huskies') and␣
 ↪row[2] == player:
      if row[3] in player_name.keys() and row[3] != row[2]:
        current_prob[row[3]] += 1
  v = [p for p in current_prob.values()]
  print(player)
  player_pass_list.append(v)
print(player_name.keys())
print(player_pass_list)
```

```
[ ]: player_shot_list = []
for player in player_name.keys():
  num_shots = 0
  for row in full_events_data:
    if (row[6] == 'Shot') and (row[1] == 'Huskies') and␣
 ↪row[2] == player:
      num_shots += 1
  player_shot_list.append(num_shots)
print(player_shot_list)
```

```
[ ]: #dont use this!!!!!!!!!!

G=nx.DiGraph() #initializes the graph
# adding Nodes
players = ['G1', 'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'D1',␣
 ↪'D2', 'D3', 'D4', 'D5', 'D6', 'D7', 'D8', 'D9', 'D10',
           'M1', 'M2', 'M3', 'M4', 'M5', 'M6', 'M7', 'M8',␣
 ↪'M9', 'M10', 'M11', 'M12', 'M13', 'Shots', 'Opponent']
#pairs = list(itertools.permutations(players, 2))
```

```
G.add_nodes_from(players)
#G.add_edges_from(pairs, edge_color='red') #we will do edges␣
 ↪with weights below
#pos = nx.spring_layout(G)
#print(pairs)


#labels_dict = {}
#for i in players:
#  labels_dict[i] = i

#nx.draw_networkx_labels(G, pos, labels = labels_dict)




#finding normalizing factors: take the number of edges␣
 ↪exiting a node and divide all elements by this


normalizations = []
index = 0

for player in player_name.keys():
    total_actions = sum(player_pass_list[index]) +␣
 ↪player_shot_list[index]
    print(player +' has action count: ' + str(total_actions))
    index+=1
    normalizations.append(total_actions)
print(normalizations)
```

```
[ ]: G=nx.DiGraph()
G.add_nodes_from(player_name.keys())
#player_pass_list is a 2d array #there should be 30 of each
#player_name.keys() are the keys
print(len(player_pass_list))
print(len(player_name.keys())) #this is a sorted dict
```

```
i = 0 #index of current player
for current_player in player_name.keys():
  j = 0 #index of recipient player
  for recipient_player in player_name.keys(): #iterating␣
 ↪through action

    if (current_player != recipient_player): #making sure we␣
 ↪dont get an edge to ourselves

      prelog_weight = player_pass_list[i][j]/normalizations[i]
      if (prelog_weight != 0.0):
        G.add_edge(current_player, recipient_player, weight =␣
 ↪-np.log(prelog_weight))

    j += 1 #iterating up one

  prelog_weight = player_shot_list[i]/normalizations[i]
  if (prelog_weight != 0.0):
    G.add_edge(current_player, 'Shot', weight = -np.
 ↪log(prelog_weight))
  i += 1

print("Nodes of graph: ")
print(G.nodes())
print("Edges of graph: ")
print(G.edges())
print('Number of edges (directed) is: ' + str(len(G.edges)))
edges,weights = zip(*nx.get_edge_attributes(G,'weight').
 ↪items())
nx.draw(G, edge_color=weights) #displays the graph

plt.show()

print(normalizations)
```

```
[ ]: nx.shortest_path(G,  target='Shot', weight = 'weight',␣
      ↪method='dijkstra')
```

```
[ ]:
```

[ ]:
```
x_average = []
y_average = []

for player in player_name.keys():
  x_origin_coord = np.array([])
  y_origin_coord = np.array([])
  for row in full_events_data:
    if row[2] == player:
      if not np.isnan(row[8]) and not np.isnan(row[9]):
        x_origin_coord = np.append(x_origin_coord, row[8])
        y_origin_coord = np.append(y_origin_coord, row[9])
  x_average.append(sum(x_origin_coord)/len(x_origin_coord))
  y_average.append(sum(y_origin_coord)/len(x_origin_coord))
print(x_average)
print(y_average)
print(player_name.keys())
```

[ ]:
```
count = 0 #iterate through all the players
distance_sq = []

goal_x = 100 #goal x position
goal_y = 50  #goal y position

for player in player_name.keys():
  temp= (x_average[count] - goal_x)**2 + (y_average[count] -␣
 ↪goal_y)**2 #calculating distance squared
  distance_sq.append(temp)
  count += 1

print(distance_sq)
#we want the closest to the goal to have the highest␣
 ↪multiplier. = 1
#we will first take 1 / distance such that the closest has␣
 ↪the largest number.
#from there, we take the max value in the inverse array and␣
 ↪divide the entire thing

distance_sq_array = np.asarray(distance_sq)
distance_inverse = 1 / distance_sq_array**2
```

```
multipliers = distance_inverse / np.amax(distance_inverse)
print(multipliers)
```

[ ]:
```
#We calculate probabilities, normalized, and take -np.log()
#for shots, we also multiply by the multiplier.

G2=nx.DiGraph()
G2.add_nodes_from(player_name.keys())
#player_pass_list is a 2d array #there should be 30 of each
#player_name.keys() are the keys
print(len(player_pass_list))
print(len(player_name.keys())) #this is a sorted dict


i = 0 #index of current player
for current_player in player_name.keys():
  j = 0 #index of recipient player
  for recipient_player in player_name.keys(): #iterating␣
 ↪through action

    if (current_player != recipient_player): #making sure we␣
 ↪dont get an edge to ourselves

      prelog_weight = player_pass_list[i][j]/normalizations[i]
      if (prelog_weight != 0.0):
        G2.add_edge(current_player, recipient_player, weight␣
 ↪= -np.log(prelog_weight))

    j += 1 #iterating up one

  prelog_weight = player_shot_list[i]/normalizations[i]
  if (prelog_weight != 0.0):
    G2.add_edge(current_player, 'Shot', weight = -np.
 ↪log(multipliers[i] * prelog_weight))
  i += 1

print("Nodes of graph: ")
print(G2.nodes())
print("Edges of graph: ")
print(G2.edges())
print('Number of edges (directed) is: ' + str(len(G2.edges)))
edges,weights = zip(*nx.get_edge_attributes(G2,'weight').
 ↪items())
nx.draw(G2, edge_color=weights) #displays the graph



plt.title('Action Graph with Weights and Multipliers')
plt.show()

print(normalizations)
```

[ ]:
```
nx.shortest_path(G2,  target='Shot', weight = 'weight',␣
 ↪method='dijkstra')
```

[ ]:
```
```

[ ]:
```
```

H Weighted Djikstras

This cell is a directed graph for Djikstras We take the probabilities of actions over all games. Then, we normalize. We also account for distances between players (using average distances)

For shots, we also have a normalizer for this. The farther someone is from a goal, the less beneficial the shot.

[ ]:
```
# Multipliers for Shots
count = 0 #iterate through all the players
distance_sq = []

goal_x = 100 #goal x position
goal_y = 50  #goal y position

for player in player_name.keys():
  temp= (x_average[count] - goal_x)**2 + (y_average[count] -␣
 ↪goal_y)**2 #calculating distance squared
```

```
  distance_sq.append(temp)
  count += 1

print(distance_sq)
#we want the closest to the goal to have the highest␣
 ↪multiplier. = 1
#we will first take 1 / distance such that the closest has␣
 ↪the largest number.
#from there, we take the max value in the inverse array and␣
 ↪divide the entire thing

distance_sq_array = np.asarray(distance_sq)
#print(distance_sq_array)
distance_inverse = 1 / distance_sq_array**10
#print(distance_inverse)
shot_multipliers = distance_inverse / np.
 ↪amax(distance_inverse)
print(shot_multipliers)

#note that from here we have an array of distance squared.
#rather than calculating multipliers for passes into an␣
 ↪array, I will calculate them while generating weights

#we need only find the normalization factor. We do this by␣
 ↪the following
#find the two closest people such that delta_r is small.␣
 ↪Then, take 1/ delta_r^2. That is our normalizer.

diff = 100000000000000000000 #effectively infinite

id1 = 0
for id1 in range(0, len(distance_sq)): #using the list for␣
 ↪length
  for id2 in range(id1+1, len(distance_sq)):
    temp = abs(distance_sq[id1] - distance_sq[id2])
    if ( temp < diff):
      diff = temp
```

```
print(diff)
#diff is the smallest difference
pass_norm = 1 / (diff)

#Thus, between two players, we should take the absolute value␣
↪of the distance squared.
#we then take 1 / delta^2
```

[ ]:
```
#We calculate probabilities, normalized, and take -np.log()
#for shots, we also multiply by the multiplier.

###Changed 1:20



#############
G2=nx.DiGraph()
G2.add_nodes_from(player_name.keys())
#player_pass_list is a 2d array #there should be 30 of each
#player_name.keys() are the keys
print(len(player_pass_list))
print(len(player_name.keys())) #this is a sorted dict


i = 0 #index of current player
for current_player in player_name.keys():
  j = 0 #index of recipient player
  for recipient_player in player_name.keys(): #iterating␣
↪through action

    if (current_player != recipient_player): #making sure we␣
↪dont get an edge to ourselves

      prelog_weight = player_pass_list[i][j]/normalizations[i]
      if (prelog_weight != 0.0):
        #we need to find the pas_multiplier
        pass_multiplier =  (1 / (  (abs(distance_sq[i] -␣
↪distance_sq[j])) ) / pass_norm

        current_type = current_player[8]
        recipient_type = recipient_player[8]

        #benefit is a multiplier for penalties, Closer␣
↪players have a smaller multipier (for shortest path)
        # ball prioritizes moving forward
        #closest to 1 indicates most favorable pass

        if (current_type == 'D'):
          if (recipient_type == 'G'):
            benefit = 0.000000001
          if (recipient_type == 'D'):
            benefit = 1.0
          if (recipient_type == 'M'):
            benefit = 1.0
          if (recipient_type == 'F'):
            benefit = 0.0000001

        if (current_type == 'G'):
          if (recipient_type == 'G'):
            benefit = 0.000000001
          if (recipient_type == 'D'):
            benefit = 1.0
          if (recipient_type == 'M'):
            benefit = 0.0001
          if (recipient_type == 'F'):
            benefit = 0.0000001

        if (current_type == 'M'):
          if (recipient_type == 'G'):
            benefit = 0.000000001
          if (recipient_type == 'D'):
            benefit = 0.9
          if (recipient_type == 'M'):
            benefit = 1.0
          if (recipient_type == 'F'):
            benefit = 0.9

        if (current_type == 'F'):
          if (recipient_type == 'G'):
            benefit = 0.000000001
          if (recipient_type == 'D'):
            benefit = 0.01
          if (recipient_type == 'M'):
            benefit = 0.1
          if (recipient_type == 'F'):
            benefit = 1.0
        #benefit is a multiplier for penalties, Closer␣
↪players have a smaller multipier (for shortest path)
        G2.add_edge(current_player, recipient_player, weight␣
↪= -np.log(prelog_weight * pass_multiplier * benefit))
      #print('HERE')

    j += 1 #iterating up one

  prelog_weight = player_shot_list[i]/normalizations[i]
  if (prelog_weight != 0.0):
    if (current_type == 'G'):
      shot_loss = 0.0000001
    if (current_type == 'D'):
      shot_loss = 0.0001
    if (current_type == 'M'):
      shot_loss = 0.1
    if (current_type == 'F'):
      shot_loss = 1.0
    G2.add_edge(current_player, 'Shot', weight = -np.
↪log(shot_multipliers[i] * prelog_weight * shot_loss))
  i += 1

print("Nodes of graph: ")
print(G2.nodes())
print("Edges of graph: ")
print(G2.edges())
print('Number of edges (directed) is: ' + str(len(G2.edges)))
edges,weights = zip(*nx.get_edge_attributes(G2,'weight').
↪items())
nx.draw(G2, edge_color=weights) #displays the graph
```

```
plt.title('Action Graph with Weights and Multipliers')
plt.show()

print(normalizations)

nx.shortest_path(G2,  target='Shot', weight = 'weight',␣
 ↪method='dijkstra')
```

I Player Locations

```
[ ]: X = x_average
     Y = y_average

     j=0
     forwards = [(X[1],Y[1]), (X[4],Y[4]), (X[10],Y[10]),␣
      ↪(X[19],Y[19]), (X[25],Y[25]), (X[26],Y[26])]
     defenders = [(X[2],Y[2]), (X[5],Y[5]), (X[8],Y[8]),␣
      ↪(X[9],Y[9]), (X[11],Y[11]), (X[14],Y[14]), (X[20],Y[20]),␣
      ↪(X[27],Y[27]), (X[28],Y[28]),
                  (X[29],Y[29])]
     midfield = [(X[3],Y[3]),(X[6],Y[6]),(X[7],Y[7]),(X[12],Y[12]),
                 ␣
      ↪(X[13],Y[13]),(X[15],Y[15]),(X[16],Y[16]),(X[17],Y[17]),
                 ␣
      ↪(X[18],Y[18]),(X[21],Y[21]),(X[22],Y[22]),(X[23],Y[23]),
                 (X[24],Y[25])]

     fX = []
     fY = []
     dX = []
     dY = []
     mX = []
     mY = []

     for i in range(0, len(forwards)):
       fX.append(forwards[i][0])
```

```
  fY.append(forwards[i][1])

for i in range(0, len(defenders)):
  dX.append(defenders[i][0])
  dY.append(defenders[i][1])

for i in range(0, len(midfield)):
  mX.append(midfield[i][0])
  mY.append(midfield[i][1])

plt.scatter(fX, fY, color = 'r', label = 'Forwards')
plt.scatter(dX, dY, color = 'b', label = 'Defenders')
plt.scatter(mX, mY, color = 'g', label = 'Midfielders')
plt.xlabel('X position')
plt.ylabel('Y position')
plt.title('Average Position of Players')
plt.legend()
plt.show()
```

```
[ ]:
```