Capstone Project: Using Machine Learning to Predict Optimal Soccer Positions

Suraj Daru
CSCI 1420

May 15, 2024

1 Abstract

Soccer is the most popular sport in the world, and scouting/team management are huge components at the professional level. Figuring out how to most optimally create a lineup, what type of substitutions to make, and how to determine the optimal position for players are just a few aspects of analysis that are integral to a successful team. In this paper I chose to focus on the later, using Neural Networks, K-Nearest Neighbors, and Principal Component Analysis to determine the most optimal positions based on Premier League dataset from Kaggle. I was able to create a model that accurately classifies outfield positions at $\sim$ 86% within the testing set after using cross-validation-loss to determine appropriate hyperparameters. Interestingly, the Linear Model performed just as well as the Nonlinear Model, and the most optimal KNN model did ever so slightly better – PCA resulted in information lost that reduced the testing accuracies. This provides a good point to improve upon and expand on classifying more specific player positions with an expanded dataset.

2 The Problem (Continued)

There are certain characteristics that pertain to defenders in particular, such as high tackling accuracy, headers won, and blocks. Likewise, midfielders are known for their pass completion, assists, and long balls to name a few. Lastly, those playing forward are expected to have a relatively high conversion rate on chances, with a higher amount of goals (and thus more shots).

Thus, I thought that the problem of choosing an optimal outfield position for a given soccer player can definitely be informed by such statistical measures. This problem takes on the structure of supervised, multi-class learning, so it would be important to consider how to adjust any raw data so a classification model could be applied.

3 Initial Approach/What worked and did not work/Changes Made

In order to solve this problem, the first choice would be to obtain a dataset that adequately describes soccer positions with various features. I decided to use this dataset from Kaggle that contains 59 total features and 571 rows (each corresponding to a player). This is particular to the Premier League, which is the highest level of soccer in England, and considered to be one of the most competitive professional leagues in the world.

I thought that this was a good representation, considering it had features such as the ones listed in the previous section that correspond to various positions, as well as the position labels themselves.

Before proceeding to make any models, I had to preprocess the data. There were a few components to this – firstly, I had to filter out the rows that corresponded to goalkeepers, as I wanted to focus on the outfield for this project. Then, I needed to get rid of any columns/features that are only relevant for goalkeepers, such as catches, saves, and sweeper clearances. Moreover, I needed to get rid of other columns that aren't indicative of positions and are dependent on many other factors, such as total wins, losses, and appearances.

Then, I needed to make sure that there were no "nan" values, and that any percentage strings were converted to floats. There were also some specific comments on the dataset by the creator about a particular column that was buggy (Goals per match) that I had to edit. Lastly, the labels of each position needed to be converted to a numerical value (0,1,2) so that they could be used in each model.

In terms of creating the models, I decided to use two different approaches: K-Nearest Neighbors, as well as Neural Networks. My K-Nearest Neighbors implementation uses data that has been split into training and testing groups (the same that are passed into the neural networks), and uses Euclidean Distance in order to find the closest k points in order to make a classification based on the majority label.

The process of creating the neural networks was similar to the deep learning assignment, however I had to change a few things. Firstly, I decided that it would be most appropriate to use a densely connected, forward feed neural network. This is as opposed to a convolutional neural network, as it is typically used for visual data like images (moreover, it wouldn't make sense to convert the tabular data to images in order to use this as such images would have no meaning: the features themselves are spatially independent from each other).

One issue I ran into was getting the right shapes within the output of my neural networks. Once I had preprocessed, there were 42 columns left, which was the amount of features, and since I had 3 categories, I made sure to have my last layer be of length 3. After looking further into the issue, I realized that the problem was my loss function.

This is because when dealing with labels (0, 1, 2), it would be a classification task and not a regression task. Thus, it would make sense to use cross entropy loss as our loss function instead of mean squared error which I was previously using (this is for continuous values, not discrete labels).

I also created another function to run a new custom example through the model – I encountered an issue here with shapes again. This is because we need to artificially construct the batch the be of size 1, which is required in the shape of the tensor when being evaluated by the model.

Thus, I experimented with the one layer neural network that I used for the deep learning assignment, and compared it to one with more parameters and nonlinear components. I also realized that regardless of the model used, I would keep the same random state as another control variable, and not shuffle the data when training/testing – this applies to the validation k-fold splits as well.

In terms of the K-Nearest-Neighbors component, I did not encounter any significant issues – I ended up using a map to keep track of the nearest points rather than a heap in my implementation. The only hyperparameter I experimented with here was the value of k.

Lastly, I had to take a look into the documentation for how to use scikit learn's PCA implementations. I then used this in order to reduce the amount of dimensions to 3. After this, I looked into the matplotlib documentation to figure out how to plot it most clearly with the labels.

4 Final Results

I decided to evaluate my models by using K-Fold Cross Validation – this was because the dataset was a bit small for my liking, and thought that this would maximize utility of the data. I did this while varying the hyperparameters in search for the most optimal set. This involved creating a list of parameter combinations with particular discrete values. This was a bit subjective, but I chose general values that I tinkered with and saw differences in test accuracies with to be within these sets of parameters. Table 1 illustrates several combinations that I iterated through and measured accuracies for. Next, I used PCA to reduce the dimensions and complexity to only have 3 categories. After doing this, I used the same hyperparameter combinations to see if PCA could further reduce loss.

Table 1: Hyperparameter Combinations and Cross-Validation Losses

Model Type	**Num Epochs**	**Learning Rate**	**Batch Size**	**Hidden Size**	**Cross Validation Loss**
Nonlinear	15	0.005	2	5	0.142
Nonlinear	15	0.005	8	5	0.228
Nonlinear	15	0.005	2	25	0.144
Nonlinear	15	0.005	8	25	0.2
Nonlinear	15	0.005	2	75	0.148
Nonlinear	15	0.005	8	75	0.192
Nonlinear	15	0.01	2	5	0.14
Nonlinear	15	0.01	8	5	0.172
Nonlinear	15	0.01	2	25	0.138
Nonlinear	15	0.01	8	25	0.182
Nonlinear	15	0.01	2	75	0.138
Nonlinear	15	0.01	8	75	0.182
Nonlinear	15	0.1	2	5	0.184
Nonlinear	15	0.1	8	5	0.188
Nonlinear	15	0.1	2	25	0.144
Nonlinear	15	0.1	8	25	0.198
Nonlinear	15	0.1	2	75	0.184
Nonlinear	15	0.1	8	75	0.186
Nonlinear	75	0.005	2	5	0.144
Nonlinear	75	0.005	8	5	0.176
Nonlinear	75	0.005	2	25	0.14
Nonlinear	75	0.005	8	25	0.16
Nonlinear	75	0.005	2	75	0.13
Nonlinear	75	0.005	8	75	0.174
Nonlinear	75	0.01	2	5	0.152
Nonlinear	75	0.01	8	5	0.168
Nonlinear	75	0.01	2	25	0.144
Nonlinear	75	0.01	8	25	0.166
Nonlinear	75	0.01	2	75	0.152
Nonlinear	75	0.01	8	75	0.156
Nonlinear	75	0.1	2	5	0.16
Nonlinear	75	0.1	8	5	0.17
Nonlinear	75	0.1	2	25	0.192
Nonlinear	75	0.1	8	25	0.17
Nonlinear	75	0.1	2	75	0.162
Nonlinear	75	0.1	8	75	0.19
Linear	15	0.005	2	5	0.148
Linear	15	0.005	8	5	0.19

Table 1: Hyperparameter Combinations and Cross-Validation Losses

Model Type	**Num Epochs**	**Learning Rate**	**Batch Size**	**Hidden Size**	**Cross Validation Loss**
Linear	15	0.01	2	5	0.146
Linear	15	0.01	8	5	0.184
Linear	15	0.1	2	5	0.17
Linear	15	0.1	8	5	0.18
Linear	75	0.005	2	5	0.152
Linear	75	0.005	8	5	0.178
Linear	75	0.01	2	5	0.144
Linear	75	0.01	8	5	0.174
Linear	75	0.1	2	5	0.2
Linear	75	0.1	8	5	0.176

Moreover, I evaluated my K-Nearest-Neighbor model on several values of k, and have included the cross-validation losses in Table 2.

k	**Cross Validation Loss**
1	0.19
2	0.595
3	0.415
5	0.442
10	0.325
25	0.545
50	0.518
100	0.565

Table 2: KNN k values and Losses

Thus, the cross validation loss was minimized at k=1, which is interesting since we would most associate this to have high variance. I then ran this model on the leave-out test set and got an accuracy of 86.139%.

On the other hand, the most optimal Nonlinear model from Table 1 has a cross-validation loss of 0.13 (Nonlinear, 75 epochs, learning rate of 0.005, batch size of 2). I ran this model on the leave-out test set and got accuracy values that were either 86.139%, 87.129%, 85.145%, and 84.149% which must be due to a few points being exactly on some decision boundary causing it to be slightly ambiguous – on average it is 85.821%. I also ran the cross-validation-loss minimizing Linear model, and received the same situation of being slightly different but around 86% each time.

Thus, we can draw an interesting conclusion from the performances of the models: it appears that the less complex linear neural network with the appropriate hyper parameter choices achieved the same accuracy on the leave out test set, even though the cross-validation was slightly higher. This corresponds to a multiclass-logistic regression model, and implies that adding nonlinearity and perhaps excess amount of neurons are unnecessary and would only lead to higher training accuracies and 'memorization of the data'.

In fact, if we run the most optimal Nonlinear model at epochs of 15, 50, 75, and 150 we get final training accuracy values of 87.12%, 88.78%, 89.027%, and 89.28% with test accuracies staying the same, showing how overfitting is taking place. The training accuracies truly start to diverge from the test accuracies at around epoch 10, and taper off in the early 100s. I found that this type of overfitting did not occur as dramatically with the most optimal linear model, which had a final training accuracy of 87.781% after 150 epochs, which does not show as significant of a deviation from the testing accuracy.

Moreover, using KNN, which only looks at the geometrically closest neighbors rather than creating a

complex decision boundary did slightly better at classifying the testing set even though it had a relatively higher cross validation loss.

I then used PCA to reduce the dimension from 42 to 3, and ran the same hyperparameter sets. I have included a select few of the lowest in Table 3.

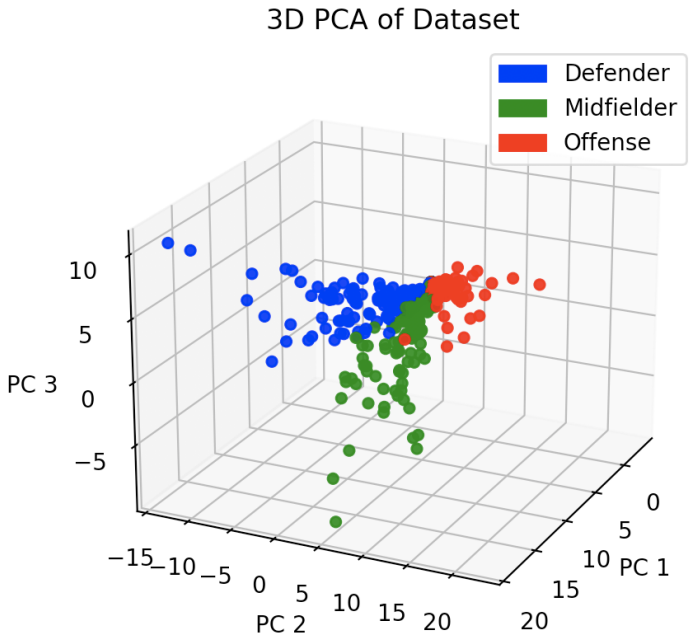

Figure 1: PCA of Dataset to 3 Dimensions

Table 3: Hyperparameter Combinations and Cross-Validation Losses for Reduced Dimension Data

Model Type	**Num Epochs**	**Learning Rate**	**Batch Size**	**Hidden Size**	**Cross Validation Loss**
Nonlinear	15	0.005	2	75	0.16
Nonlinear	75	0.005	2	5	0.16
Nonlinear	75	0.005	2	25	0.158
Nonlinear	75	0.005	2	75	0.158
Linear	75	0.01	2	N/A	0.16

I ran the two loss minimizers here with the leave-out test set and received an accuracy of 83.168% for both. Thus, reducing the dimensions seems to have caused certain important characteristics of the data to be lost, as there were no models that achieved a lower cross validation loss than 0.136, and the result cross validation loss minimizer received a lower accuracy on the leave-out set.

In terms of how PCA affected KNN, the cross validation losses are shown in table 4. I ran the cross validation loss minimizer model on the leave-out test set and received an accuracy of 84.158%. Thus, it appears that PCA also hindered KNN's classification.

Overall though, considering we have 3 classes and a random guess would have an accuracy of about 33.3%, we have achieved a model significantly above the baseline at $\sim$ 86%. Thus, I would say that the most optimal KNN, Linear, and Nonlinear models generalize the data quite well.

k	Cross Validation Loss
1	0.205
2	0.595
3	0.435
5	0.455
10	0.338
25	0.538
50	0.522
100	0.568

Table 4: KNN k values and Losses for Reduced Dimension Data

5 How I Would Improve

The first area that I could improve about my project is trying to find data that could expand this current dataset, which is a little smaller than I would have liked. The reason for this is that although the Premier League is considered one of the most competitive in the world, there are distinct types of playing styles that come across in different leagues, which were not represented here. Thus, the models that I trained only have exposure to about 20 teams, which is once again not representative of soccer as a whole. Thus, even if the data wasn't as specific as this one (which is what drew me to it, because it had a plethora of relevant fields per player), there could be a way to augment the original set to include more information. For example, miles covered per game could be a helpful statistic. Also, when trying to figure out the most optimal position, it might be helpful to have multiple rows per player in which there are different statistics for when they have played in different areas. Lastly, within the preprocessing, perhaps there could be some sort of algorithm to determine how 'important' a player has been for the team, which could take into account previous statistics, and add this as a feature. One could then use this to filter through the raw data to only use rows with a threshold past this value for consistency.

Building off of the previous point, with more/augmented data, I believe that the models could be trained to be more specific about positions. In this project, I have discretized the positions to strictly defense, midfield, and offense, however, considering that there are 10 outfield players, there are a multitude of different specific positions that can be given to a player. For example, in midfield, we can have an attacking midfielder, defensive midfielder, left midfielder, right midfielder, etc. As mentioned, this would require more data to be more specific in classification (it would be multi-class classification with more than 3 categories), so fundamentally it is a data/preprocessing issue before we would think of matching the new data with a more complex model.

Next, I would improve my hyperparameter search. Although my table above looks extensive with various hyperparameter combinations, these are merely a few possibilities – it is just due to the fact that as we add more hyperparameters categories, the amount of possibilities increases exponentially. Thus, given a significant amount of processing power from GPUs, I believe that I would be able to search more extensively for the most optimal set.

Along the same lines, given higher computing power means that I would be able to experiment extensively with more complicated models. This includes adding more layers, and using different activation functions in different combinations. This would depend on how complex the adjusted dataset would be: the reason why I held off on including these attributes with the current dataset is because it already seemed that the nonlinear model was starting to overfit with about 10 epochs, with the testing accuracy staying the same as the linear model's. Therefore, although it could result in a more optimal model, it appears that increasing complexity by adding more layers and nonlinear components would only help marginally up to a certain point, after which they are detrimental and add unnecessary computation.