Predict a Soccer Player's Overall Skill

Arturo Perez
Deep Learning CS 3891
30 April 2018

Abstract

This project deals with predicting a soccer player's overall skill using several different classification algorithms. We take advantage of the benefits of TensorFlow's built in estimators as well as create our own custom estimator. Our goal is to build a model which will have the highest accuracy rate.

1 Introduction

Known by many as the "beautiful game", Soccer is a global pastime as well as a professional sport that joins together people from all walks of life in both comradery and competition. I have decided to do my project in this theme to commemorate the start of the World Cup in June 2018.

Every sport has players who dominate the sport, Soccer is no exception. Knowing who these players are can help teams and coaches determine who they will give a spot to on their rosters. It is therefore critical that they make informed decisions.

My project's aim is to make this job a little bit easier: given a player's **proficiency at different skills**, my model will predict what their **overall rating** is on a scale [0,100). If my model's predicted overall rating falls within a range of the actual overall rating, then that prediction is considered a success.

2 The Problem and My Idea

The dataset that is used for this project was provided by Kaggle and is called the European Soccer Database. The dataset was provided in the form of a SQL database and was opened, navigated, and read by using Pandas. It provided over 180,000 unique samples for 10,000 players over a span of 8 years. This meant some players were "repeated" samples, but their skills

attributes changed over time as they improved, worsened, aged, changed teams, changed leagues, etc. Here is what a sample looked like:

player_fifa_api_id: 7763
overall_rating: 87 *(used as the label)*
potential: 87
crossing: 85
finishing: 64
heading_accuracy: 56
short_passing: 93
` volleys: 76
dribbling: 80
free_kick_accuracy: 88
long_passing: 95
ball_control: 94
acceleration: 63
sprint_speed: 55
agility: 67
reactions: 88
balance: 81
shot_power: 76
jumping: 33
stamina: 73
strength: 63
long_shots: 84
interceptions: 76
positioning: 58
vision: 94
penalties: 80
marking: 56
standing_tackle: 64
sliding_tackle: 64

As is shown above, any given sample in the database had 28 skills attributes associated with them, all on a scale of [0,100). This naturally lead to a data matrix with dimensions (181265, 28) – a lot of good data for my model to use! This also leads naturally to another question: should all skills be treated equally? Some skills may objectively be more important than others: for example, overall stamina can be said to be more important than how high a player can jump when a player has to run on average 10km per soccer match. However my model does not place emphasis on any skill over the other: they are all weighed as equally important to the overall rating of a player. I will later discuss how I did manage to place "emphasis" on certain skills, and in the conclusions will add more remarks on how actually

weighting skills to be more important than others can potentially improve the results of my model.

The learning problem for my model then becomes this: using each player's overall_rating as a label, can my model accurately predict a player's overall rating using the other 27 skills ranging from "potential" all the way to "sliding_tackle." Right away I realized I have several ways to go about this: I can train my model using all 27 skills, or I can place "emphasis" on certain skills by training the model on only a selected subset on the 27 skills. Ahead of time there was no way to know whether the model will predict better with all of the skills or only subsets of skills.

I also realized on early on that because this is essentially a classification problem with classes=100, it would be impossible for my model to accurately predict exact overall ratings and still expect to get a decent accuracy rate. Instead, my model is accurate if the predicted overall rating falls within a range of the actual overall rating. For example:

Range = ±7	Predicted Rating: 92	Actual Rating: 89	Prediction: Success
Range = ±7	Predicted Rating: 92	Actual Rating: 80	Prediction: Failure

3 Details

My model is actually 2 different learning models – one is a built-in TensorFlow DNN Classifier while the other is a custom estimator. My goal was to see which one performs better on the 27 skill classification problem. Let's begin by looking at how TensorFlow's DNN Classifier performed as I varied the number of layers and the number of hidden units. Here I am using all 27 skills, 100,000 training examples, 10,000 test examples:

- Note that for all subsequent tables, the accuracy % taken was the highest of 5 trials.

# of layers	units	Accuracy % (range = 5)
1	[100]	67.38%
2	[100, 100]	77.23%
2	[250, 100]	77.87%
3	[50, 150, 100]	81.2%
4	[150, 300, 200, 100]	82.8%

Let's compare these results with my custom estimator. We shall start by having no sort of regularization enabled, but Adam optimizer is enabled.

# of layers	units	Accuracy % (range = 5)
1	[100]	58.52%
2	[100, 100]	59.31%
2	[250, 100]	63.65%
3	[50, 150, 100]	63.72%
4	[150, 300, 200, 100]	59.58%

The TensorFlow DNN Classifier easily out beat the custom estimator on all configurations of layers and hidden units. Let's see if by adding dropout the custom estimator can perform better. Again, I am using 27 skills, 100,000 training examples, 10,000 test examples:

Custom Estimator with Dropout

# of layers	units	Accuracy % (range = 5)
1	[100]	62.64%
2	[100, 100]	62.98%
2	[250, 100]	60.41%
3	[50, 150, 100]	61.16%
4	[150, 300, 200, 100]	60.02%

Even with enabling dropout the custom estimator does not perform nearly as well as TensorFlow's DNN Classifier. Moving forward I will only be using TensorFlow's DNN Classifier. Now that I have identified the model I want to use, let's see how much I can increase the accuracy % by changing hyperparameters: first let's see how changing how many training examples we use affects the overall accuracy %. I will be using all 27 skills with a 4 layer DNN.

# of training samples	Accuracy % (range = 5)
10,000	65.22%
50,000	83.54%
100,000	78.91%
150,000	83.71%

Varying the training sample size only mattered when there were a small training size, but it did not matter much after a certain threshold of training samples was used (in this case, 50,000+).

Now comes the interesting question. Will the model perform better on subsets of skills as opposed to using all 27 skills for training? To put this to the test, I have randomly selected subsets of 5, 10, 15 skills.

# of skills	Accuracy % (range = 5)
5 (potential, finishing, ball_control, dribbling, stamina)	84.39%
5 (crossing, finishing, vision, dribbling, strength)	80.54%
5 (sliding_tackle, potential, long_passing, interceptions, strength)	81.11%
10	80.46%
10	77.83%
15	82.14%
27	83.7%

Interestingly enough there was no significant difference between subsets of small sizes, subsets of larger sizes, or using all 27 skills at once. The model was more impacted by changing the number of training samples. To conclude this section let us compare overall rating output from the model to the actual overall rating given as labels:

Predicted	Actual
46	43.0
46	43.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
60	55.0
67	65.0
67	65.0
67	65.0
67	65.0
67	65.0
67	65.0
67	65.0
67	65.0
67	66.0
67	66.0
64	66.0
64	66.0
64	66.0

4 Concluding Remarks

Using the European Soccer Database I was able to create a learning model that is able to successfully predict a soccer player's overall skill with <85% accuracy on a range of 5. If I push up the range to 8, I can even push <95% accuracy. In the end TensorFlow's built in DNN Classifier outperformed my custom estimator. After scouring the web for potential reasons on why that is, TensorFlow's built in estimators are already predefined with the best coding practices and pre-optimized. The TensorFlow community even recommends one uses built in estimators over custom iterators whenever possible. The TensorFlow built in estimator gave me great as well as fast results. One last thing: Even though this was beyond the realm of this assignment and frankly my abilities, it would be interesting to see how giving more priority to skills that objectively matter more in Soccer could have affected the learning model. The way I did it (by just choosing randomly subsets of skills) did not have much overall affect. If, for example, the learning model could understand to give twice as much priority to stamina over jumping, then perhaps the model could have predicted the overall rating with higher accuracy.