Two Approaches to Making Predictions in Fantasy Sports

Sajid Farook

June 2023

1 Introduction

I'm a big fan of Fantasy Premier League Football (FPL), but I've been pretty underwhelmed with how data is used by the community to inform our play. It seems like people rely either solely on their own intuition and experience as a fan of the sport, or they very selectively integrate data into their predictions without any consideration of *which* variables ought to be given more or less weight. There are some predictive models out there, but they are almost always hidden behind a paywall and are very inaccessible, so I thought it's due time to make one myself.

For those who aren't familiar with fantasy sports, they are a type of online game where we virtually select real-life players that we expect to perform well in their upcoming matches. If we're right (and the players we pre-selected indeed go on to score a goal and win man of the match, for instance), we get rewarded a ton of in-game points, but if we're wrong (and the players we chose end up performing badly), we score less points. This process repeats every week and, naturally, the goal is to accumulate as many points as possible. It's all about being able to make accurate prediction about future sports outcomes. For this project, I'm focusing on Fantasy Premier League Football (which is a fantasy soccer game, but the game is based in the UK, so I'll be referring to it as 'football'), but the approach I discuss here can be applied to any fantasy game, or really any form of sports betting. There over 11 million people around the world all competing against each other in FPL, so the competition is definitely tough.

The good news, though, is that I take CS109 and they don't :) In this write-up, I'm going to explain how I created two models to predict outcomes in FPL using probability concepts we learned throughout the course.

2 Objective

Ultimately, the goal here is to create a probabilistic model that predicts the number of points a football player will score in their next game. We'll model this variable as Y and condition on a number of random variables that future points are thought to be dependent on. They are the following:

- Opponent difficulty (O). The strength of the team they will be playing, measured by the average number of goals they concede per match. Intuitively, it is easier to score more fantasy points against weaker opposition. This approximately takes on values from 0.5-2.8.
- Overall quality (Q): This is the most obvious predictor; how 'good' a player actually is. This will be measured by the total number of FPL points they have accumulated throughout the season up until the point that they play the match in question. This approximately takes on values from 1-10 (with how decimals are rounded being up to my discretion), but the overwhelming majority of plausible data points would be from around 2-7.
- Form (F): The form (more recent performances) of the player, measured by their average FPL points per game in the last 5 games they played. Like in all activities, football/soccer players tend to have alternate in periods of consistently good/bad performances, so this is an important consideration. This it the same as Q, but on a shorter time frame, so once again, this approximately takes on values from 1-10 (with how decimals are rounded being up to my discretion).
- xGi (X): This is a popular statistic used in football data analytics that stands for "expected goal involvement". Sports analytic companies calculate the 'expected' number of goals/assists a player will get by weighting every shot by the probability that it will result in a goal, and then sum it for an "xGi" value. The higher the xGi, the more likely you scored or assisted a goal, and the more likely you scored higher FPL points. This approximately takes on values from 0-1.5, with how decimals are rounded being up to my discretion.

Ultimately, then, I'm trying to calculate the following:

$$E[Y|O = o, Q = q, F = f, X = x] = \sum_y Y \frac{P(y,o,q,f,x)}{P(o,q,f,x)}$$

I found a dataset of about 20,000 entries of historical FPL points and most of the data I needed off Kaggle[1]. The problem is that this isn't nearly enough data to calculate probabilities for each of the approximately 180,000,000 combinations of y, o, q, f, and x. I could probably bring that number down considerably with some rounding and by factoring in the fact that opponent difficulty,

[1] https://www.kaggle.com/ritviyer/fantasy-premier-league-dataset

O, is independent of the other predictor feature variables, but it wouldn't be nearly enough.

I came up with two solutions and decided to test both out of curiosity for which one would yield better results - the first ignores dependence of the feature variables in order to preserve precision, while the other assumes dependence at the cost of some granularity. Let's explore the results in turn:

3 "Naive Expectation"

The problem I was encountering was pretty much identical to the one that justified Naive Bayes, where we assume conditional independence of the feature variables on Y to reduce the number of probabilities that need to be computed. But this left me with two problems. The first is that even after assuming conditional independence, I felt that I still didn't have enough data to create robust predictions given that each of the four feature variables take on 20-30 values, even after some generous rounding. The second issue was that I felt that a predicting a precise, most-likely Y value would not be appropriate for this dataset since FPL points are distributed such that 2 is overwhelmingly the most common score, and I would be left unable to distinguish between a 'no-chance-of-being-anything-higher' 2 and a 'decent-chance-of-scoring-a-goal-but-still-probably-a-2' 2.

So, instead of assuming conditional independence on Y, I assumed full independence of O, Q, F, and X (a very dubious assumption). Furthermore, I kept it as an expectation rather than the argmax function we see in Naive Bayes, which left me to calculate the following expansion of Bayes Rule:

$$E[Y|O=o,Q=q,F=f,X=x] = \sum_{y=1}^{\infty} y \frac{P(O=o,Q=q,F=f,X=x|Y=y)*P(Y=y)}{P(O=o,Q=q,F=f,X=x)}$$

$$= \sum_{y=1}^{\infty} y \frac{P(O=o|Y=y)P(Q=q|Y=y)P(F=f|Y=y)P(X=x|Y=y)P(Y=y)}{P(O=o)P(Q=q)P(F=f)P(X=x)}$$

After many hours of processing and subsetting data in R[2], I arrived at a function that would take inputs for four feature variables and compute $E[Y|O, Q, F, X]$. There were a few semi-plausible inputs that didn't appear in the training data at all, so my fix was to essentially to round those extreme inputs to the most extreme input that *did* appear in the training data up to a certain point, but reject inputs that were too extreme outright by displaying an error message claiming parameters were unrealistic. The other important note here is that even after assuming full independence of the feature variables, there were just too many discrete values to deal with for the amount of data I had for Q, F, and X, so I rounded Q and F to the nearest 0.5, and rounded X to the nearest 0.05. Otherwise, this approach went pretty smoothly.

[2]The R script isn't the most readable, but it is available upon request to sajidof@gmail.com (if you are not a CS109 instructor) and should be included in my submission (if you are)

4 Latent Variables

At this point, soccer fans reading this are probably cringing at the assumption that quality, form, and xGi are independent of one another. So was I. I wanted another approach that didn't have to make this assumption and got around the issue of lacking data in some other way. The solution was to create composite 'latent variables' that condensed inputs into a much, much lower number of discrete values.

For this approach, I took the average of form (F) and xGi (X) since they measured similar things and condensed it into a new variable for 'recent performance' (R). Then, I created functions to convert values of O, Q, and R (which each previously took on around 15-25 discrete values) into composite scores from 1-6 using thresholds that I thought were appropriate[3]. Now, I only had to deal with 6*6*6=216 possible combinations of inputs. Multiplying this by around 25 possible values of Y gave me 5,400 total combinations of O, Q, R, and Y, which is still not ideal, but workable. Because I wasn't assuming any independence, I can now use the definition of conditional probability to calculate the following:

$$E[Y|O=o, Q=q, F=f, X=x] = \sum_{y=1}^{\infty} y \frac{P(O=o, Q=q, R=r, Y=y)}{P(O=o, Q=q, R=r)}$$

Once again, all that was left is processing data, which, for this approach, I did in R and later replicated in Python to integrate into my final runnable program[4]. It goes without saying, but for both approaches, I am treating each entry in the training data equally and calculating the probabilities by simply counting the number of occurrences that the respective conditions are met in the dataset.

This time, however, I found more fairly plausible inputs that were absent in the training data and lead to division by zero, which was to be expected given that the denominator is for outcomes that meet 3 conditions simultaneously. I tried expanding the range of extreme inputs that the program would 'lump in' with the most extreme input in the data set, but it didn't help much. I decided that rejecting the input outright and forcing the client to 'tone down' their inputs to force a numerical output would be better than just pretending that their input is something drastically different to what it actually is and risk the client not knowing this, so I just accepted that the client would have their inputs rejected more often under this approach. And with 5,000 data points in my test data, only 9 of them had parameters that were rejected (as opposed to the 3 using the naive expectation approach), so I didn't see this as a huge concession anyways.

[3] See appendix for screenshots of this code

[4] I only used the latent variable approach in the final product since, as you will soon see, it performed considerably better on the test data

5 Testing and Comparison

Now, it's time to compare these two approaches. Before developing both models, I reserved a random 20 percent of my data, leaving about 5,000 samples to test on. I ran each algorithm on each of the test data, and to my delight, both produced fairly similar and very realistic predictions for the majority of the test data. The difference between 2 and 10 points in FPL can often be just 2 inches that the ball swerved away from the corner of the goal, so as a fan of the game, these predictions definitely pass the eye test.

Player_Name	O	Q	F	X	Actual_Points_Outcome	Prediction_NaiveExp	Prediction_LatVars
James Ward-Prowse	1.3	3.5	4.0	0.30	3	3.76	3.76
Marten de Roon	1.0	2.5	2.5	0.20	2	2.90	3.19
Troy Deeney	1.0	4.0	3.5	0.15	2	3.40	3.69
Tom Trybull	1.3	2.0	2.0	0.00	2	2.82	3.03
Wayne Hennessey	0.9	2.0	2.0	0.00	2	2.97	3.19
Adam Lallana	1.3	4.5	2.0	0.15	2	3.13	3.91
Bruno Miguel Borges Fernandes	1.6	5.0	10.5	0.70	6	7.63	8.00
Grzegorz Krychowiak	0.8	3.5	4.0	0.05	2	3.49	3.58
Federico Fernández	0.9	4.0	2.5	0.15	6	3.27	3.42
Che Adams	1.7	2.0	2.0	0.50	2	3.47	3.72
Eric Dier	1.6	3.0	2.5	0.00	1	3.11	3.23
Bernd Leno	1.5	4.0	4.5	0.00	6	3.85	3.54
Winston Reid	1.0	4.0	4.0	0.00	9	3.62	3.44
Mason Greenwood	1.4	3.0	7.0	0.25	9	3.66	4.00
Harry Kane	1.7	5.5	7.0	0.75	15	10.31	4.94
Christian Eriksen	0.9	6.0	5.0	0.40	11	6.32	4.58
Mathew Ryan	1.4	4.0	4.5	0.00	2	3.80	3.54
David McGoldrick	1.1	2.0	2.0	0.40	1	3.22	3.42
Conor Coady	0.8	4.5	4.5	0.00	1	4.17	3.90
Jordon Ibe	1.3	3.0	2.0	0.15	5	2.86	3.03

Figure 1: Test data table

I did notice that the naive expectation approach tended to guess towards the extremes, particularly for really good players with high values for X, Q, and F, sometimes to an extent that was just laughable. This makes sense, since these are the variables that are very closely dependent, and by ignoring this dependency, the model is sort of triple-counting the causal effect of each on higher FPL points. The latent variable approach, on the other hand, was much more tame with its guesses and sometimes underestimated players and seemed to essentially max out its predictions at 6-7 points (while the naive expectation method frequently guess upwards of 20 points).

I compared the two approaches by comparing the mean squared-error of both approaches:

$$MSE = E[(Y - \hat{Y})^2]$$

Here are the results I got:

Model	MSE value
Naive Expectation	16.17
Latent Variables	9.00

It turns out the latent variable approach is considerably better. That is, for this dataset, it is worth factoring in the dependence of our feature variables at the cost of the granularity of our input.

6 Conclusion

Are these models actually going to consistently help me make better decisions in FPL? Probably not. Surpassing human intuition would require a model that considers far more variables and accounts for nuances and dependencies that just aren't captured in the approaches I have developed here. But considering the extraordinary amount of randomness in football and how simple these models are, I am thoroughly impressed with the accuracy of the predictions they made. This model is definitely something that I could consult when my intuition alone is unable to break a tie, for instance, but probably not replace said intuition entirely.

And at the very least, by comparing the two approaches, the project demonstrated the importance of accounting for dependence of random variables for predictive models such as as this one, even at the cost of some granularity. Overall, this project was a success and I'm excited to improve the model as I build my CS repertoire and explore machine learning in particular at a greater depth than was covered in CS109.

7 Appendix

```
def convertOppToLatScore(O):
    breaks = [0.5, 0.8, 1.1, 1.3, 1.5, 1.7, 2.1, 2.8]
    labels = [1, 2, 3, 4, 5, 6, 7]
    compVal = pd.cut([O], bins=breaks, labels=labels, include_lowest=True)
    return float(str(compVal[0]))

def convertFormToLatScore(F_val):
    breaks = [1.0, 3.0, 4.0, 6.0, 8.0, 10.0, 17.0]
    labels = [1, 2, 3, 4, 5, 6]
    compVal = pd.cut([F_val], bins=breaks, labels=labels, include_lowest=True)
    return float(str(compVal[0]))


def convertTPToLatScore(Q):
    breaks = [1.0, 3.0, 4.0, 6.0, 8.0, 10.0, 17.0]
    labels = [1, 2, 3, 4, 5, 6]
    compVal = pd.cut([Q], bins=breaks, labels=labels, include_lowest=True)
    return float(str(compVal[0]))
```

Figure 2: Functions converting input data to custom latent variables

```r
prob_YgOFQX <- function(Y, O, F_val, Q, X) {
  O_round <- round(O, digits = 1)
  if (O_round == 1.2) {
    avg = (prob_YgOFQX(Y, 1.3, F_val, Q, X) + prob_YgOFQX(Y, 1.1, F_val, Q, X)) /2
    return(avg)
  }
  if (O_round == 1.9) {
    avg = (prob_YgOFQX(Y, 1.8, F_val, Q, X) + prob_YgOFQX(Y, 2.0, F_val, Q, X))/2
    return(avg)
  }
  if (O_round >= 0.5 & O_round < 0.8) {
    O_round = 0.8
  }
  if (O_round > 2.1 & O_round <= 2.8) {
    O_round = 2.1
  }

  F_round <- round(F_val * 2) / 2
  Q_round <- round(Q * 2) / 2
  X_round <- round(X / 0.05) * 0.05

  OgY <- calc_OgY(O_round, Y)
  FgY <- calc_FgY(F_round, Y)
  QgY <- calc_QgY(Q_round, Y)
  XgY <- calc_XgY(X_round, Y)

  pO <- calc_O(O_round)
  pF <- calc_F(F_round)
  pQ <- calc_Q(Q_round)
  pX <- calc_X(X_round)
  pY <- calc_Y(Y)

  numerator = OgY * FgY * QgY * XgY * pY
  denominator = pO * pF * pQ * pX

  prob = numerator / denominator
  return(prob)
}
```

Figure 3: R function calculating $P(Y|O, F, X, Q)$ for naive expectation approach

```r
prob_YgOFQX <- function(Y, O, F_val, Q, X) {
  O_round <- round(O, digits = 1)
  if (O_round == 1.2) {
    avg = (prob_YgOFQX(Y, 1.3, F_val, Q, X) + prob_YgOFQX(Y, 1.1, F_val, Q, X)) /2
    return(avg)
  }
  if (O_round == 1.9) {
    avg = (prob_YgOFQX(Y, 1.8, F_val, Q, X) + prob_YgOFQX(Y, 2.0, F_val, Q, X))/2
    return(avg)
  }
  if (O_round >= 0.5 & O_round < 0.8) {
    O_round = 0.8
  }
  if (O_round > 2.1 & O_round <= 2.8) {
    O_round = 2.1
  }

  F_round <- round(F_val * 2) / 2
  Q_round <- round(Q * 2) / 2
  X_round <- round(X / 0.05) * 0.05

  OgY <- calc_OgY(O_round, Y)
  FgY <- calc_FgY(F_round, Y)
  QgY <- calc_QgY(Q_round, Y)
  XgY <- calc_XgY(X_round, Y)

  pO <- calc_O(O_round)
  pF <- calc_F(F_round)
  pQ <- calc_Q(Q_round)
  pX <- calc_X(X_round)
  pY <- calc_Y(Y)

  numerator = OgY * FgY * QgY * XgY * pY
  denominator = pO * pF * pQ * pX

  prob = numerator / denominator
  return(prob)
}
```

Figure 4: R function calculating $E(Y|O, F, X, Q)$ for naive expectation approach