XGBoost and Shapley Values for Olympics

Olexiy Pukhov

4/5/2022

The following code makes all plots and images higher resolution.

knitr::opts\_chunk$set(dpi = 300)

This is an exercise to help me get the skills for my research. I tried using XGBoost on the Olympics data. First, we need to install the many packages we need for this. We can use install.packages and then library, but using pacman needs less code to do this and is faster.

if (!require("pacman")) install.packages("pacman")

## Loading required package: pacman

pacman::p\_load(pacman, rio, tidyverse, xgboost, caTools, ggplot2, Ckmeans.1d.dp,  
 DALEXtra, mlr, caret, DiagrammeR, SHAPforxgboost, rmarkdown,  
 skimr)

## Importing Data

Now, let’s import the data and then remove the values that are highly correlated with other values. Then, let’s look at the structure of the data. Then, let’s look at the top 20 rows. There are too many rows, so they spill over to the next page. The ## represents the row #

data <- import("olympicmedals.dta")  
data = data %>%   
 select(-c(cc, year, lpop, lrgdpepc, sptinc992j\_p90p100, sptinc992j\_p99p100))  
skimr::skim(data)

Data summary

|  |  |
| --- | --- |
| Name | data |
| Number of rows | 204 |
| Number of columns | 9 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Column type frequency: |  |
| numeric | 9 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Group variables | None |

**Variable type: numeric**

| skim\_variable | n\_missing | complete\_rate | mean | sd | p0 | p25 | p50 | p75 | p100 | hist |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| sptinc992j\_p0p50 | 33 | 0.84 | 0.15 | 0.04 | 0.03 | 0.12 | 0.15 | 0.18 | 0.26 | ▁▃▇▅▂ |
| rgdpe | 28 | 0.86 | 712359.86 | 2368985.23 | 784.53 | 27290.08 | 97008.34 | 460800.94 | 20860506.00 | ▇▁▁▁▁ |
| pop | 28 | 0.86 | 43.07 | 153.75 | 0.03 | 2.81 | 9.72 | 30.77 | 1433.78 | ▇▁▁▁▁ |
| avh | 138 | 0.32 | 1849.98 | 269.24 | 1380.61 | 1650.92 | 1818.28 | 2061.05 | 2474.91 | ▅▇▆▅▂ |
| hc | 61 | 0.70 | 2.71 | 0.71 | 1.22 | 2.13 | 2.77 | 3.26 | 4.35 | ▅▅▇▇▁ |
| rnna | 29 | 0.86 | 3209645.14 | 10320354.02 | 2833.90 | 91149.59 | 356459.34 | 1935716.62 | 101703024.00 | ▇▁▁▁▁ |
| LifeLadder | 55 | 0.73 | 5.51 | 1.10 | 2.38 | 4.74 | 5.49 | 6.23 | 7.89 | ▁▅▇▇▃ |
| points | 0 | 1.00 | 10.28 | 28.75 | 0.00 | 0.00 | 0.00 | 8.00 | 232.00 | ▇▁▁▁▁ |
| rgdpepc | 28 | 0.86 | 22213.10 | 22095.72 | 251.32 | 5106.27 | 13933.31 | 33252.37 | 112941.45 | ▇▂▁▁▁ |

head(data,n = 20)

## sptinc992j\_p0p50 rgdpe pop avh hc rnna  
## 1 NA 3921.261 0.106314 NA NA 18427.50  
## 2 0.1776 NA NA NA NA NA  
## 3 0.0904 228151.016 31.825295 NA 1.481984 1367457.88  
## 4 0.1886 35890.020 2.880917 NA 2.964992 227804.64  
## 5 NA NA NA NA NA NA  
## 6 0.1274 681525.812 9.770529 NA 2.746695 4506529.00  
## 7 0.1623 991646.312 44.780677 1609.069 3.096804 3399148.50  
## 8 0.1854 41048.629 2.957731 NA 3.135995 98812.77  
## 9 NA NA NA NA NA NA  
## 10 NA 1986.163 0.097118 NA NA 10993.36  
## 11 0.1616 1280843.250 25.203198 1726.798 3.549666 5913514.00  
## 12 0.2202 498022.250 8.955102 1611.374 3.381046 2878110.75  
## 13 0.2029 162126.188 10.047718 NA NA 275832.31  
## 14 0.1402 8664.988 11.530580 NA 1.416526 18935.97  
## 15 0.2040 589449.125 11.539328 1586.431 3.149034 3498440.75  
## 16 0.1142 36740.742 11.801151 NA 1.918610 88271.80  
## 17 0.1504 43608.875 20.321378 NA 1.286242 86734.51  
## 18 0.1706 756355.562 163.046161 2418.883 2.101790 2844179.25  
## 19 0.1650 159419.969 7.000119 1645.246 3.186015 456857.28  
## 20 0.1014 74097.227 1.641172 NA 2.229507 429310.66  
## LifeLadder points rgdpepc  
## 1 NA 0 36883.7695  
## 2 2.375092 0 NA  
## 3 NA 0 7168.8579  
## 4 5.364910 0 12457.8457  
## 5 NA 0 NA  
## 6 6.458392 0 69753.2188  
## 7 5.900567 4 22144.5137  
## 8 5.488087 6 13878.4189  
## 9 NA 0 NA  
## 10 NA 0 20451.0312  
## 11 7.137368 87 50820.6641  
## 12 7.213489 10 55613.2422  
## 13 5.173389 10 16135.6230  
## 14 3.775283 0 751.4789  
## 15 6.838761 14 51081.7539  
## 16 4.407746 0 3113.3186  
## 17 4.740893 1 2145.9604  
## 18 5.279987 0 4638.9043  
## 19 5.597723 13 22773.8945  
## 20 6.173176 2 45148.9727

Let’s set the same seed for random number generation for reproducibility. Then, let’s split the data into a training and testing set. The model will be tested on the training set, and then tested on the testing set for accuracy. 75% of the data is going into the training set, and 25% of the data is going into the testing set.

set.seed(1)  
split = sample.split(data$points, SplitRatio = 0.75)  
training\_set = subset(data, split == TRUE)  
testing\_set = subset(data, split == FALSE)

The XGBoost algorithm requires that the input values are in a matrix. The algorithm only accepts numerical data. This also means that if there is categorical variables, we have to change them to dummy variables where each column is either a 0 or 1 depending on the category and there are n - 1 columns to represent the categorical data. In this dataset, there are no categorical values.

Let’s change the training and testing sets into a matrix.

training\_set = training\_set %>%   
 as.matrix()  
  
testing\_set = testing\_set %>%   
 as.matrix()

## Model Generation

Let’s make the XGBoost model, predict some new values and calculate the error in terms of MSE, MAE and RMSE on the testing set.

model = xgboost(data = training\_set[,-8], label = training\_set[,8], nrounds = 40)

## [1] train-rmse:26.374670   
## [2] train-rmse:20.696814   
## [3] train-rmse:16.510298   
## [4] train-rmse:13.157394   
## [5] train-rmse:10.600866   
## [6] train-rmse:8.590517   
## [7] train-rmse:7.086412   
## [8] train-rmse:5.897498   
## [9] train-rmse:4.995760   
## [10] train-rmse:4.268226   
## [11] train-rmse:3.628120   
## [12] train-rmse:3.085234   
## [13] train-rmse:2.665470   
## [14] train-rmse:2.329151   
## [15] train-rmse:2.062102   
## [16] train-rmse:1.835249   
## [17] train-rmse:1.636155   
## [18] train-rmse:1.482885   
## [19] train-rmse:1.340541   
## [20] train-rmse:1.242759   
## [21] train-rmse:1.168821   
## [22] train-rmse:1.025638   
## [23] train-rmse:0.927718   
## [24] train-rmse:0.834716   
## [25] train-rmse:0.776352   
## [26] train-rmse:0.731941   
## [27] train-rmse:0.682258   
## [28] train-rmse:0.635137   
## [29] train-rmse:0.569089   
## [30] train-rmse:0.540080   
## [31] train-rmse:0.487153   
## [32] train-rmse:0.438523   
## [33] train-rmse:0.405070   
## [34] train-rmse:0.364107   
## [35] train-rmse:0.315020   
## [36] train-rmse:0.277065   
## [37] train-rmse:0.258927   
## [38] train-rmse:0.232593   
## [39] train-rmse:0.215684   
## [40] train-rmse:0.200511

pred\_y = predict(model,testing\_set[,-8])  
  
  
mse = mean((testing\_set[8] - pred\_y)^2)  
mae = caret::MAE(testing\_set[8], pred\_y)  
rmse = caret::RMSE(testing\_set[8], pred\_y)  
  
cat("MSE: ", mse, "MAE: ", mae, " RMSE: ", rmse)

## MSE: 32.07732 MAE: 3.330121 RMSE: 5.663685

Let’s compare this to the initial linear model that I made before.

data2 <- import("olympicmedals.dta")  
data2 <- data2 %>%   
 filter(!is.na(data2$lpop))  
  
data2 <- data2 %>%   
 filter(!is.na(data2$lrgdpepc))  
  
set.seed(1)  
split2 = sample.split(data2$points, SplitRatio = 0.75)  
training\_set2 = subset(data2, split2 == TRUE)  
testing\_set2 = subset(data2, split2 == FALSE)  
  
model2 <- lm(points ~ lpop + lrgdpepc, data=training\_set2)  
pred\_y2 = predict(model2,testing\_set2[,-12])  
  
mse2 = mean((testing\_set2[,12] - pred\_y2)^2)  
mae2 = caret::MAE(testing\_set2[,12], pred\_y2)  
rmse2 = caret::RMSE(testing\_set2[,12], pred\_y2)  
  
cat("MSE: ", mse2, "MAE: ", mae2, " RMSE: ", rmse2)

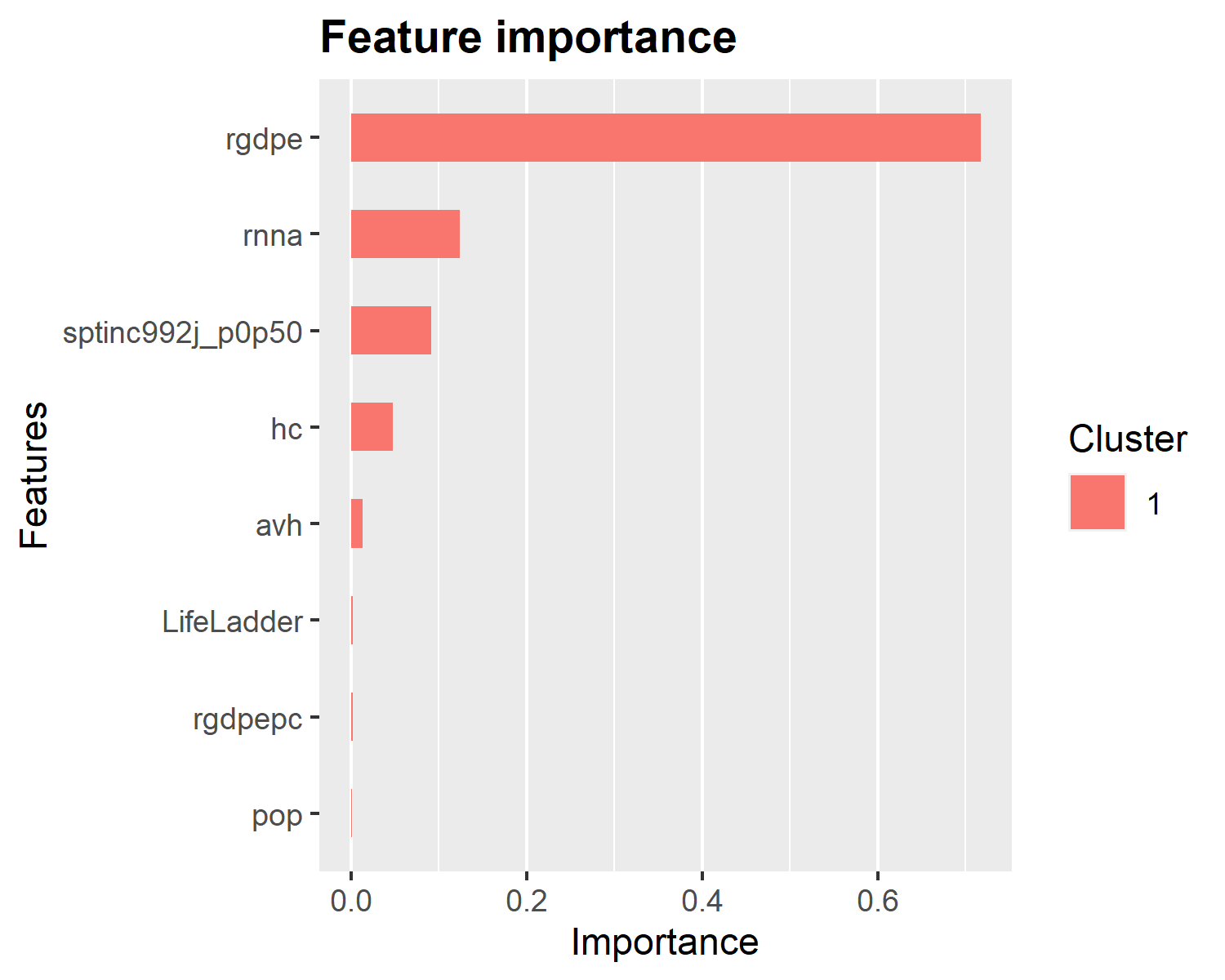
## MSE: 337.4018 MAE: 15.62729 RMSE: 18.3685

The XGBoost model is much more accurate. It has an RMSE of 5.66, compared to the linear regression model which has a RMSE of 18.3685.

## Visualizing the results

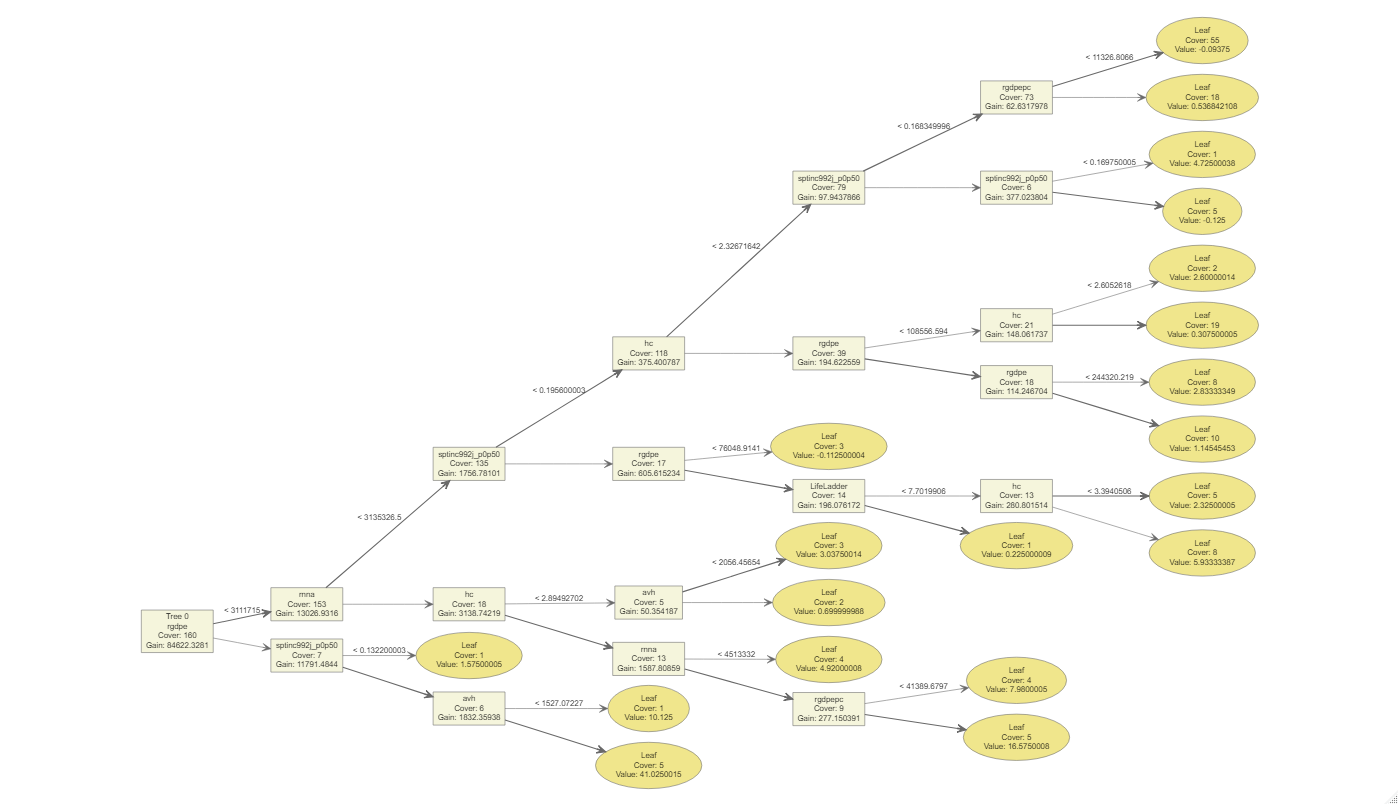
Let’s now visualize what features are important in this model. Out of all the variables we supplied to the model, which were the most important?

xgb\_imp <- xgb.importance(feature\_names = model$feature\_names,  
 model = model)  
xgb.ggplot.importance(xgb\_imp, n\_clusters = 1)

 It looks like rgdpe (Expenditure-side real GDP at chained PPPs (in mil. 2017 US$) was the most important variable in predicting the results.

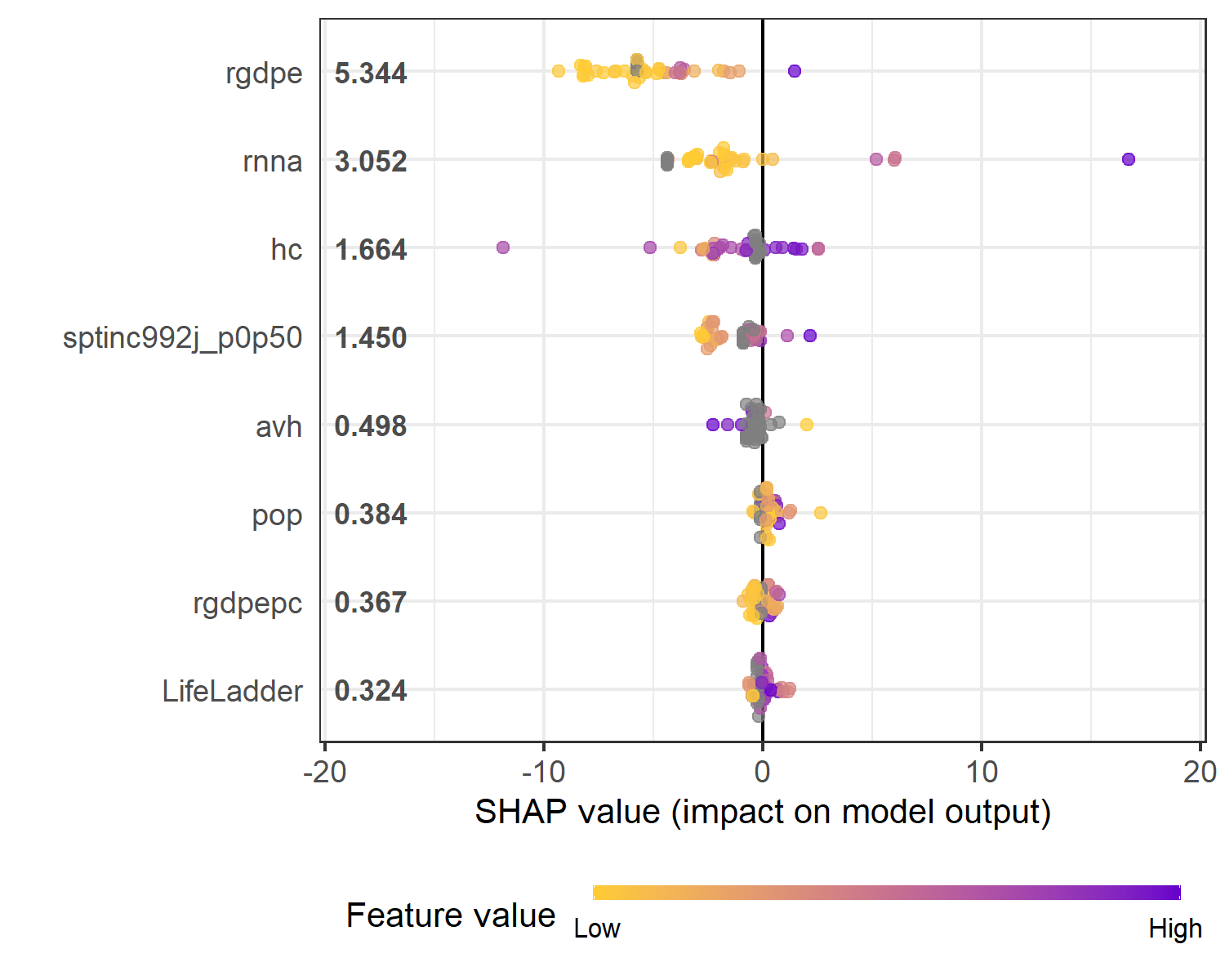
This will plot the first decision tree, which is not really useful for interpretation as XGBoost is an ensemble decision tree model, composed of many trees. However, it is still useful to give us an idea of what the algorithm is thinking about.

xgb.plot.tree(model = model, trees = 0)



Now let’s calculate the SHAP values and see which variable features were most important for our model.

shap <- shap.prep(xgb\_model = model, X\_train = testing\_set[,-8])  
shap.plot.summary(shap)

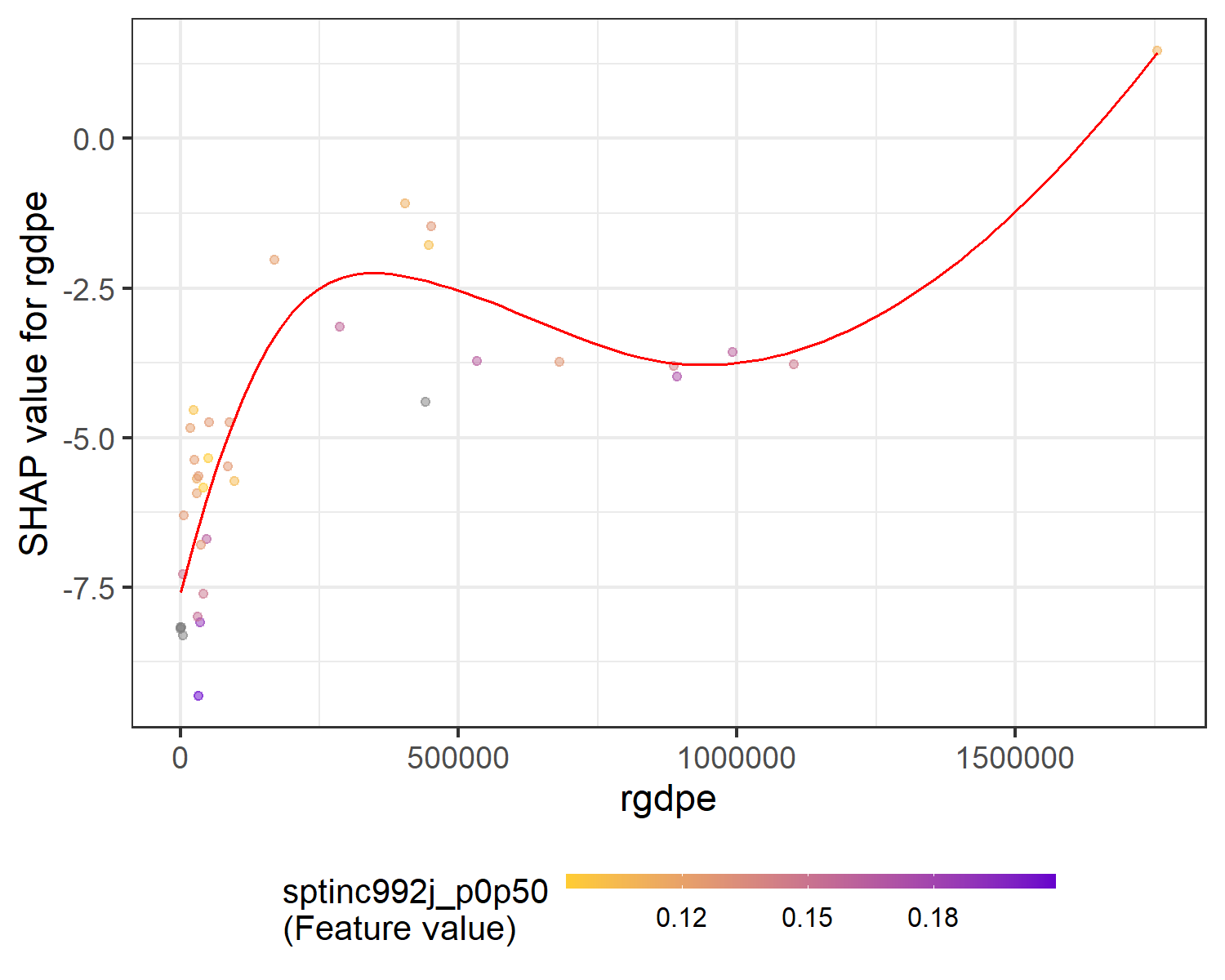
 Rgdpe is the most important valuable for predicting points won in the olympics. Let’s look at this variable in a partial dependence plot.

shap.plot.dependence(shap, "rgdpe", color\_feature = "auto",   
 alpha = 0.5, jitter\_width = 0.1)

## `geom\_smooth()` using formula 'y ~ x'

## Warning: Removed 7 rows containing non-finite values (stat\_smooth).

## Warning: Removed 7 rows containing missing values (geom\_point).

 It seems that a low rgdpe (Expenditure-side real GDP at chained PPPs (in mil. 2017 US$) hampers your ability to get points at a low value, has less an effect at higher values, and at very high values allows you to obtain higher points. Let us make sure of the results by looking again at our testing set.

max(testing\_set[,8])

## [1] 10

It seems the country with the maximum amount of points in the testing set was 10. If we compare this to our training set:

max(training\_set[,8])

## [1] 232

The maximum amount of points of a country in the training set is 232. This is an result of the countries randomly being put in the training and testing set. How do the results differ if a country with higher points was randomly chosen to be in the testing set? Let’s try a SplitRatio of 0.7.

data <- import("olympicmedals.dta")  
  
data = data %>%   
 select(-c(cc, year, lpop, lrgdpepc, sptinc992j\_p90p100, sptinc992j\_p99p100))  
  
set.seed(1)  
split = sample.split(data$points, SplitRatio = 0.7)  
training\_set = subset(data, split == TRUE)  
testing\_set = subset(data, split == FALSE)  
  
training\_set = training\_set %>%   
 as.matrix()  
  
testing\_set = testing\_set %>%   
 as.matrix()

Let’s check the testing set to see if it has a higher country with max value for points.

max(testing\_set[,8])

## [1] 68

Let’s also check the max value for points in the training set to see if it includes a high point scoring country.

max(training\_set[,8])

## [1] 232

Now let’s calculate the model again.

model = xgboost(data = training\_set[,-8], label = training\_set[,8], nrounds = 40)

## [1] train-rmse:27.064226   
## [2] train-rmse:21.260931   
## [3] train-rmse:17.006104   
## [4] train-rmse:13.642632   
## [5] train-rmse:10.957189   
## [6] train-rmse:8.904371   
## [7] train-rmse:7.340249   
## [8] train-rmse:6.071575   
## [9] train-rmse:5.093059   
## [10] train-rmse:4.339937   
## [11] train-rmse:3.710795   
## [12] train-rmse:3.165035   
## [13] train-rmse:2.737292   
## [14] train-rmse:2.358058   
## [15] train-rmse:2.060940   
## [16] train-rmse:1.819282   
## [17] train-rmse:1.568922   
## [18] train-rmse:1.391671   
## [19] train-rmse:1.254977   
## [20] train-rmse:1.144986   
## [21] train-rmse:0.995366   
## [22] train-rmse:0.869612   
## [23] train-rmse:0.750286   
## [24] train-rmse:0.683142   
## [25] train-rmse:0.635431   
## [26] train-rmse:0.596431   
## [27] train-rmse:0.561876   
## [28] train-rmse:0.494441   
## [29] train-rmse:0.451951   
## [30] train-rmse:0.425904   
## [31] train-rmse:0.397457   
## [32] train-rmse:0.375684   
## [33] train-rmse:0.330268   
## [34] train-rmse:0.319064   
## [35] train-rmse:0.286340   
## [36] train-rmse:0.274272   
## [37] train-rmse:0.263440   
## [38] train-rmse:0.257379   
## [39] train-rmse:0.232242   
## [40] train-rmse:0.210472

pred\_y = predict(model,testing\_set[,-8])  
  
  
mse = mean((testing\_set[8] - pred\_y)^2)  
mae = caret::MAE(testing\_set[8], pred\_y)  
rmse = caret::RMSE(testing\_set[8], pred\_y)  
  
cat("MSE: ", mse, "MAE: ", mae, " RMSE: ", rmse)

## MSE: 374.9142 MAE: 7.262226 RMSE: 19.3627

The RMSE of our model is now 19.36, larger than the RMSE for our linear model at 18.37. At smaller values of points, the XGBoost model works better than the linear model. However, at larger values of points, the XGboost model becomes worse than the linear model. Only a few countries won a lot of points in the olympics (USA, Russia, Great Britain, Japan and China). This is probably because of insufficient data - XGBoost is designed for very large data, and does not work well with a few outliers in a very small dataset.