

Kaggle Competition: Kobe Bryant Shot Selection

Javier Guzmán Figueira Domínguez

12/03/2018

Introducción

En este documento se tratará el problema presentado en la plataforma Kaggle bajo el título Kobe Bryant Shot Selection. El objetivo de esta competición es la de realizar una clasificación de tiros encestados y fallados por el ex-jugador de baloncesto, Kobe Bryant.

La competición proporciona un dataset con 30.697 instancias y 25 variables, incluída la variable clase. Esta variable clase está definida de forma binaria, tomando valor 1 en caso de tratarse de un tiro acertado y 0 en caso de un tiro fallido.

El conjunto de test está representado por un subconjunto de 5.000 intancias con ausencia de la etiqueta de clase. Para estas instancias, se deberá de predecir la probabilidad que cada tiro se enceste. El *score* utilizado por *Kaggle* en esta competición es la pérdida logarítmica o *log loss* (*Logarithmic loss*).

```
library(caret)

## Loading required package: lattice
## Loading required package: ggplot2
library(ggplot2)
library(glmnet)

## Loading required package: Matrix
## Loading required package: foreach
## foreach: simple, scalable parallel programming from Revolution Analytics
## Use Revolution R for scalability, fault tolerance and more.
## http://www.revolutionanalytics.com

## Loaded glmnet 2.0-13
library(earth)

## Loading required package: plotmo
## Loading required package: plotrix
## Loading required package: TeachingDemos
SEED <- 555

data <- read.csv("data.csv")

dim(data)

## [1] 30697    25
```

Análisis de la variables

En esta sección, se realizará una breve descripción y análisis de cada una de las variables predictoras del dataset y la variable clase. A continuación, se muestra el listado de variables con algunas de sus características internas.

```
str(data)
```

```
## 'data.frame': 30697 obs. of 25 variables:  
## $ action_type : Factor w/ 57 levels "Alley Oop Dunk Shot",...: 27 27 27 27 6 27 28 27 27 42 ...  
## $ combined_shot_type: Factor w/ 6 levels "Bank Shot","Dunk",...: 4 4 4 4 2 4 5 4 4 4 ...  
## $ game_event_id : int 10 12 35 43 155 244 251 254 265 294 ...  
## $ game_id : int 20000012 20000012 20000012 20000012 20000012 20000012 20000012 20000012 20000012 20000012 ...  
## $ lat : num 34 34 33.9 33.9 34 ...  
## $ loc_x : int 167 -157 -101 138 0 -145 0 1 -65 -33 ...  
## $ loc_y : int 72 0 135 175 0 -11 0 28 108 125 ...  
## $ lon : num -118 -118 -118 -118 -118 ...  
## $ minutes_remaining : int 10 10 7 6 6 9 8 8 6 3 ...  
## $ period : int 1 1 1 1 2 3 3 3 3 3 ...  
## $ playoffs : int 0 0 0 0 0 0 0 0 0 0 ...  
## $ season : Factor w/ 20 levels "1996-97","1997-98",...: 5 5 5 5 5 5 5 5 5 5 ...  
## $ seconds_remaining : int 27 22 45 52 19 32 52 5 12 36 ...  
## $ shot_distance : int 18 15 16 22 0 14 0 2 12 12 ...  
## $ shot_made_flag : int NA 0 1 0 1 0 1 NA 1 0 ...  
## $ shot_type : Factor w/ 2 levels "2PT Field Goal",...: 1 1 1 1 1 1 1 1 1 1 ...  
## $ shot_zone_area : Factor w/ 6 levels "Back Court(BC)",...: 6 4 3 5 2 4 2 2 4 2 ...  
## $ shot_zone_basic : Factor w/ 7 levels "Above the Break 3",...: 5 5 5 5 6 5 6 6 3 3 ...  
## $ shot_zone_range : Factor w/ 5 levels "16-24 ft.", "24+ ft.",...: 1 3 1 1 5 3 5 5 3 3 ...  
## $ team_id : int 1610612747 1610612747 1610612747 1610612747 1610612747 1610612747 1610612747 1610612747 1610612747 1610612747 ...  
## $ team_name : Factor w/ 1 level "Los Angeles Lakers": 1 1 1 1 1 1 1 1 1 1 ...  
## $ game_date : Factor w/ 1559 levels "1996-11-03","1996-11-05",...: 311 311 311 311 311 311 311 311 311 311 ...  
## $ matchup : Factor w/ 74 levels "LAL @ ATL","LAL @ BKN",...: 29 29 29 29 29 29 29 29 29 29 ...  
## $ opponent : Factor w/ 33 levels "ATL","BKN","BOS",...: 26 26 26 26 26 26 26 26 26 26 ...  
## $ shot_id : int 1 2 3 4 5 6 7 8 9 10 ...
```

Variable clase: *shot_made_flag*

Tal y como se ha comentado, existen 5.000 instancias que carecen de valores para la variable clase, o etiqueta. Estas son las instancias para las que debemos realizar nuestra clasificación.

```
print("Número de instancias totales con valores perdidos")  
  
## [1] "Número de instancias totales con valores perdidos"  
sum(is.na(data$shot_made_flag))  
  
## [1] 5000  
print("Instancias totales con valores perdidos, no teniendo en cuenta 'shot_made_flag'")  
  
## [1] "Instancias totales con valores perdidos, no teniendo en cuenta 'shot_made_flag'"  
sum(is.na(dplyr::select(data, -shot_made_flag)))  
  
## [1] 0
```

Por consiguiente, realizaremos una separación de ambos subconjuntos y los denominaremos conjuntos de entrenamiento o *train* y de prueba o *test*. Es obvio que el conjunto de test contendrá las 5.000 instancias con ausencia de etiqueta para la variable clase *shot_made_flag*. Por otra parte, el conjunto de entrenamiento ahora constará de un total de 25.697 instancias.

```
train <- data[!is.na(data$shot_made_flag), ]
test <- data[is.na(data$shot_made_flag), ]

train$shot_made_flag <- as.factor(train$shot_made_flag)
train$shot_made_flag <- factor(train$shot_made_flag, levels = c("1",
  "0"))
test <- dplyr::select(test, -shot_made_flag)

dim(train)

## [1] 25697    25

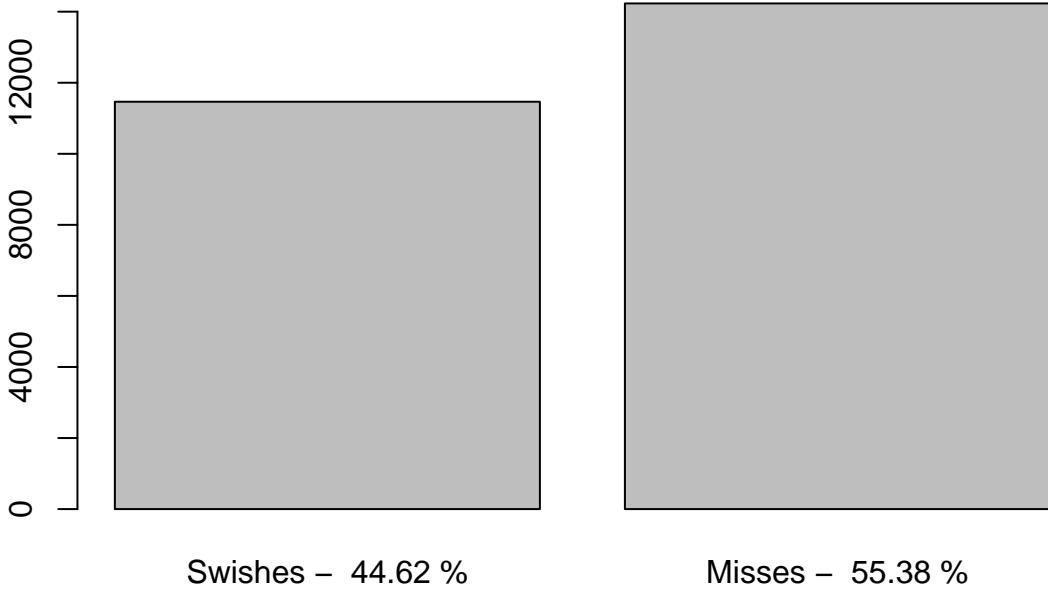
dim(test)

## [1] 5000    24

Antes de analizar las variables predictoras, observamos la distribución de la variables clase y advertimos que las categorías están bastante parejas (al menos en el conjunto de entrenamiento). De esta forma, calculamos que un 55,39% de los tiros han sido fallidos y el restante 44,62% son tiros encestados, tal y como muestra la gráfica.

shot_made_flag.misses <- format(round((length(train$shot_made_flag[train$shot_made_flag == 0])/nrow(train)) * 100, 2), nsmall = 2)
shot_made_flag.swishes <- format(round((length(train$shot_made_flag[train$shot_made_flag == 1])/nrow(train)) * 100, 2), nsmall = 2)
shot_made_flag.names <- c(paste("Swishes - ", shot_made_flag.swishes,
  "%"), paste("Misses - ", shot_made_flag.misses, "%"))

barplot(table(train$shot_made_flag), names = shot_made_flag.names)
```



Clasificación del tiro según la forma

La variable *action_type* referencia el tipo de acción mediante el que se realizó/intentó la canasta. Se definen hasta 57 tipos diferentes de acciones, siendo *Jump Shot* la más frecuente. A continuación, se muestran dichos tipos:

```
levels(data$action_type)
```

```
## [1] "Alley Oop Dunk Shot"
## [2] "Alley Oop Layup shot"
## [3] "Cutting Finger Roll Layup Shot"
## [4] "Cutting Layup Shot"
## [5] "Driving Bank shot"
## [6] "Driving Dunk Shot"
## [7] "Driving Finger Roll Layup Shot"
## [8] "Driving Finger Roll Shot"
## [9] "Driving Floating Bank Jump Shot"
## [10] "Driving Floating Jump Shot"
## [11] "Driving Hook Shot"
## [12] "Driving Jump shot"
## [13] "Driving Layup Shot"
## [14] "Driving Reverse Layup Shot"
## [15] "Driving Slam Dunk Shot"
## [16] "Dunk Shot"
## [17] "Fadeaway Bank shot"
## [18] "Fadeaway Jump Shot"
```

```

## [19] "Finger Roll Layup Shot"
## [20] "Finger Roll Shot"
## [21] "Floating Jump shot"
## [22] "Follow Up Dunk Shot"
## [23] "Hook Bank Shot"
## [24] "Hook Shot"
## [25] "Jump Bank Shot"
## [26] "Jump Hook Shot"
## [27] "Jump Shot"
## [28] "Layup Shot"
## [29] "Pullup Bank shot"
## [30] "Pullup Jump shot"
## [31] "Putback Dunk Shot"
## [32] "Putback Layup Shot"
## [33] "Putback Slam Dunk Shot"
## [34] "Reverse Dunk Shot"
## [35] "Reverse Layup Shot"
## [36] "Reverse Slam Dunk Shot"
## [37] "Running Bank shot"
## [38] "Running Dunk Shot"
## [39] "Running Finger Roll Layup Shot"
## [40] "Running Finger Roll Shot"
## [41] "Running Hook Shot"
## [42] "Running Jump Shot"
## [43] "Running Layup Shot"
## [44] "Running Pull-Up Jump Shot"
## [45] "Running Reverse Layup Shot"
## [46] "Running Slam Dunk Shot"
## [47] "Running Tip Shot"
## [48] "Slam Dunk Shot"
## [49] "Step Back Jump shot"
## [50] "Tip Layup Shot"
## [51] "Tip Shot"
## [52] "Turnaround Bank shot"
## [53] "Turnaround Fadeaway Bank Jump Shot"
## [54] "Turnaround Fadeaway shot"
## [55] "Turnaround Finger Roll Shot"
## [56] "Turnaround Hook Shot"
## [57] "Turnaround Jump Shot"

```

Puede resultar interesante mostrar la relación entre la precisión en el tiro y el tipo de acción realizada. Dado que la variable *action_type* contiene multitud de categorías, se han seleccionado aquellas que están asociadas a más de 20 lanzamientos. Las restantes se han juntado en una nueva categoría, bajo el nombre de *Others*.

```

action_type <- train$action_type
action_type.is.frequent <- sapply(levels(action_type), function(level) {
  length(action_type[action_type == level]) > 20
})
action_type.frequent.levels <- subset(levels(action_type), action_type.is.frequent)

action_type <- as.character(action_type)
action_type[!action_type %in% action_type.frequent.levels] <- "Other"
action_type <- as.factor(action_type)

```

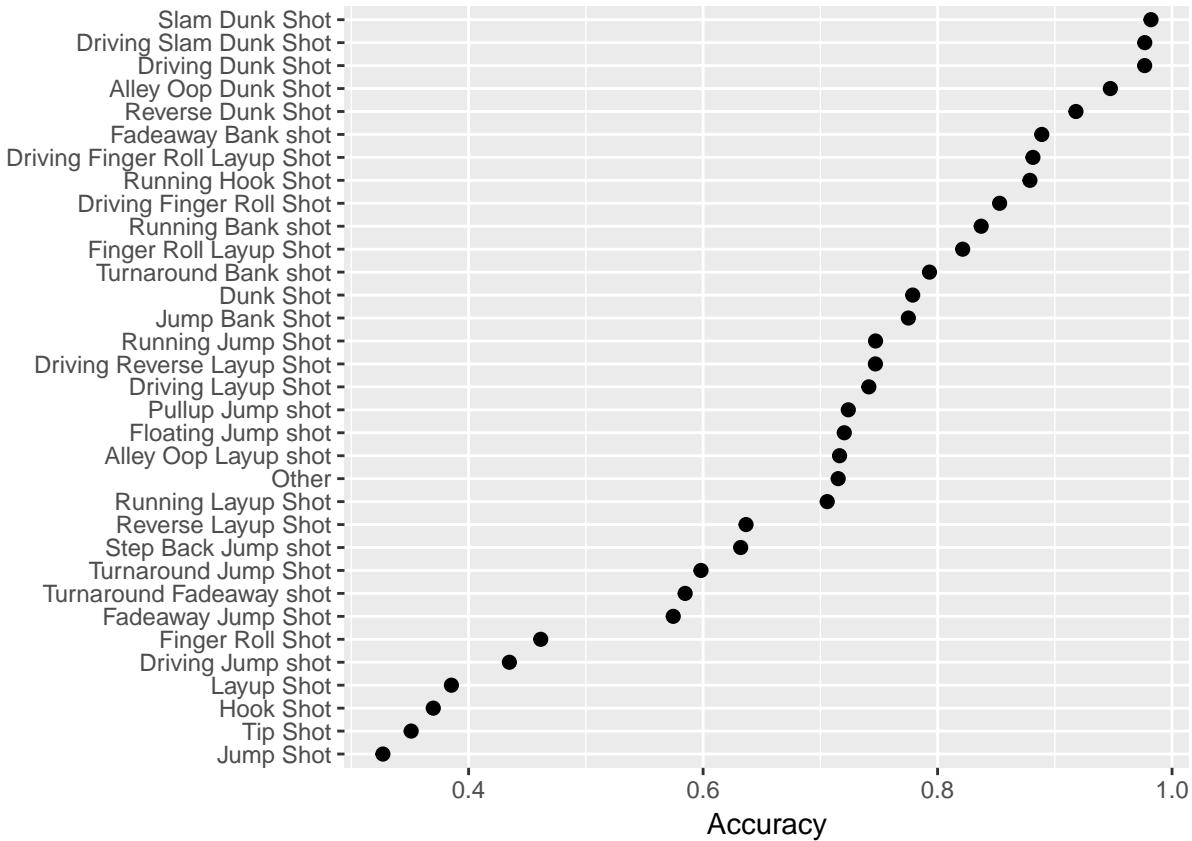
A continuación, se define una función para mostrar la proporción de aciertos y errores (en los tiros realizados),

en función de una determinada característica y ordenados en función de dicha proporción

```
plotOrderedAccuracyByFeature <- function(feature) {  
  
  temp <- prop.table(table(feature, train$shot_made_flag),  
    1)  
  temp <- as.data.frame.matrix(temp)  
  temp$shot <- rownames(temp)  
  
  ggplot(temp, aes(x = reorder(shot, `1`), y = 1)) + geom_point(aes(y = `1`),  
    size = 2, stat = "identity") + coord_flip() + xlab("") +  
    ylab("Accuracy")  
}
```

Se aprecia como, efectivamente, existe una clara diferencia de efectividad según el tipo de acción. Los tipos con mayor *accuracy* son *Slam Dunk Shot*, *Driving Slam Dunk Shot* y *Driving Dunk Shot*, mientras que *Jump Shot*, *Tip Shot*, *Hook Shot* y *Layup Shot* son las acciones con menor probabilidad de acierto.

```
plotOrderedAccuracyByFeature(action_type)
```



Otra variable que describe el tipo de tiro realizado es *combined_shot_type*. Es de tipo categórica y comprende los siguientes tipos:

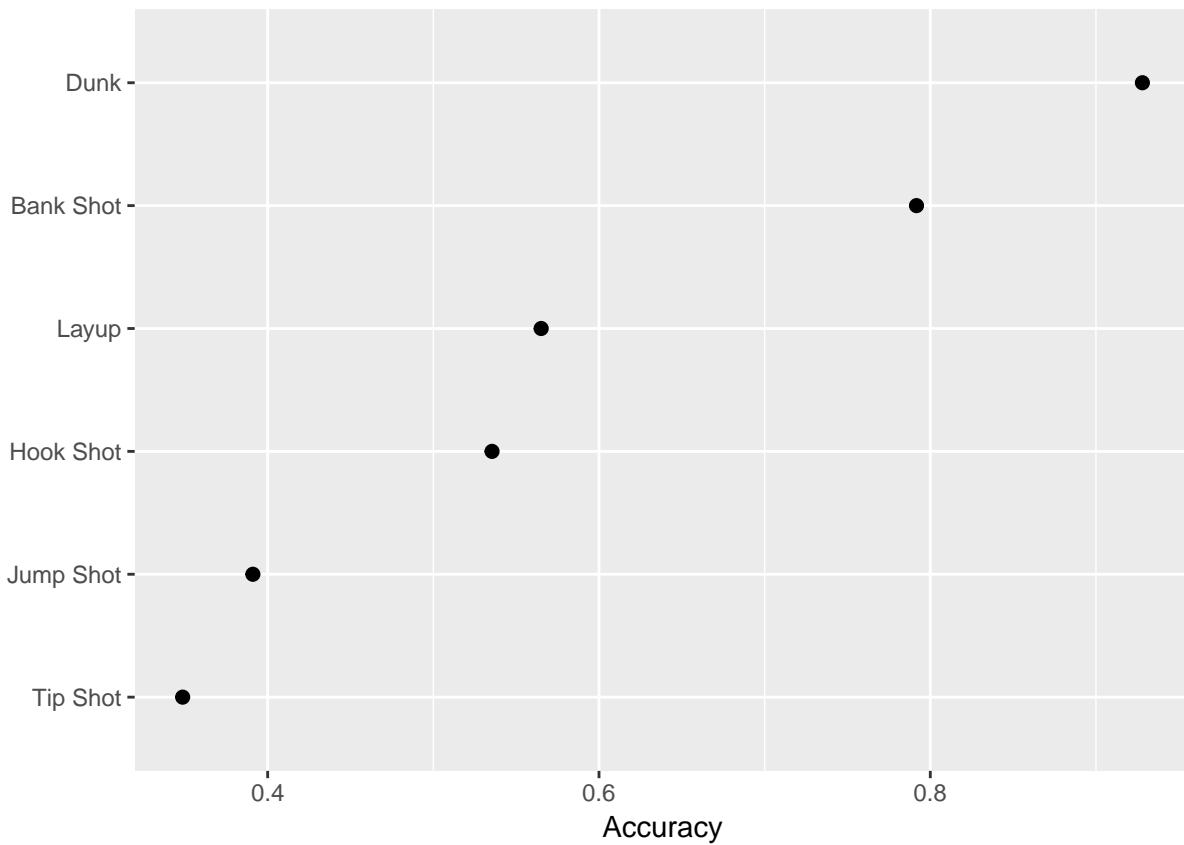
```
levels(train$combined_shot_type)
```

```
## [1] "Bank Shot" "Dunk"      "Hook Shot" "Jump Shot" "Layup"     "Tip Shot"
```

De la misma forma que se hizo con la variable *action_type*, realizaremos una comparativa entre los tipos de tiros y sus respectivas probabilidades de acierto. De igual manera, existen claras relaciones en dicha

comparativa. Observamos una mayor efectividad asociada a mates (*dunks*) y tiros libres (*Bank Shots*).

```
plotOrderedAccuracyByFeature(train$combined_shot_type)
```



Finalmente, la variable *shot_type* también nos aporta valiosa información sobre el tipo de tiro. Esta variable categórica, nos informa de si se trata de un tiro de 2 o 3 puntos.

```
levels(data$shot_type)
```

```
## [1] "2PT Field Goal" "3PT Field Goal"
```

Puede ser relevante representar su relación con la precisión en los tiros realizados. Para ello, se define una función, que de forma muy similar a la anterior, nos muestra la proporción de aciertos y errores (en los tiros realizados), en función de una característica dada.

```
plotAccuracyByFeature <- function(feature, feature.name, x.angle = 0) {  
  title <- "Shot swished"  
  ggplot(data = train, aes(x = feature)) + geom_bar(aes(fill = shot_made_flag),  
    stat = "count", position = "fill") + xlab(feature.name) +  
    ylab("Accuracy") + theme(axis.text.x = element_text(angle = x.angle,  
      hjust = ifelse(x.angle == 0, 0.5, 1))) + guides(fill = guide_legend(title = title))  
}  
  
plotAccuracyByFeature(train$shot_type, "Shot type")
```



Identificadores genéricos

Las propiedades `game_id` y `game_event_id` representan los identificadores de partido y de evento en cada partido, respectivamente. En principio, no aportan información más allá de podrían ayudarnos a estudiar las acciones agrupadas por partido.

En cuanto a las variables `team_id` y `team_name`, carecen completamente de interés. Estas recogen, respectivamente, el identificador de equipo y su nombre. Dado que el dataset contiene datos sobre los tiros realizados en un único equipo, estas características no aportan información útil.

```
unique(data$team_id)
## [1] 1610612747
levels(data$team_name)
## [1] "Los Angeles Lakers"
```

Variables referentes al partido

La variable `matchup` incluye la información de los equipos que han participado en el partido. Sin embargo, sabemos que el dataset recoge los datos de los tiros realizados por K. Bryant en Los Angeles Lakers y la variable `opponent` contiene el nombre del equipo adversario. La información útil que podemos extraer de `matchup`, es saber si una determinada acción se ha realizado como local o visitante.

```

print("Oponentes")

## [1] "Oponentes"
levels(data$opponent)

## [1] "ATL" "BKN" "BOS" "CHA" "CHI" "CLE" "DAL" "DEN" "DET" "GSW" "HOU"
## [12] "IND" "LAC" "MEM" "MIA" "MIL" "MIN" "NJN" "NOH" "NOP" "NYK" "OKC"
## [23] "ORL" "PHI" "PHX" "POR" "SAC" "SAS" "SEA" "TOR" "UTA" "VAN" "WAS"
print("Enfrentamientos")

## [1] "Enfrentamientos"
levels(data$matchup)

## [1] "LAL @ ATL"   "LAL @ BKN"   "LAL @ BOS"   "LAL @ CHA"   "LAL @ CHH"
## [6] "LAL @ CHI"   "LAL @ CLE"   "LAL @ DAL"   "LAL @ DEN"   "LAL @ DET"
## [11] "LAL @ GSW"   "LAL @ HOU"   "LAL @ IND"   "LAL @ LAC"   "LAL @ MEM"
## [16] "LAL @ MIA"   "LAL @ MIL"   "LAL @ MIN"   "LAL @ NJN"   "LAL @ NOH"
## [21] "LAL @ NOP"   "LAL @ NYK"   "LAL @ OKC"   "LAL @ ORL"   "LAL @ PHI"
## [26] "LAL @ SAC"   "LAL @ SEA"   "LAL @ TOR"   "LAL @ UTA"   "LAL @ UTH"
## [31] "LAL @ VAN"   "LAL @ WAS"   "LAL vs. ATL" "LAL vs. BKN" "LAL vs. BOS"
## [36] "LAL vs. CHA" "LAL vs. CHH" "LAL vs. CHI" "LAL vs. CLE" "LAL vs. DAL"
## [41] "LAL vs. DEN" "LAL vs. DET" "LAL vs. GSW" "LAL vs. HOU" "LAL vs. IND"
## [46] "LAL vs. LAC" "LAL vs. MEM" "LAL vs. MIA" "LAL vs. MIL" "LAL vs. MIN"
## [51] "LAL vs. NJN" "LAL vs. NOH" "LAL vs. NOK" "LAL vs. NOP" "LAL vs. NYK"
## [56] "LAL vs. POR" "LAL vs. SAC" "LAL vs. SAN" "LAL vs. SAS" "LAL vs. SEA"
## [61] "LAL vs. TOR" "LAL vs. UTA" "LAL vs. VAN" "LAL vs. WAS"

```

Así mismo, las variables *season* y *period* referencian la temporada en la que se produjo el tiro y el periodo dentro del partido, respectivamente. Estudiammos sus relaciones con la efectividad en el tiro.

```

print("Temporadas")

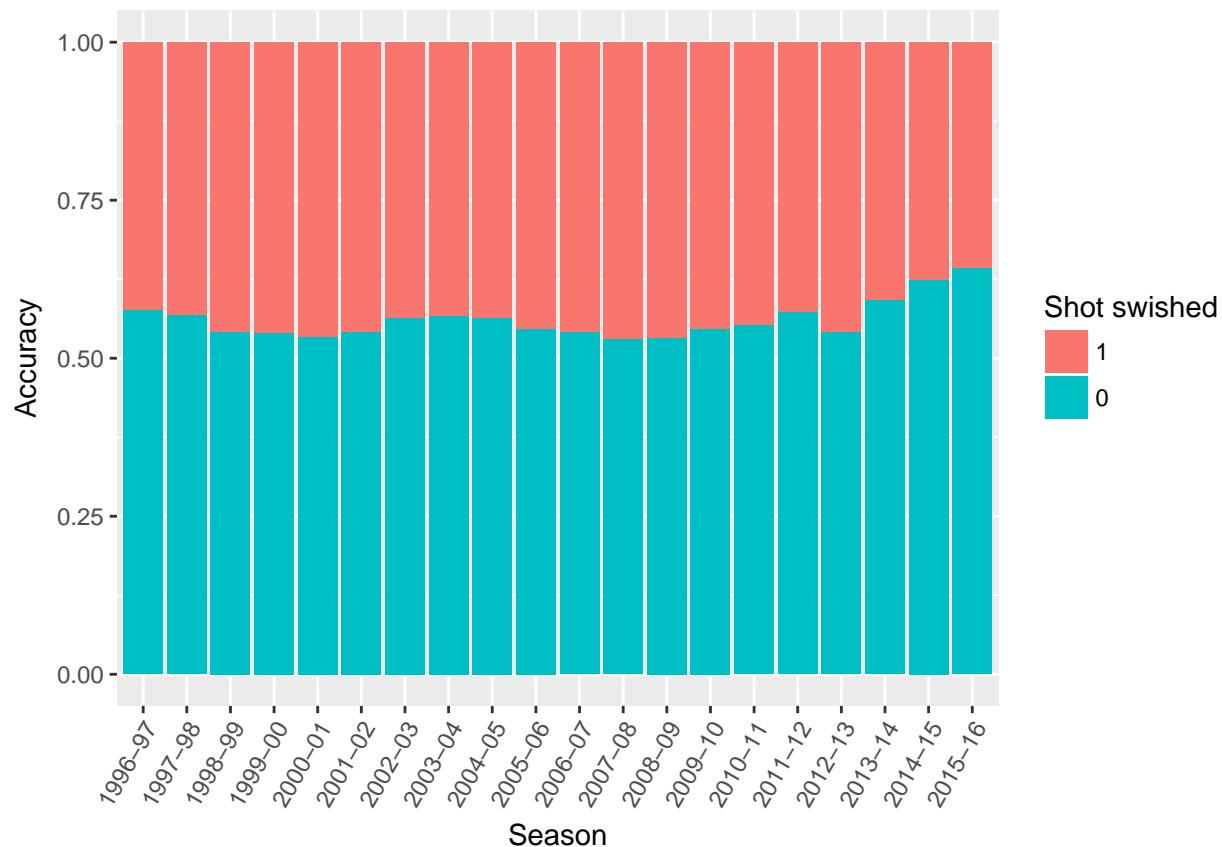
## [1] "Temporadas"
levels(data$season)

## [1] "1996-97" "1997-98" "1998-99" "1999-00" "2000-01" "2001-02" "2002-03"
## [8] "2003-04" "2004-05" "2005-06" "2006-07" "2007-08" "2008-09" "2009-10"
## [15] "2010-11" "2011-12" "2012-13" "2013-14" "2014-15" "2015-16"
print("Periodos del partido")

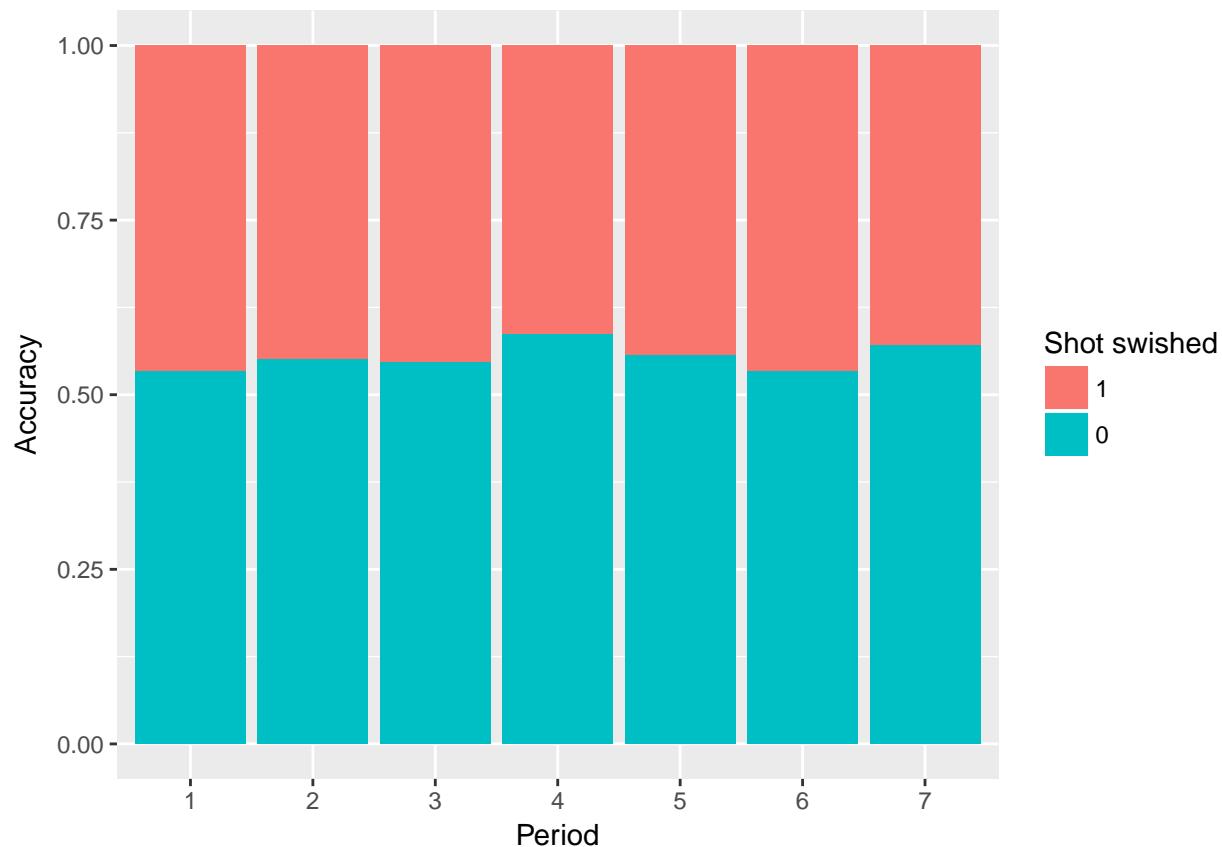
## [1] "Periodos del partido"
unique(data$period)

## [1] 1 2 3 4 5 6 7
plotAccuracyByFeature(train$season, "Season", 60)

```

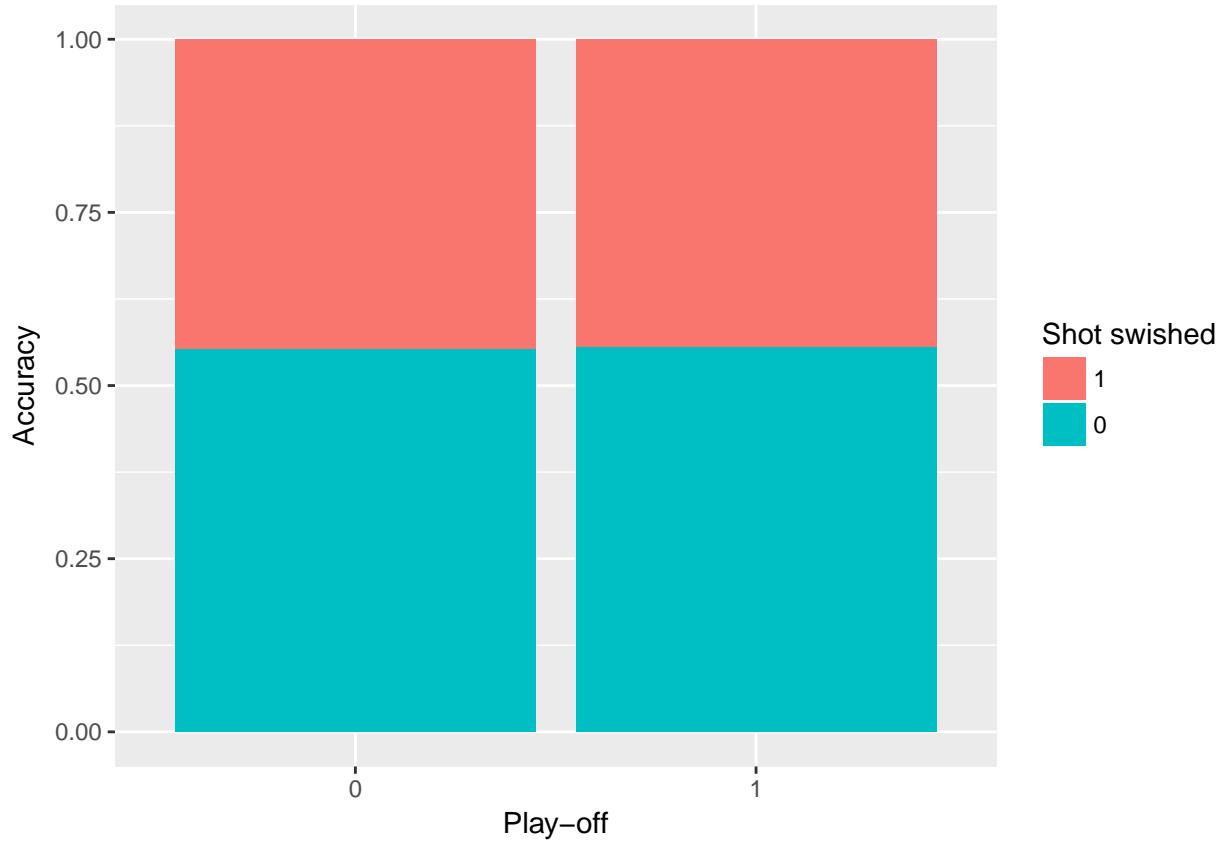


```
plotAccuracyByFeature(as.factor(train$period), "Period")
```



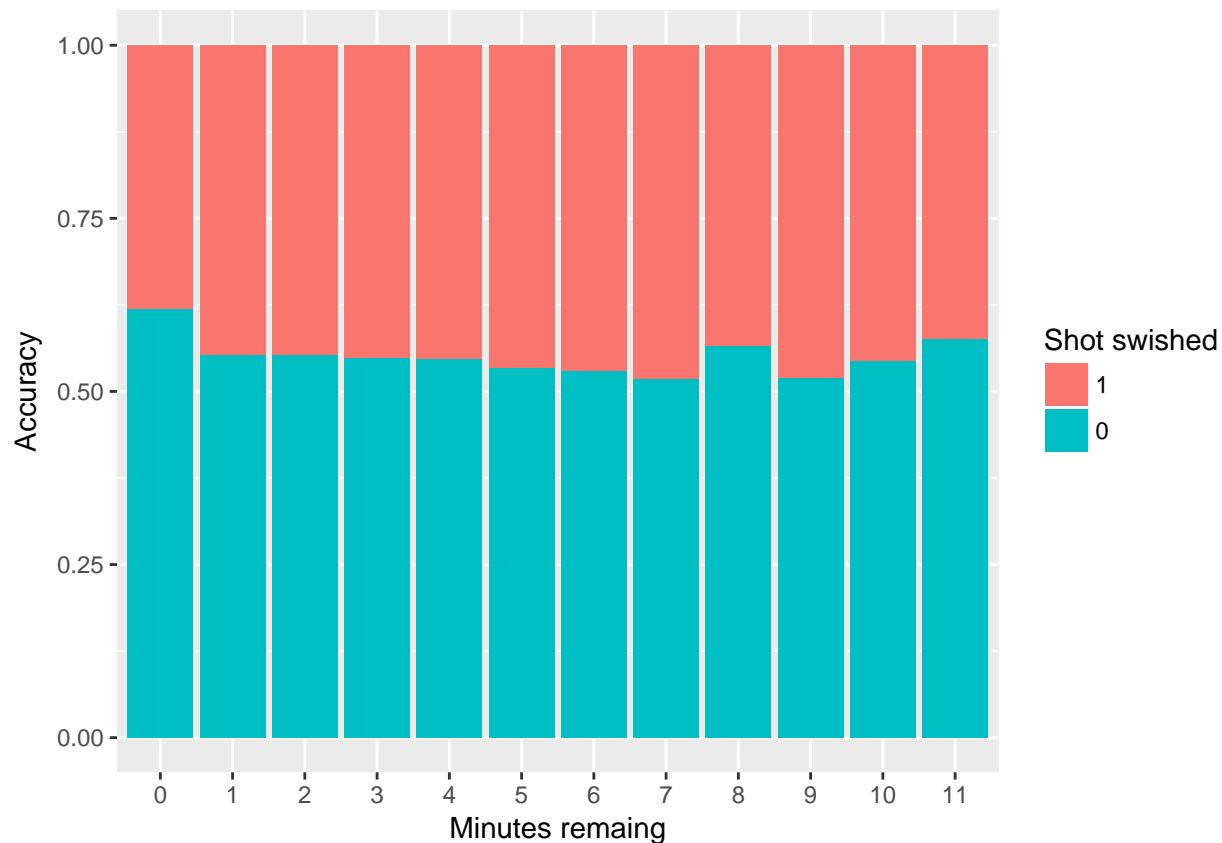
Otra variable interesante podría ser *playoffs*, ya que nos informa si un partido dado es de play-off (representado con un 1) o no (0). Sin embargo, esta relación tampoco arroja demasiada claridad.

```
unique(data$playoffs)
## [1] 0 1
plotAccuracyByFeature(as.factor(train$playoffs), "Play-off")
```

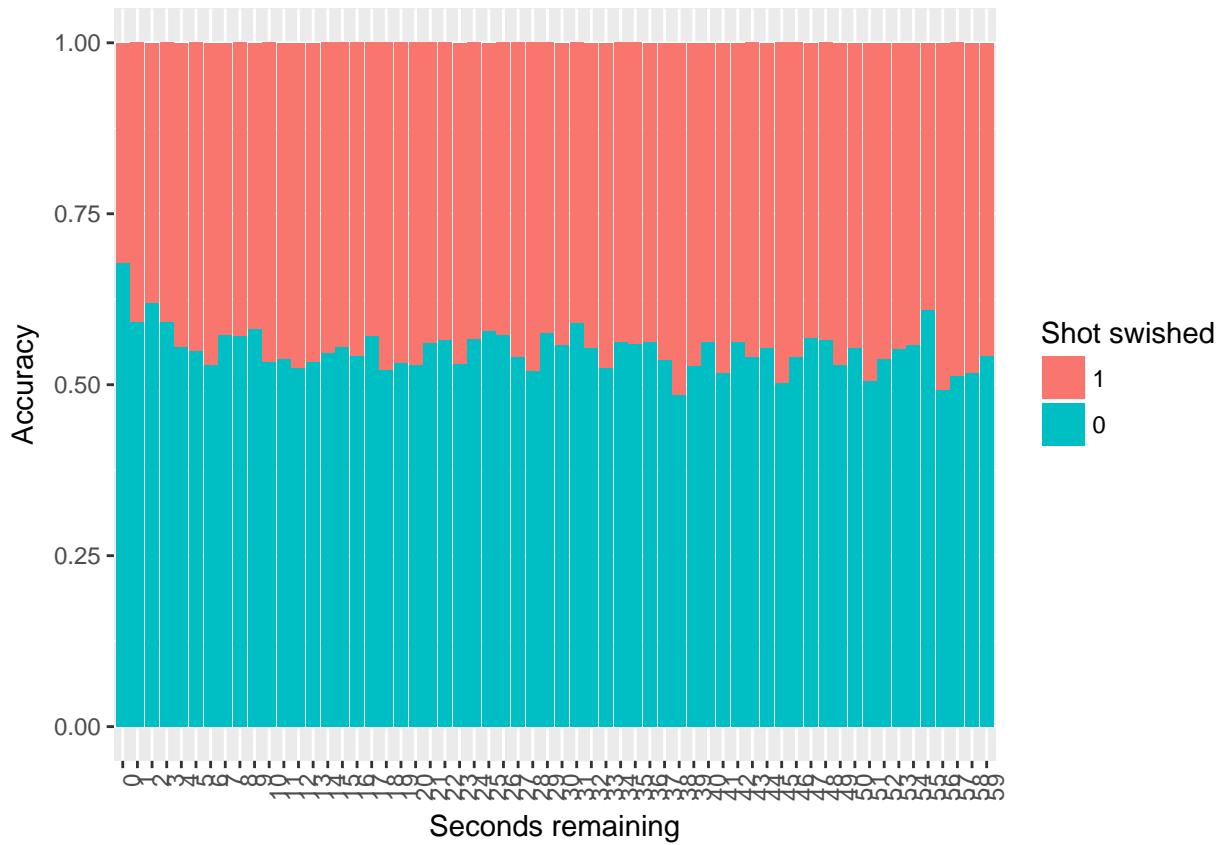


Así mismo, puede ser interesante estudiar las variables *minutes_remaining* y *seconds_remaining*. Estas propiedades expresan los segundos y minutos restantes en el momento del lanzamiento, respectivamente. Al visualizarlos, tampoco se aprecia que estas características tengan un impacto claro en la probabilidad de acierto/fallo. Se ha podido apreciar que lo mismo ocurre con las otras variables incluidas en esta categoría.

```
plotAccuracyByFeature(as.factor(train$minutes_remaining), "Minutes remaining")
```



```
plotAccuracyByFeature(as.factor(train$seconds_remaining), "Seconds remaining",  
90)
```



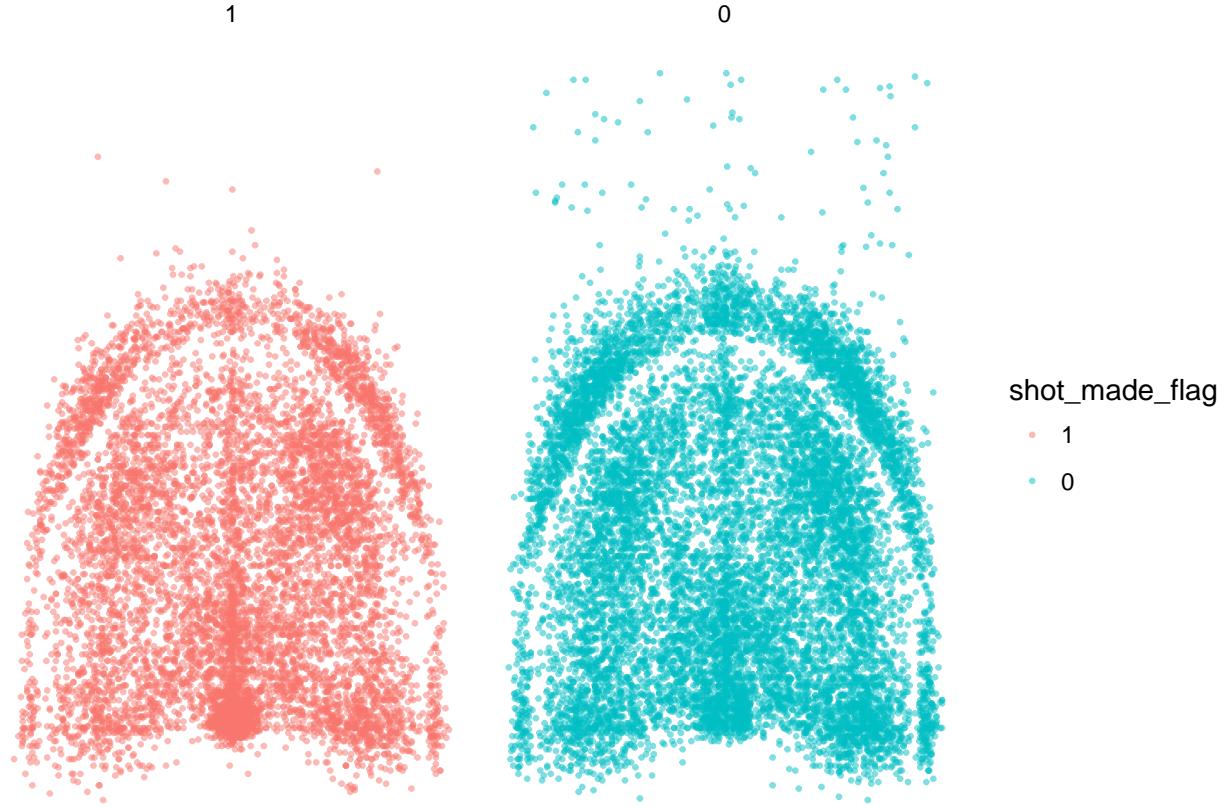
Finalmente, la variable *game_date* representa la fecha en la que se ha producido cada enfrentamiento y, por lo tanto, cada tiro. Así mismo, las variables *lat* y *lon* representan la latitud y longitud en las que se realizaron los tiros.

Variables referentes a la cancha

En primer lugar, prestamos atención a las variables *loc_x* y *loc_y*. Conjuntamente, representan la posición de la cancha de baloncesto en la que se realizó cada lanzamiento. Observamos como, a simple vista, no existe ninguna zona de la cancha con una mayor concentración de canastas o errores.

```
ggplot(train, aes(x = loc_x, y = loc_y)) + geom_point(aes(color = shot_made_flag),
  alpha = 0.5, size = 0.5) + ylim(c(-50, 400)) + theme_void() +
  facet_grid(~shot_made_flag)
```

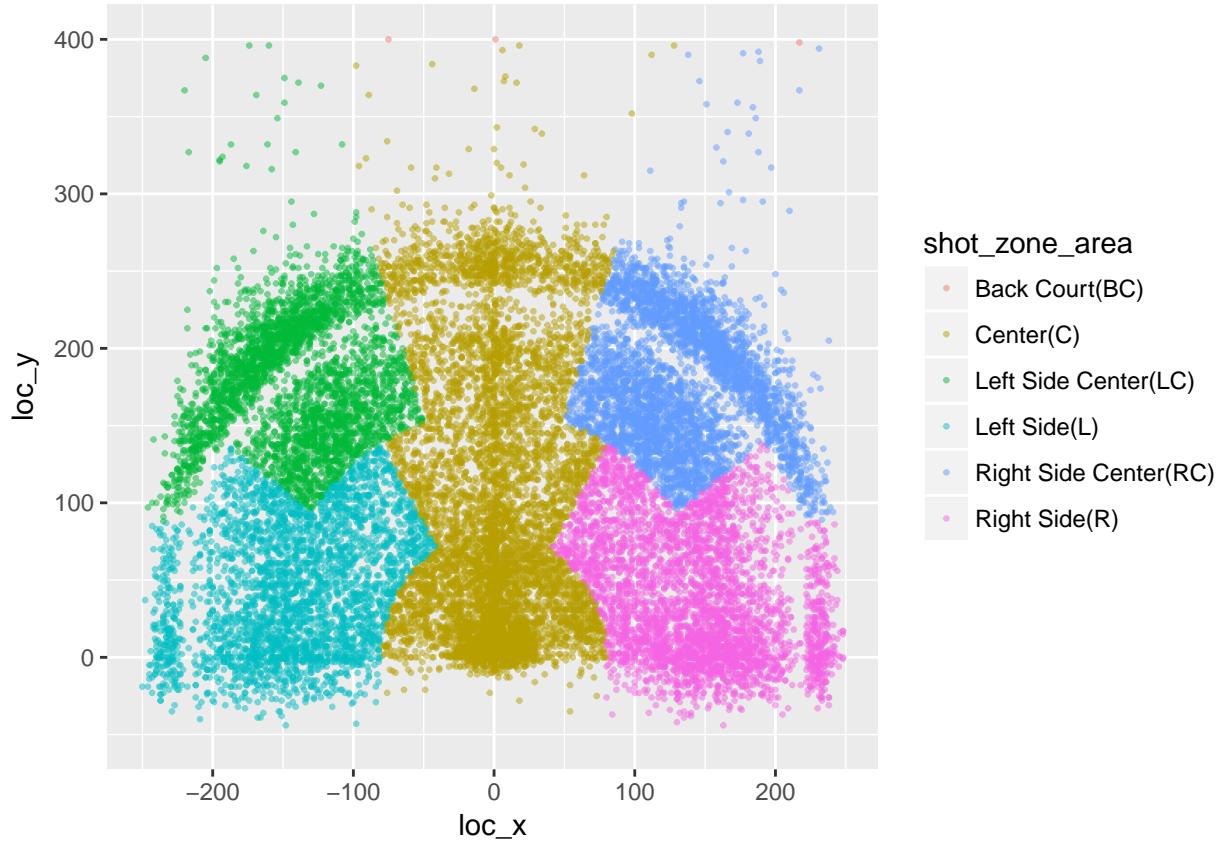
```
## Warning: Removed 69 rows containing missing values (geom_point).
```



Las variables *shot_zone_area*, *shot_zone_basic* y *shot_zone_range* representan la cancha dividida por zonas y expresan la zona en la que se produjo la acción. En primer lugar, mostramos las parcelas o divisiones de la cancha según cada variable:

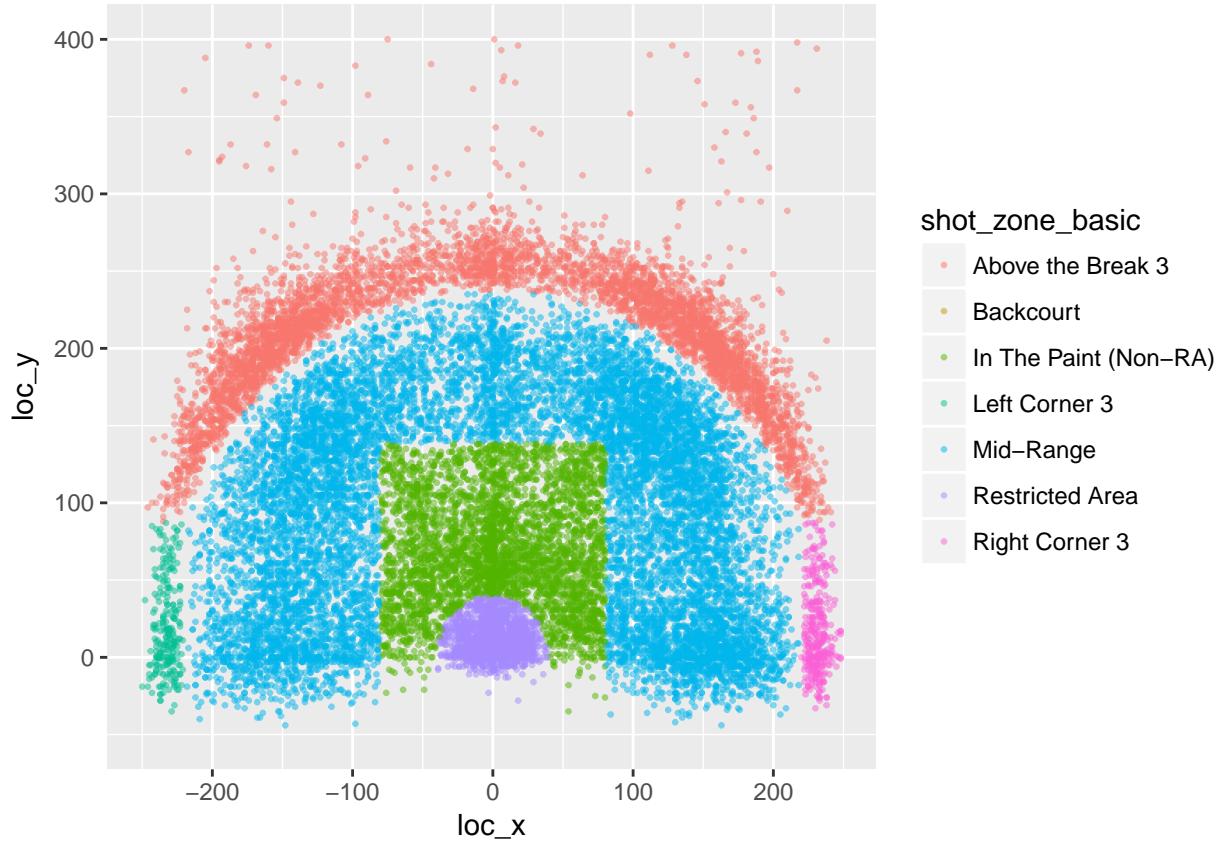
```
ggplot(train, aes(x = loc_x, y = loc_y)) + geom_point(aes(color = shot_zone_area),
alpha = 0.5, size = 0.5) + ylim(c(-50, 400)) + guides(fill = guide_legend(title = "Shot zone areas"))

## Warning: Removed 69 rows containing missing values (geom_point).
```



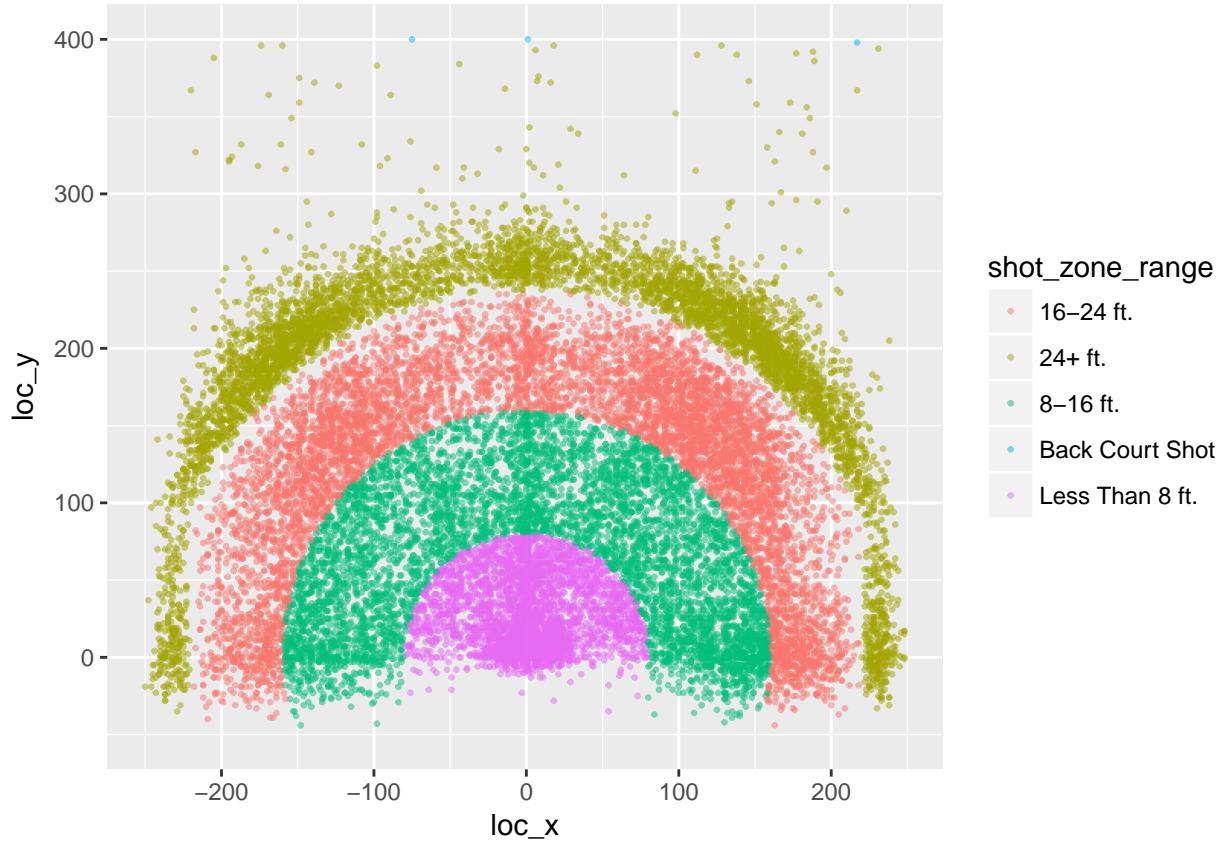
```
ggplot(train, aes(x = loc_x, y = loc_y)) + geom_point(aes(color = shot_zone_basic),
alpha = 0.5, size = 0.5) + ylim(c(-50, 400)) + guides(fill = guide_legend(title = "Shot zone basic"))

## Warning: Removed 69 rows containing missing values (geom_point).
```



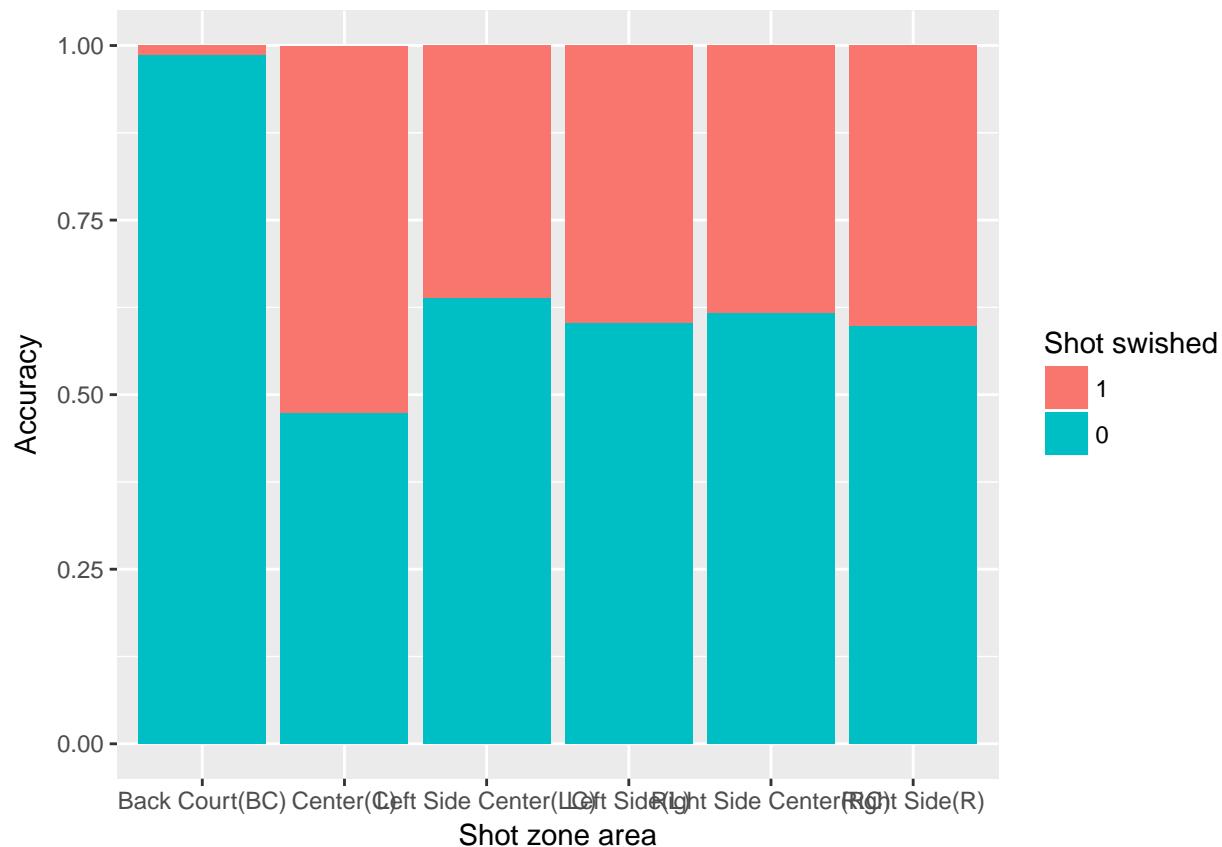
```
ggplot(train, aes(x = loc_x, y = loc_y)) + geom_point(aes(color = shot_zone_range),
alpha = 0.5, size = 0.5) + ylim(c(-50, 400)) + guides(fill = guide_legend(title = "Shot zone range"))

## Warning: Removed 69 rows containing missing values (geom_point).
```

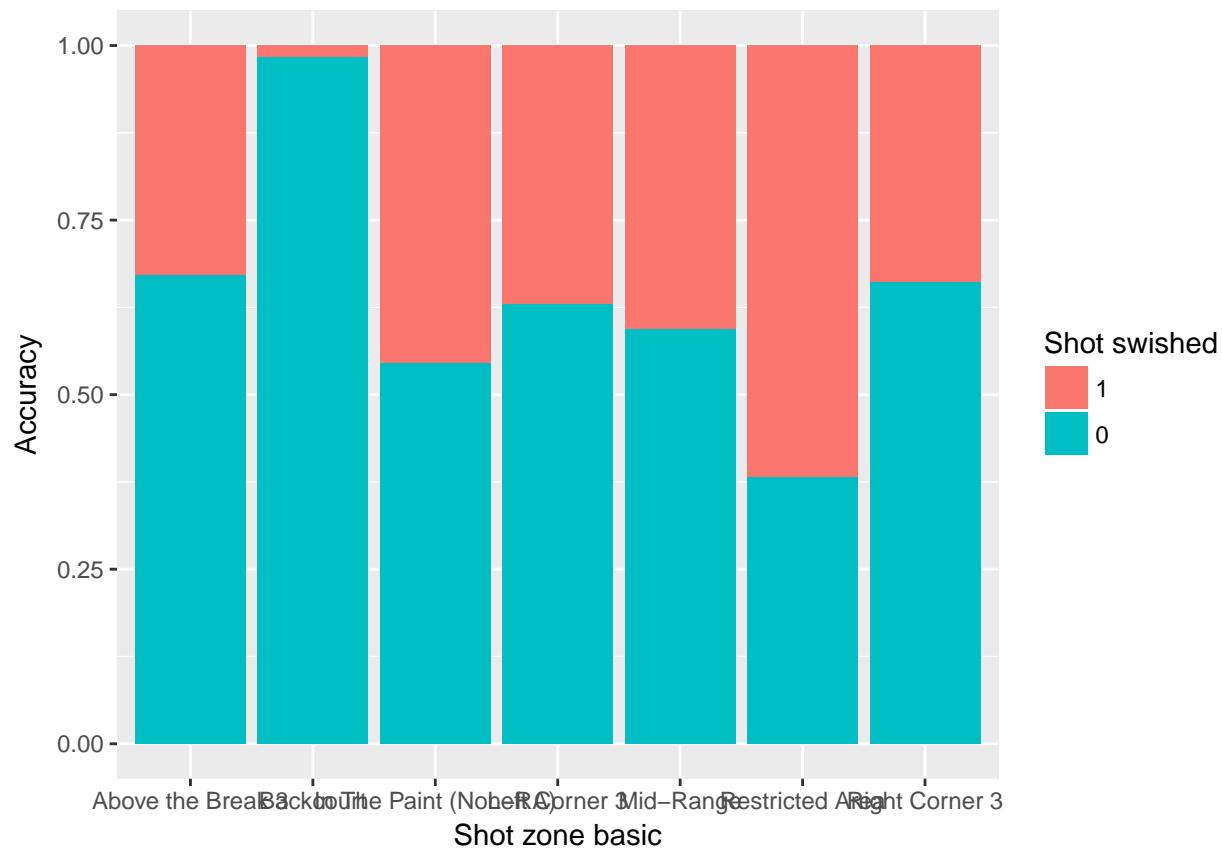


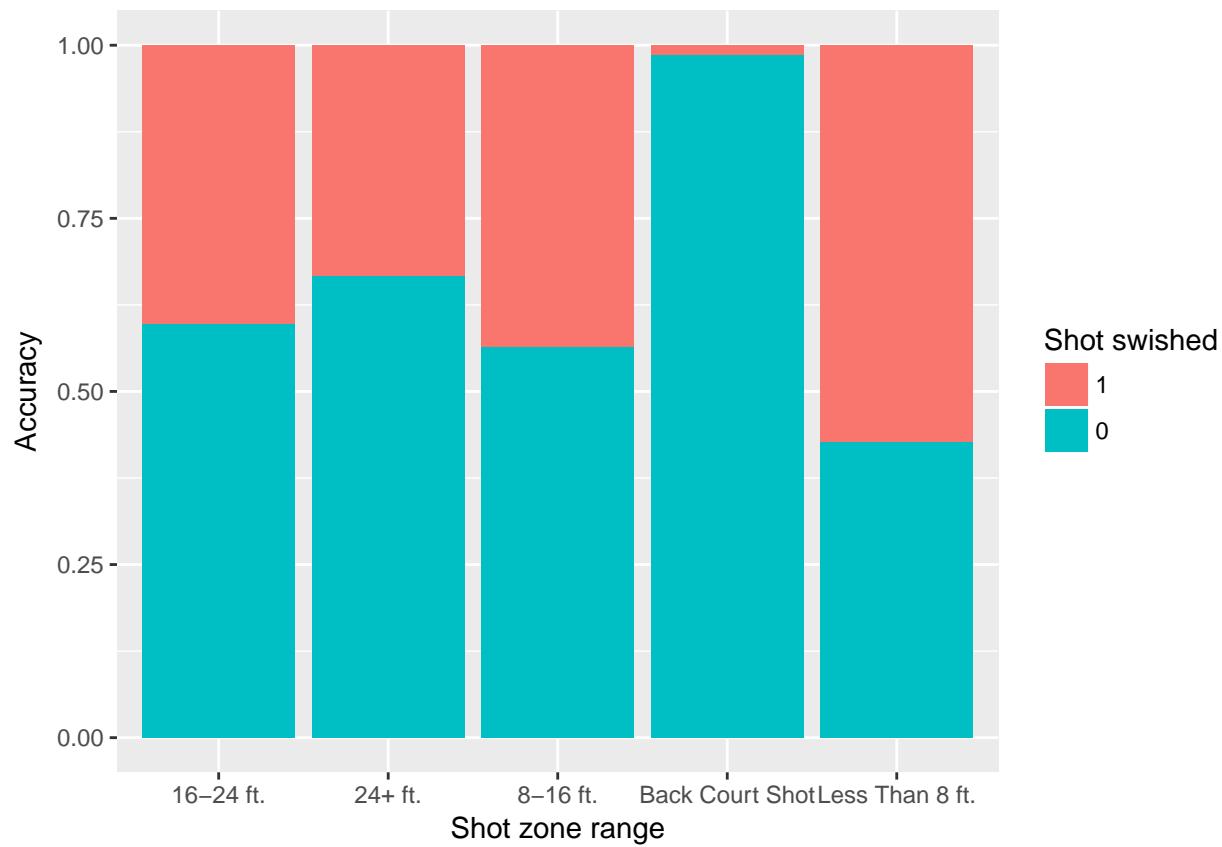
A continuación, se muestra el ratio de acierto en cada una de las zonas de la cancha.

```
plotAccuracyByFeature(as.factor(train$shot_zone_area), "Shot zone area")
```



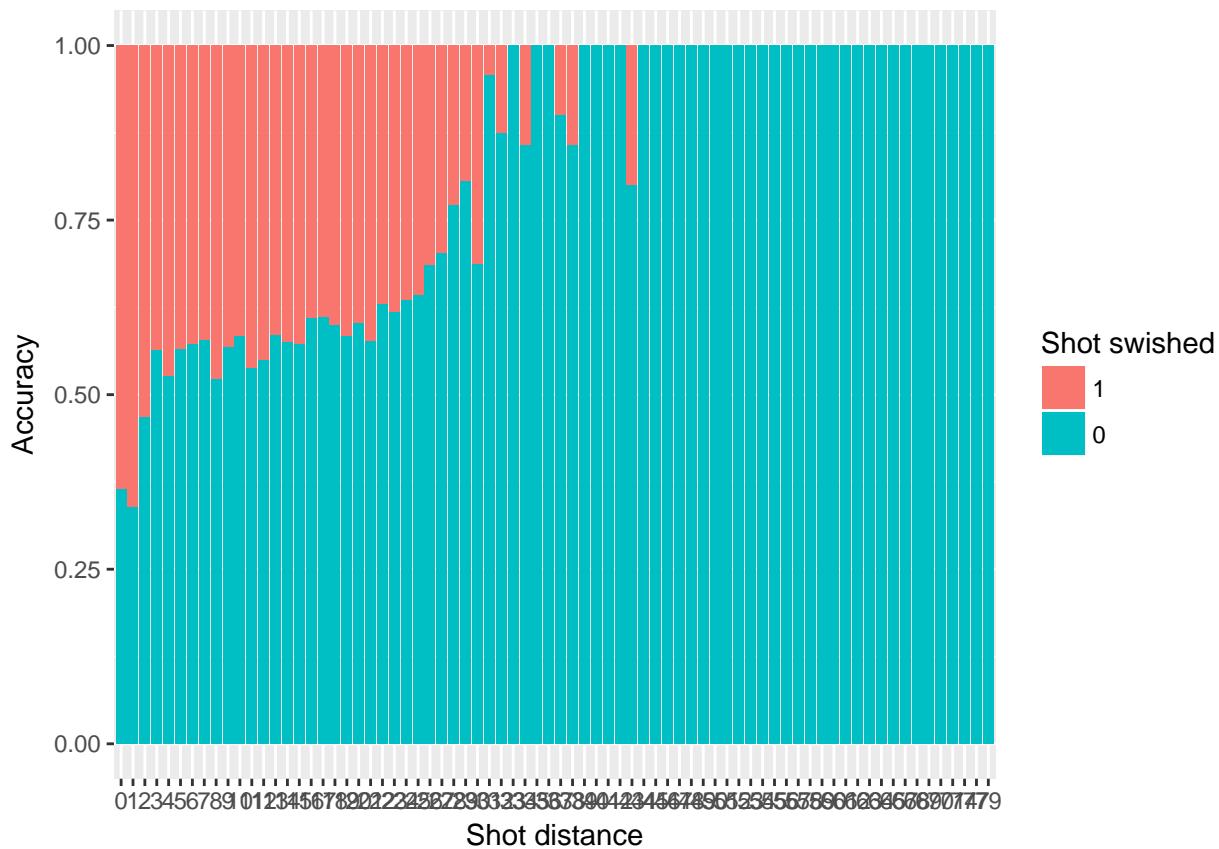
```
plotAccuracyByFeature(as.factor(train$shot_zone_basic), "Shot zone basic")
```





Finalmente, la variable *shot_distance* describe la distancia a la que se ha realizado el tiro, con respecto a la canasta.

```
plotAccuracyByFeature(as.factor(train$shot_distance), "Shot distance")
```



Transformación de los datos

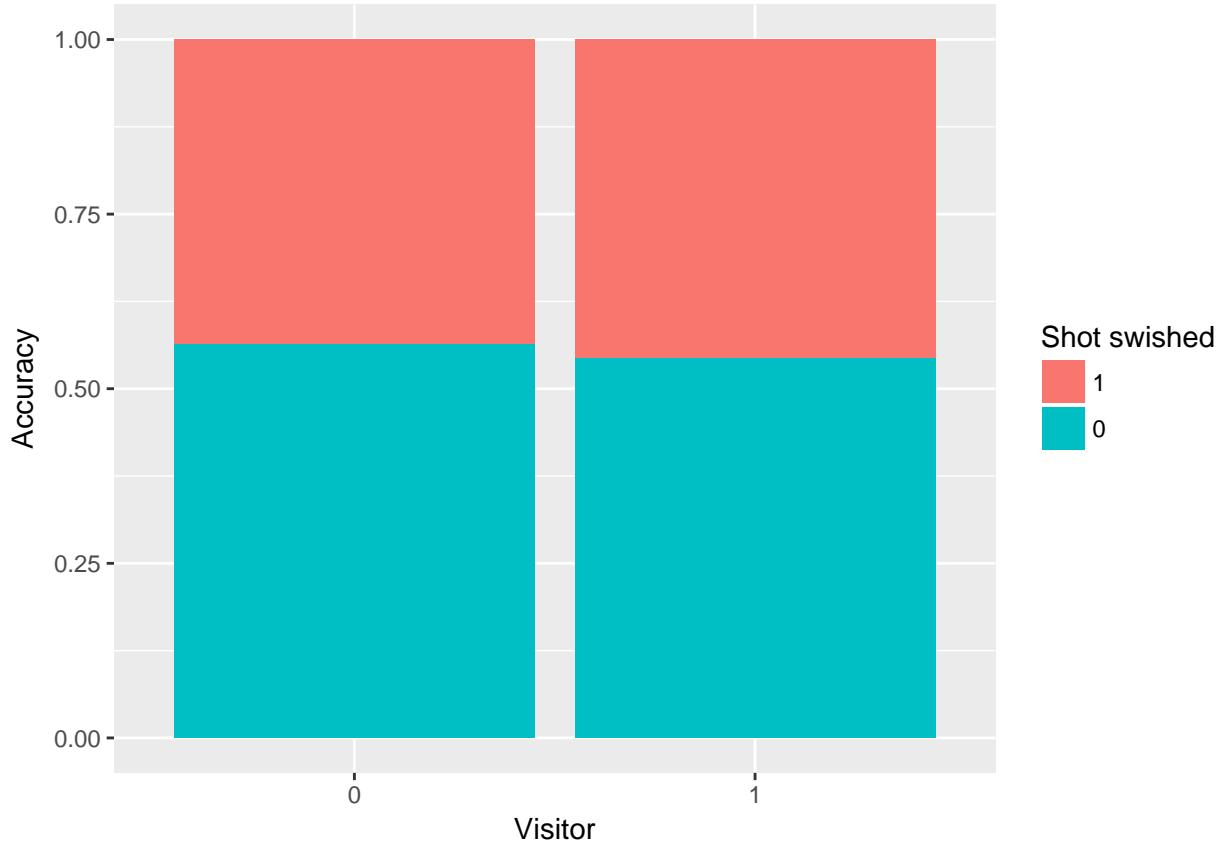
Teniendo en cuenta el análisis que características que se ha realizado, se procede a eliminar las variables que, tal y como se ha comentado en la sección anterior, no aportan información relevante.

```
train <- dplyr::select(train, -game_event_id)
train <- dplyr::select(train, -game_id)
train <- dplyr::select(train, -team_id)
train <- dplyr::select(train, -team_name)
train <- dplyr::select(train, -shot_id)
train <- dplyr::select(train, -game_date)
train <- dplyr::select(train, -lat)
train <- dplyr::select(train, -lon)
```

A continuación, utilizamos la variable *matchup* para obtener información algo más valiosa que el nombre del enfrentamiento. Los enfrentamientos que contienen el símbolo “@*” representan enfrentamientos como equipo local. Por consiguiente, los partidos que contienen “VS” en el nombre, son partidos jugados como visitante. Ergo, creamos la variable *visitor** con el fin de clarificar el rol del equipo en el partido. Aún así, no se observa cambio en la efectividad en función de jugar un partido como visitante o como local.

```
train$visitor <- ifelse(grepl("@", train$matchup), 0, 1)
train$visitor <- as.factor(train$visitor)
train <- dplyr::select(train, -matchup)

plotAccuracyByFeature(as.factor(train$visitor), "Visitor")
```



Existen algunos tipos de métodos analíticos que tienen problemas al tratar variables categóricas. Por lo tanto, se procede a la conversión de dicho tipo de variables a numéricas. Para ello, se asigna a cada categoría un entero, generado de forma incremental en cada característica.

```
categorical.features <- c("action_type", "combined_shot_type",
  "season", "shot_type", "shot_zone_area", "shot_zone_basic",
  "shot_zone_range", "opponent", "visitor")

for (feature in categorical.features) {
  if (is.factor(train[, feature])) {
    levels(train[, feature]) <- c(1:length(levels(train[, feature])))
    train[, feature] <- as.numeric(train[, feature])
  }
}
```

Sin embargo, al tratarse de una tarea de clasificación, necesitamos que nuestra variable clase sea de tipo categórico. Así que procedemos a asignar el tipo *SUCCESS* a aquellos tiros acertados (1) y *ERROR* a las canastas fallidas (0).

```
train$shot_made_flag <- as.factor(ifelse(train$shot_made_flag ==
  1, "SUCCESS", "ERROR"))
train$shot_made_flag <- factor(train$shot_made_flag, levels = c("SUCCESS",
  "ERROR"))
```

Dado que el conjunto de test no contiene las etiquetas de la variable clase, no podremos validar nuestro modelo contra el conjunto de test. Aunque sí podremos comprobar el *score* que la plataforma *Kaggle* asigne a nuestra predicción. Por lo tanto, particionaremos el conjunto de entrenamiento en dos subconjuntos: uno

para un entrenamiento parcial del modelo y otro para realizar la validación de tal modelo.

Para realizar tal partición, se podía valorar el uso de la función *createFolds* con $k = 2$, pero se obtendrían dos subconjuntos balanceados aunque de dimensiones casi idénticas. Por otra parte, la función *createResample* crea las particiones utilizando *bootstrapping*, con lo que podríamos tener instancias presentes en ambos subconjuntos. Nos interesa crear dos subconjuntos balanceados y de unas proporciones de 70% para el subconjunto de entrenamiento y 30% para el de test (con respecto al conjunto original). Para ello, utilizamos la función *createDataPartition* con el parámetro $p = .7$.

```
set.seed(SEED)
inTrain <- caret::createDataPartition(y = train$shot_made_flag,
  p = 0.7, list = FALSE)
subset.train <- train[inTrain, ]
subset.test <- train[-inTrain, ]
```

Entrenamiento y validación

En primer lugar, utilizamos la función *trainControl* para generar los parámetros que, más tarde, utilizaremos para controlar el entrenamiento del modelo. De esta forma controlaremos el tipo de estimación del error. En este caso, utilizaremos una validación cruzada de 10 hojas o *folds* (por defecto). Dicha validación la repetiremos 3 veces, utilizando el parámetro *repeats = 3* conjuntamente con el parámetro *method = "repeatedcv"* (ya que si empleasemos *method = "cv"* no podríamos seleccionar repeticiones). Así mismo, nos interesa obtener las probabilidades con las que cada instancia pertenece a cada clase, para ello utilizamos el parámetro *classProbs = TRUE*.

La evaluación de esta competición se realiza utilizando la métrica *Log Loss*, por lo que será interesante utilizar dicha métrica para seleccionar el modelo óptimo. *Logarithmic loss* mide el rendimiento de un modelo de clasificación, en el cual la predicción viene dada por un valor de probabilidad entre 0 y 1. Idealmente, un modelo perfecto predicaría con una *Log Loss* de 0, dado que la métrica se incrementa cuando la probabilidad predicha diverge de la etiqueta real.

Al no ser una métrica que proporcione por defecto el paquete *caret*, definimos la función *LogLoss* para incluirla por medio del parámetro *summaryFunction*.

```
LogLoss <- function(data, lev = NULL, model = NULL) {
  obs <- data[, "obs"]
  cls <- levels(obs) # find class names
  probs <- data[, cls[2]] # use second class name
  probs <- pmax(pmin(as.numeric(probs), 1 - 1e-15), 1e-15) # bound probability
  logPreds <- log(probs)
  log1Preds <- log(1 - probs)
  real <- (as.numeric(data$obs) - 1)
  out <- c(mean(real * logPreds + (1 - real) * log1Preds)) *
    -1
  names(out) <- c("LogLoss")
  out
}

set.seed(SEED)
control <- caret::trainControl(method = "repeatedcv", repeats = 3,
  classProbs = TRUE, summaryFunction = LogLoss)
```

glmnet

Uno de los métodos elegidos para generar nuestro modelo de clasificación es *glmnet* (*Lasso and Elastic-Net Regularized Generalized Linear Models*). Este algoritmo utiliza descenso cíclico coordinado, en el cual optimiza sucesivamente la función objetivo sobre cada parámetro, hasta llegar a converger. También utiliza los parámetros *alpha* y *lambda*. El primero se emplea para modificar el valor de “mezcla” de la regularización *elastic net*; tomando valores entre 1 (lasso) y 0 (ridge). El segundo parámetro *lambda* se calcula en función de del valor de *alpha* y el número de valores de la secuencia (por defecto 100).

```
glmnet.info <- caret::getModelInfo("glmnet")
glmnet.info$glmnet$parameters
```

```
##   parameter    class           label
## 1     alpha   numeric      Mixing Percentage
## 2     lambda  numeric Regularization Parameter
```

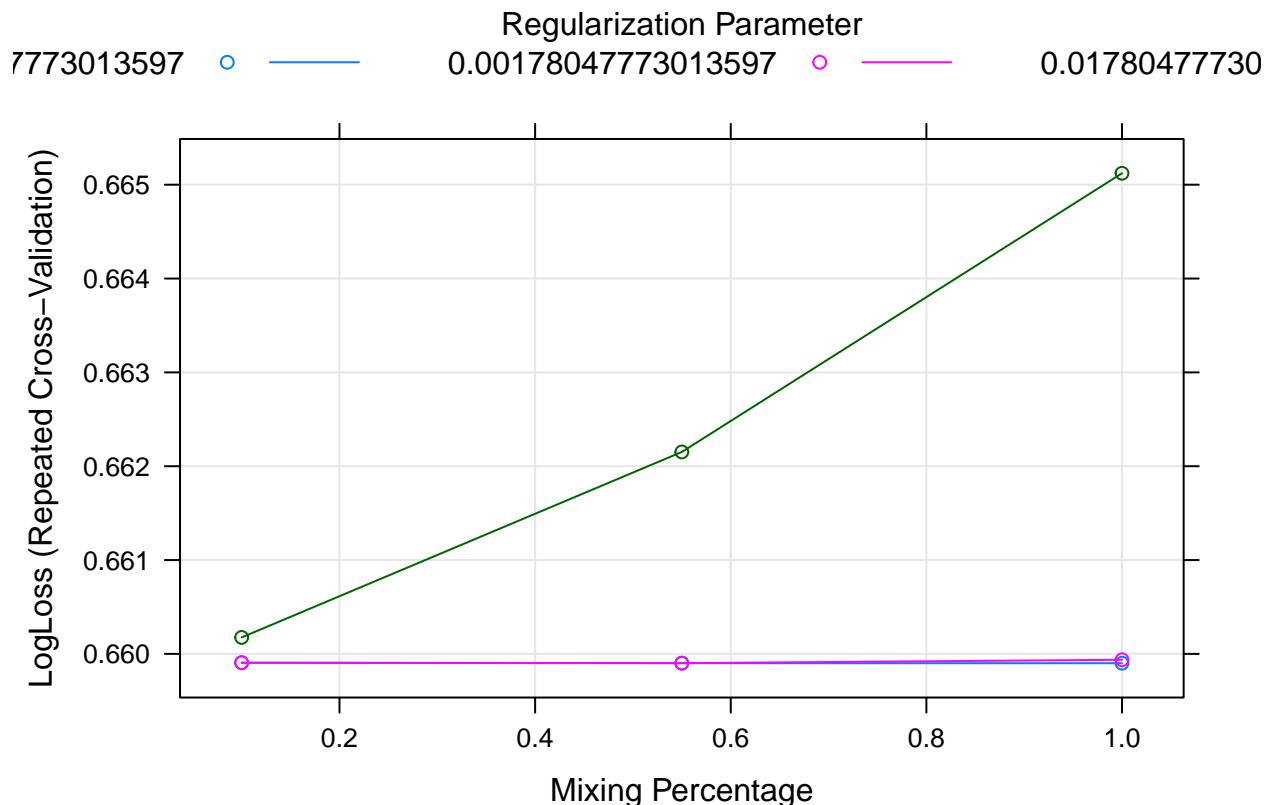
A continuación, en el modelo *modelGLM.default* realizamos un entrenamiento utilizando 3^2 combinaciones de dichos parámetros. Y en el modelo *modelGLM.custom* aplicamos el parámetro *tuneLength = 20*, por lo que combinaremos 20 valores de cada uno de los parámetros, obteniendo un total de 20^2 combinaciones.

Por medio del parámetro *preProcess*, se aplican las siguientes acciones de preprocessado: *center* sustrae la media de cada variable a todos valores de la misma; mientras *scale* divide dichos valores por la desviación típica. En cuanto al método *pca* aplica la técnica de análisis de componentes principales (PCA), con el fin de reducir la dimensionalidad del conjunto de datos.

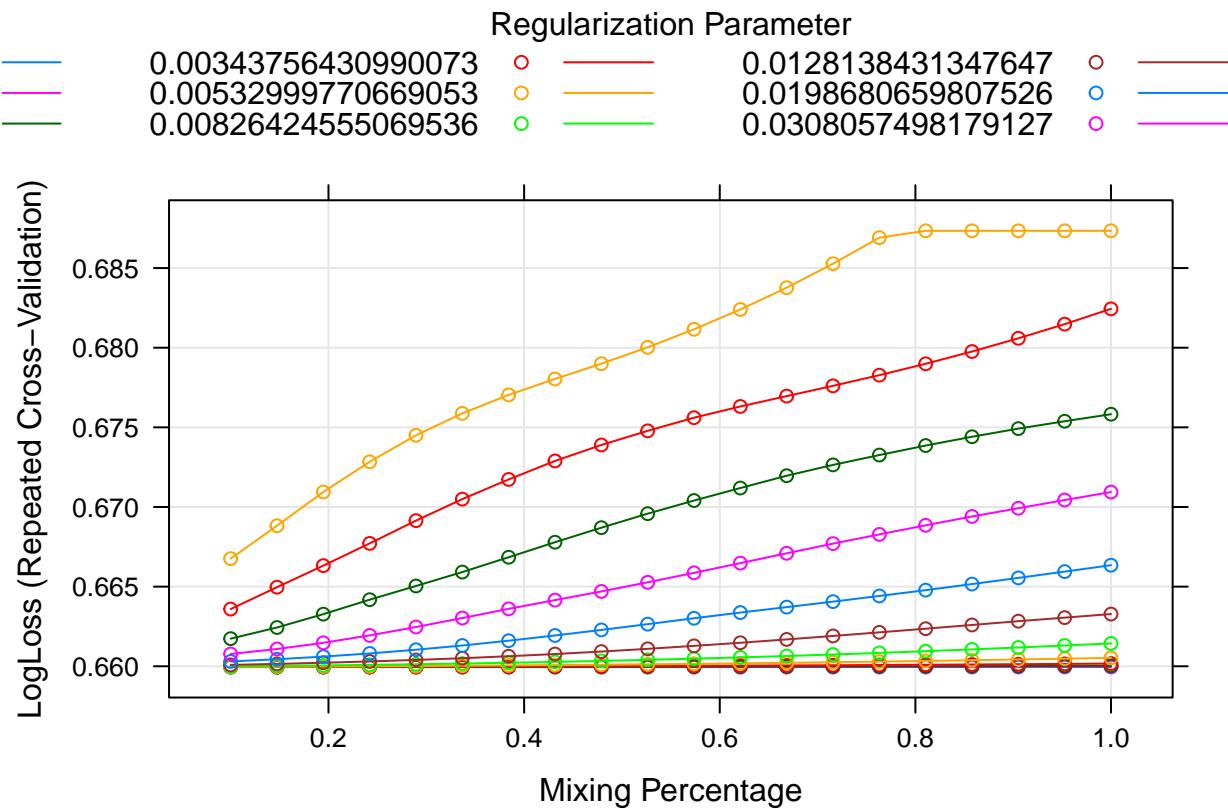
```
modelGLM.default <- caret::train(shot_made_flag ~ ., data = subset.train,
  method = "glmnet", trControl = control, preProcess = c("center",
  "scale", "pca"), metric = "LogLoss", maximize = FALSE)
```

```
modelGLM.custom <- caret::train(shot_made_flag ~ ., data = subset.train,
  method = "glmnet", trControl = control, preProcess = c("center",
  "scale", "pca"), tuneLength = 20, metric = "LogLoss",
  maximize = FALSE)
```

```
plot(modelGLM.default)
```



```
plot(modelGLM.custom)
```



fda

A continuación, entrenaremos el modelo utilizando un análisis discriminante flexible (*Flexible Discriminant Analysis*). Este tipo de modelo de clasificación se basa en una combinación de modelos de regresión lineal. Utiliza un *scoring* óptimo para transformar la variable de respuesta, de forma que los datos sean más fáciles de separar linealmente. Así mismo, utiliza múltiples *splines* adaptativos para generar la superficie discriminante.

```
fpa.info <- caret::getModelInfo("fda")
fpa.info$fda$parameters
```

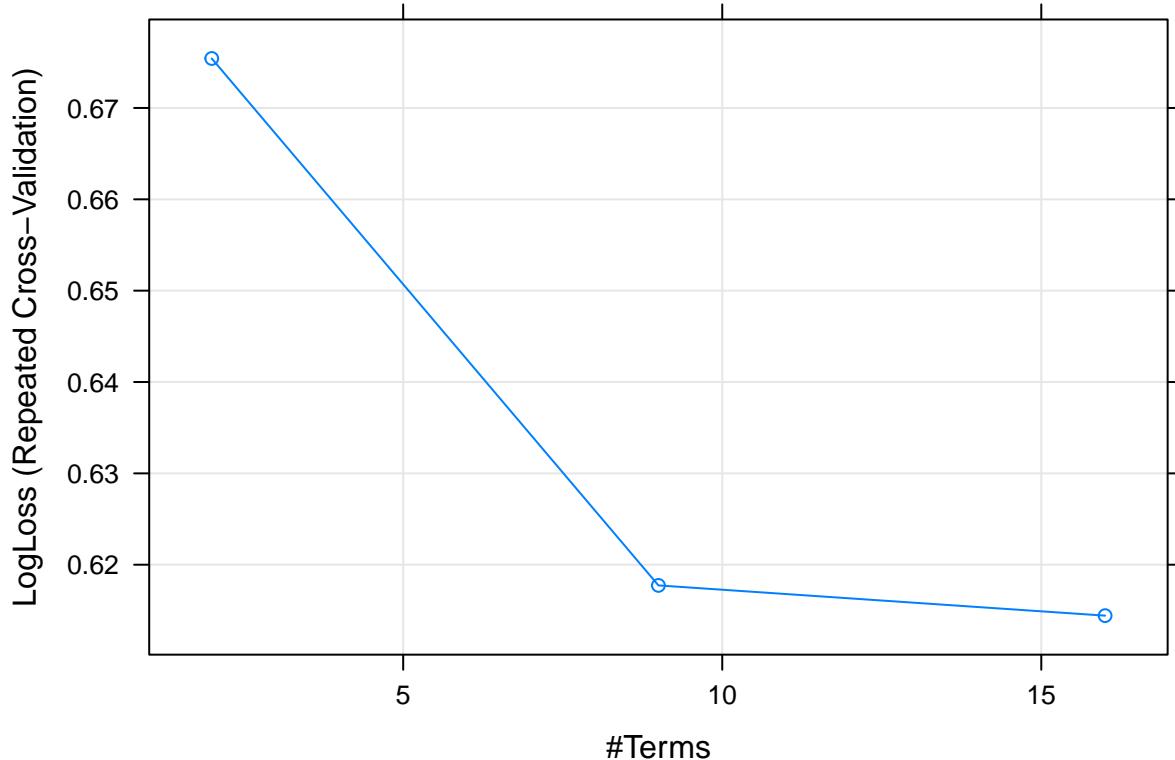
```
##   parameter    class      label
## 1     degree  numeric Product Degree
## 2     nprune  numeric      #Terms
```

Tal y como podemos observar, *fda* contiene dos parámetros: *degree* y *nprune*. A continuación, se procede a entrenar el modelo con el conjunto de entrenamiento que creamos en la sección anterior. En primer lugar, empleamos el número por defecto de valores por parámetro utilizando un total de 3^2 combinaciones. Así mismo, a través del argumento *tunelength* asignamos 20 con el fin de obtener una mayor amplia variedad de 20^2 combinaciones.

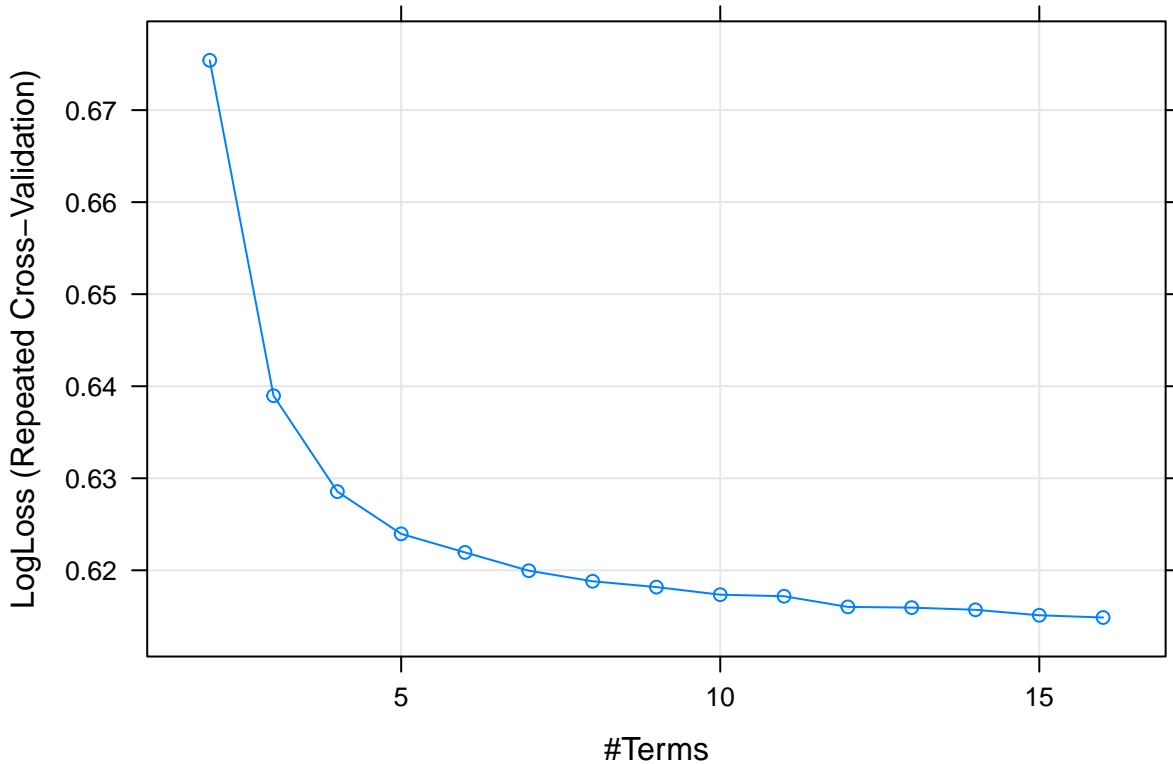
```
modelFDA.default <- caret::train(shot_made_flag ~ ., data = subset.train,
  method = "fda", trControl = control, preProcess = c("center",
  "scale"), metric = "LogLoss", maximize = FALSE)

modelFDA.custom <- caret::train(shot_made_flag ~ ., data = subset.train,
  method = "fda", trControl = control, preProcess = c("center",
```

```
"scale"), tuneLength = 20, metric = "LogLoss", maximize = FALSE)  
plot(modelFDA.default)
```



```
plot(modelFDA.custom)
```



Validación

Ahora se procede a realizar la predicción sobre el subconjunto de validación. Dado que necesitamos obtener las probabilidades asociadas a la clasificación de cada instancia en cada clase. Así mismo, dado que el *score* utilizado por *Kaggle* es *Log Loss*, una vez obtenidas las predicciones sobre el subconjunto de test, se procede a calcular dicha métrica para todos los modelos entrenados.

```
validation.partition.y <- ifelse(subset.test$shot_made_flag ==
  "SUCCESS", 1, 0)
validation.partition.test <- dplyr::select(subset.test, -shot_made_flag)

prediction.glm.default <- predict(modelGLM.default, newdata = validation.partition.test,
  type = "prob")
score.glm.default.validation <- MLmetrics::LogLoss(y_pred = prediction.glm.default$SUCCESS,
  y_true = validation.partition.y)
score.glm.default.validation

## [1] 0.661074

prediction.glm.custom <- predict(modelGLM.custom, newdata = validation.partition.test,
  type = "prob")
score.glm.custom.validation <- MLmetrics::LogLoss(y_pred = prediction.glm.custom$SUCCESS,
  y_true = validation.partition.y)
score.glm.custom.validation

## [1] 0.6610684
```

```

prediction.fda.default <- predict(modelFDA.default, newdata = validation.partition.test,
  type = "prob")
score.fda.default.validation <- MLmetrics::LogLoss(y_pred = prediction.fda.default$SUCCESS,
  y_true = validation.partition.y)
score.fda.default.validation

## [1] 0.6107533

prediction.fda.custom <- predict(modelFDA.custom, newdata = validation.partition.test,
  type = "prob")
score.fda.custom.validation <- MLmetrics::LogLoss(y_pred = prediction.fda.custom$SUCCESS,
  y_true = validation.partition.y)
score.fda.custom.validation

## [1] 0.6107533

```

Dado que ambos modelos (*fda* y *glmnet*) han sido entrenados con las mismas hojas (no se ha modificado la semilla), podemos comparar los resultados de la crossvalidación de los mismos. Podemos observar que el *p*-valor es menor a $< 2.2e-16$, por consiguiente la probabilidad de que rechacemos la hipótesis nula erroneamente, es muy baja. Ergo, podemos decir que hay una diferencia real entre ambos modelos. También se aprecia que la diferencia de medias entre ambos modelos es de *0.04432*. Al ser una media positiva y *fda* el segundo modelo, se puede decir que este obtiene unos mejores valores con *fda*.

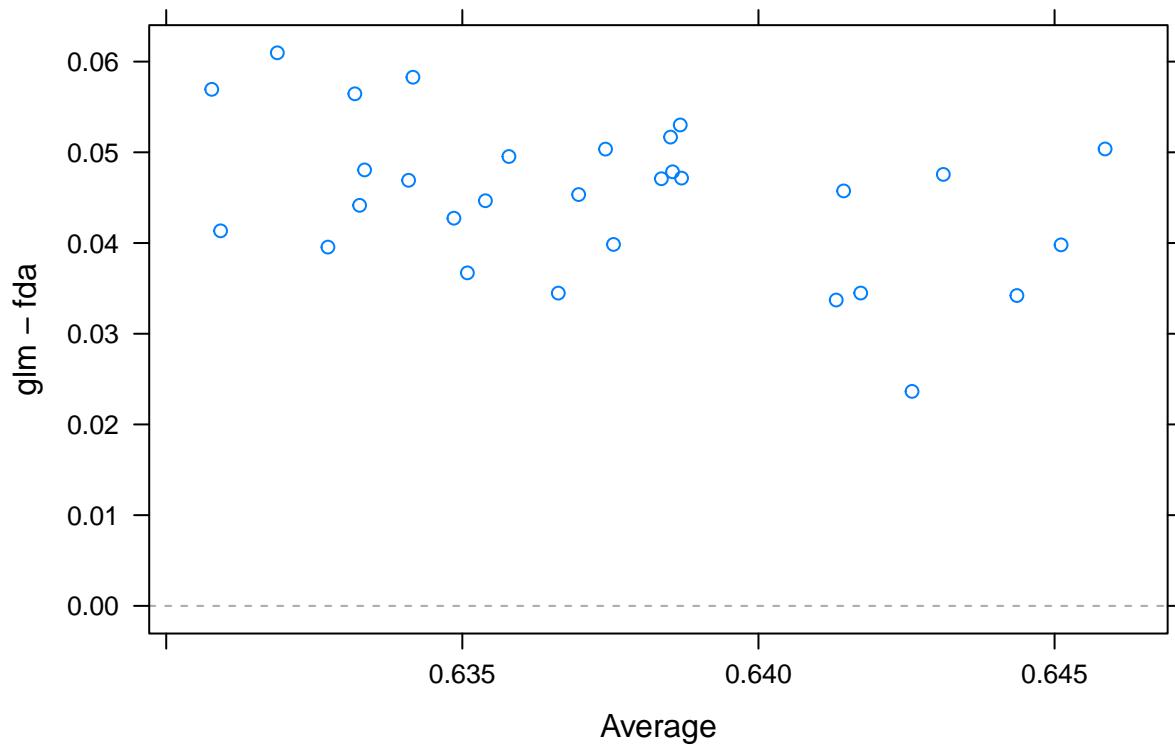
```

resamples.result <- caret::resamples(list(glm = modelGLM.custom,
  fda = modelFDA.custom))
summary(resamples.result)

##
## Call:
## summary.resamples(object = resamples.result)
##
## Models: glm, fda
## Number of resamples: 30
##
## LogLoss
##      Min.    1st Qu.     Median      Mean    3rd Qu.      Max. NA's
## glm 0.6515897 0.6574023 0.6600950 0.6599560 0.6625657 0.6710369    0
## fda 0.6013916 0.6110606 0.6138912 0.6148677 0.6191463 0.6307716    0
lattice::xyplot(resamples.result, what = "BlandAltman")

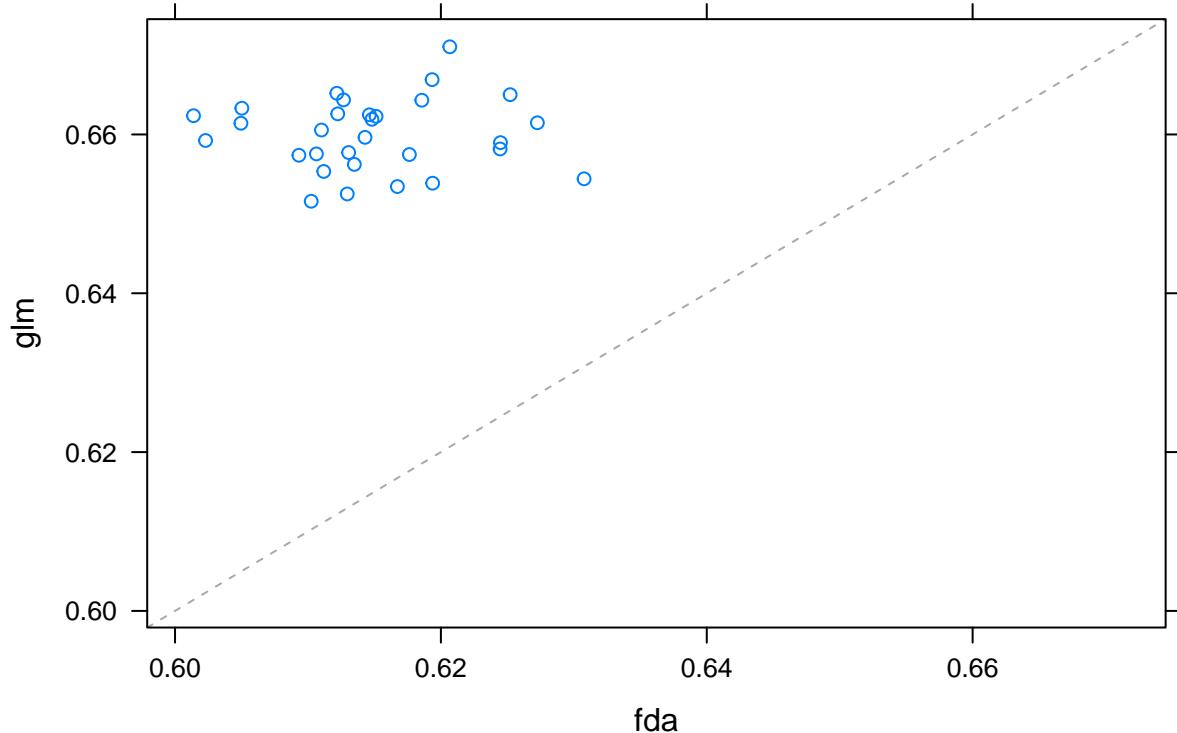
```

LogLoss



```
lattice::xyplot(resamples.result, what = "scatter")
```

LogLoss



```
diffs <- diff(resamples.result)
summary(diffs)

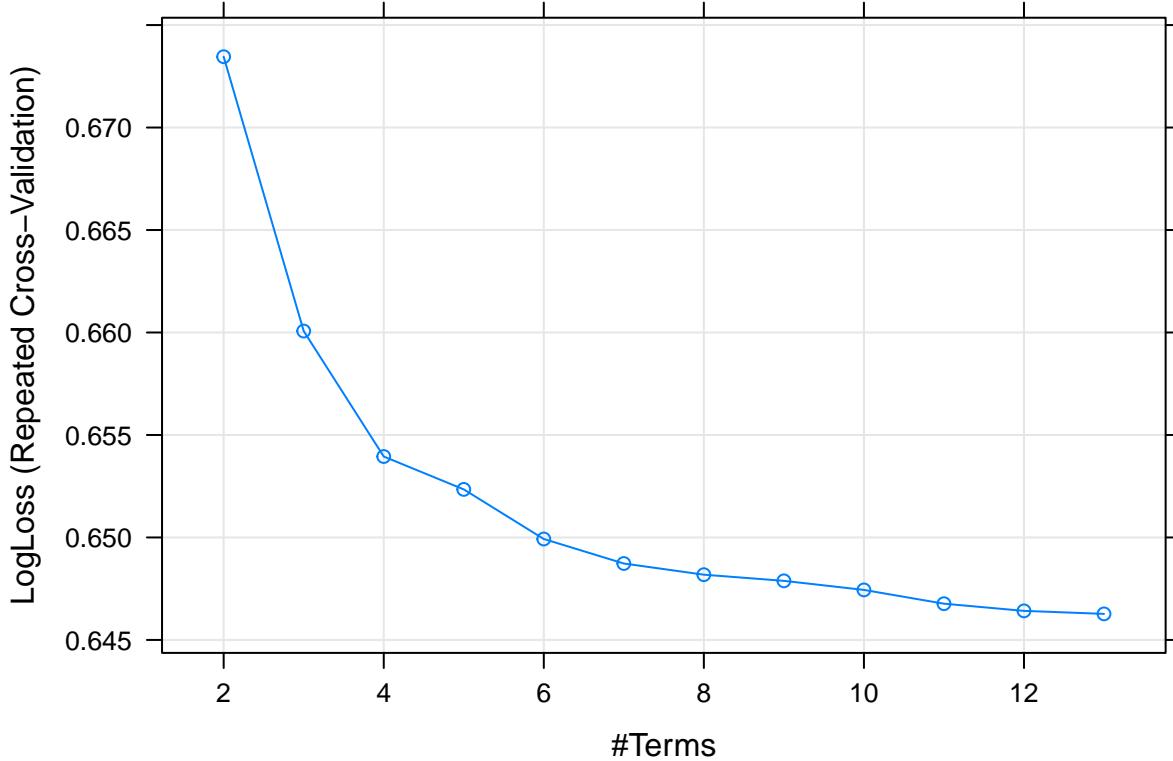
##
## Call:
## summary.diff.resamples(object = diffs)
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## LogLoss
##      glm      fda
##      glm      0.04509
##      fda < 2.2e-16
```

Generación del modelo final y predicción

Ahora procedemos a realizar el entrenamiento del modelo utilizando todas las instancias que conforman el conjunto de entrenamiento que nos proporciona *Kaggle*. Para ello, utilizamos el método que mejores valores ha obtenido en la validación cruzada: *fda* con *tuneLength* = 20.

```
model <- caret::train(shot_made_flag ~ ., data = train, method = "fda",
  trControl = control, preProcess = c("center", "scale", "pca"),
  tuneLength = 20, metric = "LogLoss", maximize = FALSE)
```

```
plot(model)
```



Antes de generar una clasificación sobre el conjunto de test, necesitamos eliminar y transformar las variables tal y como lo hicimos en el conjunto de entrenamiento. El modelo ha sido entrenado para “comprender” unas determinadas variables de determinados tipos, por lo que es necesario aplicar las mismas operaciones.

```
shot_id <- test$shot_id

test <- dplyr::select(test, -game_event_id)
test <- dplyr::select(test, -game_id)
test <- dplyr::select(test, -team_id)
test <- dplyr::select(test, -team_name)
test <- dplyr::select(test, -shot_id)
test <- dplyr::select(test, -game_date)
test <- dplyr::select(test, -lat)
test <- dplyr::select(test, -lon)

test$visitor <- ifelse(grepl("@", test$matchup), 0, 1)
test$visitor <- as.factor(test$visitor)
test <- dplyr::select(test, -matchup)

for (feature in categorical.features) {
  if (is.factor(test[, feature])) {
    levels(test[, feature]) <- c(1:length(levels(test[, feature])))
    test[, feature] <- as.numeric(test[, feature])
  }
}
```

```
}
```

Finalmente, generamos la clasificación sobre el conjunto de test que nos ha proporcionado *Kaggle*. El *score* proporcionado por dicha plataforma, basado en la métrica *LogLoss*, es de **0.64530**.

```
prediction <- predict(model, newdata = test, type = "prob")
prediction.table <- data.frame(shot_id = shot_id, shot_made_flag = prediction$SUCCESS)
write.csv(prediction.table, "prediction.csv", row.names = FALSE)
```

Conclusiones

Cabe destacar que los valores de *Logarithmic Loss* obtenidos a lo largo de flujo de trabajo, no han sido muy satisfactorios. Sin embargo, la mejor puntuación que consta en esta competición ha sido de *0.56528*. Esto se debe a la elevada dificultad de generar un modelo fiable con los datos del dataset. Tal y como observamos al estudiar cada una de las características, no existe ninguna variable que muestre una clara relación con la variable clase. Además, pensado en el dominio del problema, existen muchas circunstancias de los partidos de las que no tenemos información y pueden ser decisivas. Por ejemplo, distancia con respecto al jugador más próximo en el momento del tiro, estado de forma del jugador, estado de forma del equipo, cansancio, minutos jugados por el jugador en el momento de cada tiro, etc.

Bibliografía

- <https://cran.r-project.org/web/packages/fda/fda.pdf>
- <http://trymachinelearning.com/machine-learning-algorithms/dimensionality-reduction/flexible-discriminant-analysis/>
- https://www.researchgate.net/publication/2889611_Flexible_Discriminant_Analysis_by_Optimal_Scoring
- https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html
- https://en.wikipedia.org/wiki/Elastic_net_regularization
- <https://cran.r-project.org/web/packages/glmnet/index.html>
- http://wiki.fast.ai/index.php/Log_Loss