

# Análisis de batallas en World of Warcraft

Sergi Fornés

## Índice

<b>Introducción</b>	<b>1</b>
<b>Datos</b>	<b>1</b>
<b>Análisis Exploratorio</b>	<b>3</b>
<b>Aprendizaje estadístico</b>	<b>10</b>
Análisis de Componentes Principales . . . . .	10
Clustering . . . . .	13
Predicción de ganadores . . . . .	18
Predicción de asesinatos . . . . .	23

## Introducción

El objetivo de este trabajo es analizar batallas del videojuego *World of Warcraft* y crear diversos modelos que sean capaces de predecir si un jugador ganará una batalla y cuántos enemigos habrá matado. *World of Warcraft* es un videojuego de rol multijugador en línea en el que cada jugador se crea su propio personaje para luchar. En el momento de la creación del personaje se debe elegir en cual de las dos facciones jugará, los personajes de la misma facción son aliados, mientras que los de la facción contraria son enemigos. También se elige la clase del personaje, la cual influye en sus habilidades y características.

Entre otras cosas, los personajes pueden participar en campos de batalla en los que dos equipos luchan para ganar. Hay diversos campos de batalla y en cada uno el objetivo es diferente, el equipo que primero cumpla el objetivo gana la batalla.

También hay que mencionar que los personajes entran en el campo de batalla con un rol específico, el rol puede ser de ataque (*dps*) o de apoyo (*heal*). Hay algunas clases que únicamente pueden ser de ataque, mientras que otras pueden ser tanto de ataque como de apoyo, aunque no se puede cambiar de rol durante una batalla.

## Datos

Los datos para realizar este análisis han sido recogidos por un jugador durante algunos meses participando en campos de batalla. Se pueden encontrar en el siguiente *link*.

Es importante señalar que World of Warcraft actualiza su juego añadiendo parches y expansiones cada cierto tiempo, cambiando las estadísticas de los personajes del juego, por lo que estos datos no sirven para hacer predicciones en batallas actuales ya que fueron extraídos durante la sexta expansión (*Legión*).

Estos datos están divididos en 10 ficheros diferentes, aunque se podría decir que son 5, ya que la mitad son actualizaciones de los ficheros iniciales. Al existir diferentes objetivos según el campo de batalla, en 4 campos específicos se ofrece información extra relacionada con dichos objetivos. Con lo cual, la información de estos campos de batalla estará guardada en diferentes ficheros con más columnas, y la información del resto de campos de batalla estará en un fichero común.

Los ficheros cuentan con un total de 14.650 observaciones con las siguientes variables comunes:

- **Code:** Código del campo de batalla. Consiste en un par de letras pertenecientes al nombre del campo de batalla junto con un número que es el identificador de la batalla. Diversos personajes con el mismo Code han participado en la misma batalla como aliados o enemigos.
- **Faction:** Facción a la que pertenece el personaje. Es una variable categórica con 2 posibles valores: **Alliance** o **Horde**.
- **Class:** Clase del personaje. Es una variable categórica con 12 posibles valores: **Hunter**, **Death Knight**, **Paladin**, **Rogue**, **Druid**, **Shaman**, **Priest**, **Demon Hunter**, **Warlock**, **Warrior**, **Monk** o **Mage**.
- **KB:** Número de enemigos que ha rematado el personaje. Es una variable numérica.
- **D:** Número de veces que el personaje ha muerto. Es una variable numérica.
- **HK:** Número de asistencias del personaje, es decir, cantidad de veces que ha participado en un asesinato, ya sea realizándole algún tipo de daño o sanando al aliado que lo ha rematado. Es una variable numérica.
- **DD:** Daño realizado por el personaje. Es una variable numérica.
- **HD:** Sanación realizada por el personaje. Es una variable numérica.
- **Honor:** Cantidad de puntos que ha ganado el personaje durante la batalla. Es una variable numérica.
- **Win:** Variable que toma valor 1 si el personaje ha ganado la batalla.
- **Lose:** Variable que toma valor 1 si el personaje ha perdido la batalla.
- **Rol:** Rol que desempeña el personaje en el campo de batalla. Es una variable categórica con 2 posibles valores: Ataque (**dps**) o apoyo (**heal**).
- **BE:** Variable binaria que toma valor 1 si el personaje ha luchado en el campo de batalla durante algún evento en el que se obtienen más puntos de honor.

Una de las variables que se quiere predecir es **Win**, la cual se transforma cambiando los valores NA por 0 y convirtiéndola en una categoría. Además existe multicolinealidad perfecta con la variable **Lose**, por lo que esta última se omite.

La otra variable de interés que se quiere predecir es **KB**, la cantidad de enemigos asesinados por el personaje.

De la variable **Code** únicamente nos interesa el campo de batalla en el que ha participado el personaje, por lo que se transforma en una nueva variable categórica llamada **Battleground** sin el identificador de la batalla. Esta variable tiene 10 posibles valores: Cuenca de Arathi (AB), Batalla por Guilneas (BG), Cañón del Céfiro (DG), Ojo de la Tormenta (ES), Playa de los Ancestros (SA), Minas Lonjaplatas (SM), Costa Hirviente (SS), Templo de Kotmogu (TK), Cumbres Gemelas (TP) o Garganta Grito de Guerra (WG).

La variable **Honor** es calculada al finalizar la partida en función de las demás variables basadas en el rendimiento del personaje durante la partida. El interés de este trabajo está en predecir las variables **Win** y **KB** una vez acabada la partida, pero antes de conocer los puntos obtenidos, por lo tanto, no se usa esta variable para hacer la predicción. La variable **BE** tampoco es usada en la predicción.

También se omiten las demás variables específicas de los campos de batalla, puesto que son de poco interés.

```
data <- tibble()
# Se leen uno a uno los ficheros y se guardan en un data frame
for(file in list.files("data")){
  read_csv(paste("data/", file, sep = "")) %>%
```

```

dplyr::select(Code, Faction, Class, KB, D, HK, DD, HD, Win, Rol) %>%
  rbind(data) -> data
}

# Se arreglan las variables
data <- data %>%
  replace_na(list(Win = 0)) %>%
  mutate(Battleground = as.factor(substr(Code, 1, 2)),
         Faction = as.factor(Faction),
         Class = as.factor(Class),
         Win = as.factor(Win),
         Rol = as.factor(Rol)) %>%
  dplyr::select(-Code)

# Resumen de las variables
kable(summary(data[,1:5]), format = "pipe") %>%
  kable_styling()

```

Faction	Class	KB	D	HK
Alliance:7275	Shaman :1582	Min. : 0.000	Min. : 0.000	Min. : 0.00
Horde :7375	Druid :1482	1st Qu.: 0.000	1st Qu.: 1.000	1st Qu.:13.00
	Warrior:1477	Median : 2.000	Median : 3.000	Median :24.00
	Hunter :1325	Mean : 3.023	Mean : 3.024	Mean :25.83
	Paladin:1318	3rd Qu.: 5.000	3rd Qu.: 5.000	3rd Qu.:37.00
	Mage :1295	Max. :33.000	Max. :13.000	Max. :95.00
	(Other):6171			

```

kable(summary(data[,6:10]), format = "pipe") %>%
  kable_styling()

```

DD	HD	Win	Rol	Battleground
Min. : 0	Min. : 0	0:7226	dps :11793	WG :3028
1st Qu.: 15656	1st Qu.: 6062	1:7424	heal: 2857	BG :2948
Median : 34532	Median : 12318			TK :2566
Mean : 41311	Mean : 26353			SM :2540
3rd Qu.: 59125	3rd Qu.: 26805			ES : 916
Max. :246000	Max. :325000			TP : 889
				(Other):1763

Con un vistazo rápido al resumen de las variables se puede ver como los valores tienen buena pinta. No quedan valores NA y no hay valores fuera del rango esperado, por lo que ya se pueden empezar a explorar estos datos en mayor profundidad.

## Análisis Exploratorio

En este apartado se obtendrá una visión más detallada de los datos, sobretodo de las variables de interés Win y KB.

En la tabla anterior se puede ver como las variables **Faction** y **Win** se distribuyen uniformemente. Esto tiene

sentido debido a que los equipos son de aproximadamente el mismo tamaño, así que en cada batalla la mitad de los personajes serán de una facción que ganará y la otra mitad serán de la otra facción que perderá.

```
# Se usa la función kable del paquete knitr para crear tablas en un buen formato y
#personalizables
kable(round(prop.table(table(data$Rol)), 3),
      col.names = c("Rol", "Frecuencia relativa"),
      align = c("l","c"),
      format = "pipe") %>%
kable_styling()
```

Rol	Frecuencia relativa
dps	0.805
heal	0.195

En cuanto a la variable Rol, un 80% de los personajes son de ataque mientras que el 20% restante son de apoyo. Esta distribución está creada por el juego, los equipos se crean siguiendo esta proporción.

```
# Se usa la función fct_inorder del paquete forcats para ordenar los niveles de los
#factores en función de la cantidad de observaciones
# Se usa la función ggarrange del paquete ggpubr para poder imprimir diferentes gráficos
#dentro de un mismo plot
plotClass <- ggplot(data) +
  geom_hline(yintercept = length(which(data$Class == "Shaman")),
             color = "darkgray",
             linetype = 2) +
  geom_bar(aes(x = fct_infreq(Class), fill = Class),
           color = "black",
           show.legend = FALSE) +
  theme(axis.text.x = element_text(angle = 15)) +
  ggtitle("Distribución de las clases de los personajes") +
  scale_fill_manual(values = c("#C41E3A", "#A330C9", "#FF7COA", "#AAD372",
                             "#3FC7EB", "#00FF98", "#F48CBA", "#FFFFFF",
                             "#FFF468", "#0070DD", "#8788EE", "#C69B6D")) +
  geom_hline(yintercept = 0, color = "black") +
  xlab("Clase") +
  ylab("Frecuencia absoluta")

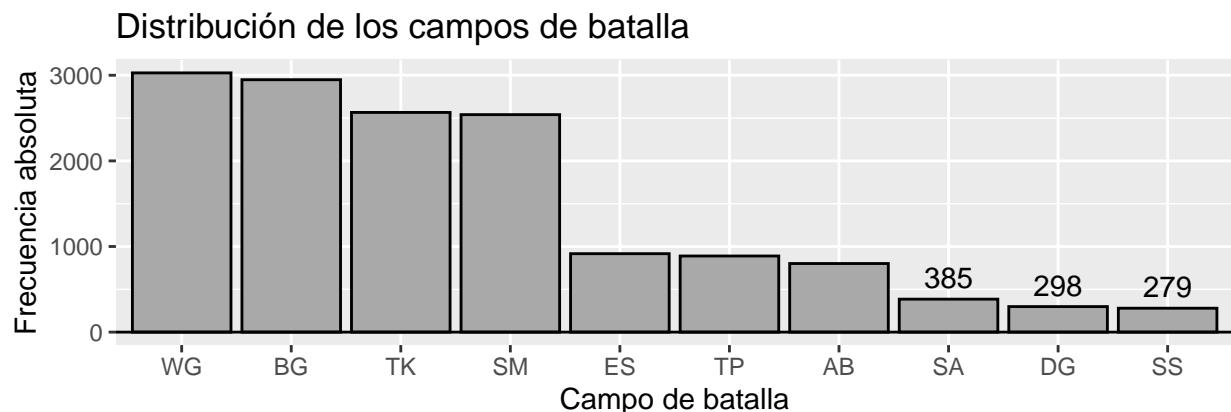
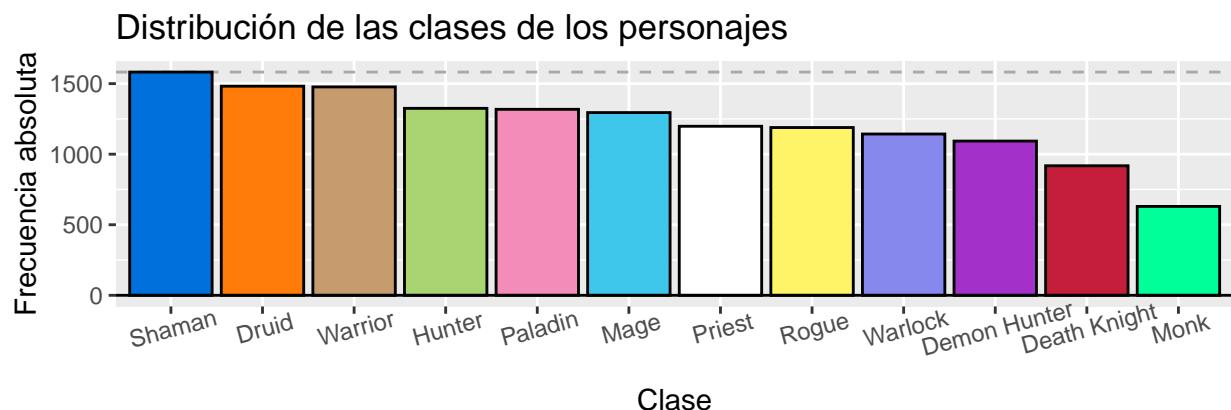
plotBG <- ggplot(data) +
  geom_bar(aes(x = fct_infreq(Battleground), fill = "darkgray", color = "black") +
  ggtitle("Distribución de los campos de batalla") +
  geom_hline(yintercept = 0, color = "black") +
  xlab("Campo de batalla") +
  ylab("Frecuencia absoluta") +
  geom_text(aes(label = length(which(data$Battleground == "SA"))),
            x = "SA",
            y = length(which(data$Battleground == "SA")) + 250,
            check_overlap = TRUE) +
  geom_text(aes(label = length(which(data$Battleground == "DG"))),
            x = "DG",
            y = length(which(data$Battleground == "DG")) + 250,
            check_overlap = TRUE) +
```

```

geom_text(aes(label = length(which(data$Battleground == "SS"))),
          x = "SS",
          y = length(which(data$Battleground == "SS")) + 250,
          check_overlap = TRUE)

ggarrange(plotClass, plotBG, nrow = 2)

```



En el primer gráfico se puede observar la distribución de la variable `Class`. Las tres clases menos frecuentes, `Demon Hunter`, `Death Knight` y `Monk`, se han añadido a lo largo de las expansiones, mientras que las demás siempre han estado en el juego. Esta puede ser la causa de su baja frecuencia. Por otro lado, las diferencias entre las demás clases pueden provenir simplemente de la mayor popularidad de algunas clases o también de un posible desbalanceamiento de estas. Que las clases estén desbalanceadas quiere decir que algunas pueden tener mayor rendimiento que otras en los campos de batalla, por lo tanto algunas clases tendrán mejores incentivos para participar. A continuación se va a comprobar si las clases están balanceadas comparando sus ratios de victorias.

```

# Se usan las funciones kable_styling y add_header_above del paquete kableExtra para
# añadir un encabezado y centrar la tabla
kable(round(prop.table(table(data$Class, data$Win), margin = 1), 3),
      align = c("c","c")) %>%
  kable_styling() %>%
  add_header_above(c(" " = 1, "Win" = 2))

```

	Win	
	0	1
Death Knight	0.473	0.527
Demon Hunter	0.466	0.534
Druid	0.520	0.480
Hunter	0.562	0.438
Mage	0.520	0.480
Monk	0.498	0.502
Paladin	0.487	0.513
Priest	0.497	0.503
Rogue	0.461	0.539
Shaman	0.435	0.565
Warlock	0.495	0.505
Warrior	0.502	0.498

La hipótesis nula es que las clases no están desbalanceadas, por lo que la proporción de victorias de cada una de ellas debería ser del 50 %. Sea  $p_i$  la proporción de victorias de la clase  $i$ .

$$H_0 : p_i = 0.5$$

$$H_1 : p_i \neq 0.5$$

Se calcula el estadístico de contraste  $T$  para cada clase  $i$ .

$$T_i = \frac{\hat{p}_i - p_i}{\sqrt{\frac{p_i \cdot (1-p_i)}{n_i}}} \sim N(0, 1)$$

donde  $\hat{p}_i$  es la proporción observada y  $n_i$  es el número de observaciones de la clase  $i$ .

```
pHat <- prop.table(table(data$Class, data$Win), margin = 1)[,2]
# Se calculan los estadísticos T
Ti <- (pHat - 0.5) / sqrt((0.5 * (1 - 0.5)) / as.vector(table(data$Class)))
# Se calculan los p-valores
pValue <- round(ifelse(pnorm(Ti) > 0.5, 1 - pnorm(Ti), pnorm(Ti)), 4)
kable(tibble(Clase = levels(data$Class),
            Proporción = round(pHat, 3),
            n = as.vector(table(data$Class)),
            `T` = round(Ti, 3),
            `p-valor` = pValue),
      align = c("l", "c", "c", "c", "c"),
      format = "pipe") %>%
kable_styling()
```

Clase	Proporción	n	T	p-valor
Death Knight	0.527	918	1.650	0.0494
Demon Hunter	0.534	1093	2.269	0.0116
Druid	0.480	1482	-1.507	0.0660
Hunter	0.438	1325	-4.533	0.0000
Mage	0.480	1295	-1.473	0.0704
Monk	0.502	630	0.080	0.4682
Paladin	0.513	1318	0.937	0.1745

Clase	Proporción	n	T	p-valor
Priest	0.503	1198	0.231	0.4086
Rogue	0.539	1189	2.697	0.0035
Shaman	0.565	1582	5.179	0.0000
Warlock	0.505	1143	0.325	0.3725
Warrior	0.498	1477	-0.130	0.4482

Se elige un nivel de significación de  $\alpha = 0.05$ , pero el contraste es bilateral, por lo que el p-valor que rechaza la hipótesis nula debe ser inferior a 0.025. Entonces podemos decir con una seguridad del 95 % que las clases Demon Hunter, Hunter, Rogue y Shaman no ganan la mitad de los campos de batalla. Concretamente, la clase Hunter pierde más del 50 % de las batallas y las clases Demon Hunter, Rogue y Shaman ganan más del 50 %. Este resultado puede significar que estas clases están desbalanceadas, aunque otra hipótesis podría ser que los jugadores que usan estas clases son mejores/peores jugadores que la media.

Por otro lado, en el gráfico con la distribución de la variable Battleground se ve claramente que las frecuencias están desequilibradas. Los jugadores pueden elegir el campo de batalla en el que quieren jugar, por lo que el jugador que recogió los datos seguramente tenía preferencia por cuatro campos de batalla en particular. En el campo de batalla de Costa Hirviente (SS) tan solo hay datos de 279 personajes, lo que supone poco menos de un 2 % de las observaciones totales. Supongo que este desequilibrio no tendrá efectos negativos en las predicciones.

```
# Se usa la función annotate_figure del paquete ggpublisher para añadir un título común a la
#figura creada con ggarrange
plotHK_D <- ggplot(data) +
  geom_jitter(aes(x = HK, y = D, color = Win), alpha = 0.4) +
  scale_color_manual(values = c("#9d9d9d", "#00ccff")) +
  theme(panel.background = element_rect(color = "black", fill = "white"),
        panel.grid = element_blank()) +
  xlab("Asistencias") +
  ylab("Muertes")

plotX <- ggplot(data) +
  geom_density(aes(x = HK, fill = Win), alpha = 0.6) +
  scale_fill_manual(values = c("#9d9d9d", "#00ccff")) +
  theme_void()

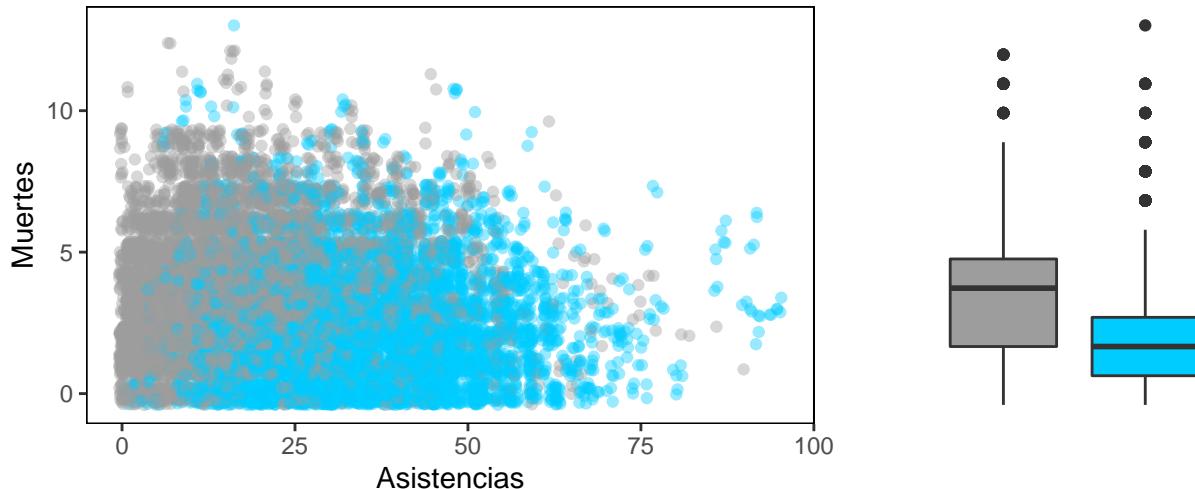
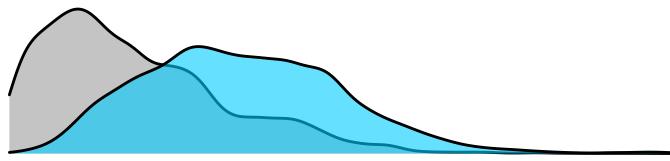
plotY <- ggplot(data) +
  geom_boxplot(aes(x = Win, y = D, fill = Win)) +
  scale_fill_manual(values = c("#9d9d9d", "#00ccff")) +
  theme_void()

fig <- ggarrange(plotX, NULL, plotHK_D, plotY,
                 ncol = 2, nrow = 2, align = "hv",
                 widths = c(2, 1), heights = c(1, 2),
                 common.legend = TRUE)

annotate_figure(fig,
               top = text_grob("Distribución de ganadores en función de muertes y asistencias"))
```

### Distribución de ganadores en función de muertes y asistencias

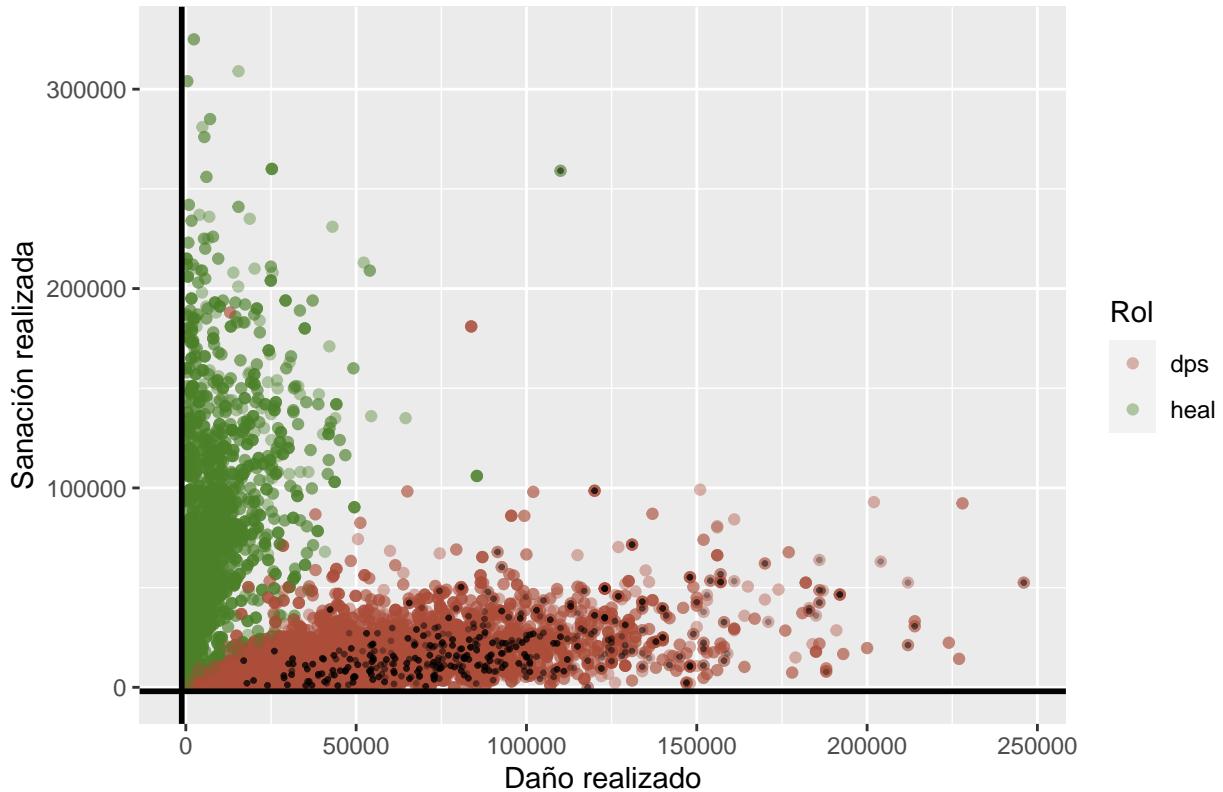
Win  0  1



Como era de esperar, los personajes con mayor número de asistencias y menos muertes propias ganan más campos de batalla, aunque ambos grupos están bastante solapados.

```
# Se busca el 5% de personajes con mayor número de asesinatos
data5KB <- data %>%
  filter(KB >= quantile(data$KB, 0.95))
ggplot() +
  geom_point(data, mapping = aes(x = DD, y = HD, color = Rol), alpha = 0.4) +
  scale_color_manual(values = c("#AB4D39", "#4B8028")) +
  xlab("Daño realizado") +
  ylab("Sanación realizada") +
  ggtitle("Distribución del Rol por daño y sanación realizada") +
  geom_point(data5KB, mapping = aes(x = DD, y = HD), size = 0.5, alpha = 0.3) +
  geom_hline(yintercept = -2000, color = "black", size = 1) +
  geom_vline(xintercept = -1200, color = "black", size = 1)
```

## Distribución del Rol por daño y sanación realizada



Los roles de **dps** y **heal** están perfectamente separados por las dimensiones de daño realizado y sanación realizada. Aunque ambos roles puedan atacar y curar, la gran masa de **dps** se centra en un alto daño realizado y baja sanación realizada, mientras que los **heal** todo lo contrario.

En el mismo gráfico también se pueden ver los personajes que más veces han asesinado enemigos durante la batalla. Concretamente, los puntos negros son el 5 % de los personajes con mayor número de asesinatos KB, esto es, los que han realizado 9 asesinatos o más. Se puede ver como todos son **dps** pero no necesariamente han realizado una cantidad de daño muy elevado, ya que la variable KB contabiliza un asesinato si el personaje ha sido quien lo ha rematado.

Puede que haya clases que tengan mayor facilidad de dar el último golpe al enemigo. Para ver esto, se imprime la distribución de los asesinatos en función de las clases. Se ha visto que los personajes **heal** no tienen valores altos en KB, por lo que se dejan fuera del gráfico para no tirar a la baja la distribución de los **dps**.

```
filter(data, Rol == "dps") %>%
  ggplot() +
  geom_boxplot(aes(x = Class, y = KB, fill = Class),
               show.legend = FALSE,
               outlier.color = NA) +
  theme(axis.text.x = element_text(angle = 15)) +
  ylim(0,9) +
  scale_fill_manual(values = c("#C41E3A", "#A330C9", "#FF7COA", "#AAD372",
                             "#3FC7EB", "#00FF98", "#F48CBA", "#FFFFFF",
                             "#FFF468", "#0070DD", "#8788EE", "#C69B6D")) +
  ggtitle("Distribución de asesinatos de dps por clase",
          subtitle = "Truncada en el cuantil 0.95") +
```

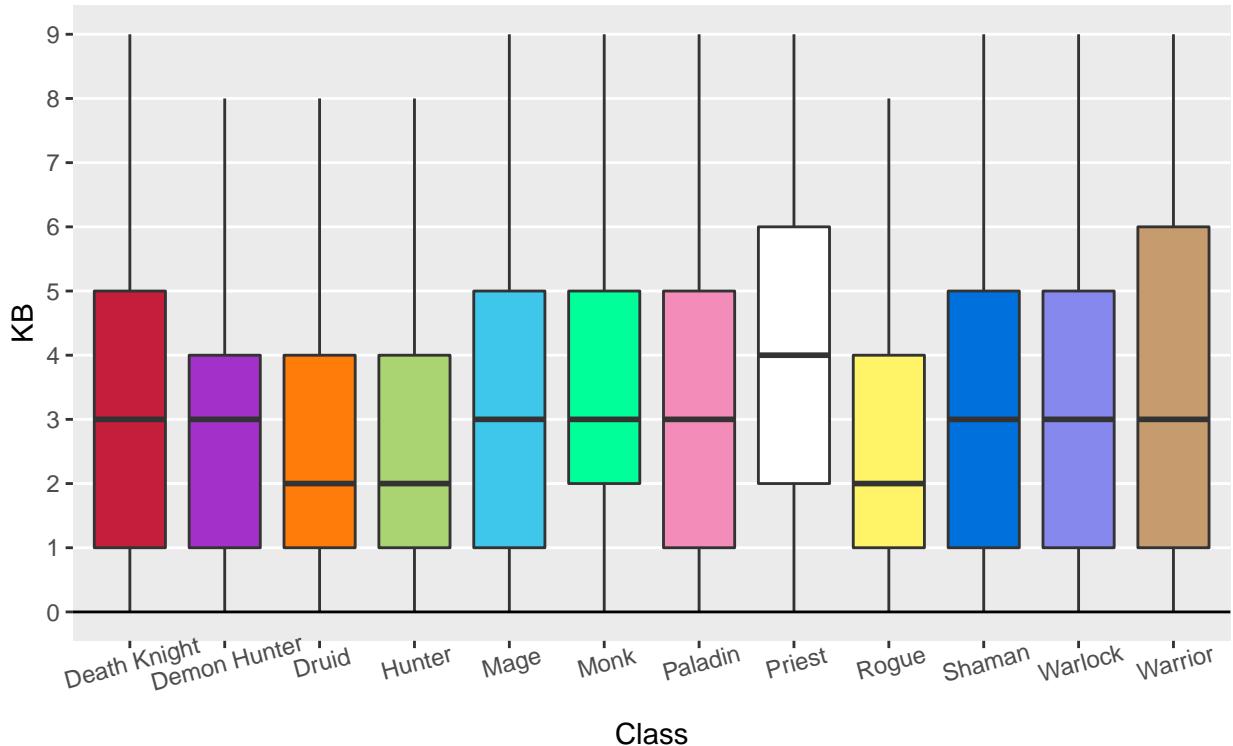
```

scale_y_continuous(limits = c(0,9), breaks = 0:9) +
theme(panel.grid.minor = element_blank(), panel.grid.major.x = element_blank()) +
geom_hline(yintercept = 0, color = "black")

```

## Distribución de asesinatos de dps por clase

Truncada en el cuantil 0.95



Se puede observar que las diferencias entre clases no son muy elevadas, pero si que hay clases como **Priest** o **Warrior** que se llevan más asesinatos que otras clases como **Druid**, **Hunter** o **Rogue**.

Una vez conocemos el comportamiento de los datos, se pueden aplicar algoritmos complejos para verlos desde otras perspectivas y hacer predicciones.

## Aprendizaje estadístico

En este apartado se reducirá la dimensión de los datos aplicando la técnica de Análisis de Componentes Principales, se buscarán grupos en estas nuevas dimensiones con el uso de algoritmos de Clustering, y se ajustarán modelos para predecir las variables **Win** y **KB**.

## Análisis de Componentes Principales

Las variables reducibles son **KB**, **D**, **HK**, **DD** y **HD**, ya que son las únicas cuantitativas, aunque la variable **KB** no se reducirá ya que posteriormente se quiere encontrar un modelo para predecirla. Si no se usa en el Análisis de Componentes Principales, se podrán usar los componentes principales para predecirla. Por lo tanto, los componentes principales representarán una parte del rendimiento de los personajes durante la batalla.

Se realiza la reducción de dimensiones con las variables estandarizadas, ya que las escalas son muy dispares.

```

acpFit <- prcomp(data[,4:7], center = TRUE, scale = TRUE)
acpTable <- round(summary(acpFit)$importance, 4)
rownames(acpTable) <- c("Desviación Típica",
                        "Proporción de Varianza",
                        "Proporción Acumulada")
kable_styling(kable(acpTable, align = c("c", "c", "c", "c"), format = "pipe"))

```

	PC1	PC2	PC3	PC4
Desviación Típica	1.1659	1.0945	0.9979	0.6686
Proporción de Varianza	0.3398	0.2995	0.2490	0.1118
Proporción Acumulada	0.3398	0.6393	0.8882	1.0000

Según la tabla de importancia de los componentes principales, los tres primeros recogen la mayoría de la varianza de estas variables. Contienen un 88.8% de la información. Según el criterio del valor medio se podrían usar solo los dos primeros, pero entre ellos tan solo recogen el 63.9% de la información.

```

kable(round(acpFit$rotation[,1:3], 3), align = c("c", "c", "c"), format = "pipe") %>%
  kable_styling()

```

	PC1	PC2	PC3
D	-0.037	-0.489	0.824
HK	0.696	0.383	0.049
DD	0.715	-0.338	0.043
HD	-0.061	0.707	0.563

Interpretación de los tres primeros Componentes Principales:

- PC1: Este índice recoge mucha información de las variables HK y DD, las cuales están relacionadas con el ataque. A mayor número de asistencias y daño realizado, mayor es este índice.
- PC2: Este componente se centra positivamente en la variable HD, y en las demás en menor medida. Está relacionado con el apoyo, puesto que puntúa mucho la sanación realizada, pero también las asistencias, el poco daño realizado y las pocas muertes.
- PC3: El tercer componente principal da mucha importancia a la variable D, y en menor medida a HD. Es un indicador que tiene un alto valor si el personaje muere mucho y realiza sanación.

Para facilitar la visualización, únicamente se imprimen las observaciones sobre los dos primeros índices.

```

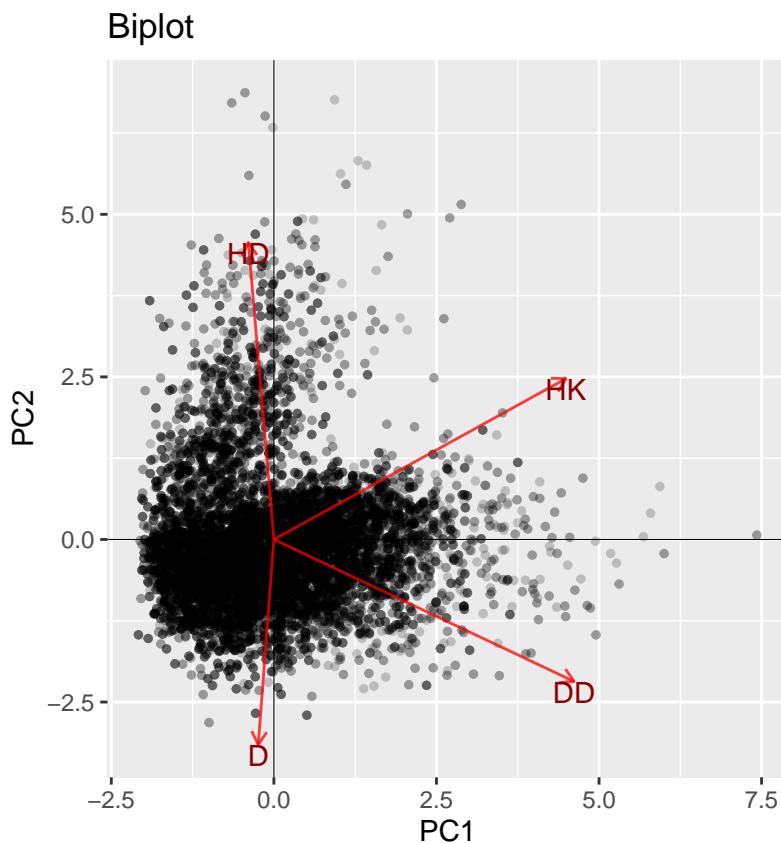
# Función para crear un biplot
acp_biplot <- function(acp) {
  data <- data.frame(obsnames = 1:nrow(acp$x), acp$x)
  plot <- ggplot(data) +
    geom_point(aes(x = data[, "PC1"], y = data[, "PC2"]),
               alpha = 0.2,
               size = 1,
               color = "black")
  plot <- plot + geom_hline(yintercept = 0, aes(0), size = 0.2) +
    geom_vline(aes(0), size = 0.2, color = "black", xintercept = 0)
  datapc <- data.frame(varnames = rownames(acp$rotation), acp$rotation)
}

```

```

mult <- min((max(data[, "PC2"]) - min(data[, "PC2"])) /
             (max(datapc[, "PC2"]) - min(datapc[, "PC2"]))), 
            (max(data[, "PC1"]) - min(data[, "PC1"])) /
             (max(datapc[, "PC1"]) - min(datapc[, "PC1"])))
datapc <- transform(datapc,
                      v1 = 0.7 * mult * (get("PC1")),
                      v2 = 0.7 * mult * (get("PC2")))
plot <- plot +
  geom_segment(data = datapc,
               aes(x = 0, y = 0, xend = v1, yend = v2),
               arrow = arrow(length = unit(0.2, "cm")),
               alpha = 0.75, color = "red")
plot <- plot + coord_equal() +
  geom_text(data = datapc,
            aes(x = v1, y = v2, label = varnames),
            size = 4,
            vjust = 1,
            color = "darkred")
plot + ggtitle("Biplot") + xlab("PC1") + ylab("PC2")
}
acp_biplot(acpFit)

```



```

# Se insertan los CP en el data frame
data <- cbind(data, acpFit$x[,1:3])

```

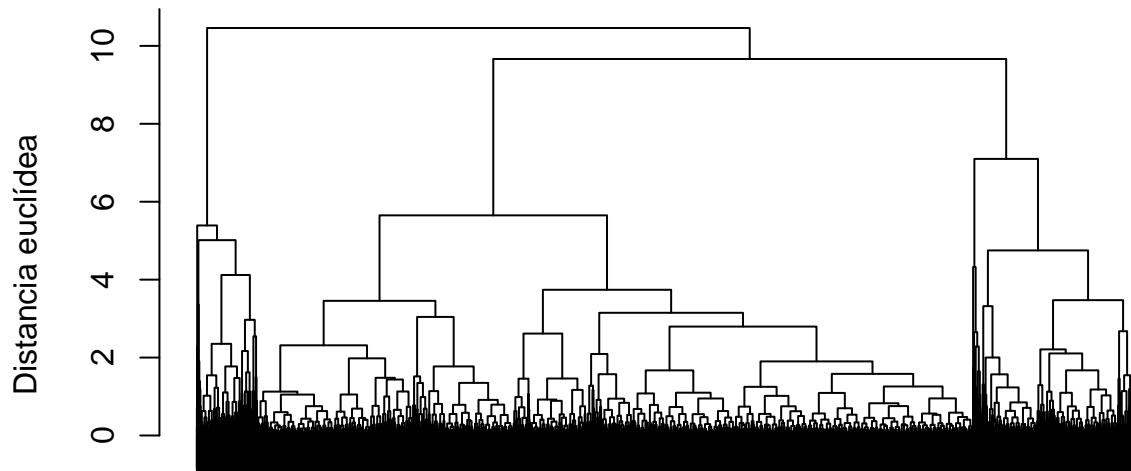
En el biplot se pueden intuir tres regiones con muchas observaciones. En la primera, el valor de PC1 es positivo, es decir, aquí están los personajes que más han atacado y más han participado en asesinatos. En una segunda zona con PC1 negativo y PC2 positivo, se encuentran los personajes que más han sanado y menos han muerto. Y en la tercera región donde los dos componentes principales son negativos y convergen las otras dos zonas, hay personajes que han muerto mucho, han ayudado poco y han hecho poco daño y sanación. Por lo tanto, parece que en la primera zona hay personajes que han realizado una buena batalla como `dps`, en la segunda lo mismo pero como `heal`, y en la tercera han realizado una mala batalla en general.

## Clustering

A continuación se busca separar las observaciones entre diferentes grupos en las dimensiones de los dos primeros componentes principales. El cluster jerárquico con conexión completa y distancia euclídea como métrica funciona relativamente bien al separar las observaciones en 3 grupos diferentes.

```
hcComplete <- hclust(dist(data[,11:12]), method = "complete")
plot(hcComplete,
      labels = FALSE,
      ylab = "Distancia euclídea",
      xlab = NA,
      main = "Dendograma con conexión Completa",
      sub = NA)
```

**Dendograma con conexión Completa**



```
hcCut <- as.factor(cutree(hcComplete, 3))
data <- cbind(data, hcCut)
ggplot(data) +
```

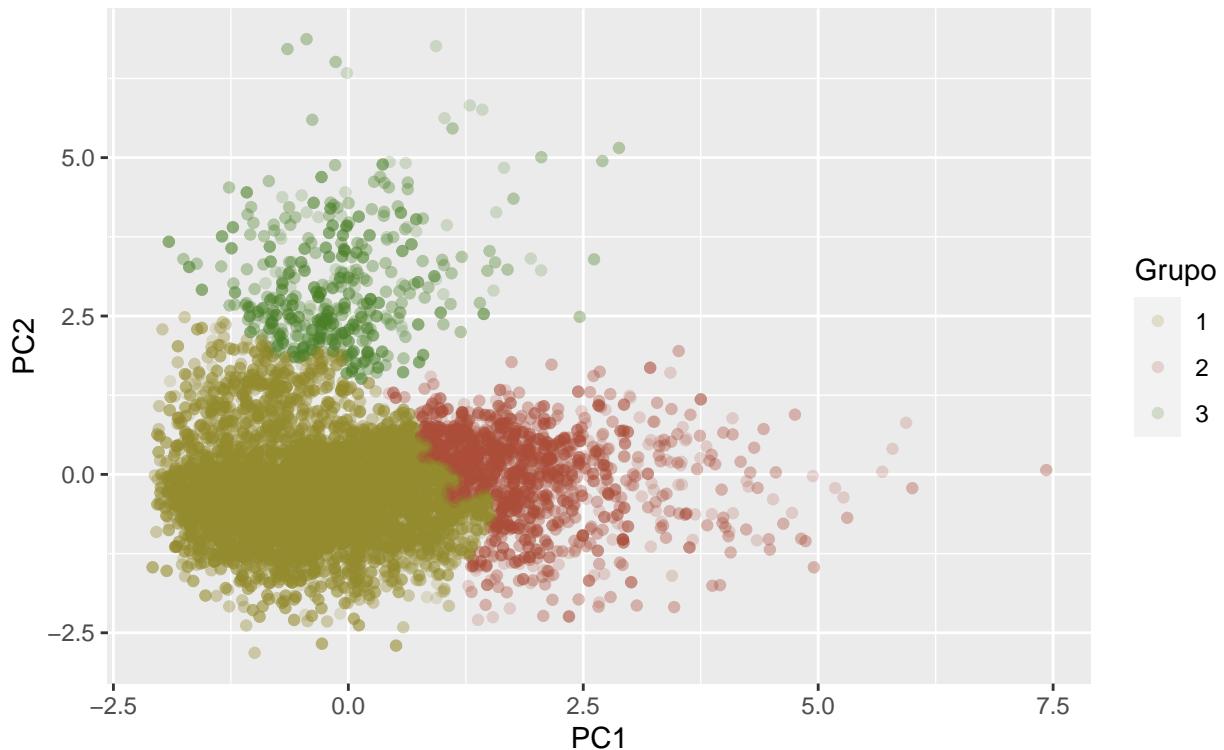
```

geom_point(aes(x = PC1, y = PC2, color = hcCut), alpha = 0.2) +
  labs(color = "Grupo") +
  ggtitle("Separación por clusters",
          subtitle = "Cluster Jerárquico con conexión Completa") +
  scale_color_manual(values = c("#948B2E", "#AB4D39", "#4B8028"))

```

## Separación por clusters

### Cluster Jerárquico con conexión Completa



Este algoritmo separa las observaciones en las tres regiones intuidas en el apartado anterior. Los personajes que han realizado una muy buena partida pertenecen a los grupos 2 o 3, dependiendo de su rol. Si es de apoyo, al grupo 3, mientras que si es de ataque, al grupo 2. Por otro lado, los personajes que pertenecen al grupo 1 no han destacado por su rendimiento durante la partida.

Con cualquier otro tipo de conexión, el clustering jerárquico funciona muy mal debido a que se crea un gran grupo con muchas observaciones, y otros muchos grupos diminutos.

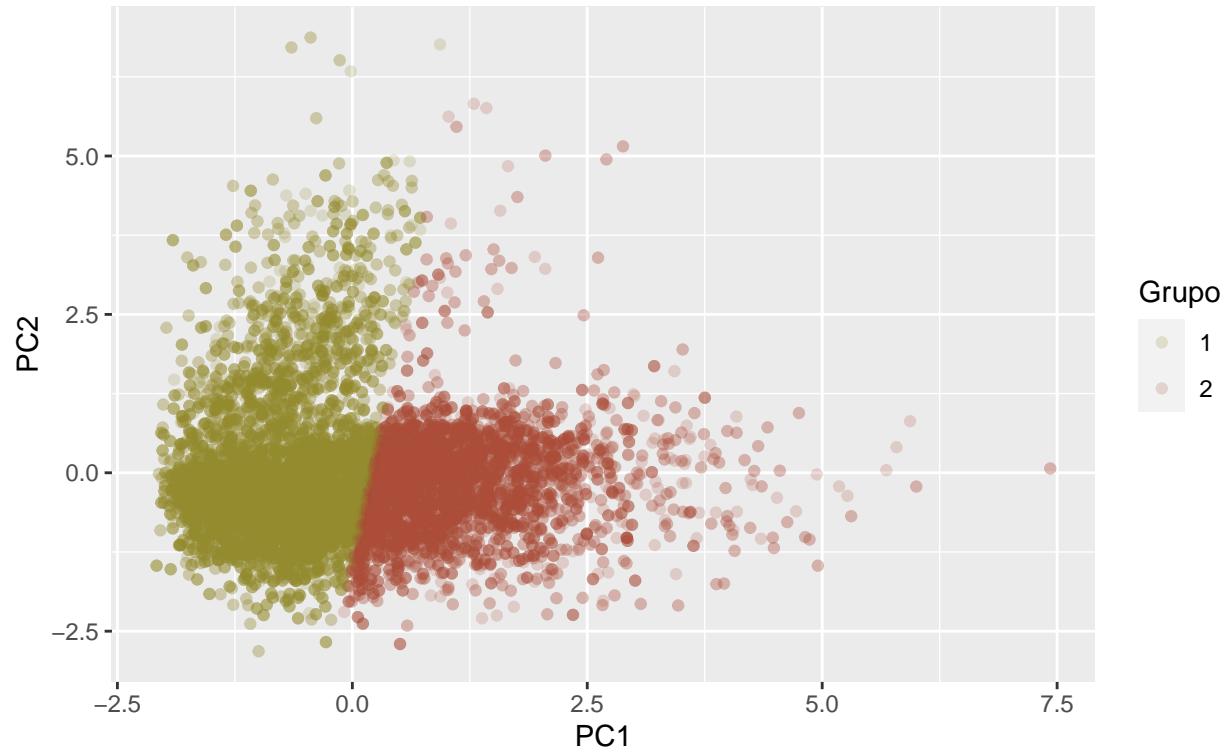
Ahora se prueba el algoritmo de k-means clustering con 2 y 3 grupos.

```

set.seed(1)
kmc2 <- kmeans(data[,11:12], 2)
data <- cbind(data, kmc2 = as.factor(kmc2$cluster))
ggplot(data) +
  geom_point(aes(x = PC1, y = PC2, color = kmc2), alpha = 0.2) +
  labs(color = "Grupo") +
  ggtitle("Separación por clusters", subtitle = "k-means clustering con 2 grupos") +
  scale_color_manual(values = c("#948B2E", "#AB4D39"))

```

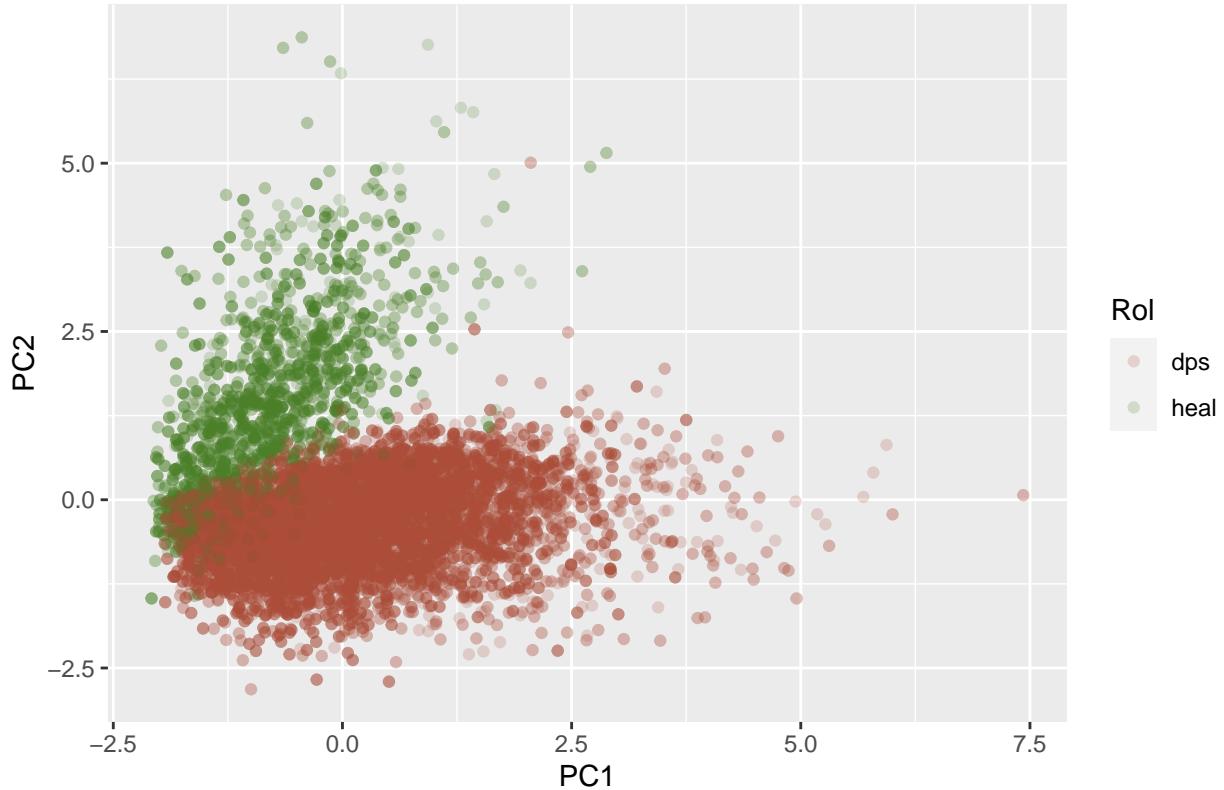
## Separación por clusters k-means clustering con 2 grupos



Con  $K = 2$ , el algoritmo crea dos grupos de tamaño similar, aunque la región que separa ambos grupos parece muy artificial. En la sección anterior se ha comentado que las observaciones de la derecha del gráfico podrían ser personajes `dps`, mientras que los de arriba podrían ser `heal`.

```
ggplot(data) +  
  geom_point(aes(x = PC1, y = PC2, color = Rol), alpha = 0.2) +  
  ggtitle("Separación por Rol") +  
  scale_color_manual(values = c("#AB4D39", "#4B8028"))
```

## Separación por Rol



```
kable(table(data$Rol, data$kmc2)) %>%
  kable_styling() %>%
  add_header_above(c(" " = 1, "Grupo" = 2))
```

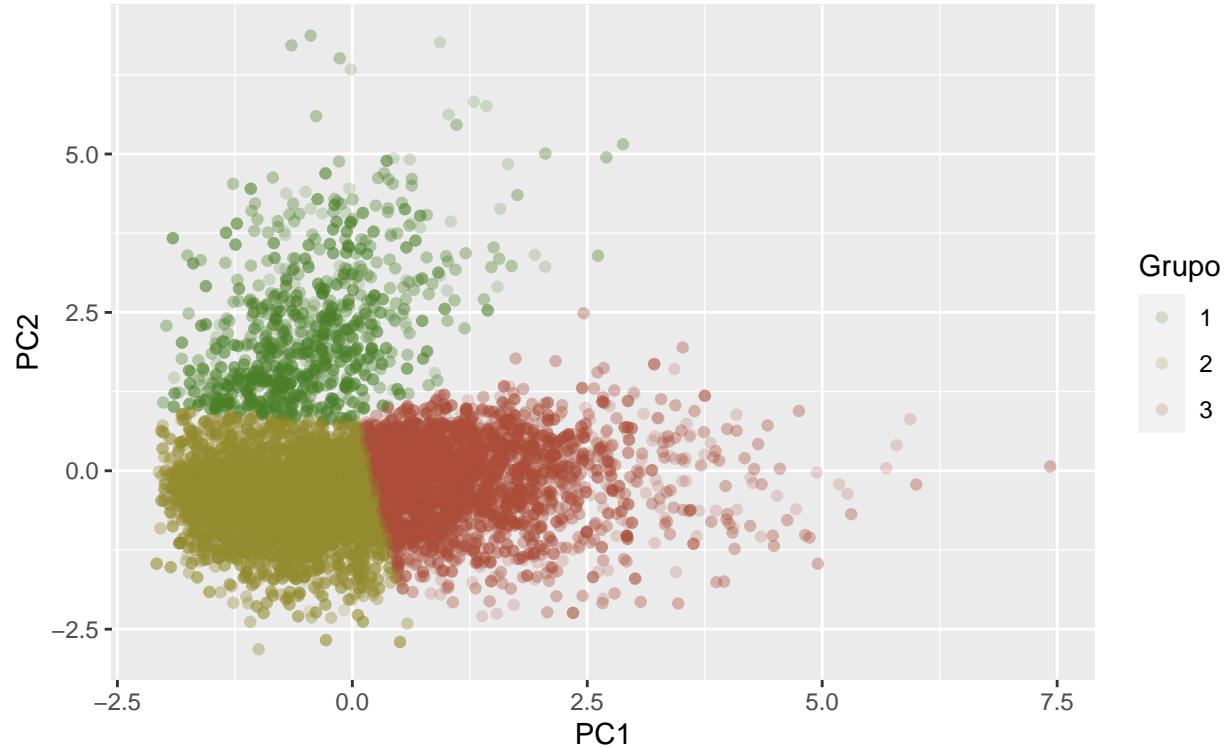
		Grupo	
		1	2
dps	1	6095	5698
	2	2745	112

Comparando los roles con los grupos creados por el algoritmo, son algo parecidos, aunque el grupo 1 creado por el clustering recoge muchas más observaciones de las que debería para separar bien los roles. Por lo tanto, parece que este algoritmo únicamente separa ambos grupos en función del valor de PC1. Los personajes que hayan atacado mucho estarán en el grupo 2, mientras que los demás estarán en el 1.

```
set.seed(1)
kmc3 <- kmeans(data[,11:12], 3)
data <- cbind(data, kmc3 = as.factor(kmc3$cluster))
ggplot(data) +
  geom_point(aes(x = PC1, y = PC2, color = kmc3), alpha = 0.2) +
  labs(color = "Grupo") +
  ggtitle("Separación por clusters", subtitle = "k-means clustering con 3 grupos") +
  scale_color_manual(values = c("#4B8028", "#948B2E", "#AB4D39"))
```

## Separación por clusters

### k-means clustering con 3 grupos



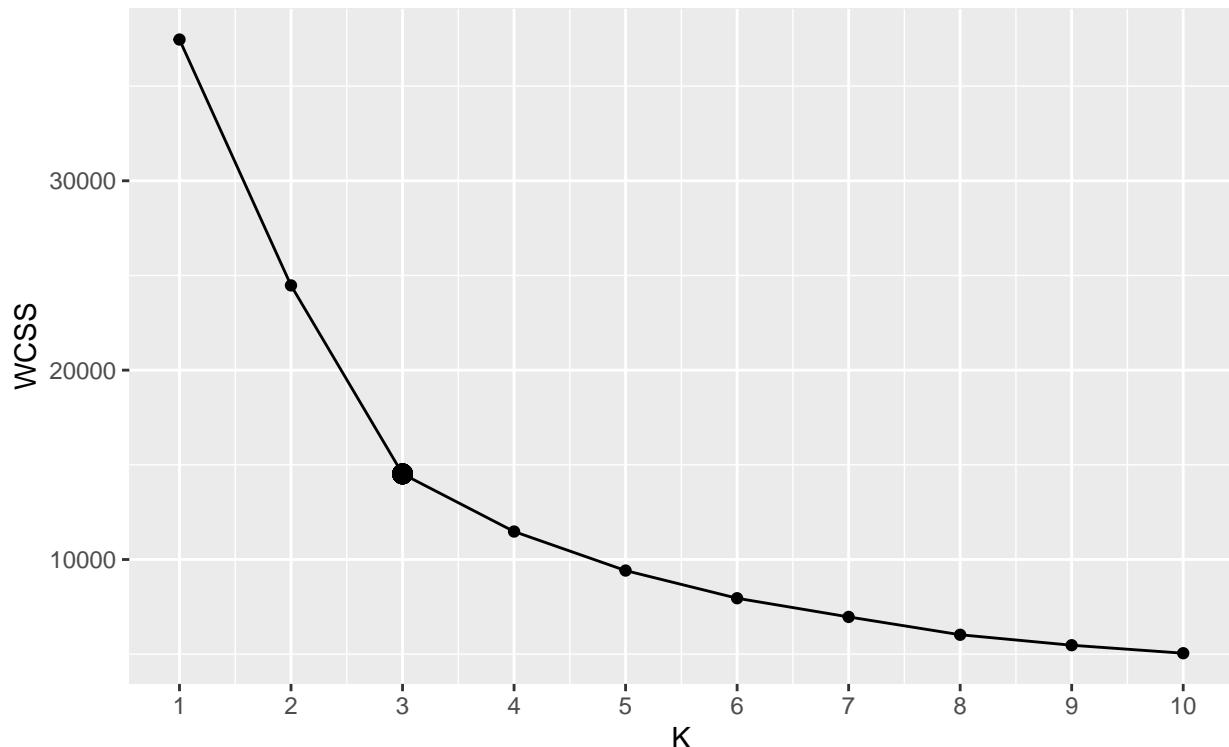
Por otro lado, con  $K = 3$ , el algoritmo separa las observaciones de manera parecida al cluster jerárquico. Cuyos grupos son los personajes `dps` con buen rendimiento, los `heal` con buen rendimiento, y los personajes con mal rendimiento.

Para saber el número óptimo de clusters para el algoritmo k-means, se va a comparar la suma de los cuadrados al centro del cluster (WCSS) para valores  $K = \{1, \dots, 10\}$ .

```
set.seed(1)
WCSS <- vector()
for(i in 1:10) {
  WCSS[i] <- sum(kmeans(data[,11:12], i)$withinss)
}
dataWCSS <- tibble(K = 1:10, WCSS)
ggplot(dataWCSS, mapping = aes(x = K, y = WCSS)) +
  geom_line() +
  geom_point() +
  geom_point(x = 3, y = WCSS[3], size = 3) +
  scale_x_continuous(breaks = 1:10) +
  ggtitle("Método del codo", subtitle = "Elección del K óptimo")
```

## Método del codo

### Elección del K óptimo



Según la técnica del codo, parece ser que  $K = 3$  es un buen número de clusters. Así que tiene sentido separar las observaciones entre personajes con buen `dps`, con buen `heal` y personajes malos.

## Predicción de ganadores

En este apartado se quiere ajustar un modelo que permita predecir si un personaje ha ganado la batalla dadas las características del personaje y su rendimiento durante la batalla. Es decir, se quiere clasificar la variable `Win` en función de `Faction`, `Class`, `Rol`, `Battleground`, `KB`, `PC1`, `PC2` y `PC3`. Se estimarán dos modelos de clasificación y se compararán sus precisiones para obtener el mejor modelo. En primer lugar se ajustará una Regresión Logística y luego un Extreme Gradient Boosting. La Regresión Logística es un algoritmo relativamente simple que permite calcular probabilidades de que la variable dependiente tome un valor concreto usando variables explicativas tanto cuantitativas como cualitativas. Por otro lado, Extreme Gradient Boosting es un algoritmo muy potente basado en árboles de decisión.

## Regresión Logística

Antes de estimar el modelo se separan las observaciones en dos subconjuntos, uno con el 80 % de los datos que se usará para entrenar el modelo, y otro con el 20 % restante para obtener una medida de la precisión del modelo. La precisión se mide prediciendo observaciones no usadas para estimar el modelo, así evitamos un posible problema de sobreajuste. También es necesario escalar la variable `KB`, ya que es la única variable cuantitativa que no está estandarizada.

```
# Se estandariza KB y se dividen las observaciones en conjunto train y test
dataSc <- data %>%
  dplyr::select(Win, 1:2, 9:10, KB, 11:13)
```

```

dataSc[,6] <- scale(dataSc[,6])
set.seed(1)
iTrain <- sample(nrow(dataSc), nrow(dataSc) * 0.8, replace = FALSE)
dataScTrain <- dataSc[iTrain,]
dataScTest <- dataSc[-iTrain,]

```

Una vez se tienen los datos preparados, se ajusta el modelo de regresión logística con los datos de entrenamiento, se predicen los datos test y se crea la matriz de confusión para comparar los valores predichos con los reales.

```

# Estimación del modelo
rlFit <- glm(Win ~ ., data = dataScTrain, family = binomial)
# Obtención de probabilidades
rlProbs <- predict(rlFit, type = "response", newdata = dataScTest[-1])
# Predicciones en función de las probabilidades
rlPred <- ifelse(rlProbs > 0.5, 1, 0)
# Matriz de confusión
kable(table(dataScTest[,1], rlPred)) %>%
  kable_styling() %>%
  add_header_above(c(" " = 1, "Predicción" = 2))

```

		Predicción	
		0	1
0	0	1142	325
	1	306	1157

```

# Cálculo de la precisión
rlAcc <- mean(dataScTest[,1] == rlPred)

```

El modelo de Regresión Logística predice bastante bien, clasifica un 78.46 % de las observaciones correctamente.

```

# Coeficientes del modelo
rlCoef <- round(summary(rlFit)$coefficients, 4)
Sig <- as.factor(ifelse(rlCoef[,4] > 0.05, "No", "Sí"))
kable(cbind(as.tibble(rlCoef), Sig),
      align = c("c", "c", "c", "c", "c"),
      format = "pipe") %>%
  kable_styling()

```

	Estimate	Std. Error	z value	Pr(> z )	Sig
(Intercept)	-0.7786	0.1429	-5.4502	0.0000	Sí
FactionHorde	1.0648	0.0473	22.4968	0.0000	Sí
ClassDemon Hunter	0.3639	0.1251	2.9083	0.0036	Sí
ClassDruid	0.1095	0.1200	0.9132	0.3612	No
ClassHunter	0.0283	0.1196	0.2369	0.8128	No
ClassMage	-0.0700	0.1203	-0.5816	0.5608	No
ClassMonk	-0.1042	0.1511	-0.6896	0.4905	No
ClassPaladin	0.1043	0.1223	0.8522	0.3941	No
ClassPriest	-0.2405	0.1320	-1.8221	0.0684	No

	Estimate	Std. Error	z value	Pr(> z )	Sig
ClassRogue	0.3654	0.1221	2.9935	0.0028	Sí
ClassShaman	0.3487	0.1258	2.7719	0.0056	Sí
ClassWarlock	-0.2473	0.1232	-2.0075	0.0447	Sí
ClassWarrior	0.4364	0.1188	3.6728	0.0002	Sí
Rolheal	-0.2736	0.1148	-2.3838	0.0171	Sí
BattlegroundBG	0.6252	0.1203	5.1989	0.0000	Sí
BattlegroundDG	-0.4589	0.1871	-2.4528	0.0142	Sí
BattlegroundES	-0.3671	0.1462	-2.5115	0.0120	Sí
BattlegroundSA	-1.1430	0.1816	-6.2942	0.0000	Sí
BattlegroundSM	0.2197	0.1207	1.8195	0.0688	No
BattlegroundSS	0.4186	0.1917	2.1833	0.0290	Sí
BattlegroundTK	0.1733	0.1202	1.4420	0.1493	No
BattlegroundTP	0.5225	0.1398	3.7377	0.0002	Sí
BattlegroundWG	0.2461	0.1178	2.0898	0.0366	Sí
KB	0.0010	0.0378	0.0259	0.9793	No
PC1	0.9088	0.0355	25.5978	0.0000	Sí
PC2	0.9775	0.0362	27.0228	0.0000	Sí
PC3	-0.5870	0.0292	-20.0789	0.0000	Sí

Según este modelo, que un personaje pertenezca a la facción Horde aumenta la probabilidad de ganar el campo de batalla. La clase también puede tener efecto, aunque no todas influyen de la misma manera. Llama la atención el hecho de que ser `heal` reduzca la probabilidad de ganar, lo que pelear con este rol también depende de la clase, así que el efecto puede estar compensado por otra vía. También sorprende que la variable KB no sea estadísticamente significativa al 5 %, aunque seguramente sea porque los componentes principales ya recogen la información necesaria del rendimiento del personaje durante la batalla.

## Extreme Gradient Boosting

Para estimar el modelo de Extreme Gradient Boosting también es necesario separar las observaciones en conjuntos de entrenamiento y de test, pero antes de estimar el modelo final es necesario obtener los hiperparámetros óptimos del modelo. Para ello se divide el conjunto de entrenamiento en 5 cajas y se buscan los mejores hiperparámetros para cada caja. Debido a la gran cantidad de combinaciones que existen para distintos valores de los hiperparámetros, el algoritmo tarda bastante.

Los posibles valores de los hiperparámetros que se prueban son:

Hiperparámetro	Valores
<code>nrounds</code>	{200}
<code>max_depth</code>	{3, 6, 10}
<code>eta</code>	{0.01, 0.1, 0.3}
<code>gamma</code>	{0.01}
<code>colsample_bytree</code>	{0.5, 1}
<code>min_child_weight</code>	{0, 1}
<code>subsample</code>	{0.5, 1}

```
# División entre subconjuntos train y test
dataAux <- dplyr::select(data, Win, 1:2, 9:10, KB, 11:13)
dataTrain <- dataAux[iTrain,]
dataTest <- dataAux[-iTrain,]
```

```

# División en 5 cajas de los datos train
set.seed(1)
folds <- createFolds(dataTrain$Win, k = 5)

# Se prepara la parrilla con los hiperparámetros a probar
xgbGrid <- expand.grid(nrounds=c(200),
  max_depth = c(3, 6, 10),
  eta = c(0.01, 0.1, 0.3),
  gamma = c(0.01),
  colsample_bytree = c(0.5, 1),
  min_child_weight = c(0, 1),
  subsample = c(0.5, 1))
xgbTune <- matrix(nrow = 5, ncol = 7)
dimnames(xgbTune) <- list(c("k1", "k2", "k3", "k4", "k5"),
  c("nrounds", "max_depth", "eta", "gamma",
  "colsample_bytree", "min_child_weight", "subsample"))
# Se especifican los parámetros que se usarán para estimar el modelo, específicamente
# 5-fold CV
fitControl <- trainControl(method = "cv", number = 5)

set.seed(1)
for (i in 1:5) {
  # Se elige la caja
  foldTrain <- dataTrain[folds[[i]],]
  # Se estima el modelo de cada caja
  xgbFit <- train(Win ~ .,
    data = foldTrain,
    method = "xgbTree",
    trControl = fitControl,
    tuneGrid = xgbGrid,
    metric = "Accuracy")
  # Se actualiza la variable que guarda los hiperparámetros óptimos
  xgbTune[i,] <- as.matrix(xgbFit$bestTune)
}

```

Los mejores modelos de cada caja  $k$  tienen los siguientes hiperparámetros:

```

kable(xgbTune, align = c("c","c","c","c","c","c","c"), format = "pipe") %>%
  kable_styling()

```

	nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
k1	200	10	0.1	0.01	0.5	0	1.0
k2	200	10	0.1	0.01	1.0	0	1.0
k3	200	6	0.3	0.01	1.0	0	1.0
k4	200	6	0.1	0.01	1.0	0	1.0
k5	200	10	0.1	0.01	1.0	1	0.5

Por lo tanto, parece que `max_depth = 10`, `eta = 0.1`, `gamma = 0.01`, `colsample_bytree = 1`, `min_child_weight = 0` y `subsample = 1` pueden ser los hiperparámetros óptimos. Así que se estima otro Extreme Gradient Boosting con estos hiperparámetros y todos los datos de entrenamiento.

```

# Se crea la parrilla con los hiperparámetros que se quieren usar
xgbBstGrid <- expand.grid(nrounds=c(200),
  max_depth = c(10),
  eta = c(0.1),
  gamma = c(0.01),
  colsample_bytree = c(1),
  min_child_weight = c(0),
  subsample = c(1))
# Se estima el modelo final
set.seed(1)
xgbBstFit <- train(Win ~ .,
  data = dataTrain,
  method = "xgbTree",
  tuneGrid = xgbBstGrid,
  metric = "Accuracy")

```

Una vez obtenido el modelo, se predicen las observaciones del conjunto test y se comparan los valores predichos con los reales para obtener su precisión.

```

# Se predicen los datos test
xgbPred <- predict(xgbBstFit, dataTest)
# Se compara las predicciones con los valores reales
xgbAcc <- mean(as.character(xgbPred) == as.character(dataTest$Win))
# Matriz de confusión
kable(table(dataTest$Win, xgbPred)) %>%
  kable_styling() %>%
  add_header_above(c(" " = 1, "Predicción" = 2))

```

		Predicción	
		0	1
0	0	1410	57
	1	52	1411

El modelo Extreme Gradient Boosting ha hecho un trabajo espectacular, prediciendo correctamente un 96.28 % de las observaciones de los datos test.

```

# Importancia de las variables
kable(round(varImp(xgbBstFit, scale = FALSE)$importance * 100, 2),
  col.names = c("Porcentaje"),
  align = c("c"),
  format = "pipe") %>%
  kable_styling()

```

	Porcentaje
PC2	31.81
PC1	27.11
PC3	14.30
KB	4.92
FactionHorde	3.69
BattlegroundWG	1.91
BattlegroundTK	1.73

	Porcentaje
BattlegroundBG	1.73
BattlegroundTP	1.71
BattlegroundSM	1.36
ClassDruid	1.08
ClassWarlock	0.72
ClassPaladin	0.71
ClassDemon Hunter	0.70
ClassShaman	0.70
ClassRogue	0.69
ClassWarrior	0.68
BattlegroundSA	0.67
ClassHunter	0.64
ClassPriest	0.62
ClassMage	0.58
BattlegroundSS	0.55
BattlegroundES	0.55
ClassMonk	0.41
BattlegroundDG	0.30
Rolleal	0.13

En la tabla se puede ver la contribución de cada variable al modelo. Un porcentaje más alto implica que la variable explicativa es más importante. Se observa claramente que el rendimiento durante la batalla es el factor más importante a la hora de ganar la batalla, principalmente por los dos primeros componentes principales. Con menor importancia se tendría la facción del personaje, seguido de los campos de batalla y las clases. Y por último, el rol es la variable menos importante.

### Mejor modelo

Para elegir el mejor modelo se deben comparar los porcentajes de observaciones correctamente clasificadas. Aunque también se debe tener en cuenta el modelo trivial, que en este caso clasificaría todas las observaciones como ganadores ya que hay más ganadores que perdedores. Este modelo clasificaría correctamente un 50.68 % de las observaciones.

Modelo	Precisión
Trivial	50.68 %
Regresión Logística	78.46 %
Extreme Gradient Boosting	96.28 %

Clarísimamente, el modelo que mejor logra clasificar estos datos es el Extreme Gradient Boosting.

### Predicción de asesinatos

Ahora se quiere estimar un modelo para predecir la cantidad de enemigos que ha rematado un personaje. Se busca la mejor regresión de la variable KB sobre las demás, exceptuando D, HK, DD y HD debido a que su información ya está contenida dentro de los componentes principales. Se estiman dos modelos diferentes para ver cuál es mejor.

```

# Gráficos para ver si existe linealidad
plotPC1 <- ggplot(data = data, mapping = aes(x = PC1, y = KB)) +
  geom_jitter() +
  geom_smooth()

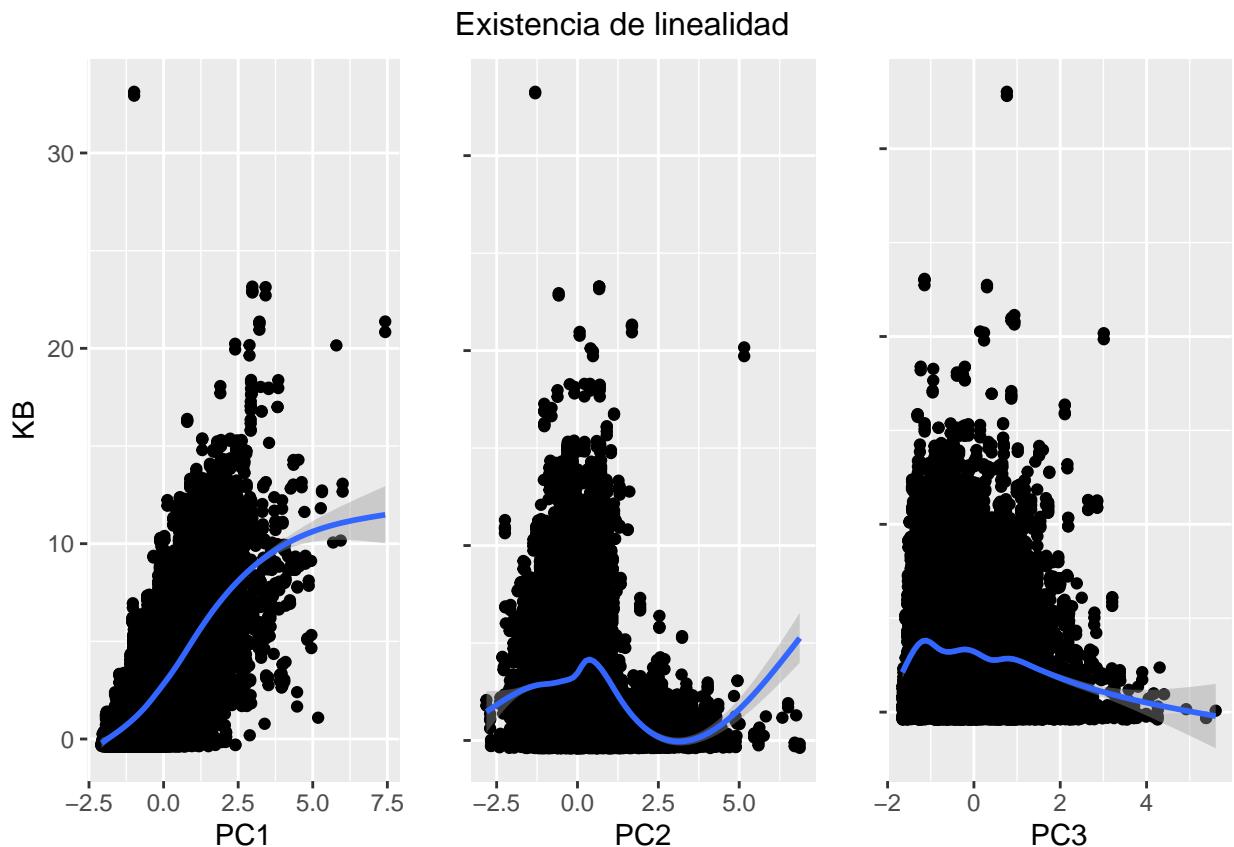
plotPC2 <- ggplot(data = data, mapping = aes(x = PC2, y = KB)) +
  geom_jitter() +
  geom_smooth() +
  ylab("") +
  scale_y_continuous(labels = NULL)

plotPC3 <- ggplot(data = data, mapping = aes(x = PC3, y = KB)) +
  geom_jitter() +
  geom_smooth() +
  ylab("") +
  scale_y_continuous(labels = NULL)

fig2 <- ggarrange(plotPC1, plotPC2, plotPC3,
                   ncol = 3, nrow = 1,
                   widths = c(1, 1, 1), heights = c(1))

annotate_figure(fig2,
               top = text_grob("Existencia de linealidad"))

```



Viendo las distribuciones de puntos se pueden descartar los modelos lineales, en el segundo gráfico se ve claramente que no existe relación lineal entre PC2 y la variable dependiente KB.

Por lo tanto primero se aplica una regresión Support Vector Machine, ya que es útil para regresiones no lineales, y luego se estima un modelo Random Forest, otro modelo basado en árboles parecido al Extreme Gradient Boosting pero más simple. En este caso en el que modelamos una regresión, se compararán las sumas de los cuadrados de los errores para elegir el mejor modelo.

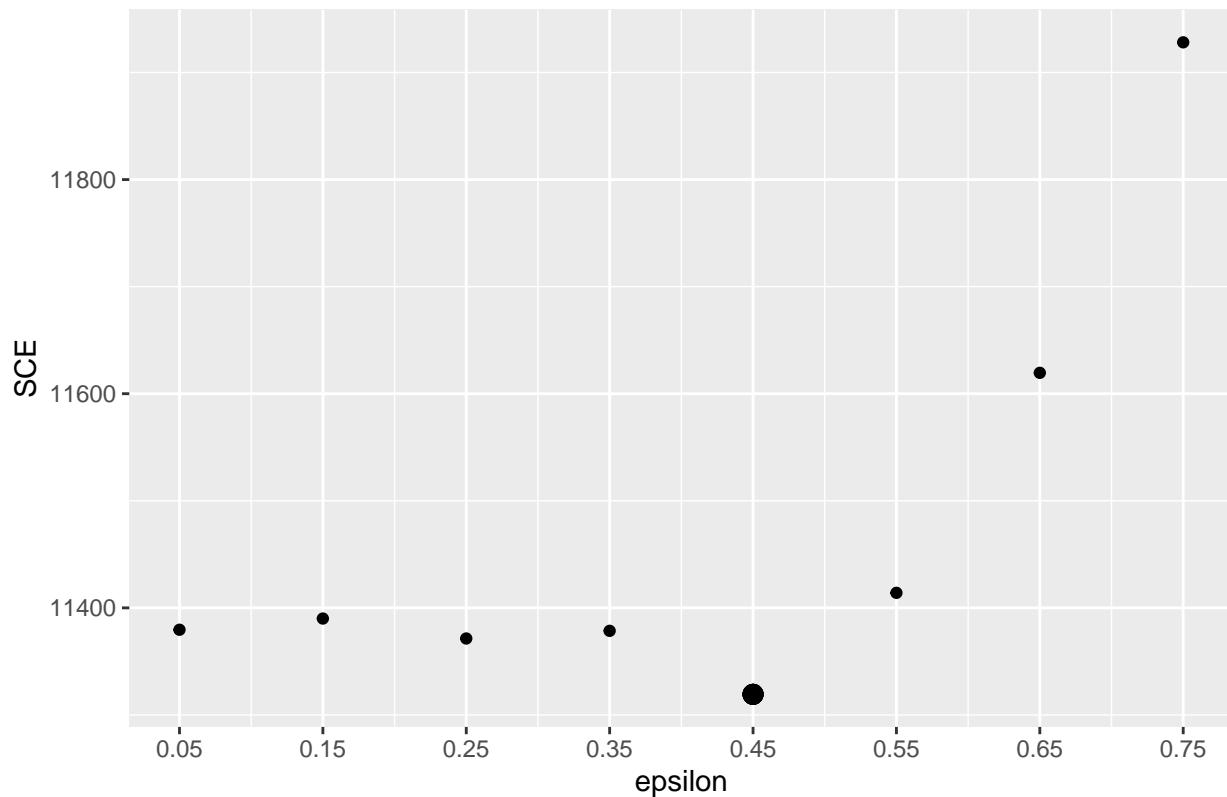
## Support Vector Machine

En este caso también se calculará la medida de error del modelo dividiendo los datos entre datos de entrenamiento y datos de validación.

Debido a la no linealidad de los datos, se utiliza un núcleo de base radial Gaussiana (los núcleos polinomial y sigmoide tienen peor capacidad de predicción). También se debe elegir el hiperparámetro `epsilon` con el que se especifica el umbral de error que se quiere tolerar. Se prueban diferentes valores para ver cuál es el óptimo.

```
# Se prueban 8 valores diferentes de epsilon
svmSCEeps <- c()
eps <- c(seq(from = 0.05, to = 0.75, by = 0.1))
for(i in 1:8) {
  # Se crea el modelo
  svmFit <- svm(KB ~ .,
    data = dataTrain,
    type = "eps-regression",
    kernel = "radial",
    epsilon = eps[i])
  # Se hace la predicción
  svmPred <- predict(svmFit, newdata = dataTest[,-6])
  # Se calcula la SCE
  svmSCEeps[i] <- sum((dataTest$KB - svmPred)^2)
}
ggplot(data.frame(epsilon = eps, SCE = svmSCEeps)) +
  geom_point(aes(x = epsilon, y = SCE)) +
  scale_x_continuous(breaks = eps) +
  geom_point(x = 0.45, y = svmSCEeps[5], size = 3) +
  ggtitle("Elección de epsilon óptimo")
```

## Elección de epsilon óptimo



```
svmSCE <- svmSCEeps [5]
```

Con `epsilon` = 0.45 se obtiene el mejor rendimiento del modelo, con una suma de cuadrados de los errores de 11319.

## Random Forest

A continuación se estima el modelo Random Forest. La metodología es muy similar a la del Extreme Gradient Boosting, aunque en este modelo únicamente tenemos el hiperparámetro `mtry` el cual representa el número de variables elegidas aleatoriamente para realizar la siguiente división del árbol de decisión.

Para elegir el valor de `mtry` óptimo, dividimos los datos de entrenamiento en 5 cajas y en cada una de ellas buscamos el valor óptimo del hiperparámetro. Se prueban los valores  $\{3, 4, 6, 9, 12\}$ .

```
# Se eligen los diferentes valores que se quieren probar
rfGrid <- expand.grid(mtry = c(3,4,6,9,12))
# Se crea la parrilla para ver el mejor valor de cada caja
rfTune <- matrix(nrow = 5, ncol = 1)
dimnames(rfTune) = list(c("k1", "k2", "k3", "k4", "k5"), c("mtry"))
# Se especifican los parámetros que se usarán para estimar los modelos
fitControl <- trainControl(method = "cv", number = 5)

set.seed(1)
for (i in 1:5) {
  # Se elige la caja
```

```

foldTrain <- dataTrain[folds[[i]],]
# Se estima el modelo de cada caja
rfFit <- train(KB ~ .,
                 data = foldTrain,
                 method = "rf",
                 trControl = fitControl,
                 tuneGrid = rfGrid,
                 metric = "Rsquared")
# Se actualiza la variable que guarda los hiperparámetros óptimos
rfTune[i,] <- as.matrix(rfFit$bestTune)
}

kable(rfTune, align = c("c","c"), format = "pipe") %>%
  kable_styling()

```

	mtry
k1	9
k2	12
k3	9
k4	9
k5	9

El valor óptimo parece ser `mtry = 9`. El problema es que se tienen 8 variables explicativas, por lo que el algoritmo da problemas aunque realmente las variables `Class` y `Battleground` sean variables categóricas multinomiales. Entonces se codifican las variables categóricas en formato One Hot Encode, pero no se usarán las variables `Faction.Alliance`, `Class.Death.Knight`, `Win.0`, `Rol.dps` ni `Battleground.AB` para no tener problemas de multicolinealidad.

Ahora se estima el modelo final con `mtry = 9` y todos los datos de entrenamiento, se predicen los datos de validación y se obtiene la suma de cuadrados de los errores.

```

# Se seleccionan las variables que queremos usar en el modelo
dataAux <- data %>%
  dplyr::select(1:3, 8:13)

# La función dummyVars del paquete caret permite codificar las variables categóricas en
# formato One Hot Encoder
dmy <- dummyVars(~ ., data = dataAux)
dataOhe <- data.frame(predict(dmy, newdata = dataAux))

# Se evita multicolinealidad
dataOhe <- dplyr::select(dataOhe,
                         -Faction.Alliance,
                         -Class.Death.Knight,
                         -Win.0,
                         -Rol.dps,
                         -Battleground.AB)

# División entre datos train y test
dataOheTrain <- dataOhe[iTrain,]
dataOheTest <- dataOhe[-iTrain,]

```

```

# Se estima el modelo con el hiperparámetro óptimo y los datos train
set.seed(1)
rfBstFit <- randomForest(KB ~ ., data = dataOheTrain, mtry = 9)

# Se predicen los datos test
rfPred <- predict(rfBstFit, newdata = dataOheTest[,-13])

# Se comparan las predicciones con los valores reales
rfSCE <- sum((dataOheTest$KB - rfPred)^2)

```

Con el modelo final de Random Forest se ha obtenido una suma de cuadrados de los errores de 2619.

También se puede obtener la disminución media del error cuadrático medio de las variables explicativas. Esta es una medida de la importancia global de la variable y representa la disminución de la impureza de los nodos producida por la variable en cuestión. A mayor valor, más importancia tiene la variable a la hora de hacer la predicción. En la tabla siguiente se pueden observar estos índices.

```

# Se compara la importancia de las variables
kable(arrange(tibble(Variable = row.names(rfBstFit$importance),
                     `Disminución Media del ECM` = round(rfBstFit$importance, 2)),
               desc(`Disminución Media del ECM`)),
      align = c("l", "c"),
      format = "pipe") %>%
kable_styling()

```

Variable	Disminución Media del ECM
PC1	58434.14
PC2	13961.01
PC3	11399.61
Rol.heal	9101.30
Win.1	3667.68
Class.Warrior	2254.76
Battleground.BG	1450.21
Faction.Horde	1351.71
Battleground.WG	1098.40
Battleground.TK	1069.70
Class.Priest	1049.69
Class.Paladin	1042.21
Battleground.SM	903.46
Battleground.ES	809.98
Class.Mage	793.02
Class.Warlock	786.06
Class.Demon.Hunter	754.70
Class.Shaman	737.53
Battleground.SA	692.64
Class.Druid	686.41
Class.Hunter	684.72
Battleground.TP	558.35
Class.Rogue	547.90
Battleground.DG	469.70
Class.Monk	271.97
Battleground.SS	153.51

La variable que más impacto tiene sobre los asesinatos del personaje es el primer componente principal, que básicamente es un indicador del ataque del personaje. Los demás componentes principales que incluyen el rendimiento del personaje durante la batalla también tienen mucha importancia, seguidos del rol del personaje y del resultado de la partida. En cuanto a la influencia de la clase y del campo de batalla, se tiene que tener en cuenta que el personaje base es un *Death Knight* que ha participado en la batalla Cuenca de Arathi (AB), ya que estas son las variables que no hemos usado para eliminar la multicolinealidad. Por lo que estos valores de importancia son relativos a este personaje base.

### Mejor modelo

Igual que en la comparación de los modelos de clasificación, se deben comparar los modelos estimados con un modelo trivial. En este caso, el modelo trivial de una regresión predice todas las observaciones con la media de la variable dependiente. Para poder comparar los modelos con la suma de cuadrados de los errores, las cantidades de observaciones predichas deben ser iguales.

```
# Media de SCE de datos test
trivPred <- rep(mean(dataTest$KB), nrow(dataTest))
# Rendimiento del modelo
trivSCE <- sum((dataTest$KB - trivPred)^2)
```

Modelo	SCE
Trivial	30454
Support Vector Machine	11319
Random Forest	2619

La capacidad de predicción del modelo Random Forest es mucho mejor que la de los demás modelos.