Introducción a R

Aplicaciones de Análisis Multivariado

Luis Román García Omar Trejo Navarro

Primavera 2014

Índice

1. ¿Qué es R?

R es **software para realizar análisis de datos**. Científicos de datos, estadísticos, analistas y *quants*, entre otros, hacen uso de R en sus diferentes disciplinas para análisis estadístico, visualización de datos y modelación predictiva. En R existen funciones para casi cualquier tipo de manipulación de datos, modelos estadísticos, y gráficas que un analista podría necesitar. Además de proporcionar la gran mayoría de los métodos utilizados actualmente, como mucha de la investigación en estadística, y áreas afines, se realiza actualmente con R, también se tiene acceso a recursos muy valiosos de vanguardia y que están siendo desarrollados hoy en día.

R es un **lenguaje de programación**. Para hacer análisis de datos con R se utilizan *scripts* y funciones. R es un lenguaje de programación orientado a objetos, interactivo y diseñado por estadísticos. El lenguaje provee objetos, operadores y funciones que permiten la exploración, modelado, y visualización de datos de forma natural. Análisis estadísticos completos y robustos pueden lograrse con relativamente pocas líneas de código.

R es un **proyecto de código abierto**. No solamente se puede descargar y utilizar sin costo alguno, sino que el código que se encuentra detrás de sus funciones está abierto para que el público en general lo consulte, estudie y modifique. Esto implica que tiene estándares muy altos de calidad porque mucha gente lo puede criticar y mejorar, y en realidad lo han hecho durante los años. Finalmente, otra gran ventaja de que sea abierto es que se desarrollan continuamente interfaces para que interactue con otros tipos de software de manera directa.

R es una **comunidad**. Fue creado inicialmente por Ross Ihaka y Robert Gentleman en la Universidad de Auckland en 1993. Actualmente hay al rededor de 20 estadísticos y computólogos, de todo el mundo, que liderean el proyecto. Además, miles de personas contribuyen continuamente con mejoras y funcionalidad agregada a través de *paquetes*. R tiene actualmente al rededor de dos millones de usuarios por todo el mundo. Esto explica la existencia de una gran comunidad, que interactua en línea, integrada con personas desde nivel básico hasta reconocidos investigadores.

Este manual ayudará a explorar ideas básicas y cómo usarlas para la materia de *Estadística Aplicada III*. Sin embargo, para aprender realmente a usar R será necesario explorar y experimentar uno mismo. Este documento, los ejemplos presentados y el código utilizado pueden ser descargados en https://github.com/otrenav/IaR. En esa página se encuentra la versión más actualizada de este documento.

A ambos nos interesa fomentar el uso de R, por lo que si tienen alguna duda no duden en contactarnos, y si podemos, con gusto les ayudaremos:

- **Omar Trejo Navarro**: otrenav@gmail.com
- **Luis Román García**: luis.roangarci@gmail.com

En caso de que encuentre errores en este documento pedimos que nos notifiquen. En cuanto

tengamos las correcciones listas actualizaremos el documento en línea.

2. Fundamentos

En esta sección abordaremos los fundamentos del análisis de datos en R. El material que se muestra a continuación es un resúmen del curso *R Programming* que se ofrece en Coursera.

Siempre que se muestre un > al inicio del código quiere decir que ese código se introduce directamente en la consola de R. En la sección de fundamentos, si el símbolo > no está al inicio de una línea es porque es lo que devuelve R al ejecutar algún comando. Escogemos ese símbolo porque es el mismo que muestra la consola de R y para que permita al lector diferenciar entre código y *output* de R.

Siempre que mostremos elementos en cursiva nos referimos a elementos de R. Por ejemplo, *matriz* habla de una matriz en el sentido de R, mientras que matriz habla de una matriz en el sentido matemático. Las diferencias son sutiles, pero queremos aclararlas porque en general una *matriz* contiene más información que una matriz (específicamente una *matriz* contiene atributos).

A medida que avancemos en los ejemplos, el lector deberá familiarizarse con varios aspectos de lenguaje, y consecuentemente dejaremos de comentar mucho de lo que hagamos. En caso de que el lector no entienda el código puede recurrir a secciones anteriores para ver si ahí lo explicamos, y si no lo hicimos nos gustaría que nos lo notifiquen para explicarlo debidamente.

2.1. Instalación

Para instalar R debemos ir a `http://cran.itam.mx/` (el *mirror* de R que se encuentra en el ITAM) y descargar la versión correspondiente a nuestro sistema (Windows, Mac o Linux). Además, si se quiere instalar RStudio[1] (lo recomendamos para principiantes), debemos ingresar a `https://www.rstudio.com/ide/download/desktop` y descargar la versión correspondiente a nuestro sistema.

Una vez que se tienen los archivos de instalación, ambos programas se pueden instalar como cualquier otro programa.

[1] RStudio es un entorno de desarrollo interactivo que ofrece al usuario una interfaz más intuitiva que la línea de comandos de R. Tiene varias características que lo hacen apto para principiantes como páneles para datos, gráficas, navegadores de archivos, documentación, entre otros.

2.2. Ayuda

Cuando se tengan dudas acerca de R muchas veces saldremos adelante con una rápida búsqueda en Google. Sin embargo, no debemos olvidar que se tienen los siguientes recursos disponibles.

- Foros en internet (http://r.789695.n4.nabble.com/)
- Buscadores en internet (http://www.rseek.org/)
- Manuales (http://cran.r-project.org/manuals.html)

Si se tienen dudas sobre cómo utilizar funciones específicas de R, podemos ver la descripción de la función con:

```
1 > ? prcomp
```

Hay veces que algunas funciones son ocultas por paquetes que se han cargado (esto pasa porque tienen el mismo nombre y queda disponible la que pertenece al último paquete que se cargó en memoria). Para ver en qué paquetes tenemos disponible alguna función podemos ejecutar:

```
1 > ?? prcomp
```

Si lo que queremos es ver el código de una función ejecutamos el nombre de la función sin paréntesis[2].

```
1 > prcomp
```

2.3. Directorio de trabajo

Cuando trabajamos en R el *directorio de trabajo* es un directorio en nuestro sistema de donde se leerán, cargarán y guardarán los cambios que hagamos, tanto código como datos[3].

Para ver en qué directorio nos encontramos debemos ejecutar:

```
1 > getwd()
```

Para cambiar a otro directorio podemos utilizar:

[2]Si la función se encuentra compilada (para aumentar eficiencia) no aparecerá directamente el código en pantalla, como es el caso con `prcomp`. En esos casos consultar: http://stackoverflow.com/questions/19226816/how-can-i-view-the-source-code-for-a-function para ver cómo acceder al código. En la mayoría de los casos no es complicado.

[3]Se puede trabajar directamente en otros directorios, pero se deben escribir las rutas completas.

```
> setwd("/Users/Usuario/Documents/R/")
```

Debemos recordar que en general siempre que se trabajen con archivos y datos dentro de R, se debe respetar la sintaxis de la ruta que utilice el sistema que estamos utilizando. En este caso el comando anterior se puede ejectuar en un sistema Mac (la sintaxis de rutas es la misma en Linux, pero cambia la estructura de las carpetas). Si quisieramos ejectuar el comando en Windows deberiamos utilizar:

```
> setwd("C:/Users/Usuario/Documents/R/")
```

Para ver los archivos que se encuentran en el directorio en el que estamos actualmente se utiliza:

```
> dir()
```

Cada vez que se creen nuevos *scripts* deberemos guardarlos para no perder el código. Posterioremente, para ejecutar el código utilizamos[4]:

```
> source("ejemplo.R")
```

2.4. Tipos de datos

En esta sección tratamos los siguientes temas:

- *Atomic classes*
- *Vectors*
- *Factors*
- *Missing values*
- *Data frames*
- *Names*

Un aspecto importante de R son los objetos. Todo dentro de R es un objeto. Estos objetos son creados para poder tratar diferentes problemas de maneras más eficaces y eficientes. Algunos de los objetos más comunes son *vectores*, *listas*, *arreglos*, *matrices*, *tablas*, y *data frames*[5] Sin embargo, hay mucho más, y al inicio, varios de los errores con los que se encon-

[4] Aquí suponemos que nos encontramos en el *directorio de trabajo* donde se encuentra el código y que el nombre del archivo es `ejemplo.R`.

[5] En ocasiones no traduciremos las palabras que se utilizan en R porque la traducción probablemente sería errónea y porque confundiría al lector cuando consulte documentación que probablemente estará en inglés. Cuando sea el caso, dejaremos el nombre del objeto en inglés.

trará uno al usar R tendrán que ver con uso de objetos de manera incorrecta. En caso de que el lector tenga un problema con su código, recomendamos que revise el tipo de objetos que está utilizando. Para ver el tipo de objeto de `prcomp`, por ejemplo, usamos:

```
> class(prcomp)
[1] "function"
```

R tiene las siguientes clases atómicas (los objetos más básicos):

- Caractéres
- Numéricos (números reales)
- Enteros (números enteros)
- Complejos (números complejos)
- Lógicos (falso/verdadero)

El tipo de objeto no atómico más básico es un *vector*. Cada *vector* sólo puede tener elementos de un tipo de objeto atómico, excepto las *listas*[6].

Cada objeto tiene atributos. Los atributos representan características específicas a cada objeto. Algunos ejemplos de atributos son:

- Nombres (`names`, `dimnames`)
- Dimensión (`dim`)
- Clase (`class`)
- Longitud (`length`)

El operador de asignación <- indica el valor que una variable será asiganada[7]. El signo # indica que todo lo que sigue en la línea es un comentario y debe ser ignorado por el interprete de R.

```
> x <- 1     # Asignación
> x          # Impresión implícita
[1] 1
> print(x)   # Impresión explícita
[1] 1
> mensaje <- "Hola"  # Asignación
> mensaje   # Impresión implícita
[1] "Hola"
```

[6]Una lista puede tener elementos de cualquier tipo, incluso listas. Se pueden pensar como una versión más flexible, en algunos aspectos, que los vectores.

[7]Otra forma de asignar a una variables es con =, igual que en otros lenguajes de programación. En la comunidad de R se prefiere <- para asignación de variables y = para asignación de parámetros dentro de funciones. Cuando se asignan parámetros en funciones sólo se puede utilizar =.

El número al inicio de cada salida indica en qué entrada del vector de *output* se encuentra el dato que se presenta al inicio de esa línea. Tiene mucho más sentido cuando se ven *outputs* más largos.

```
> x <- 1:20
> x
[1]  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
[16] 16 17 18 19 20
```

A continuación mostramos diferentes formas de crear vectores:

```
> x <- c(0.5, 0.6)          # Numérico
> x <- c(TRUE, FALSE)       # Lógico
> x <- c(T, F)              # Lógico (abreviado)
> x <- c("a", "b", "c")     # Caracter
> x <- 9:29                 # Entero
> x <- c(1 + 0i, 2 + 4i)    # Complejo
```

Usando la función `vector()` podemos crear un vector[8]:

```
> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0
```

Si mezclamos objetos al crear vectores no obtendremos error (aunque suene contraintuitivo porque dijimos que los vectores sólo pueden contener un sólo tipo de objeto). Lo que pasará es que se creará un objeto que represente el mínimo común denominador. Para ejemplificar:

```
> y <- c(1.7, "a")      # Caracter
> class(y)
[1] "character"
> y
[1] "1.7" "a"
> y <- c(TRUE, 2)       # Numérico
> class(y)
[1] "numeric"
> y
[1] 1 2
> y <- c("a", TRUE)     # Caracter
> class(y)
[1] "character"
> y
[1] "a" "TRUE"
```

[8]Por default el valor con el que se crea el vector, si no se especificó, es cero.

Se puede forzar a que un objeto sea representado como un objeto de otro tipo con las funciones `as.*()`[9]. Lo que muestra R después de cada comando es la conversión del vector original al tipo de datos que se solicita con la función `as.*()`.

```
> x <- 0:6
> x
[1] 0 1 2 3 4 5 6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Antes de ir más a fondo debemos explicar la existencia de `NaN` y `NA`. Cuando se pide que un tipo de data se convierta a otro, donde no existe una interpretación pertinente, obtenemos `NA` en lugar de la representación esperada y una advertencia:

```
> x <- c("a", "b", "c")
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

Por otro lado, un `NaN` se da cuando se ejecuta un comando que esperaríamos que devuelva un número pero obtenemos en su lugar una representación de *not a number* (`NaN`). También podemos obtener `Inf` o `-Inf`. Las interpretaciones son las mismas que en matemáticas:

```
> 1 / 0
[1] Inf
> 0 / 0
[1] NaN
> -1 / 0
[1] -Inf
> 1 / 0 + 1 / 0
[1] Inf
> 1 / 0 - 1 / 0
[1] NaN
```

[9] El $\star$ significa que en su lugar pueden ir varios nombres diferentes. En este caso podría ser remplazado con `as.numeric()`, `as.factor()`, `as.matrix()`, entre otros.

Tanto `NaN` como `NA` se consideran `NAs`, mientras que sólo `NaN` se considera `NaNs`.

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

La existencia de estos elementos se prueba con `is.na()`, `is.nan()` y `is.finite()`, respectivamente, y nos sirven para identificar valores faltantes en las bases de datos que utilizaremos. Una forma práctica de quitar `NAs` es[10].:

```
> x <- c(1, 2, NA, 4, NA, 5)
> NAs <- is.na(x)
> x[!NAs]
[1] 1 2 4 5
```

Si lo que queremos es encontrar los casos donde la i-ésima posición en cada vector es válida, entonces usamos:

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> buenos <- complete.cases(x, y)
> buenos
[1] TRUE TRUE FALSE TRUE FALSE TRUE
> x[buenos]
[1] 1 2 4 5
> y[buenos]
[1] "a" "b" "d" "f"
```

Ya que mencionamos la existencia de `NA`, `NaN` y `Inf` continuaremos con otros objetos. Las *matrices* en R son vectores que además tiene un atributo de *dimensión*[11]. El atributo de *dimensión* es en sí mismo un vector de longitud dos (`nrow`, `ncol`).

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
```

[10] El símbolo `!` significa negación y se coloca antes de lo que se quiere negar. En este caso queremos los casos que no son `NAs`.

[11] El elemento con el que se crean las *matrices* es por default `NA`.

```
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Una forma alternativa de crear *matrices* es especificando los elementos que queremos que tengan. Cuando se hace de esta forma se debe notar que R llena la *matriz* por columnas.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Otra forma de crear *matrices* es cambiando el atributo de *dimensión* de un *vector*:

```
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Más aún, otra forma de crear *matrices* es uniendo *vectores*:

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
      x   y
[1,]  1  10
[2,]  2  11
[3,]  3  12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

Otro tipo de objetos son las *listas*. Cuando nos referimos al elemento de una lista usamos `[[]]` mientras que cuando nos referimos a los elementos de un vector sólamente usamos `[]`.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1
[[2]]
[1] "a"
[[3]]
[1] TRUE
[[4]]
[1] 1+4i
```

Los *factor* son datos categóricos, no necesariamente ordenados. Este tipo de datos son tratados de manera especial por funciones como `lm()` y `glm()` (las veremos más adelante en este documento). El uso más común de estas variables es funcionar como etiquetas porque es más fácil interpretar resultados cuando indican *hombre* y *mujer*, en lugar de `0` y `1`.

```
> x <- factor(c("hombre", "hombre", "mujer", "hombre", "mujer"))
> x
[1] hombre hombre mujer hombre mujer
Levels: hombre mujer
> x
hombre  mujer
     3      2
> unclass(x)
[1] 1 1 2 1 2
attr(,"levels")
[1] "hombre" "mujer"
```

Más adelante cuando necesitemos definir *grupos control*, deberemos definir cuáles son los controles. Escoger grupos control es importante en modelos lineales y modelos lineales generales. El control de cada `factor` es el primer elemento de los niveles. En este caso sería *hombre*.

```
> x <- factor(c("hombre", "hombre", "mujer", "hombre", "mujer"), ↩
   levels = c("hombre", "mujer")
> x
[1] hombre hombre mujer hombre mujer
Levels: hombre mujer
```

Los *data frames* se utilizan para guardar datos de diferentes tipos, cumplen la función de una base de datos. Se representan como un tipo especial de listas, y pueden tener diferentes tipos de objetos en cada columna. Tienen un atributo adicional llamado *row.names*, y pueden ser convertidos a matrices con `data.matrix()`. Para crear un *data frame* normalmente utilizamos `read.table()` o `read.csv()`, pero también lo podemos hacer con la función `data.frame()`.

```
> x <- data.frame(var = 1:4, iab = c(T, T, F, F))
> x
   var    iab
1    1   TRUE
2    2   TRUE
3    3  FALSE
4    4  FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

Cualquier objeto puede recibir un nombre, y en ocasiones esto facilita mucho el análisis.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("var", "iab", "le")
> x
 var  iab   le
   1    2    3
> names(x)
[1] "var" "iab" "le"
```

```
> m <- matrix(1:4, nrow = 2, ncol = 3)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
   c  d
a  1  3
b  2  4
```

Hasta aquí hemos tratado los tipos de datos más básicos de R. En la siguiente sección mostraremos como se pueden utilizar los objetos anidados.

2.5. Objetos anidados

Existen tres formas de extraer objetos anidados (objetos que se encuentran dentro de otros objetos):

- `[]`, se puede utilizar para seleccionar más de un elemento y siempre regresa un objeto del mismo tipo que el objeto original.
- `[[]]`, se utiliza para extraer elementos de una lista o de un *data frame*, y la clase del objeto que regresa no necesariamente será del mismo tipo que el objeto original.

- $, se utiliza para extraer elementos de una lista o de un *data frame* por nombre.

A continuación ilustraremos como accesamos a estos objetos anidados de diferentes formas:

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
```

Podemos acceder a los elementos de un *vector* de forma condicional. Esto quiere decir que buscamos aquellos elementos del *vector* que cumplan cierta condición. En este caso la condición es que sean mayor que a (los elementos que cumplen esta condición son b, c y d):

```
> x[x > "a"]
[1] "b" "c" "c" "d"
> u <- x > "a"
> u
[1] FALSE TRUE TRUE TRUE TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Los elementos de *matrices* pueden ser extraídos como lo haríamos normalmente usando notación matemática:

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Podemos referirnos a todos los elementos de una fila o columna si dejamos su respectivo índice en blanco:

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

Cuando escogemos elementos de una *matriz* y la *dimensión* del resultado es de una o varias *dimensiones* menor que la del objeto original, tenemos una reducción implícita de dimensión. En el ejemplo a continuación la tercera línea muestra un *vector* en lugar de una *matriz*. Esto es deseable la mayoría de las veces, pero en ocasiones puede ocasionar problemas. Para

preservar la dimensión del objeto debemos pasar el argumento `drop = FALSE`. La quinta línea muestra una *matriz*:

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
     [,1] [,2] [,3]
[1,]    1    3    5
```

Si damos nombres a las diferentes columnas, cuando necesitamos extraer alguna de ellas es más fácil hacerlo por nombre que por índice. Si son muchas columnas probablemente no recordemos en cuál se encuentra la variable que queremos extraer, pero el nombre es más fácil de recordar. No debemos recordar exactamente en qué índice está el elemento que queremos, podemos simplemente llamarlo por nombre con `$`.

```
> x <- list(vari = 1:4, able = 0.6)
> x[1]
$vari
[1] 1 2 3 4
> x[[1]]
[1] 1 2 3 4
> x$able
[1] 0.6
> x[["able"]]
[1] 0.6
> x["able"]
$able
[1] 0.6
```

Una ventaja de utilizar `[[]]` es que podemos extraer elementos después de algún cálculo, mientras que `$` no puede ser utilizado en esos casos porque busca nombres de manera literal.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "Hola")
> name <- "foo"
> x[[name]]     # Elemento calculado
[1] 1 2 3 4
> x$name        # No exite (busca literal)
NULL
> x$foo         # Existe
[1] 1 2 3 4
```

2.6. Operaciones vectorizadas

Una ventaja importante de lenguajes de programación que trabajan con operaciones vectorizadas es que las pueden realizar de manera muy eficiente. En este caso realizamos una suma, una multiplicación y una división, todas elemento a elemento:

```
> x <- 1:4
> y <- 6:9
> x + y
[1] 7 9 11 13
> x * y
[1] 6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Si queremos realizar operaciones con matrices debemos utilizar el operador `%*%`.

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)
> x * y      # Operación elmento a elemento
     [,1]  [,2]
[1,]   10    30
[2,]   20    40
> x / y
     [,1]  [,2]
[1,]  0.1   0.3
[2,]  0.2   0.4
> x %*% y   # Multiplicación matricial
     [,1]  [,2]
[1,]   40    40
[2,]   60    60
```

Otro aspecto de las operaciones vectorizadas es que puedes encontrar qué elementos cumplen cierta característica. En este caso queremos aquellos elementos de `x` que son mayores que dos y mayores o iguales que dos, además utilizamos el cálculo para extraerlos del *vector*.

```
> x > 2
[1] FALSE FALSE TRUE TRUE
> x[x > 2]
[1] 3 4
> x >= 2
[1] FALSE TRUE TRUE TRUE
> x[x >= 2]
[1] 2 3 4
```

2.7. Lectura y almacenamiento

Las principales funciones para leer y almacenar datos son:

- `read.table()`[12], para datos tabulados.
- `readLines()`, para archivos de texto.
- `source()`, para código de R.
- `load()`, para *workspaces*.

Por ser la que más utilizaremos en este documento, explicaremos la función `read.table()`, las demás se dejan al lector. Algunos de los parámetros de la función `read.table()` son:

- `file`, nombre del archivo (puede ser en internet).
- `header`, valor lógico indicando si existen encabezados.
- `sep`, caracter que indica la separación de los datos.
- `colClasses`, vector de caracteres que indica la clase de cada columns.
- `nrows`, el número de filas en el archivo.
- `comment.char`, caracter que indica el símbolo de comentarios.
- `skip`, número de filas que se ignorarán desde el inicio.
- `stringsAsFactors`, valor lógico que indica si los caracteres deben ser codificados como *factors*.

Para leer archivos con extensión .CSV utilizamos `read.table()` con el parámetro `sep = ","`[13]:

```
1 > datos <- read.table("archivo.csv", sep = ",")
```

Otra forma de leer archivos con extensión .CSV podemos utilizar `read.csv()`:

```
1 > datos <- read.csv("archivo.csv")
```

En general existen muchas más maneras de leer datos. Se pueden leer datos de formatos de compresión o de otros programas como SPSS. Para el lector interesado, recomendamos leer el manual. A continuación mostramos el caso de datos en formato .SAV o .POR, formatos que utiliza el software SPSS de IBM. Simplemente debemos cargar en memoria el paquete `foreign` que permite leer y guardar datos en varios formatos.

[12] Existe una función llamada `read.csv()` que es exactamente igual a `read.table()` con `sep = ","` porque el que se encuentra por default en R es con espacios.

[13] En estos ejemplos asumimos que existen los archivos que estamos leyendo en el comando y que nos encontramos en el *directorio de trabajo* donde se encuentran estos archivos. Además cuando guardamos archivos, quedan en ese mismo *directorio de trabajo* porque no lo cambiaremos.

```
> require(foreign)
> datos <- read.sav("archivo.sav", header = TRUE)
```

También es posible leer datos directo de archivos con extensión .XLS y .XLSX (formatos propiedad de Microsoft y que se utilizan en archivos de Excel). Para lograr esto hay varias formas, aquí mostraremos la que usa el paquete `XLConnect`. Para que esta forma funcione es necesario tener una versión actualizada de Java en el sistema. Primero debemos instalar el paquete `XLConnect` y después lo cargamos en memoria. Ahora debemos abrir el archivo (*workbook*) y ya que abrimos el archivo obtenemos los datos de la hoja que nos interesa.

```
> install.packages("XLConnect")
> require(XLConnect)
> wb <- loadWorkbook("archivo.xlsx")
> datos <- readWorksheet(we, sheet = "Sheet 1", header = TRUE)
```

Si se quiere trabajar con bases de datos muy grandes se recomienda leer la página de `read.table()` en el manual de R. Ahí se pueden encontrar muchos tips de cómo hacer más eficiente la lectura de datos. En general, entre más parámetros se especifiquen al llamar la función `read.table()` la lectura se hará con mayor eficiencia.

En cuanto a cómo almacenar datos, solamente explicaremos el caso de archivos .CSV porque es el formato que recomendamos utilizar y porque es sencillo de utilizar[14].

```
> write.csv(datos, file = "datos.csv")
```

2.8. Información de objetos

Una vez que tenemos los datos cargados en memoria nos interesa entender la estructura básica que presentan para saber qué tipos de análisis son más adecuados y con qué partes de la base de datos debemos tener más cuidado. Para lograr esto utilizamos las funciones `head()`, `summary()` y `str()`. La primera sirve para ver los primeros elementos de una base de datos, la segunda sirve para ver algunas estadísticas sobre cada tipo e dato en la base de datos y la tercera muestra la cantidad de observaciones por variable, y es particularmente útil cuando se quiere analizar la estructura general de listas anidadas.

En el siguiente ejemplo obtenemos 100 observaciones de una distribución normal con media dos y varianza cuatro. Después vemos los primeros elementos del vector donde guardamos las observaciones con `head()`, obtenemos una serie de estadísticas con `summary()` y por último vemos que es de tipo numérico con cien observaciones y sus primeros elementos con `str()`.

[14] Para este ejemplo suponemos que existe un *data frame* en memoria llamado *datos* y que nos encontramos en el *directorio de trabajo* en donde queremos guardar los datos.

```
> x <- rnorm(100, 2, 4)
> head(x)
[1]  8.8858508 -1.7233194 -5.4726760  0.1517995  4.8811799  ↩
   2.6323957
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-8.8570 -0.3367  1.6260  1.4330  3.6820 10.3200
> str(x)
num [1:100] 8.886 -1.723 -5.473 0.152 4.881 ...
> f <- gl(40, 10)
```

El lector debe notar que al correr el ejemplo anterior en su sistema probablemente no obtenga los mismo números que obtuvimos nosotros. Esto representa un problema a la hora de realizar estudios o ejemplos que sean verificables y repetibles. Para superar este problema debemos hacer uso de las semillas. Una semilla, sin ahondar mucho en el tema, permite que el generador de números pseudo-aleatorios de R genere los mismo números y así hacer repetibles los experimentos que hagamos. En el siguiente ejemplo se muestra cómo utilizar semillas y cómo podemos aplicar las mismas funciones del ejemplo anterior a *matrices*.

```
> set.seed(123454321)
> m <- matrix(rnorm(50), 10, 5)
> head(m)
           [,1]       [,2]        [,3]       [,4]        [,5]
[1,] -1.9427595 -1.1577822 -1.76338227  1.1072043  1.56230798
[2,]  1.4254461  0.4190303 -0.03476158 -0.1130442 -0.31753878
[3,] -0.9034276 -1.3272208  0.68913452 -1.4872828 -0.01320191
[4,] -0.6337974 -0.7988829 -1.15240556 -0.3040702 -0.68325655
[5,]  1.7360916 -0.9236041 -1.40937628 -0.3651207  1.37978598
[6,] -2.0509408 -0.5640992 -1.28606062  0.6082227 -0.34194015
> summary(m)
       V1                 V2                  V3
 Min.   :-2.0509   Min.   :-1.32722   Min.   :-1.7634
 1st Qu.:-1.1042   1st Qu.:-0.89242   1st Qu.:-1.2526
 Median :-0.5369   Median :-0.67324   Median :-0.7625
 Mean   :-0.1090   Mean   :-0.46454   Mean   :-0.7212
 3rd Qu.: 1.3488   3rd Qu.: 0.04303   3rd Qu.:-0.3502
 Max.   : 1.7716   Max.   : 0.80825   Max.   : 0.6891
       V4                 V5
 Min.   :-1.48728   Min.   :-0.68326
 1st Qu.:-0.48022   1st Qu.:-0.32477
 Median :-0.20856   Median : 0.08176
 Mean   :-0.09525   Mean   : 0.33173
 3rd Qu.: 0.43036   3rd Qu.: 1.05706
 Max.   : 1.10720   Max.   : 1.56231
```

```
> str(m)
num [1:10, 1:5] -1.943 1.425 -0.903 -0.634 1.736 ...
```

Si ahora el lector repite los comandos se dará cuenta que obtendrá exactamente los mismos números. Los mismos comandos que hemos visto aplicados en *vectores* y *matrices* también se pueden utilizar en *data frames*. En este ejemplo cargamos varias bases de datos en memoria, pero sólo utilizamos una que tiene que ver con medicioens de calidad de aire. Podemos observar, por el comando `str()` que la base de datos *airquality* contiene 153 observaciones de 6 variables. Además tenemos cinco variables que fueron medidas como enteros y una que fue medida como número real. Si separamos la base de datos por la variable *month* (en esta base se midio durante cinco meses) obtenemos una lista anidada, y podemos ver los resultados de cada mes. Viendo los resultados mensuales notamos que en el segundo mes se realizó una observación menos que en el resto de los meses. Estos son los tipos de detalles de los cuales podemos darnos cuenta con estos comandos.

```
> require(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> str(airquality)
'data.frame':   153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
> s <- split(airquality, airquality$Month)
> str(s)
List of 5
 $ 5:'data.frame':  31 obs. of  6 variables:
  ..$ Ozone  : int [1:31] 41 36 12 18 NA 28 23 19 8 NA ...
  ..$ Solar.R: int [1:31] 190 118 149 313 NA NA 299 99 19 194 ...
  ..$ Wind   : num [1:31] 7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 ...
  ..$ Temp   : int [1:31] 67 72 74 62 56 66 65 59 61 69 ...
  ..$ Month  : int [1:31] 5 5 5 5 5 5 5 5 5 5 ...
  ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
 $ 6:'data.frame':  30 obs. of  6 variables:
  ..$ Ozone  : int [1:30] NA NA NA NA NA NA 29 NA 71 39 ...
  ..$ Solar.R: int [1:30] 286 287 242 186 220 264 127 273 291 ...
```

```
  ..$ Wind   : num [1:30] 8.6 9.7 16.1 9.2 8.6 14.3 9.7 6.9 ...
  ..$ Temp   : int [1:30] 78 74 67 84 85 79 82 87 90 87 ...
  ..$ Month  : int [1:30] 6 6 6 6 6 6 6 6 6 6 ...
  ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
 $ 7:'data.frame':  31 obs. of  6 variables:
  ..$ Ozone  : int [1:31] 135 49 32 NA 64 40 77 97 97 85 ...
  ..$ Solar.R: int [1:31] 269 248 236 101 175 314 276 267 ...
  ..$ Wind   : num [1:31] 4.1 9.2 9.2 10.9 4.6 10.9 5.1 6.3 ...
  ..$ Temp   : int [1:31] 84 85 81 84 83 83 88 92 92 89 ...
  ..$ Month  : int [1:31] 7 7 7 7 7 7 7 7 7 7 ...
  ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
 $ 8:'data.frame':  31 obs. of  6 variables:
  ..$ Ozone  : int [1:31] 39 9 16 78 35 66 122 89 110 NA ...
  ..$ Solar.R: int [1:31] 83 24 77 NA NA NA 255 229 207 222 ...
  ..$ Wind   : num [1:31] 6.9 13.8 7.4 6.9 7.4 4.6 4 10.3 8 ...
  ..$ Temp   : int [1:31] 81 81 82 86 85 87 89 90 90 92 ...
  ..$ Month  : int [1:31] 8 8 8 8 8 8 8 8 8 8 ...
  ..$ Day    : int [1:31] 1 2 3 4 5 6 7 8 9 10 ...
 $ 9:'data.frame':  30 obs. of  6 variables:
  ..$ Ozone  : int [1:30] 96 78 73 91 47 32 20 23 21 24 ...
  ..$ Solar.R: int [1:30] 167 197 183 189 95 92 252 220 230 259 ...
  ..$ Wind   : num [1:30] 6.9 5.1 2.8 4.6 7.4 15.5 10.9 10.3 ...
  ..$ Temp   : int [1:30] 91 92 93 93 87 84 80 78 75 73 ...
  ..$ Month  : int [1:30] 9 9 9 9 9 9 9 9 9 9 ...
  ..$ Day    : int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
```

2.9. Gráficas básicas

Al igual que los demás temas, el tema de gráficas será tratado de manera muy breve. Sin embargo, en los ejemplos de aplicaciones de estadística multivariada haremos uso extensivo de `ggplot2` y un poco de `rgl`. Se recomienda al lector repasar cada uno de esos ejemplos hasta que entienda que hace cada línea.

Para empezar a ver algunos ejemplos de gráficas y el código que las generó podemos ejecutar:

```
> example(points)
```

Gráficar en R puede parecer difícil sobre todo si se quiere ajustar mucho los detalles. Además existen varios métodos y sistemas a la hora de graficar, los más comúnes son:

- `graphics`, incluye funciones básicas para graficar. Contiene funciones como `plot`, `hist`, `boxplot`, entre otras.

- `lattice`, mejora el sistema básico de gráficas de R. Contiene funciones como: `xyplot`, `bwplot`, `levelplot`, entre otras.
- `ggplot2`, mejora el sistema básico de gráficas de R. Es independiente de lattice y no se pueden combinar. Es un intento por generalizar y crear una gramática de gráficas.
- `rgl`, permite la creación de gráficas dinámicas que representan tres dimensiones.

Aunque aquí se dará un breve ejemplo de cómo se grafica en R con el paquete básico de gráficas, a lo largo del documento utilizaremos `ggplot2` y `rgl` por ser los que más utilizamos.

```
1 > x <- rnorm(100)
2 > hist(x)
```

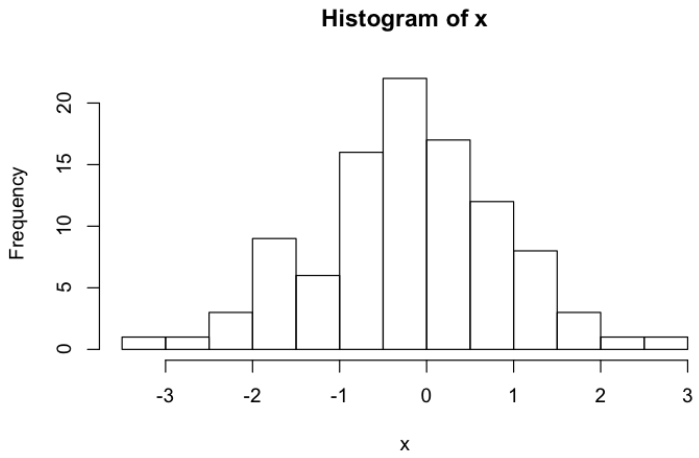

Figura 1: Histograma

Debemos notar que aunque no pasamos parámetros a la función `hist()`, automáticamente incluyó título y nombres en los ejes. A continuación mostramos como hacer un *scatter plot* con dos *vectores*.

```
1 > y <- rnorm(100)
2 > plot(x, y)
```

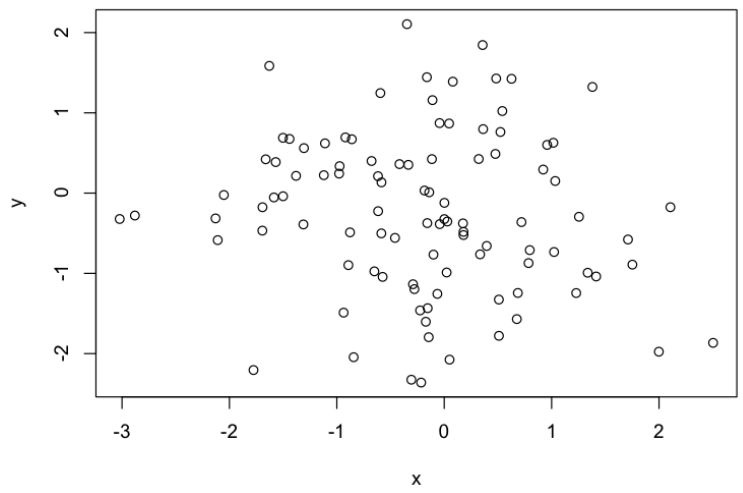

Figura 2: Scatter plot

En la siguiente gráfica ajustamos algunos parámetros, colocamos una leyenda, corremos una regresión lineal y la graficamos sobre la gráfica anterior.

```
1 > plot(x, y, xlab = "Peso", ylab = "Altura",
2     main = "Scatterplot", pch = 1, col = "blue")
3 > legend("topright", legend = "Datos", pch = 1, col = "blue")
4 > linea <- lm(y ~ x)
5 > abline(linea, col = "red")
```

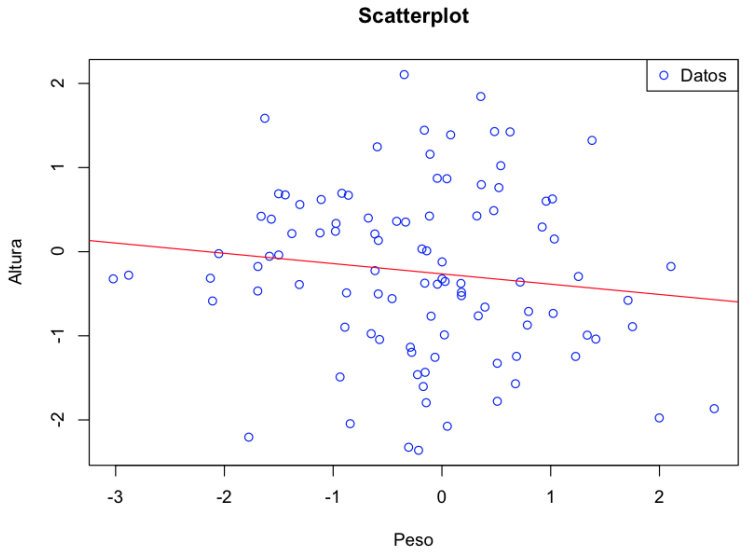

Figura 3: Scatter plot con regresión lineal y leyenda

Esto es todo lo que mostraremos por ahora de gráficas. En los ejemplos que desarrollaremos a continuación el lector podrá darse cuenta de cómo utilizar los sistemas `ggplot2` y `rgl`. No explicaremos muchos de los parámetros que utilizaremos, pero al lector interesado se le recomienda consultar http://ggplot2.org/ y http://rgl.neoscientists.org/about.shtml.

2.10. Conclusión

Con esto concluímos los fundamentos de R. Mucho del gran potencial de R se encuentra en entender bien los fundamentos, y estamos conscientes de que hemos dejado mucho de lado. Hemos tratado de mostrar lo mínimo necesario para que se entiendan los ejemplos que mostraremos a continuación. Sin embargo, en varios partes incluímos más información de la mínima necesaria para entender el resto del documento. Esto lo hicimos para mostrar algunas características de R que aunque no son necesarias para entender el resto del documento son muy útiles.

Hay temas muy importantes que no hemos tratado aquí y que se recomienda al lector que los estudie más a fondo si planea usar R para análisis de datos. Específicamente hemos dejado de lado estructuras lógicas, simulación, distribuciones de probabilidad, ambientes de desarrollo, excepciones, funciones, efectos secundarios, programación orientada a objetos, pruebas estadísticas y *debugging*, entre otros.

Por último, se recomienda al lector que visite el sitio web https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml y estudie los usos y costumbres aceptadas R. Esto facilitará mucho la lectura del código en futuro tanto para la persona que lo escribió como para los demás, y permitirá enfocarse en el análisis y no en el código.

3. Aplicaciones de Estadística Multivariada

Ya que se explicaron los fundamentos de R procedemos a mostrar aplicaciones de estadística multivariada. La primera sección muestra la preparación de los datos que serán utilizados en el análisis de componentes principales que se muestra en la sección que le sigue. Utilizamos una base de datos conformada por variables de las elecciones federales del año 2012 y datos de marginación del año 2010. Estas dos primeras secciones se deben considerar como un sólo ejemplo, pero quisimos hacer la separación para mostrar por un lado la limpieza de datos y por el otro la técnica de análisis. Esto con el fin de facilitar el entendimiento para aquellas personas que van empezando con R.

Para los ejemplos de correspondencias (simple y múltiple) utilizamos la base de datos *Tea*, para el análisis de discriminación lineal usamos la base de datos *Iris*[15] y para el ejemplo de

[15]Las bases de datos *Tea* e *Iris* se encuentran dentro de R; las podemos cargar en memoria con `data(tea)`

regresión logística utilizamos una base de datos derivada de *Global Terrorism Database*. En estos ejemplos no se mostrará la limpieza de datos porque las bases de datos utilizadas ya están limpias.

Asumiremos que el lector conoce las matemáticas detrás de los métodos, por lo que no las explicaremos y no interpretaremos los resultados, solamente mostraremos como se ejecutan las técnicas[16]. Además, el lector debe recordar que todo el código mostrado a continuación se encuentra publicado en línea por lo que no es necesario que lo copie de este documento a su sistema.

El lector notará que en el código de estos ejemplos deja de aparecer el símbolo >. Esto es porque el código que se presenta a continuación no está diseñado para ser introducido directamente a la consola de R —está diseñado para ser parte de un `script` que será ejecutado desde R. Lo que hacemos es explicar las partes de los `script`.

Finalmente, antes de comenzar, cabe aclarar que mucho del código mostrado es para generar buenas gráficas. El análisis es relativamente sencillo de realizar, pero el ajustar las gráficas a detalle puede resultar complejo en ocasiones (y en mucho código). Es posible ahorrarse mucho código a expensas de tener gráficas de calidad sub-óptima. Para aquellos que prefieran gráficas fáciles de implementar mostramos una gráfica sencilla de hacer, en caso de que sea posible. El lector podrá comparar estas gráficas con aquellas donde se ha tomado más tiempo en su elaboración. Al final es cuestión de preferencias acerca de qué gráficas desea utilizar el lector.

3.1. Base de datos

Primero debemos descargar las BDs[17]. Siempre que sea posible debemos bajar la información en un formato *plano*, i.e. que contenga solamente texto. Esto nos facilitará la lectura y evitará potenciales problemas por fórmulas dentro de los datos. En este caso descargaremos la BD del Instituto Federal Electora (IFE) en formato `.TXT` y la del Consejo Nacional de Población (CONAPO) en format `.XLSX`. Las páginas de donde se pueden obtener las BDs son:

- Elecciones Federales 2012: http://siceef.ife.org.mx/pef2012/SICEEF2012.html
- Índice de Marginación 2010: http://www.conapo.gob.mx/es/CONAPO/Indices_de_Marginacion_2010_por_entidad_federativa_y_municipio

y `data(iris)`, y posteriormente llamarlas con sus respectivos nombres.

[16] La teoría detrás de cada método puede ser estudiada de las notas de Estadística Aplicada III del Prof. Rubén Hernández Cid. Tratamos de mantener una relación biyectiva entre esas notas y estos ejemplos.

[17] Asumimos que todos los archivos se guardarán en el mismo directorio en el que se trabajará con R para evitar el uso de rutas a los archivos.

Lo más probable es que las BDs no tengan el formato que necesitamos para utilizarlas correctamente, por lo que tenemos que *limpiarlas* primero. Específicamente en este caso observamos que:

1. La BD de CONAPO
   a) Presenta números en porcentajes.
   b) Contiene información extra estorbando la lectura.
   c) Tiene nombres de variables imprácticos.
   d) Contiene la clave de entidad federativa.
   e) No contiene los nombres de los estados.
2. La BD del IFE
   a) Se encuentra limpia.
   b) Presenta números en cantidades absolutas.
   c) Utiliza separación no-estándar (barras).
   d) No contiene la clave de entidad federativa.
   e) Contiene los nombres de los estados.

Por lo anterior, debemos abrir la BD de CONAPO en Excel y quitar todo lo que no sea útil. Específicamente hay que eliminar las filas 1, 2, 4, 5, 6, 7, 8, 2465, 2466, 2467 y 2468. Si no hacemos este paso, R no leerá bien la información. También cambiaremos los nombres de las variables que utilizaremos a: `clv_ent_fed`, `clv_mncp`, `mncp`, `pob_tot`, `analf`, `prm`, `drnj`, `ee`, `ag`, `hac`, `pt`, `hbt`, `smin`, `ind_marg`, `grd_marg`, `ind_marg_0_100` y `lugar`. Finalmente hay que exportar la información a un archivo `.CSV`. De lo demás nos ocuparemos dentro de R.

Una vez que tenemos los archivos `.CSV` podemos empezar a usar R. Antes que nada, debemos cargar las BDs y quitar los datos que fueron leídos de más. Esto lo hacemos con:

```
1 ife <- read.table("IFE_2011.txt", header = TRUE, sep = "|")
2 conapo <- read.csv("CONAPO_2010.csv", header = TRUE)
3 conapo <- conapo[1:2456, 1:17]
```

Si se desea se puede ver un resumen de las BDs con `summary()`, y si se desea ver las primeras entradas de las BD para observar la estructura, se puede hacer con `head()`. En ambos casos, hay que pasar el objeto que queremos analizar a las funciones.

Queremos que ambas BDs estén juntas, pero para hacerlo debemos:

1. Calcular los agregados a nivel estado.

2. Juntar ambas BDs por estado.

La BD de CONAPO está relativamente en orden. Solamente debemos obtener las filas que contienen información relevante:

```
conapo_limpia <- matrix(0, 32, 10)
conapo_limpia[, 1] <- c(1:32)
colnames(conapo_limpia) <- colnames(conapo[, c(1,5:13)])
for (i in 1:32) {
    for (j in 1:9) {
        conapo_limpia[i, j + 1] <-
            mean(as.matrix(conapo[conapo$clv_ent_fed == i, ][j + 4]))
    }
}
```

La BD del IFE es mucho más difícil de limpiar porque no viene con cantidades porcentuales, por lo que las debemos calcularlas por nuestra cuenta[18]. Lo primero es crear los objetos donde estarán nuestros datos limpios:

```
colnames(ife) <- tolower(colnames(ife))
ife_limpia <- matrix(0, 32, 8)
ife_limpia[, 1] <- c(1:32)
entidades <- unique(ife[, 1])
ife_limpia[, 2] <- as.character(entidades)
```

Para facilitar la lectura de las gráficas, utilizaremos las abreviaturas oficiales con tres letras y colocaremos los nombres de las columnas de la BD:

```
ife_limpia[, 3] <- c("AGU", "BCN", "BCS", "CAM", "COA", "COL",
    "CHP", "CHH", "DIF", "DUR", "GUA", "GRO", "HID",
    "JAL", "MEX", "MIC", "MOR", "NAY", "NLE", "OAX",
    "PUE", "QUE", "ROO", "SLP", "SIN", "SON", "TAB",
    "TAM", "TLA", "VER", "YUC", "ZAC")
colnames(ife_limpia) <- c(
    "clv_ent_fed",
    "ent_fed",
    "abr_ent_fed",
    "pan",
    "pri",
    "prd",
    "nulos",
```

[18] Esto implica que tendremos que hacer algunos supuestos. Por ejemplo, asumiremos que la diferencia entre las el número registrado de personas en el padrón y el total de votos emitidos son las personas decidieron abstenerse de votar. Claramente aquí hay defunciones, votos repetidos de manera ilegal y votos comprados, y deberían de ser tomados en cuenta en un análisis más minucioso, pero aquí no lo haremos.

```
    "abst")
```

Ahora calcularemos los votos porcentuales para cada partido, votos nulos y abstenciones. Debemos tener cuidado con las divisiones 0/0 porque introducen `NaN` al análisis y esto afecta cálculos posteriores. Por mismo asumiremos `0/0 = 0`. Además para calcular los votos para el PRI y el PRD tomaremos en cuenta los votos que provienen por alianzas también.

```
for (i in 1:32) {
    total <- as.matrix(ife[ife$nombre_estado == entidades[i], ↩
       ][21])

    # %PAN
    pan <- as.matrix(ife[ife$nombre_estado == entidades[i], ][6])
    p_pan <- pan/total
    # Remover NaN's por falta de votos (0/0)
    p_pan[is.nan(p_pan) | is.na(p_pan)] <- 0
    ife_limpia[i, 4] <- mean(p_pan)

    # %PRI con alianza (PVEM)
    pri <- as.matrix(ife[ife$nombre_estado == entidades[i], ][7])
    pri_pvem <- as.matrix(ife[ife$nombre_estado == entidades[i], ↩
       ][13])
    p_pri <- (pri + pri_pvem) / total
    # Remover NaN's por falta de votos (0/0)
    p_pri[is.nan(p_pri) | is.na(p_pri)] <- 0
    ife_limpia[i, 5] <- mean(p_pri)

    # %PRD con alianzas (PT, MC, PT-MC)
    prd <- as.matrix(ife[ife$nombre_estado == entidades[i], ][8])
    prd_pt_mc <- as.matrix(ife[ife$nombre_estado == entidades[i], ↩
       ][14])
    prd_pt <- as.matrix(ife[ife$nombre_estado == entidades[i], ↩
       ][15])
    prd_mc <- as.matrix(ife[ife$nombre_estado == entidades[i], ↩
       ][16])
    p_prd <- (prd + prd_pt_mc + prd_pt + prd_mc) / total
    # Remover NaN's por falta de votos (0/0)
    p_prd[is.nan(p_prd) | is.na(p_prd)] <- 0
    ife_limpia[i, 6] <- mean(p_prd)

    # %Nulos = mean([nul_j/tot_j])
    nulos <- as.matrix(ife[ife$nombre_estado == entidades[i], ↩
       ][19])
    p_nulos <- nulos / total
    p_nulos[is.nan(p_nulos) | is.na(p_nulos)] <- 0
```

```
    ife_limpia[i, 7] <- mean(p_nulos)

    #  %Abstencionismo =
    #       mean([(lista_nominal_j - tot_j) / lista_nominal_j])
    # Nota: cuidado con los 0's en lista_nominal
    total_slnz <- as.matrix(ife[
        ife$nombre_estado == entidades[i] &
        ife$lista_nominal != 0, ][21])
    lista_nominal <- as.matrix(ife[
        ife$nombre_estado == entidades[i] &
        ife$lista_nominal != 0, ][22])
    abst <- lista_nominal - total_slnz
    p_abst <- abst/lista_nominal
    p_abst[is.nan(p_abst) | is.na(p_abst)] <- 0
    ife_limpia[i, 8] <- mean(p_abst)
}
```

Ahora fusionamos ambas BDs y las guardamos `BDs.csv`:

```
datos <- merge(ife_limpia, conapo_limpia,
    by = "clv_ent_fed",
    sort = FALSE)
write.table(as.matrix(datos), "BDs.csv",
    row.names = FALSE,
    quote = FALSE,
    sep = ",")
```

3.2. Análisis de Componentes Principales

El análisis de componentes principales (ACP) crea una serie de nuevos ejes ortogonales entre sí que contienen la misma información que el conjunto original pero que exhiben la mayor varianza posible en los datos. Esto sirve para poder encontrar las relaciones implícitas en los datos y encontrar dimensiones que representan un porcentaje alto de la información. En ocasiones encontramos que con uno, dos o tres ejes es suficiente para explicar cerca del 80 % de la información (mejor dicho varianza, y depende de qué tan correlacionadas están las variables). Esto nos permite realizar análisis gráfico.

3.2.1. Marginación

Primero cargamos los paquetes necesario para graficar y tener disponibles algunas funciones que facilitan el análisis.

```
require(ggplot2)
require(GGally)
require(rgl)
```

Ahora importamos los datos que utilizaremos en el análisis (son los datos que creamos en la sección anterior). Escogemos las variables para en análisis que son las variables que miden el nivel de marginación, escogemos el radio del círculo que utilizaremos para mostrar las correlaciones y extraemos las abreviaturas de los nombre de los estados.

```
bds <- read.csv("BDs.csv", header = TRUE)
r <- 2                  # Radio del círculo
vars <- c(9:17)         # Variables de marginación
comps <- c(1:9)         # Número de componentes
datos <- bds[, vars]    # Datos que usaremos
abrv <- as.character(bds[, 3])  # Abreviaturas
meds <- as.character(colnames(bds[, c(vars)]))  # Mediciones
```

Un primer paso es analizar las correlaciones bivariadas, y observamos que las correlaciones entre varias variables son positivas (algunas de manera significativa), con lo que podemos intuir que el ACP nos permitirá una reducción de dimensinoes significativa.

```
ggpairs(datos, alpha = 0.5)
```

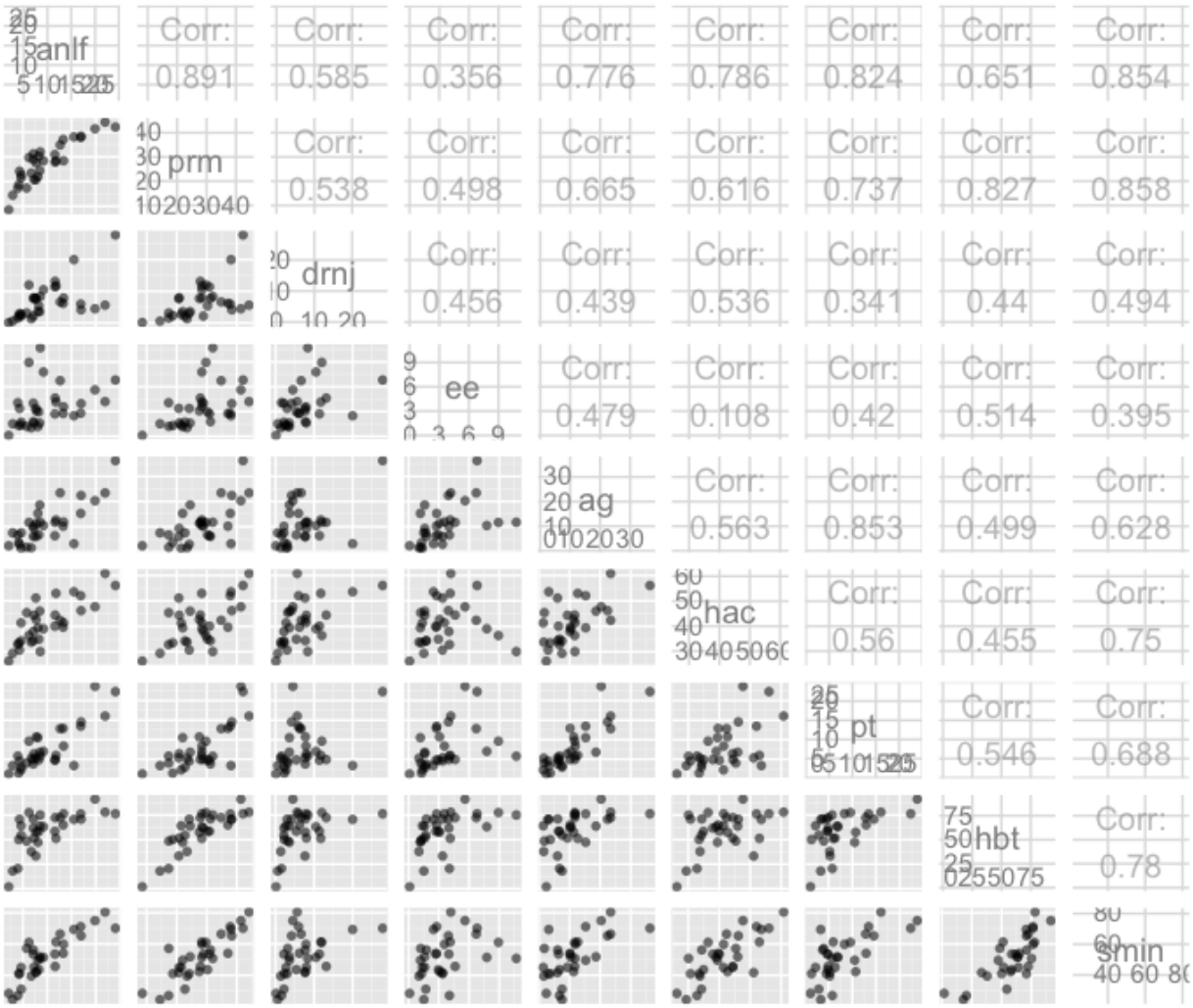

Figura 4: Correlaciones bivariadas

Ahora sí podemos llevar a cabo nuestro ACP, y extraer las componentes, las correlaciones, y las desviaciones estándar. Estamos multiplicando las correlaciones por `r` modificando el tamaño de los vectores a la hora de gráficar para facilitar la lectura de la gráfica. Se anima al lector a cambiar el valor de `r` y observar el efecto en las gráficas.

```
pca <- prcomp(datos, center = TRUE, scale. = TRUE)
componentes <- pca$x
correlaciones <- cor(datos, componentes) * r
dsvstd <- data.frame(cbind(comps, pca$sdev))
```

Fácilmente se puede crear una gráfica de barras que muestre la desviación estándar de cada componente:

```r
# Gráfica fácil
plot(pca)
```

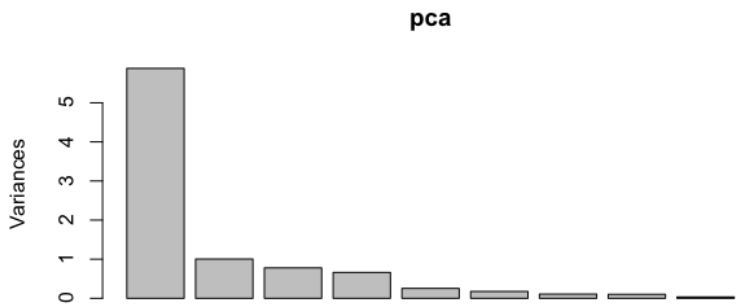

Figura 5: Desviación estándar por componente

Sin embargo, se puede crear una mucho mejor gráfica de las desviaciones estándar de cada componenente principal.

```r
ggplot() +
    geom_hline(
        yintercept = 1,
        color = "gray") +
    geom_path(
        data = dsvstd,
        aes(x = comps, y = V2),
        color = "dodgerblue3") +
    geom_point(
        data = dsvstd,
        aes(x = comps, y = V2),
        size = 4,
        color = "dodgerblue3") +
    labs(
        x = "",
        y = "Desviacion Estandar") +
    scale_x_continuous(
        breaks = comps,
        labels = colnames(componentes))
```

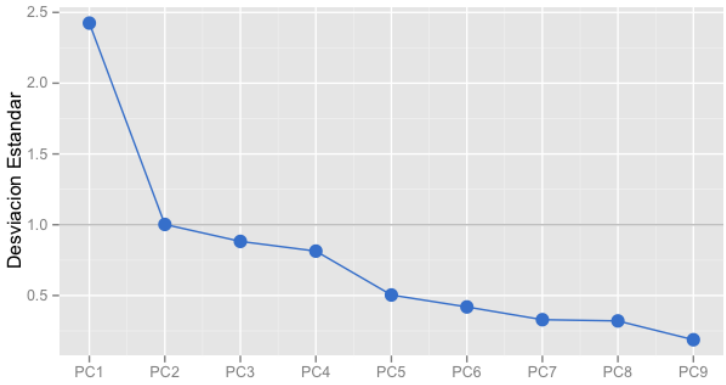

Figura 6: Desviación estándar por componente

También se puede crear fácilmente una gráfica del plano generado por las primeras dos componentes con las proyecciones de las variables y los estados.

```
1 # Gráfica fácil
2 biplot(pca)
```

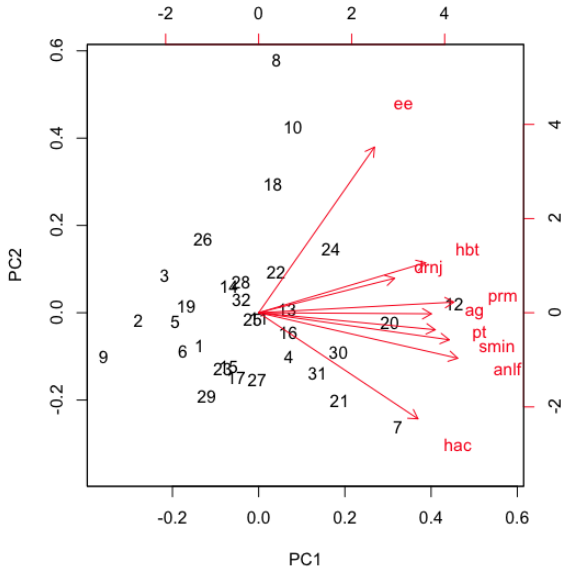

Figura 7: Biplot de estados y variables

Sin embargo, nuevamente podemos generar mucho mejores gráfcias. Para esto debemos crear la matriz de vectores que utilizaremos para graficar las correlaciones en los círculos de correlaciones y crear el círculo mismo que utilizaremos en la gráfica.

```r
# Coordenadas de correlaciones
k <- 1
corrs <- replicate(3, matrix(0, 9, 4), simplify = FALSE)
for (i in 1:2) {
    for (j in (i + 1):3) {
        corrs[[k]] <- data.frame(x = rep(0, length(comps)),
                                 y = rep(0, length(comps)),
                                 xend = correlaciones[, i],
                                 yend = correlaciones[, j])
        k <- k + 1
    }
}

# Círculo de correlaciones
circulo <- function(center = c(0, 0), npoints = 100) {
    t <- seq(0, 2 * pi, length = npoints)
    x <- center[1] + r * cos(t)
    y <- center[1] + r * sin(t)
    return(data.frame(x = x, y = y))
}
corcir <- circulo(c(0, 0), npoints = 100)
```

Ahora graficamos los estados sobre el plano generado por las componentes principales uno y dos junto con las variables.

```r
# Gráfica: CP1 vs CP2
eje_x <- paste("PC1 (",
    round(pca$sdev[1] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "", collapse = "")
eje_y <- paste("PC2 (",
    round(pca$sdev[2] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "", collapse = "")
ggplot() +
    geom_hline(
        yintercept = 0,
        color = "gray") +
    geom_vline(
        xintercept = 0,
        color = "gray") +
    geom_path(
        data = corcir,
        aes(x = x, y = y),
        color = "gray") +
    geom_segment(aes(
        x = corrs[[1]][1],
```

```
        y = corrs[[1]][2],
        xend = corrs[[1]][3],
        yend = corrs[[1]][4]), color = "gray") +
    geom_point(aes(
        x = corrs[[1]][[3]],
        y = corrs[[1]][[4]]),
        color = "gray") +
    geom_text(size = 3, aes(
        x = correlaciones[, 1] + 0.2,
        y = correlaciones[, 2],
        label = meds),
        color = "dodgerblue3") +
    geom_text(size = 3, aes(
        x = componentes[, 1],
        y = componentes[, 2],
        label = abrv),
        color = "orangered3") +
    labs(
        x = eje_x,
        y = eje_y) +
    theme(
        axis.title = element_text(size = 10)) +
    coord_equal()
```

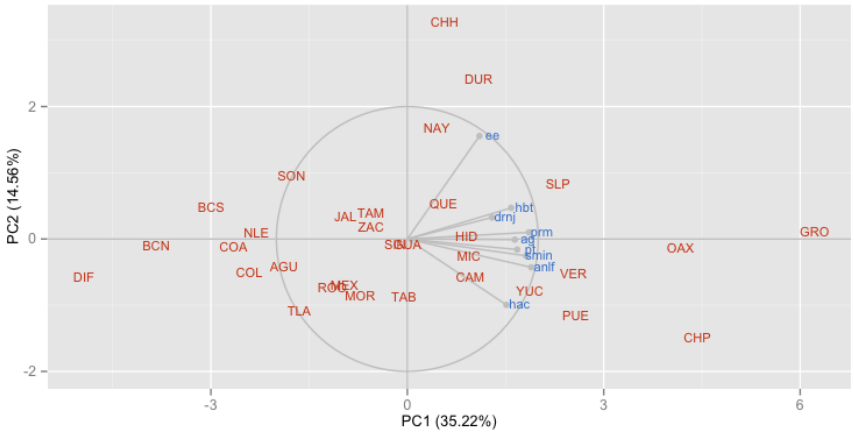

Figura 8: Biplot de estados y variables (CP1 y CP2)

Lo mismo hacemos para las componentes uno y tres:

```
eje_x <- paste("PC1 (",
    round(pca$sdev[1] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "", collapse = "")
```

```
eje_y <- paste("PC3 (",
    round(pca$sdev[3] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "", collapse = "")
ggplot() +
    geom_hline(
        yintercept = 0,
        color = "gray") +
    geom_vline(
        xintercept = 0,
        color = "gray") +
    geom_path(
        data = corcir,
        aes(x = x, y = y),
        color = "gray") +
    geom_segment(aes(
        x = corrs[[2]][1],
        y = corrs[[2]][2],
        xend = corrs[[2]][3],
        yend = corrs[[2]][4]), color = "gray") +
    geom_point(aes(
        x = corrs[[2]][[3]],
        y = corrs[[2]][[4]]),
        color = "gray") +
    geom_text(size = 3, aes(
        x = correlaciones[, 1] + 0.2,
        y = correlaciones[, 3],
        label = meds),
        color = "dodgerblue3") +
    geom_text(size = 3, aes(
        x = componentes[, 1],
        y = componentes[, 3],
        label = abrv),
        color = "orangered3") +
    labs(
        x = eje_x,
        y = eje_y) +
    theme(
        axis.title = element_text(size = 10)) +
    coord_equal()
```

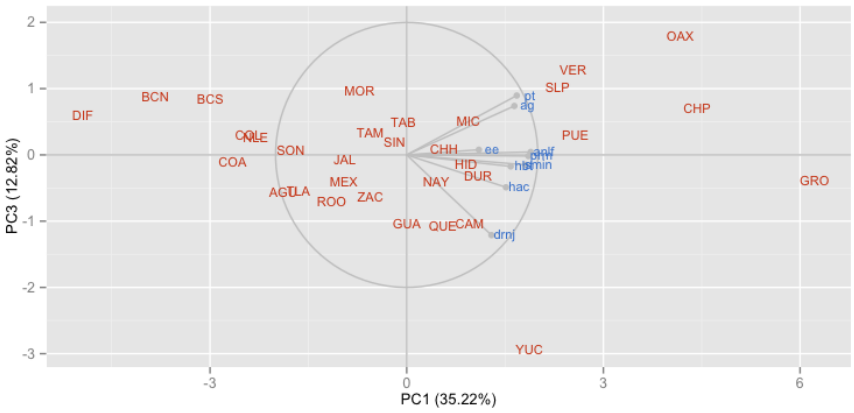

Figura 9: Biplot de estados y variables (CP1 y CP3)

Por último, el mismo procedimiento pero para las componentes dos y tres se deja como ejercicio al lector, solamente mostramos la gráfica.

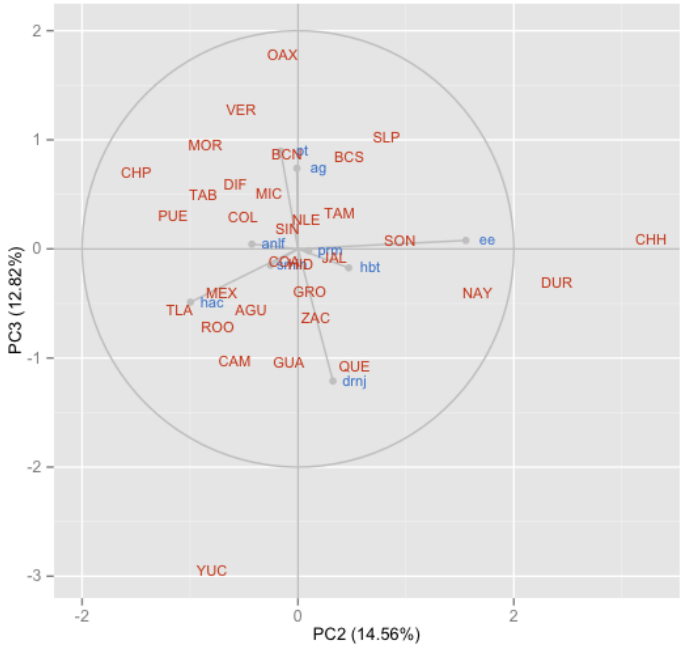

Figura 10: Biplot de estados y variables (CP2 y CP3)

Ahora preparamos la matriz que nos permitirá graficar los vectores de las correlaciones.

```
corrs3D <- as.matrix(corrs3D)
c3D <- matrix(data = NA, nrow = 2 * length(vars), ncol = 3)
for (i in 1:length(vars)) {
```

```
    c3D[2 * i - 1, 1:3] <- corrs3D[i, 1:3]
    c3D[2 * i, 1:3] <- corrs3D[i, 4:6]
}
```

Con lo siguiente creamos el texto que será colocado en el eje de cada componente explicando la cantidad de varianza explicada por esa dimensión y graficamos la caja que contiene a los estados y sus proyecciones sobre las componentes uno, dos y tres, junto con la esfera de correlaciones.

```
eje_x <- paste("PC1 (",
    round(pca$sdev[1] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "",
    collapse = "")
eje_y <- paste("PC2 (",
    round(pca$sdev[2] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "",
    collapse = "")
eje_z <- paste("PC3 (",
    round(pca$sdev[3] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "",
    collapse = "")

plot3d(
    componentes[, 1],
    componentes[, 2],
    componentes[, 3],
    xlab = eje_x,
    ylab = eje_y,
    zlab = eje_z,
    box = TRUE,
    type = 'n') +
# Estados
text3d(
    componentes[, 1],
    componentes[, 2],
    componentes[, 3],
    abrv,
    size = 3,
    color = "orangered3") +
# Variables
segments3d(
    c3D,
    color = "gray") +
text3d(
    correlaciones[, 1:3],
```

```
37      size = 3,
38      color = "dodgerblue3",
39      texts = rownames(correlaciones))
```

A continuación mostramos la gráfica generada desde diferentes ángulos. Cada ángulo mostrado corresponde a las proyecciones hechas sobre los planos de las combinaciones de las diferentes componentes. El lector debe notar las similitudes de estas gráficas con las mostradas anteriormente (en dos dimensiones).

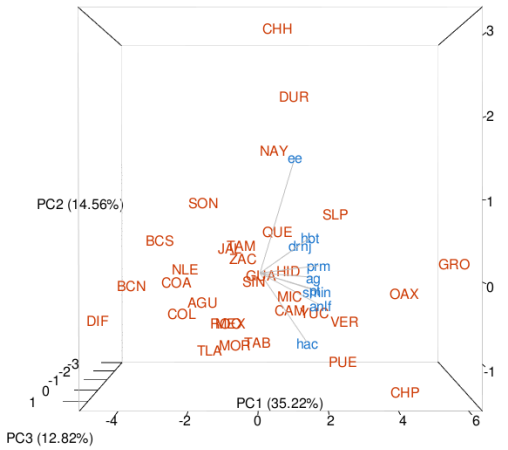

Figura 11: Biplot 3D (CP1 y CP2)

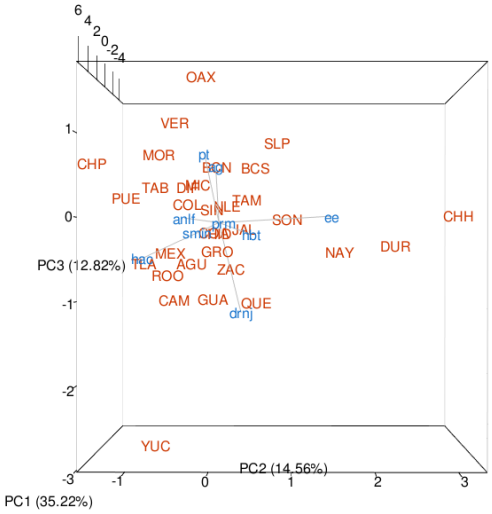

Figura 12: Biplot 3D (CP2 y CP3)

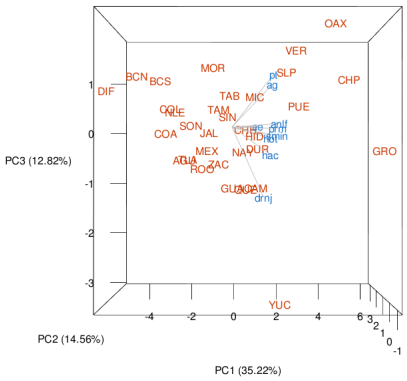

Figura 13: Biplot de 3D (CP1 y CP3)

3.2.2. Elecciones

Para realizar el análisis de componentes principales respecto a las variables de las elecciones federales el procedimiento es similar, pero en lugar de elegir las variables de marginación se deben elegir las de las elecciones y se debe ajustar el número de componentes que vamos a utilizar:

```
# vars <- c(9:17)        # Variables de marginación
# comps <- c(1:9)        # Número de componentes
vars <- c(4:8)           # Variables de marginacion
comps <- c(1:5)          # Numero de componentes
```

Lo demás es casi idéntico, pero al final se hacen algunos ajustes de en las gráficas. Se deja al lector esto como ejercicio, y sólamente se presentan los resultados[19].

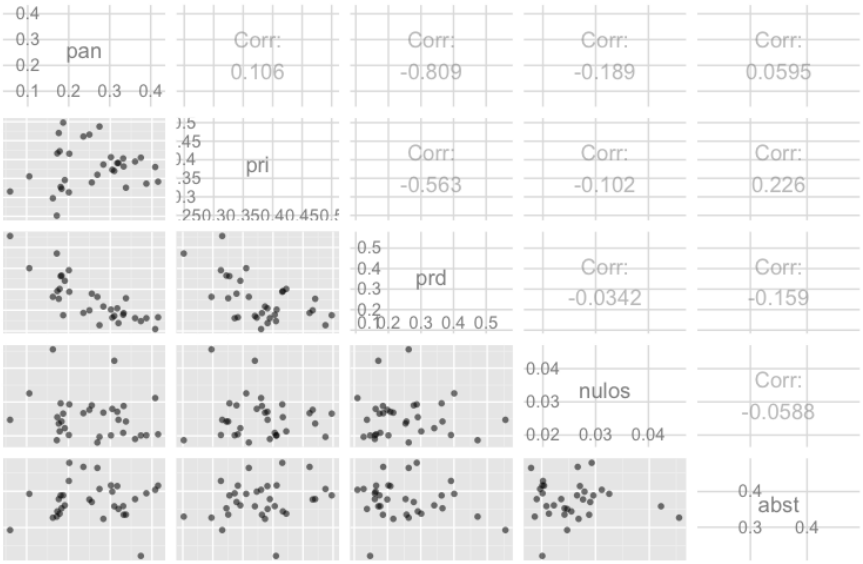

Figura 14: Correlaciones bivariadas

[19]La gráfica que representa tres dimensiones al ser ejecutada en el sistema, igual que la anterior, se puede girar con el cursor.

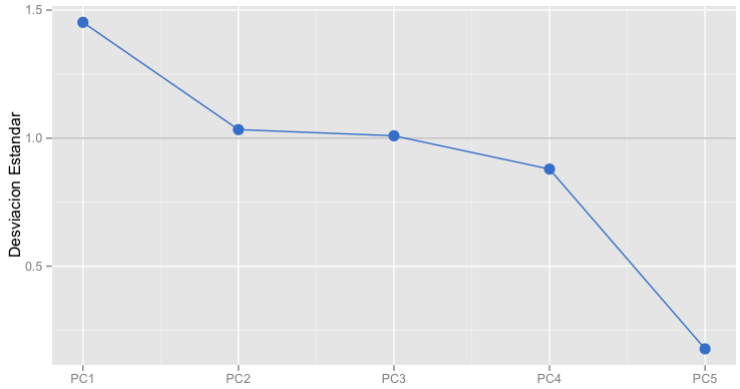

Figura 15: Desviación estándar por componente

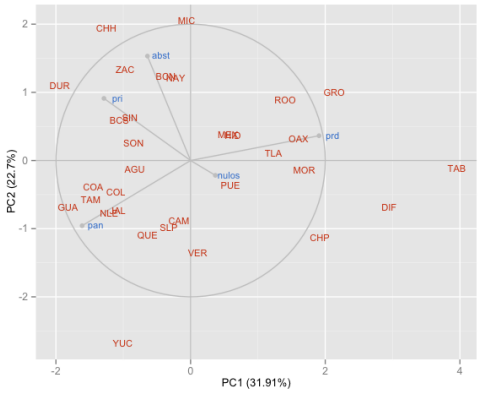

Figura 16: Biplot (CP1 y CP2)

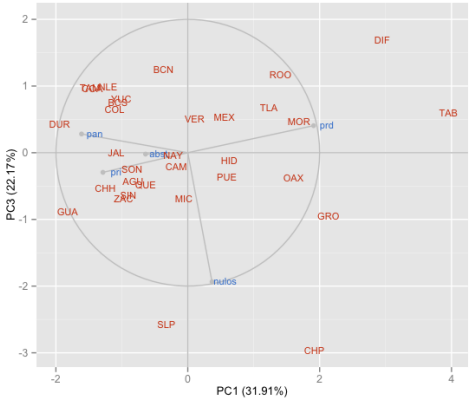

Figura 17: Biplot (CP1 y CP3)

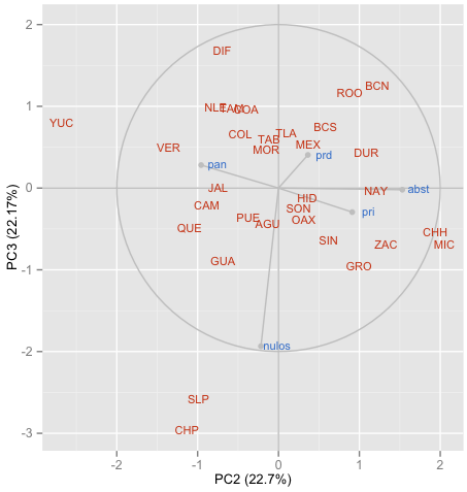

Figura 18: Biplot (CP2 y CP3)

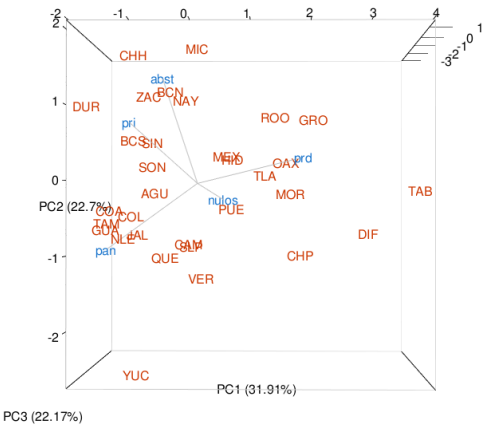

Figura 19: Biplot 3D (CP1 y CP2)

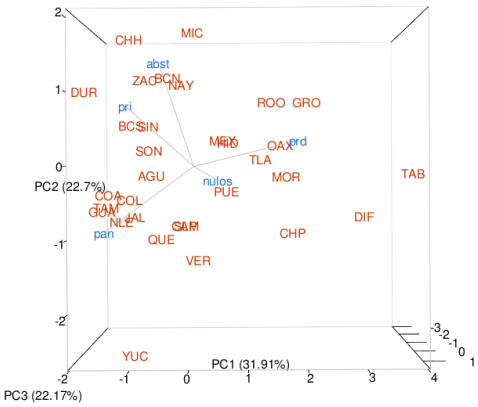

Figura 20: Biplot 3D (CP2 y CP3)

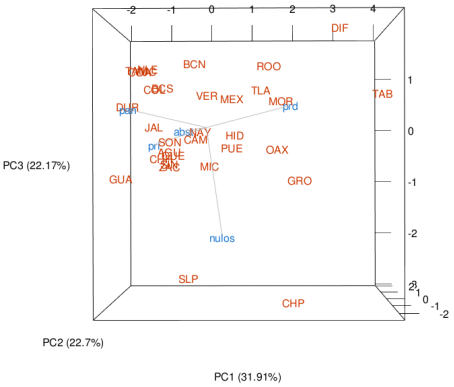

Figura 21: Biplot de 3D (CP1 y CP3)

3.2.2.1. Clustering jerárquico

El análisis de componentes principales no sólo sirve para llevar a cabo análisis exploratorio de datos. Esta técnica es muy poderosa y también puede ser utilizada para abordar problemas de predicción y clasificación —en este caso la usaremos con *clustering jerárquico*[20]. Es un algoritmo de clasificación no-supervisado y el objetivo es agrupar a los individuos respecto a un conjunto de características que los distingan a unos de otros y que dan una idea del grado de *similaridad* o *disimilaridad* que existe entre los individuos.

Lo primero que debemos hacer es calcular la distancia entre cada individuo (estados en este caso). Una vez hecho lo anterior formamos el dendrograma y lo graficamos tomando en

[20]Existen otros tipo de *clustering*, por ejemplo *K-Means*. Aunque no se mostrará en este documeto cómo realizar éste último, al lector no se le debe dificultar demasiado su implementación después de entender lo expuesto en este documento.

cuenta tres grupos en la separación[21].

```
dist <- dist(datos)
hclust <- hclust(dist, method = "complete")
plot(hclust, labels = abrv, hang = -1,
    xlab = "Entidades Federativas",
    ylab = "Distancias",
    main = "Dendrograma")
rect.hclust(hclust, 3)
```

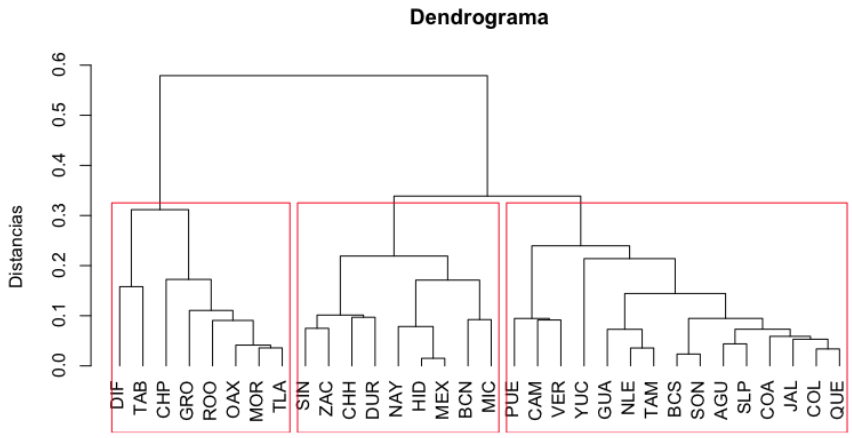

Figura 22: Dendrograma con tres grupos

Ahora creamos la misma gráfica en tres dimensiones que habíamos creado anteriormente, pero coloreando cada estado respecto al grupo al que pertenencen de acuerdo al dendrograma[22].

```
# Colores que usaremos para los grupos y
# van de acuerdo al color representativo
# de cada partido.
colores <- groups.3 <- cutree(hclust, 3)
colores[colores == 1] <- "forestgreen"
colores[colores == 2] <- "orangered3"
colores[colores == 3] <- "dodgerblue3"

# Nombres de los ejes para incluir
```

[21]En este ejemplo utilizamos *complete linkage* que es otro nombre para la métrica del *vecino más lejano*. El lector puede modificar el ejemplo escogiendo un número diferente de grupos para el corte o cambiando la métrica utilizada.

[22]Debemos recordar al lector que al ejecutar este ejemplo en su computadora podrá mover el cubo de la gráfica para observarla desde diferentes ángulos y a distintas distancias.

```r
# el porcentage de varianza explicada
eje_x <- paste("PC1 (",
    round(pca$sdev[1] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "",
    collapse = "")
eje_y <- paste("PC2 (",
    round(pca$sdev[2] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "",
    collapse = "")
eje_z <- paste("PC3 (",
    round(pca$sdev[3] / sum(pca$sdev) * 100, digits = 2), "%)",
    sep = "",
    collapse = "")
# Gráfica 3D
plot3d(
    componentes[, 1],
    componentes[, 2],
    componentes[, 3],
    xlab = eje_x,
    ylab = eje_y,
    zlab = eje_z,
    box = TRUE,
    type = 'n') +
# Estados
text3d(
    componentes[, 1],
    componentes[, 2],
    componentes[, 3],
    abrv,
    size = 3,
    color = colores) +
# Variables
segments3d(
    c3D,
    color = "gray") +
text3d(
    correlaciones[, 1:3],
    size = 3,
    color = "mediumslateblue",
    texts = rownames(correlaciones))
```

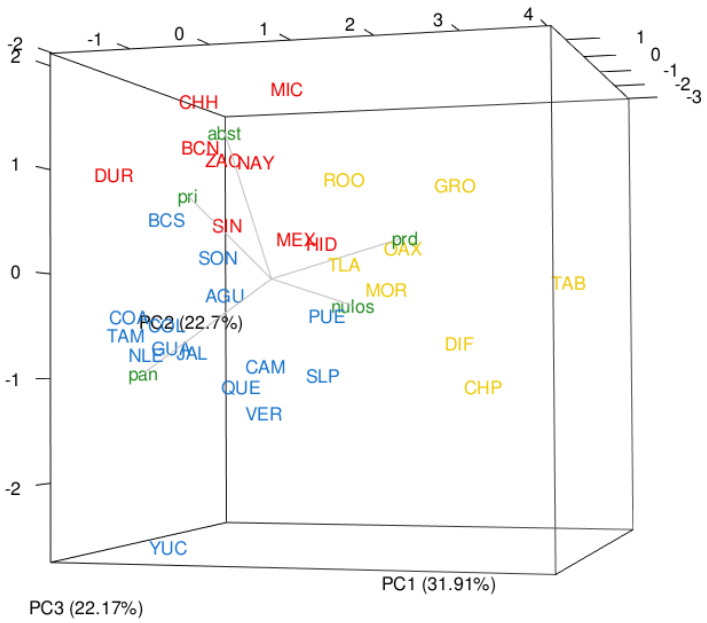

Figura 23: Biplot 3D con clusters

3.3. Análisis Canónico de Correlaciones

Esté método surge de la necesidad de encontrar la relación entre dos grupos de variables medidas en un mismo grupo de *individuos*. Buscamos saber qué tan bien se explican los grupos de variables entre sí. Debemos tener cuidado porque estamos proyectando tres espacios diferentes, el del primer grupo de variables, el del segundo grupo de variables y el de los indiviudos[23]. Debemos mantener esto en mente a la hora de extraer conclusiones. Dicho lo anterior a continuación se presenta la implementación en R.

Primero cargamos los paquetes necesarios:

```
require(ggplot2)
require(GGally)
require(CCA)
```

Ahora cargamos los datos, elegimos las variables de elección como el primer grupo de variables y las variables de las elecciones como el segundo grupo de variables. Naturalmente el rol de los *individuos* será ocupado por los estados.

```
bds <- read.csv("BDs.csv", header = TRUE)
vars.elec <- c(4:8)      # Variables de elección
```

[23]Para profundizar en este punto el lector deberá revisar las notas del Prof. Rubén Hernández Cid de *Estadística Aplicada III*.

```
vars.marg <- c(9:17)     # Variables de marginación
abrv <- as.character(bds[, 3])  # Abreviaciones
r <- 2                   # Radio de círculo
alpha <- 0.7             # Nivel de transparencia
```

En primera instancia es bueno observar las correlaciones bivariadas para darnos una idea. Eso se puede lograr con los siguientes comandos, pero no se mostrarán las gráficas porque esencialmente son las mismas correlaciones bivariadas que mostramos en el ejemplo de componentes principales.

```
ggpairs(bds[, vars.elec], alpha = 0.5)
ggpairs(bds[, vars.marg], alpha = 0.5)
```

A continuación mostramos cómo se aplica el análisis y cómo se pueden observar las correlaciones y los factores. No mostraremos el output de la última línea por ser muy extenso.

```
cca <- cc(bds[, vars.elec], bds[, vars.marg])
cca$cor              # Correlacioens
cca[3:4]             # Coeficientes
cca$scores           # Factores
```

Fácilmente podemos graficar los resultados:

```
# Gráfica fácil
plt.cc(cca,
       int = 1,
       ind.names = abrv,
       var.label = TRUE)
```

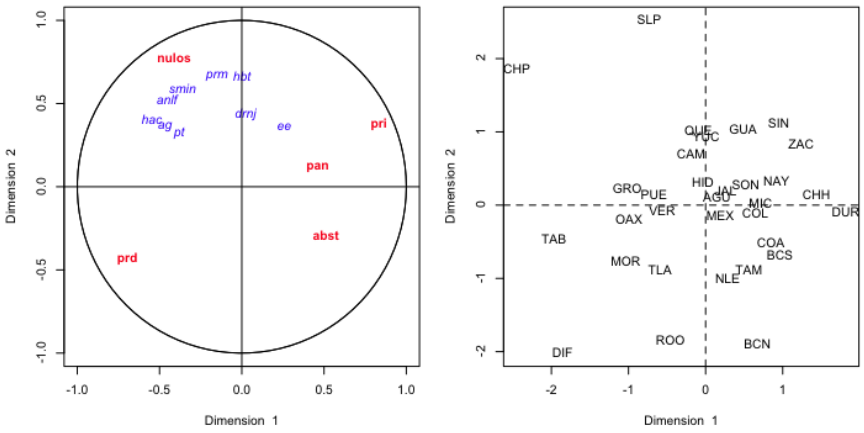

Figura 24: Variables (izquierda) e individuos (derecha)

Sin embargo, al igual que en los casos pasados, podemos mejorar significativamente esta gráfica. A continuación graficamos los tres espacios en una gráfica. Presentamos las variables de la elección de color azul, las variables de marginación en color naranja y los estados de color verde.

Primero creamos el objeto que contendrá los puntos que usaremos para graficar el círculo.

```
# Círculo
circulo <- function(center = c(0, 0), npoints = 100) {
    t <- seq(0, 2 * pi, length = npoints)
    x <- center[1] + r * cos(t)
    y <- center[1] + r * sin(t)
    return(data.frame(x = x, y = y))
}
corcir <- circulo(c(0, 0), npoints = 100)
```

Ahora creamos los objetos que contienen las correlaciones que graficaremos. Originalmente $r = 1$, pero para modificar el aspecto de la gráfica, y dado que lo que nos importa de las variables no es la magnitud si no el ángulo en la gráfica, podemos multiplicar por un factor r para escalar la gráfica[24].

```
vars.uno <- cbind(
    matrix(0, nrow = length(cca$scores$corr.X.xscores[,1]), ncol = ↩
        2),
    cca$scores$corr.X.xscores[, 1:2])
vars.dos <- cbind(
    matrix(0, nrow = length(cca$scores$corr.Y.xscores[,1]), ncol = ↩
        2),
    cca$scores$corr.Y.xscores[, 1:2])
vars.uno <- vars.uno * r
vars.dos <- vars.dos * r
```

Ahora sí podemos producir la gráfica. Primero se crean los ejes que representan el cero para fácil visualización y colocamos el objeto del círculo. Después agregamos las variables de la elección, junto con los segmentos que muestras los ángulos entre las mismas, añadimos un punto para mejorar la estética y colocamos el texto que identifica a qué variable pertenece ese vector. Hacemos lo mismo para las variables de marginación y finalmente colocamos las etiquetas de los estados y los nombres de los ejes.

```
# Gráfica de variables (dims 1 y 2)
ggplot() +
    geom_hline(
        yintercept = 0,
```

[24]Si $r < 1$ reducimos el tamaño, mientras que si $r > 1$ ampliamos el tamaño del círculo y los vectores que representan las correlaciones.

```
        color = "gray") +
    geom_vline(
        xintercept = 0,
        color = "gray") +
    geom_path(
        data = corcir,
        aes(x = x, y = y),
        color = "gray") +
    # Variables de elecciones
    geom_segment(aes(
        x = vars.uno[, 1],
        y = vars.uno[, 2],
        xend = vars.uno[, 3],
        yend = vars.uno[, 4]),
        color = "dodgerblue3",
        alpha = alpha) +
    geom_point(aes(
        x = vars.uno[, 3],
        y = vars.uno[, 4]),
        color = "dodgerblue3",
        alpha = alpha) +
    geom_text(size = 4, aes(
        x = cca$scores$corr.X.xscores[, 1] * r + 0.05,
        y = cca$scores$corr.X.xscores[, 2] * r + 0.05,
        label = rownames(cca$scores$corr.X.xscores)),
        color = "dodgerblue3",
        alpha = alpha) +
    # Variables de marginación
    geom_segment(aes(
        x = vars.dos[, 1],
        y = vars.dos[, 2],
        xend = vars.dos[, 3],
        yend = vars.dos[, 4]),
        color = "orangered3",
        alpha = alpha) +
    geom_point(aes(
        x = vars.dos[, 3],
        y = vars.dos[, 4]),
        color = "orangered3",
        alpha = alpha) +
    geom_text(size = 4, aes(
        x = cca$scores$corr.Y.xscores[, 1] * r + 0.05,
        y = cca$scores$corr.Y.xscores[, 2] * r + 0.05,
        label = rownames(cca$scores$corr.Y.xscores)),
        color = "orangered3",
```

```
        alpha = alpha) +
    # Estados
    geom_text(size = 4, aes(
        x = cca$scores$xscores[, 1],
        y = cca$scores$xscores[, 2],
        label = abrv),
        color = "forestgreen") +
    labs(
        x = "Dimension 1",
        y = "Dimension 2") +
    theme(
        axis.title = element_text(size = 10)) +
    coord_equal()
```

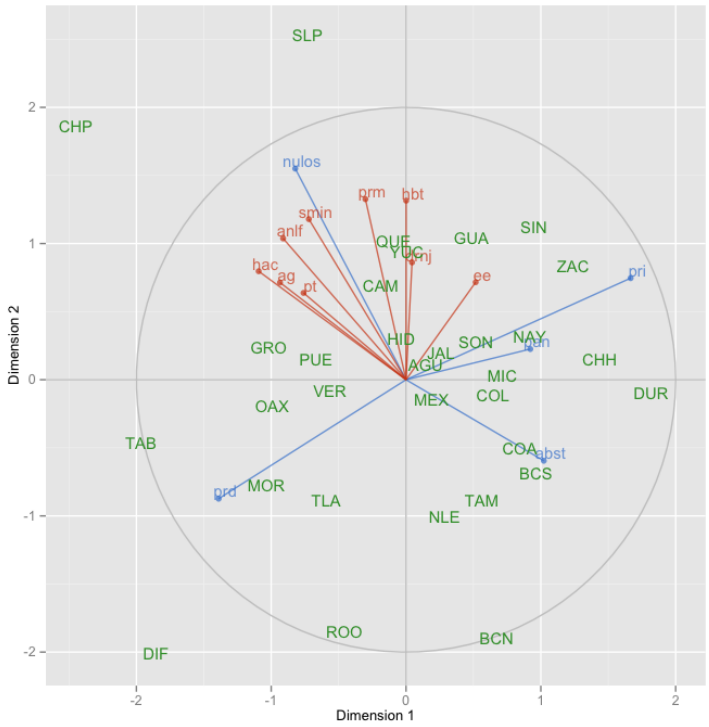

Figura 25: Gráfica de variables e individuos

En esta gráfica se puede apreciar las relaciones entre las variables de ambos conjuntos y los estados. Además, el lector debe notar que esta gráfica nos muestra una historia un poco diferente a las de los análisis de componentes principales que se explica por las relaciones entre ambos grupos de variables. Hay otras gráficas que se pueden crear cuando se realiza un análisis canónico de correlaciones, y las agregaremos en un futuro a este documento.

3.4. Análisis de Correspondencias Simple

En esta sección mostramos cómo llevar a cabo un análisis de correspondencias simple. Asumimos que el lector ha entendido los ejemplos pasados por lo que no explicaremos el código a detalle como lo hemos hecho hasta ahora —sólo explicaremos las ideas fundamentales.

Como se mencionó anteriormente, utilizamos la base de datos *Tea* provista por R.

```
require(ca)
require(MASS)
require(FactoMineR)

data(tea)
# Variables: tipo y edad
datos <- tea[, c("Tea", "age_Q")]
```

Creamos la tabla de contingencia asociada a los datos y la tabla condicional con márgenes que se deriva.

```
tab_cont <- table(datos)
```

Tabla 1: Tabla de contingencia

	age_Q				
Tea	15-24	25-34	35-44	44-59	+60
black	11	11	16	18	18
Earl Grey	77	48	21	34	13
green	4	10	3	9	7

Ahora creamos el histograma respecto a los porcentages para observar si cambian las proporciones entre las variables.

```
# Barplot condicional
barplot(t(prop.table(as.matrix(tab_cont), 1)),
        beside = TRUE,
        legend = TRUE)
```

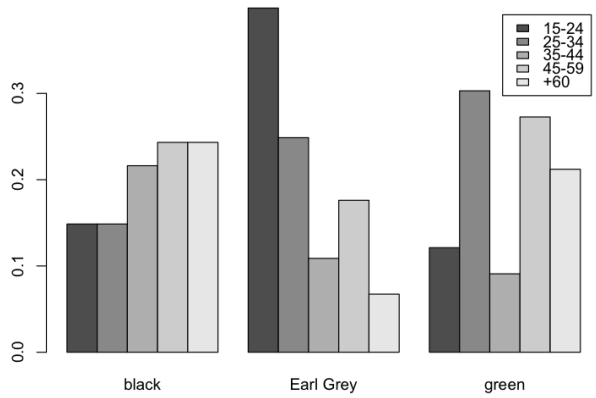

Figura 26: Análisis de correspondencias simple

Debemos realizar la prueba χ^2 de independencia para saber si podemos rechazar la hipótesis de independencia entre las variables.

```
chisq.test(tab_cont)
```

Podemos observar que los datos sugieren que sí existe una relación entre las variables porque el *valor-p* es $2.405e-06$, como se muestra en los resultados de la prueba que se muestran a continuación:

```
    Pearson's Chi-squared test

data:  tab_cont
X-squared = 40.6678, df = 8, p-value = 2.405e-06
```

Finalmente, llevamos a cabo el análisis de correspondencias simple. La gráfica que se muestra es creada automáticamente cuando se ejecuta el comando.

```
tab_cont.sca.CA <- CA(tab_cont,
                      ncp = 3,
                      row.sup = NULL,
                      col.sup = NULL,
                      axes = c(1, 2))
```

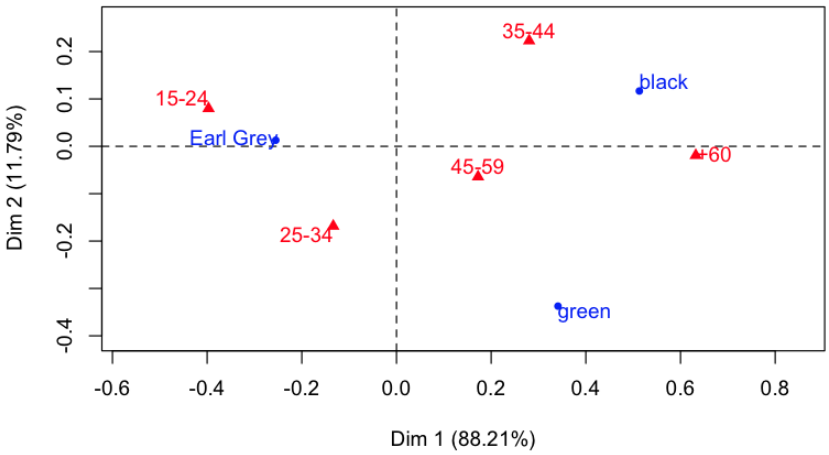

Figura 27: Análisis de correspondencias simple

3.5. Análisis de Correspondencias Múltiple

Al igual que la sección anterior, utilizamos la base de datos *Tea* provista por R para mostrar el análisis de correspondencias múltiple. Esta técnica es una generalización de la técnica de análisis de correspondencias simple para poder utilizar varias variables en el análisis. En este caso utilizamos cuatro variables.

```
require(FactoMineR)
require(ggplot2)

data(tea)
data <- tea[, c("Tea", "how", "where", "age_Q")]
head(data)
```

Obtenemos los nombres de las variables y el número de categorías en cada variable, y ejecutamos el análisis. Después se muestra cómo obtener los valores propios y las proyecciones tanto de las variables como de las observaciones.

```
categories <- apply(data, 2, function(x) nlevels(as.factor(x)))
categories

# MCA
data.mca <- MCA(data, graph = FALSE)
data.mca

# Valores propios
data.mca$eig

```

```
# Coordenadas de variables
data.mca.vars <- data.frame(data.mca$var$coord,
                        Variable = rep(names(categories), categories↩
                            ))

# Coordenadas de observaciones
data.mca.obs <- data.frame(data.mca$ind$coord)
```

Si se desea gráficar los resultados de forma fácil tenemos:

```
# Gráfica fácil
plot(data.mca)
```

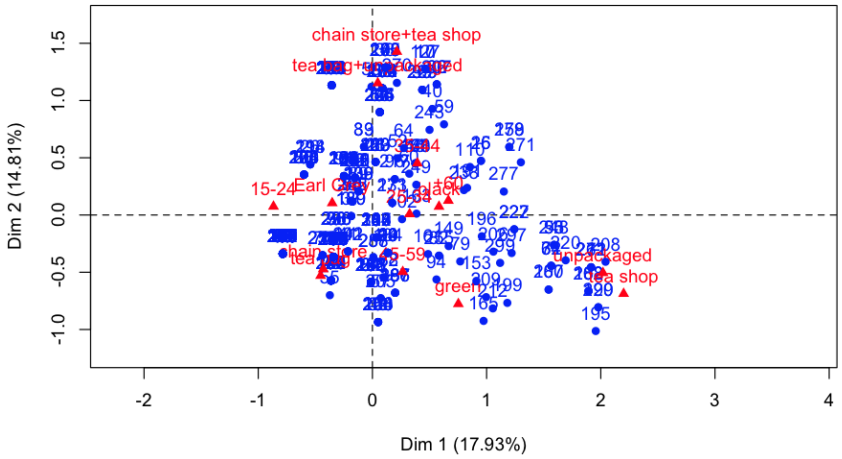

Figura 28: Análisis de correspondencias múltiple

Sin embargo, como en los casos anteriores, podemos mejor sustancialmente la gráfica. Ahora se muestran las variables, junto con las observaciones y un *contour* que nos indica una aproximación a la densidad que exhiben los datos.

```
# Gráfica de variables y observaciones
ggplot(data = data.mca.obs, aes(x = Dim.1, y = Dim.2)) +
    geom_hline(yintercept = 0, colour = "gray70") +
    geom_vline(xintercept = 0, colour = "gray70") +
    geom_point(colour = "gray50", alpha = 0.5) +
    geom_density2d(colour = "gray80") +
    geom_text(data = data.mca.vars,
              aes(x = Dim.1, y = Dim.2,
                  label = rownames(data.mca.vars), colour = ↩
                     Variable)) +
    scale_colour_discrete(name = "Variable") +
```

```
    xlab("") +
    ylab("")
```

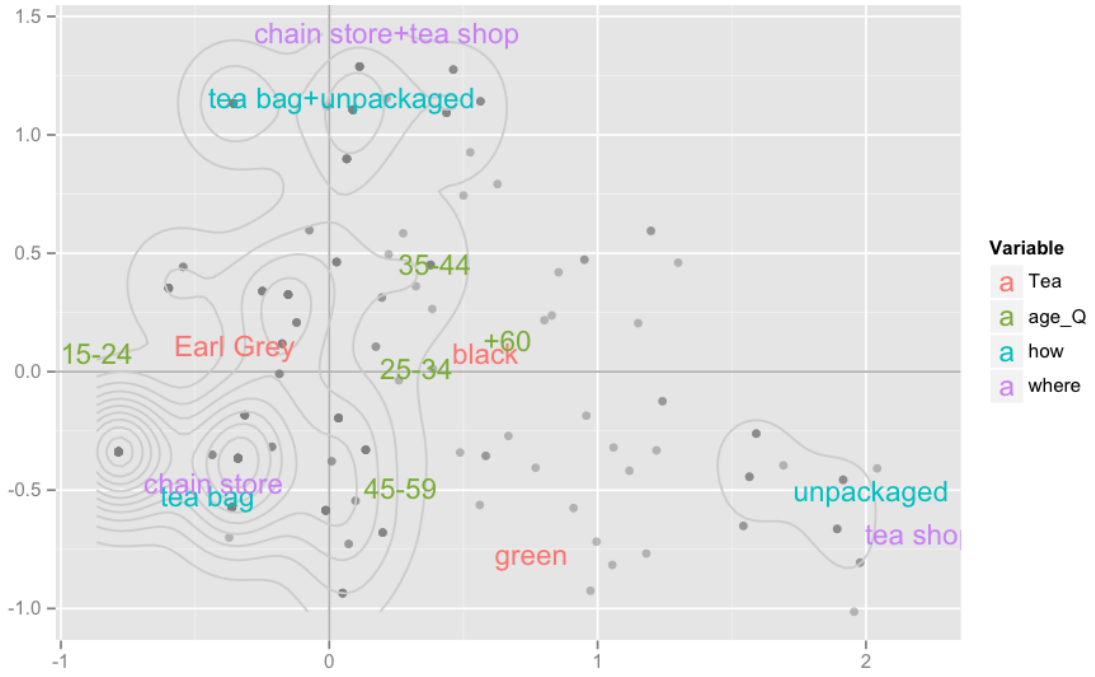

Figura 29: Análisis de correspondencias múltiple

3.6. Discriminación Lineal

En el análisis de discriminación lineal buscamos una combinación lineal de los datos que nos permita minimizar la varianza dentro (intra) de los grupos y maximizar la varianza entre (inter) los grupos para poder clasificar nuevas observaciones. Es un método de clasficación supervisado (porque se conocen de antemano las características que acompañana a los individuos de los grupos). A continuación mostramos cómo se ejecuta este análisis utilizando la base de datos *Iris* incluída en R.

```
library(MASS)
attach(iris)
datos <- iris
```

Lo primero que hacemos es un *scatterplot* para ver qué tan bien separados se encuentran los grupos. Como podemos ver, están bien separados por lo que podemos esperar muy buenos resultados.

```
panel.pearson <- function(x, y, ...) {
```

```
    horizontal <- (par("usr")[1] + par("usr")[2]) / 2
    vertical <- (par("usr")[3] + par("usr")[4]) / 2
    text(horizontal,
         vertical,
         format(abs(cor(x, y)), digits = 2),
         cex = 1.7)
}
pairs(datos[1:4],
      pch = 21,
      bg = c("red","green3","blue")[unclass(datos$Species)],
      upper.panel = panel.pearson)
```

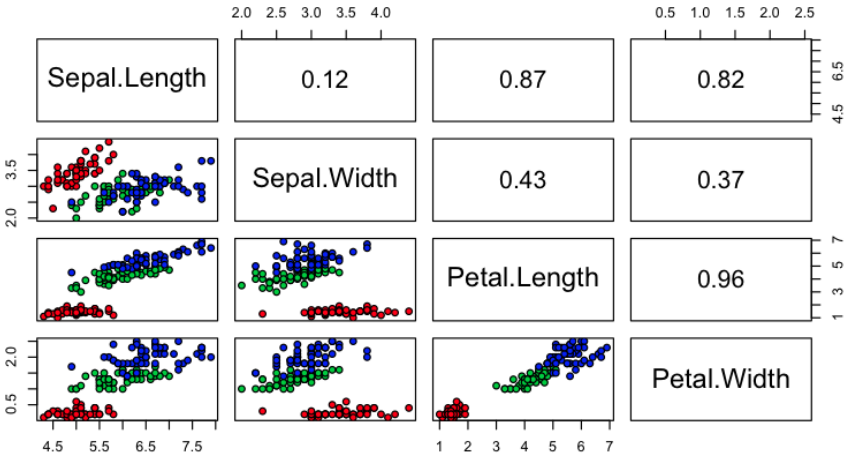

Figura 30: Observaciones agrupadas y correlaciones

Ahora implementamos el análisis.

```
datos.lda <- lda(Species ~ .,
    data = datos,
    na.action = "na.omit")
```

Para poder ver los resultados ejecutamos lo siguiente.

```
datos.lda
```

Los resultados obtenidos son:

```
Call:
lda(Species ~ ., data = datos, na.action = "na.omit")

Prior probabilities of groups:
```

```
    setosa versicolor  virginica
 0.3333333  0.3333333  0.3333333

Group means:
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026

Coefficients of linear discriminants:
                    LD1         LD2
Sepal.Length  0.8293776  0.02410215
Sepal.Width   1.5344731  2.16452123
Petal.Length -2.2012117 -0.93192121
Petal.Width  -2.8104603  2.83918785

Proportion of trace:
   LD1    LD2
0.9912 0.0088
```

Para evaluar qué tan bien se realiza la clasificación con este método utilizamos lo siguiente[25]

```r
# Resultados de clasificación
ct <- table(datos$Species, datos.lda$class)
ct
prop.table(ct, 1)

# Porcentage de clasificaciones correctas
sum(diag(prop.table(ct)))
```

Con esto observamos que la clasificación se realiza bastante bien.

Tabla 2: Tabla de resultados de clasificación

	setosa	versicolor	virginica
setosa	50 (100 %)	0 (0 %)	0 (0 %)
versicolor	0 (0 %)	48 (96 %)	2 (4 %)
virginica	0 (0 %)	1 (2 %)	49 (98 %)

Fácilmente podemos graficar las proyecciones utilizando las dos funciones discriminatorias.

[25] En un análisis serio, debemos crear dos grupos: uno con el que entrenamos el algoritmo y otro con el que probamos los resultados. Esto no se hizo aquí para mantener el análisis simple, pero ciertamente se debería de realizar.

```r
# Gráfica fácil
plot(datos.lda)
```

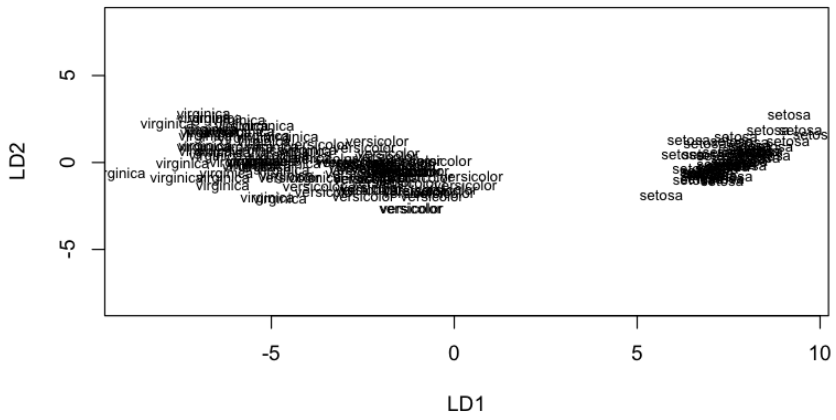

Figura 31: Análisis de discriminación lineal

Como es costumbre, podemos mejorar sustancialmente esta gráfica con lo siguiente:

```r
# Scatterplot con las primeras dos
# fuciones discriminantes
proyecciones <- cbind(scale(as.matrix(iris[, -5]), scale = FALSE)
    * % datos.lda$scaling, iris[, 5, drop = FALSE])
ggplot(data = proyecciones,
     aes(x = LD1, y = LD2, col = Species)) + geom_point()
```

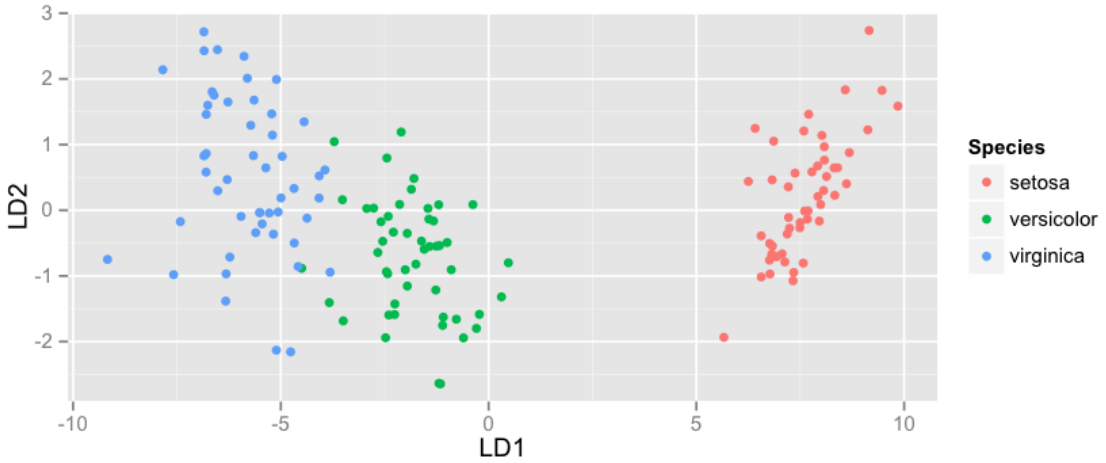

Figura 32: Análisis de discriminación lineal

Por último, podemos graficar la distribución de los datos a lo largo del eje generado por la primer función discriminante.

```
plot(datos.lda, dimen = 1, type = "both")
```

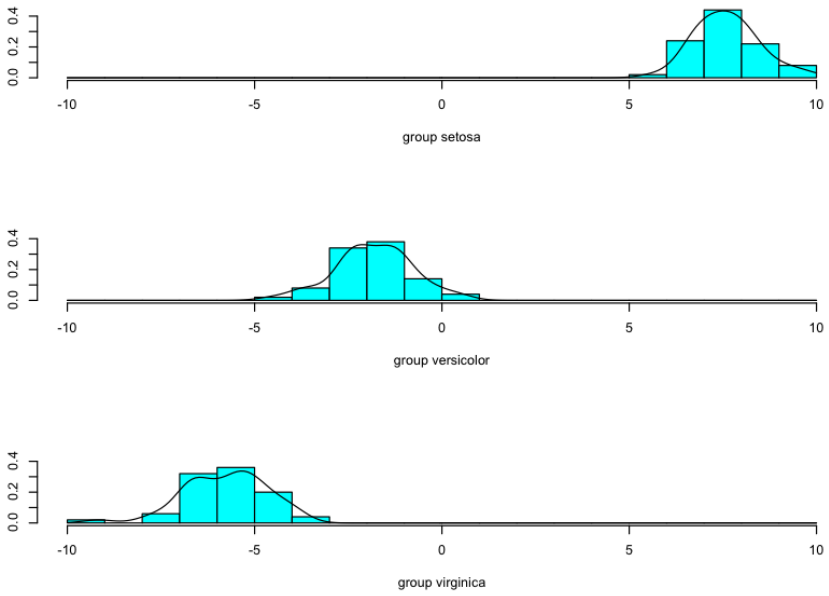

Figura 33: Distribución con la primer función discriminante

3.7. Regresión Logística

En esta sección mostramos cómo se puede utilizar la regresión logística con variables categóricas. Utilizamos una base de datos acerca de ataques terroristas que limpiamos a partir de la base de datos *Global Terrorism Database*[26]. El proceso de limpieza no se muestra, pero la base de datos que utilizamos en el análisis se puede descargar en la página de este documento (está en la introudcción). El objetivo será clasificar correctamente si hubo muertos utilizando el tipo de arma y el objetivo del ataque como variables independientes.

Lo primero que hacemos es cargar los paquetes que utilizaremos, cargar la base de datos y definir el grupo control que utilizaremos.

```
require(reshape2)
require(GGally)

data <- read.csv("GTD.csv", header = TRUE)
```

[26]Es una base de datos publicada por el la Universidad de Maryland y es utilizada por el Departamento de Defensa de los Estados Unidos. Se puede descargar de http://www.start.umd.edu/gtd/contact/.

```r
data$target <- relevel(data$target, ref = "Airports & Aircraft")
data$weapon <- relevel(data$weapon, ref = "Bombs")
data$death <- relevel(data$death, ref = "yes")
```

A continuación partimos la base de datos en dos, la parte que utilizaremos para el entrenamiento (`data.train`) (80 % de las observaciones) y la parte que utilizaremos para las pruebas (`data.test`).

```r
set.seed(123454321)
trn_sample <- sample(dim(data)[1], dim(data)[1] * .8)
data.train <- data[ trn_sample, ]
data.test  <- data[-trn_sample, ]
```

Ahora ejecutamos el modelo sobre los datos de entrenamiento, y desplegamos los resultados. Debemos notar que R no muestra automáticamente los momios, muestra el logit de los momios. Por lo tanto para ver los momios debemos exponencias los coeficientes de la regresión. También mostramos cómo se puede obtener los intervalos de confianza, aunque estos últimos no se muestran.

```r
log_fit <- glm(death ~ weapon + target,
    data = data.train,
    family = "binomial")

summary(log_fit)
exp(coef(log_fit))      # Momios
exp(confint(log_fit))   # 95 % CI de momios
```

Los resultados significativos son los siguientes (no se muestran aquellos que obtuvieron un *valor-p* mayor a 0.05):

	Estimate	Std. Error	Z-Value	$\Pr(>\|z\|)$
(Intercept)	1.30162	0.33362	3.901	$9.56e-05$
weaponFirearms	−1.19050	0.09389	−12.680	$< 2e-16$
weaponIncendiary	2.69036	0.23970	11.224	$< 2e-16$
weaponMelee	−0.96122	0.36747	−2.616	0.00890
targetMilitary	−1.38763	0.34694	−4.000	$6.34e-05$
targetPolice	−1.01602	0.34591	−2.937	0.00331
targetPrivate Citizens	−0.88885	0.34663	−2.564	0.01034
targetReligious Figures	−0.86847	0.43677	−1.988	0.04677
targetTelecommunication	2.23971	0.79622	2.813	0.00491
targetTerrorists	−2.64570	0.72434	−3.653	0.00026
targetUtilities	2.57885	0.44336	5.817	$6.00e-09$

Ahora queremos saber qué tan bien podemos clasificar con los resultados obtenidos hasta

ahora. Así que realizamos las predicciones sobre los datos de entrenamiento con el *umbral de corte* en 0.5, creamos la *tabla de confusión* y obtenemos el porcentage total de clasificación correcta.

```
pred <- predict(log_fit, newdata = data.test, type = "response")
pred <- pred >= .5
table(pred, data.test$bnm)
sum(diag(table(pred, data.test$bnm)))/dim(data.test)[1]
```

Tabla 3: Resultados de clasificación

pred	no	yes
no	191 (16.46 %)	31 (2.67 %)
yes	405 (34.91 %)	533 (45.94 %)

Obtenemos un porcentage total de clasificación correcta de 62.41 %. Sin embargo, podemos mejorar este resultado. Para hacerlo debemos explorar los errores tipo uno y tipo dos, y encontrar el mejor punto de corte dado el modelo. Para hacerlo realizamos iteraciones cambiando el punto de corte y observando la clasificación obtenida, con lo cual podemos obtener el punto de corte óptimo.

```
est <- data.frame(matrix(0, ncol = 3, nrow = 300))
colnames(est) <- c("Cutoff", "Porcentage", "Mediciones")
for (i in 1:100) {
    pred <- predict(log_fit, newdata = data.test, type = "response"↩
       )
    pred <- pred >= i/100
    t <- table(pred, data.test$bnm)
    est[i, 1] <- i/100
    est[i + 100, 1] <- i/100
    est[i + 200, 1] <- i/100
    est[i + 300, 1] <- i/100
    est[i, 2] <- sum(diag(t))/dim(data.test)[1]
    est[i + 100, 2] <- t[2]/dim(data.test)[1]
    est[i + 200, 2] <- t[3]/dim(data.test)[1]
    est[i, 3] <- "Correctos"
    est[i + 100, 3] <- "Falso positivo"
    est[i + 200, 3] <- "Falso negativo"
}
est <- na.omit(est)
est <- subset(est, Cutoff != .99 & Cutoff != 1.00)
cutoff <- est[which.max(est[est$Mediciones == "Correctos", 2]), 1]
ggplot(data = est,
    aes(x = Cutoff,
```

```
        y = Porcentage,
        colour = Mediciones)) +
    geom_line() +
    geom_vline(xintercept = 0.5,
        color = "darkgreen",
        alpha = 0.3) +
    geom_vline(xintercept = cutoff,
        color = "red",
        alpha = 0.2)
```

La gráfica que se muestra a continuación exhibe los errores tipo uno y tipo dos, y el porcentage total de predicción correcta. La línea vertical de color verde muestra el punto de corte original (0.5) y la de color rojo muestra el punto de corte óptimo (0.69).

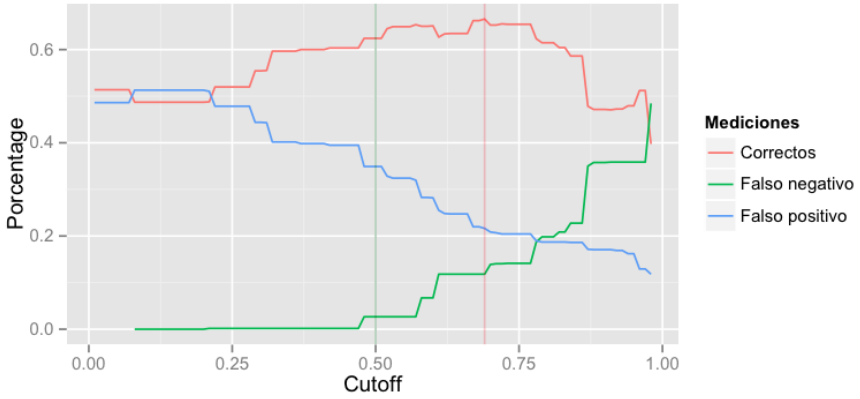

Figura 34: Errores tipo uno y dos, y éxitos de clasificación

```
pred <- predict(log_fit, newdata = data.test, type = "response")
pred <- pred >= cutoff
table(pred, data.test$bnm)
prop.table(table(pred, data.test$bnm))
sum(diag(table(pred, data.test$bnm)))/dim(data.test)[1]
```

Los resultados son mejores una vez realizado este procedimiento. Obtenemos un porcentage total de clasificación correcta de 66.55 %.

Tabla 4: Resultados de clasificación con *cutoff* ajustado

pred	no	yes
no	345 (29.74 %)	137 (11.81 %)
yes	251 (21.63 %)	427 (36.81 %)

3.8. Conclusión

Esperamos que este documento sea de ayuda a nuestros compañeros del ITAM, y de otras universidades, que quieren aprendar un poco más sobre R y sus usos para la investigación estadística.

Lo que mostramos en este documento es sólo una pequeña muestra de lo que se puede hacer con un *software* tan flexible como lo es R, y esperamos que sirva para motivar el mayor uso de este programa.

Referencias

[1] Rubén Hernández Cid, *Notas de Estadística Aplicada III*, Instituto Tecnológico Autónomo de México (ITAM), 2013.

[2] Robert Gentleman Kurt Hornik Giovanni Parmigiani, *An Introduction to Applied Multivariate Analysis with R*, Springer, 2011.

[3] Wojtek J. Krzanowski, *Recent Advances in Descriptive Multivariate Analysis*, Royal Statistical Society, 2010.