Ejemplos de tratamiento de datos con R

En este documento vamos a ver algunos ejemplos de cómo poner en práctica lo visto en documentos anteriores: secuenciar diferentes etapas de pre-procesamiento de datos y definir algunas estrategias de análisis de esos datos, como pasos previos a la ingesta de esos datos por parte de un modelo de *machine learning*, o como estrategia para analizar y comprender mejor los datos con los que estamos trabajando.

1. Ejemplo de pre-procesado de datos

El pre-procesado de datos en una etapa crucial en todo proceso de *data science* o *machine learning*, ya que ayuda a mejorar la calidad y consistencia de los datos con que se trabaja. Se extraen las características relevantes, se limpian y formatean los datos y se preparan para etapas posteriores. Veremos aquí un ejemplo de algunos pasos habituales en este pre-procesamiento, partiendo de este CSV que contiene datos sobre clientes de una tienda: nacionalidad, edad, salario anual y si compró o no un determinado producto.

1.1. Librerías necesarias

Comenzaremos importando los paquetes necesarios:

```
# Para análisis de datos
library(dplyr)
# Para generación de conjuntos de entrenamiento y test
library(caTools)
```

1.2. Carga inicial de los datos

Lo primero que haremos será **cargar los datos** en un *data frame*. Podemos usar la instrucción `View` para mostrar los datos que hemos cargado (en IDEs como RStudio, que dispone de visualizador).

```
datos <- read.csv("ejemplo_dataset.csv")
View(datos)
```

1.3. Filtrado de columnas relevantes

Ahora elegiremos las columnas que nos interesan: todas menos el *Id*

```
datos <- datos %>%
    select(-Id)
```

1.4. Gestión de valores nulos (*missing values*) y duplicados

En una colección de datos en R, los valores inexistentes se identifican con el término `NA` (*Not Available*). Podemos detectar si hay valores faltantes en una colección con la instrucción `is.na`, que devuelve un booleano por cada posición de la colección, o la instrucción `anyNA`, que devuelve un resultado global para todo el conjunto

```
v <- c(1, 4, NA, 8)
is.na(v)            # FALSE FALSE TRUE FALSE
anyNA(v)            # TRUE
```

Para los conjuntos de datos que tienen valores nulos podemos querer hacer alguna operación para reemplazarlos. Por ejemplo, sustituirlos por la media de valores del conjunto. Podemos hacerlo con algo así:

```
v2 <- ifelse(is.na(v),
             mean(v, na.rm=TRUE),
             v)
```

Lo que hace la instrucción anterior es crear un vector donde aplica la función `ifelse` a cada elemento. Si es nulo, se sustituye por la media de los valores de *v*. Si no es nulo, se deja el propio valor.

En el caso de querer utilizar otro valor como reemplazo (por ejemplo, la moda o la mediana), tendríamos que cambiar la función a utilizar en la expresión anterior. En el caso de la mediana podemos reemplazar la función *mean* anterior por `median`, y en el caso de la moda tenemos que calcularla manualmente, aunque basta con hacer algo como esto:

```
mode <- function(v) {
   uniqv <- unique(v)
   uniqv[which.max(tabulate(match(v, uniqv)))]
}
```

En algunas ocasiones nos puede interesar eliminar valores duplicados de un conjunto de datos. La instrucción `unique` nos permite hacerlo:

```
v <- c("Uno", "Dos", "Uno", "Tres", "Dos")
v2 <- unique(v)  # "Uno" "Dos" "Tres"
```

Aplicado al ejemplo que estamos tratando, si observamos los datos de entrada, falta alguno en las columnas de edad y salario. Reemplazaremos los valores faltantes por la media de la columna correspondiente:

```
datos$Age <- ifelse(is.na(datos$Age),
                    mean(datos$Age, na.rm = TRUE),
                    datos$Age)
datos$Salary <- ifelse(is.na(datos$Salary),
                    mean(datos$Salary, na.rm = TRUE),
                    datos$Salary)
```

1.5. Codificación de columnas categóricas

A continuación vamos a **codificar los datos categóricos** para poder realizar operaciones matemáticas con ellos (por ejemplo, a la hora de establecer una regresión lineal u operaciones similares). La columna *Purchased* tiene valores de *Yes* o *No*, que codificaremos como 1 y 0 respectivamente:

```
datos$Purchased <- factor(datos$Purchased,
                    levels=c("No", "Yes"),
                    labels=c(0, 1))
# El 0 queda como nulo, hay que recodificarlo de nuevo
datos$Purchased[is.na(datos$Purchased)] <- 0
```

En cuanto a la columna del país, podríamos también codificarla con valores numéricos consecutivos. Por ejemplo, 0 para España, 1 para Francia, etc. Sin embargo, este tipo de codificación puede otorgar un orden arbitrario a los datos, y que el modelo que construyamos después "aprenda" que España es mejor o peor que Francia porque su código es menor o mayor. Para evitar esta situación realizamos una codificación sin orden establecido, llamada ***one hot encoding***, donde añadimos una columna por cada posible país, y se pone a 1 la columna del país correspondiente.

Podemos utilizar para esto librerías adicionales, como *mltools* o *caret*. Sin embargo es más difícil controlar cómo se van a llamar las columnas generadas, y cómo se van a añadir al *data frame* existente. No es difícil hacer este proceso a mano:

```
datos %>%
    mutate(France = ifelse(Country=="France", 1, 0)) %>%
    mutate(Germany = ifelse(Country=="Germany", 1, 0)) %>%
    mutate(Spain = ifelse(Country=="Spain", 1, 0)) %>%
    # Eliminamos la columna original
    select(-Country)

# Opcionalmente podemos reorganizar las columnas
datos <- datos[, c("France", "Germany", "Spain", "Age", "Salary", "Purchased")]
```

El resultado será algo así:

France	Germany	Spain	Age	Salary	Purchased
1	0	0	44.00000	72000.00	1
0	0	1	27.00000	48000.00	0
...	...	...	...	...	...

1.6. Escalado de los datos

La operación de escalado de datos es muy habitual cuando trabajamos con valores heterogéneos, de forma que igualamos la magnitud de los datos que utilizamos. Por ejemplo, imaginemos que estamos trabajando con edades de personas y salarios anuales. Son valores muy dispares, porque las edades oscilan normalmente entre 0 y 100 años, y los salarios alcanzan varias decenas o centenas de miles de euros, o dólares. Si hacemos operaciones matemáticas con los datos, como sumas o multiplicaciones, claramente el salario va a ser mucho más determinante que la edad en el resultado final. Para evitar esta desigualdad se escalan o normalizan los datos a un rango común, como puede ser de 0 a 1 o de -1 a 1.

Para escalar datos en R usamos la función nativa `scale`, a la que le indicamos los datos a escalar. Por ejemplo, si queremos escalar las columnas 2 y 3 de un *data frame* haremos algo así:

```
datos[, 2:3] <- scale(datos[, 2:3])
```

Vamos ahora escalar los valores numéricos (edad y salario) de nuestro ejemplo:

```
datos[, c("Age", "Salary")] <- scale(datos[, c("Age", "Salary")])
```

1.7. División en conjuntos de entrenamiento y test

Una tarea muy habitual en muchos procesos de *machine learning* es dividir el conjunto de datos de que se dispone en una parte para entrenar el modelo y otra para validarlo una vez entrenado. Para hacer esto en R podemos emplear el paquete `caTools` que hemos incorporado al principio del ejemplo. Deberemos instalarlo previamente con `install.packages("caTools")`, si no lo tenemos instalado.

Podemos, opcionalmente, establecer una semilla aleatoria fija, de modo que siempre se genere el mismo subconjunto de entrenamiento y test. Esto se consigue con la instrucción `set.seed` nativa de R:

```
set.seed(1)
```

Finalmente, llamamos al método `sample.split` de `caTools`. Le tenemos que pasar como parámetros:

- Los valores de una columna cualquiera del *data frame*
- El *SplitRatio*, o porcentaje de valores que queremos destinar a entrenamiento

Como resultado, `sample.split` nos devuelve un vector de booleanos, donde a TRUE estarán las casillas reservadas para entrenamiento y a FALSE las destinadas para test. Con este vector podemos establecer los conjuntos de datos para cada cosa, usando la función `subset`. Aquí vemos un ejemplo de todos estos pasos:

```
datos <- # Cargar data frame
muestra <- sample.split(datos$Columna, SplitRatio=0.8)
entrenamiento <- subset(datos, muestra==TRUE)
test <- subset(datos, muestra==FALSE)
```

Aplicado a nuestro ejemplo, dividiremos ahora los datos en una parte para entrenamiento y otra para test, usando *caTools*:

```
set.seed(1)
split <- sample.split(datos$Purchased, SplitRatio=0.8)
train <- subset(datos, split==TRUE)
test <- subset(datos, split==FALSE)
```

> Aquí tienes el ejemplo completo del código fuente que hemos utilizado en este apartado.

2. Ejemplo de análisis exploratorio de datos

El análisis exploratorio de datos (en inglés *EDA, Exploratory Data Analysis*) es un proceso de análisis de los datos de un problema para extraer características relevantes, comprender mejor los datos e incluso intuir o inferir otros nuevos. Es un proceso creativo, a menudo continuación del pre-procesado anterior. Podemos estudiar variaciones o tendencias en los datos, mostrar gráficos representativos de ciertas características, etc.

Comenzaremos incluyendo los paquetes que vamos a necesitar para nuestro ejemplo:

```
library(ggplot2)  # Para gráficos
library(tibble)   # Para dinamizar el tratamiento de data frames
library(dplyr)    # Para funciones varias de tratamiento de datos
```

2.1. Cargando los datos

Para hacer las pruebas vamos a utilizar un *dataset* ya incorporado con *ggplot2* como muestra, llamado *diamonds*. Podemos consultar su estructura aquí y, como vemos, es una tabla con características de unos 50.000 diamantes diferentes:

- *carat* es el peso del diamante (valor numérico continuo)
- *cut* es la calidad del corte, una columna categórica con los valores *Fair*, *Good*, *Very Good*, *Premium* e *Ideal*. Son valores graduados, donde unos son mejores/peores que otros.
- *color* es el color del diamante, desde *D* (peor color) hasta *J* (mejor color). También es una columna categórica y graduada.
- *clarity* es una medida de la claridad del diamante, con los valores *I1* (el peor), *SI2*, *SI1*, *VS2*, *VS1*, *VVS2*, *VVS1* e *IF* (el mejor). De nuevo, columna categórica y graduada.
- *x*, *y* y *z* hacen referencia a la longitud, anchura y profundidad en milímetros, respectivamente. Son valores numéricos continuos
- *depth* mide la profundidad del diamante, como una operación matemática de las tres variables anteriores
- *table* mide la anchura en la punta del diamante relativa al punto más ancho
- *price* es el precio del diamante en dólares

Algunos de estos datos se pueden pre-procesar si se considera necesario para nuestro propósito. Por ejemplo, codificar numéricamente las columnas categóricas, o normalizar lo datos numéricos. Sin embargo, para otras operaciones, como ciertos gráficos que podemos obtener, no interesa esta codificación o escalado previo, porque perdemos los valores originales que queremos estudiar.

Podemos obtener un vistazo preliminar de los valores del *dataset* con la función `summary`, que nos mostrará un resumen por columna con su valor máximo, mínimo, media, conteo de valores por categoría en columnas categóricas, etc.

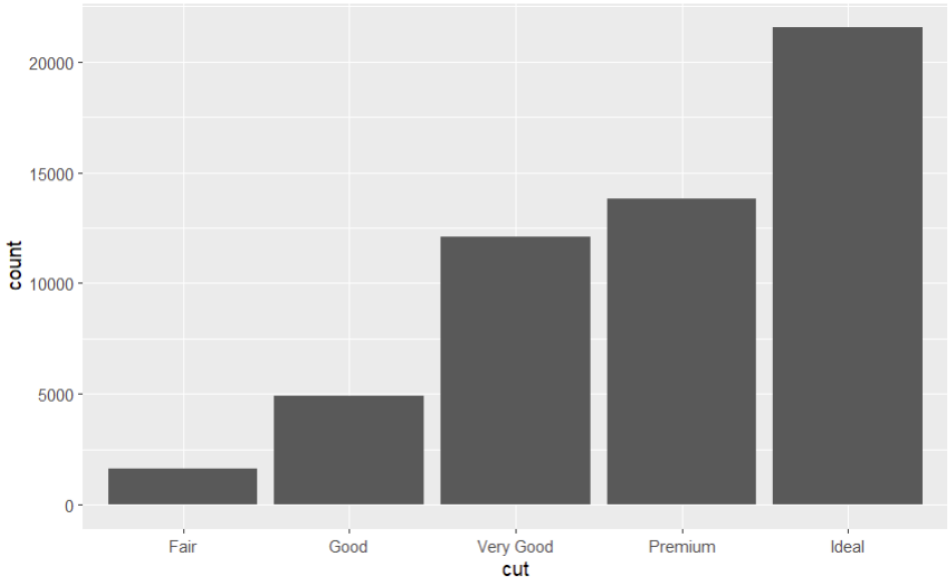
```
datos <- diamonds
summary(datos)
```

2.2. Algunos gráficos representativos

Vamos a representar algunos gráficos representativos de nuestro *dataset* usando *ggplot2*.

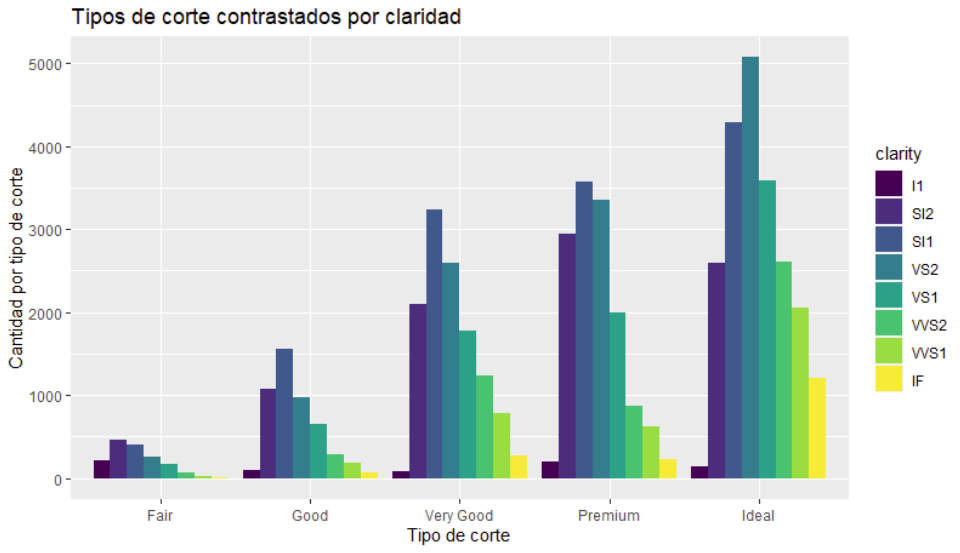
- Por ejemplo, podemos construir un gráfico de barras verticales con `geom_bar`, que muestre el conteo de valores de cada tipo de corte (columna *cut*):

```
ggplot(data=datos, mapping=aes(x=cut)) +
  geom_bar()
```



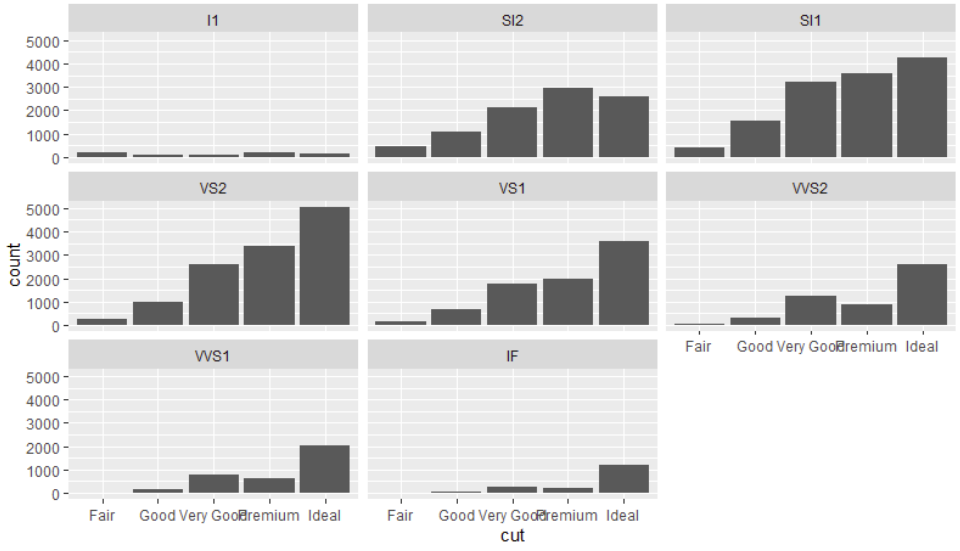
- Podríamos también enriquecer el gráfico anterior comparando la variable estudiada (*cut*) con otra (por ejemplo, la claridad del diamante). Podemos indicar incluso que cada tipo de claridad se pinte de un color diferente, acumulándose así distintos bloques en cada barra de corte, cada uno representando un tipo distinto de claridad.

```
ggplot(data=datos, mapping=aes(x=cut, fill=clarity)) +
  # position="dodge" hace que las cajas se pinten adyacentes
  # de izquierda a derecha, no apiladas
  geom_bar(position="dodge") +
  xlab("Tipo de corte") +
  ylab("Cantidad por tipo de corte") +
  ggtitle("Tipos de corte contrastados por claridad")
```



- Mostramos ahora un gráfico de barras como el primero, pero separado para cada claridad, acumulando los distintos gráficos con *facet*:

```
ggplot(data=datos, mapping=aes(x=cut)) +
   geom_bar() +
   facet_wrap(~ clarity)
```



- Histograma de pesos de los diamantes (10 divisiones)

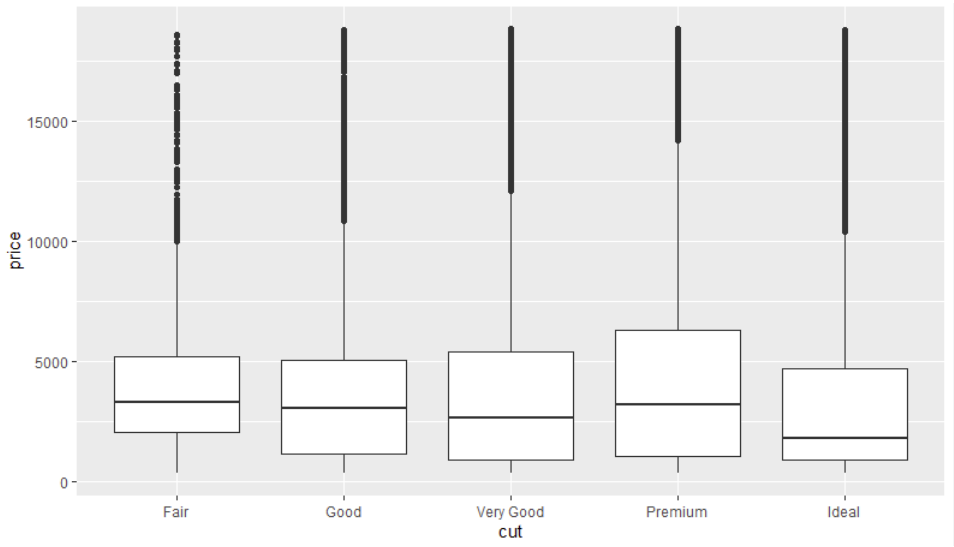
```
ggplot(data=datos, mapping=aes(x=carat)) +
   geom_histogram(bins=10)
```

- Gráfico de barras de tipo de corte para diamantes con peso entre 1 y 2

```
datos %>%
  filter(between(carat, 1, 2)) %>%
  ggplot(mapping=aes(x=cut)) + geom_bar()
```

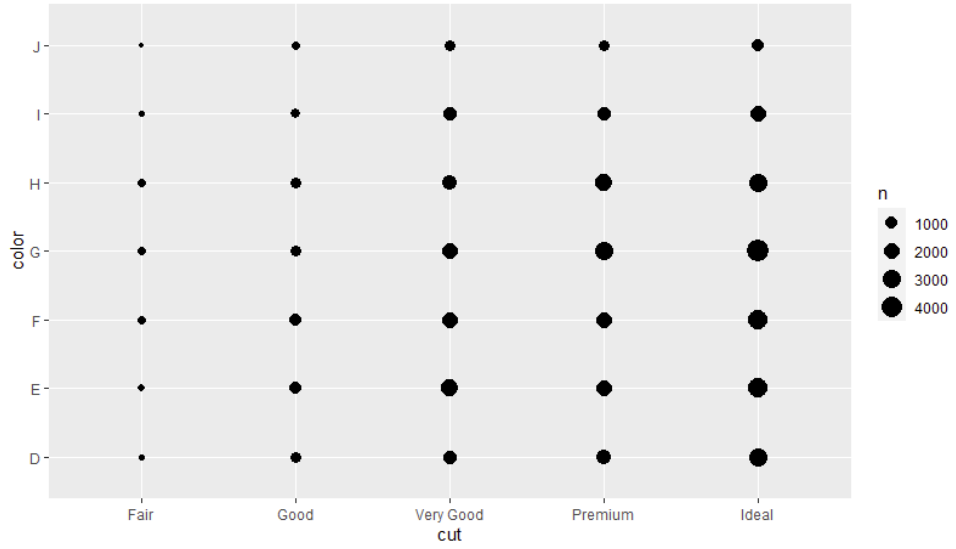
- Gráfico de cajas del tipo de corte frente al precio. Este tipo de gráficos permite detectar anomalías fuera de los percentiles de las cajas para cada valor categórico (eje X):

```
ggplot(data=datos, mapping=aes(x=cut, y=price)) +
   geom_boxplot()
```



- Contraste del tipo de corte con el color: este gráfico muestra puntos más gruesos donde las frecuencias son más grandes, viendo qué tipos de corte se corresponden más con qué colores

```
ggplot(data=datos, mapping=aes(x=cut, y=color)) +
   geom_count()
```



2.3. Otras estadísticas relevantes

Además de mostrar representaciones gráficas también podemos obtener datos numéricos o textuales del *dataset*:

- Conteo por tipo de corte:

```
datos %>% count(cut)
```

- Contraste del tipo de corte con la claridad (columnas *cut* y *clarity*): con la instrucción `table` podemos indicar dos nombres de columna, y se crea una tabla donde las filas son los valores de una columna y las columnas los valores de otra, contrastando así cuántos elementos con un valor X en una columna tienen un valor Y en la otra.

```
table(datos$cut, datos$clarity)
```

- Proporción de tipo de corte por tipo de claridad: mostramos en una tabla qué porcentaje de diamantes tienen el tipo de corte y tipo de claridad indicado:

```
round(prop.table(table(datos$cut, datos$clarity)) * 100, 2)
```

- Distribución de los pesos de los diamantes (*carat*) en franjas de 0.5:

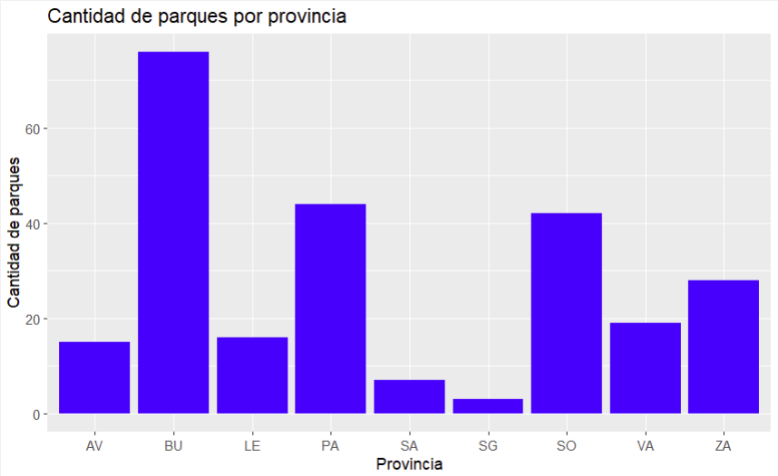
```
datos %>%
   count(cut_width(carat, 0.5))
```

Aquí tienes el ejemplo completo del código fuente que hemos utilizado en este apartado.

Ejercicio 1:

Utiliza este CSV sobre parques eólicos en Castilla y León. Se piden dos operaciones:

- Reemplazar los valores nulos de la columna de potencia total (*potencia*) por la media de esa columna
- Construir un gráfico de barras con el conteo de parques eólicos por provincia



AYUDA: vídeo con solución del ejercicio