# Web Scraping (descarga de datos web)

A la hora de bajarse datos web, lo primero que tenemos que hacer es averiguar en qué nodo del HTML se encuentra dicha información. Selectorgadget es una forma fácil de determinar qué selector extrae los datos que nos interesan sin necesidad de conocer HTML: Ve a esta página <http://selectorgadget.com/> y pon el programa en tu barra de marcadores.

|  |  |
| --- | --- |
|  | Entra en la página web.  Arrastra el link (caja naranja) a la barra de marcadores (caja roja). |

Ya lo tienes. A partir de ahora, en cualquier página en la que estés puedes pinchar en selectorgadget para seleccionar información. Cuando lo hagas, te saldrá el siguiente recuadro en la parte inferior derecha de tu pantalla:



Necesitarás las siguientes librerías en R, acuérdate de instalarlas e invocarlas antes de hacer web scraping.

**library(rvest)**

# Ejemplo: Extraemos datos del Lobo de Wall Street de wikipedia



library(rvest)

datos\_pelicula <- read\_html ("https://es.wikipedia.org/wiki/El\_lobo\_de\_Wall\_Street")

**# Obtenemos la ficha técnica (la columna de la derecha con los datos de dirección, producción, etc)**

ficha\_tecnica <- datos\_pelicula %>%

html\_nodes("table") %>%

.[1] %>% # Sabemos lo que queremos extraer gracias a selectorgadget

html\_table()

ficha\_tecnica<-as.data.frame(ficha\_tecnica)

ficha\_tecnica<-ficha\_tecnica[2:dim(ficha\_tecnica)[1],1:2]

**# Obtenemos las referencias**

referencias<- datos\_pelicula %>%

html\_nodes("span.reference-text") %>% # Sabemos lo que queremos extraer gracias a selectorgadget

html\_text()

referencias<-as.data.frame(referencias)

**# Obtenemos el contenido de la página con el argumento**

argumento<- datos\_pelicula %>%

html\_nodes("p") %>% # Sabemos lo que queremos extraer gracias a selectorgadget

html\_text()

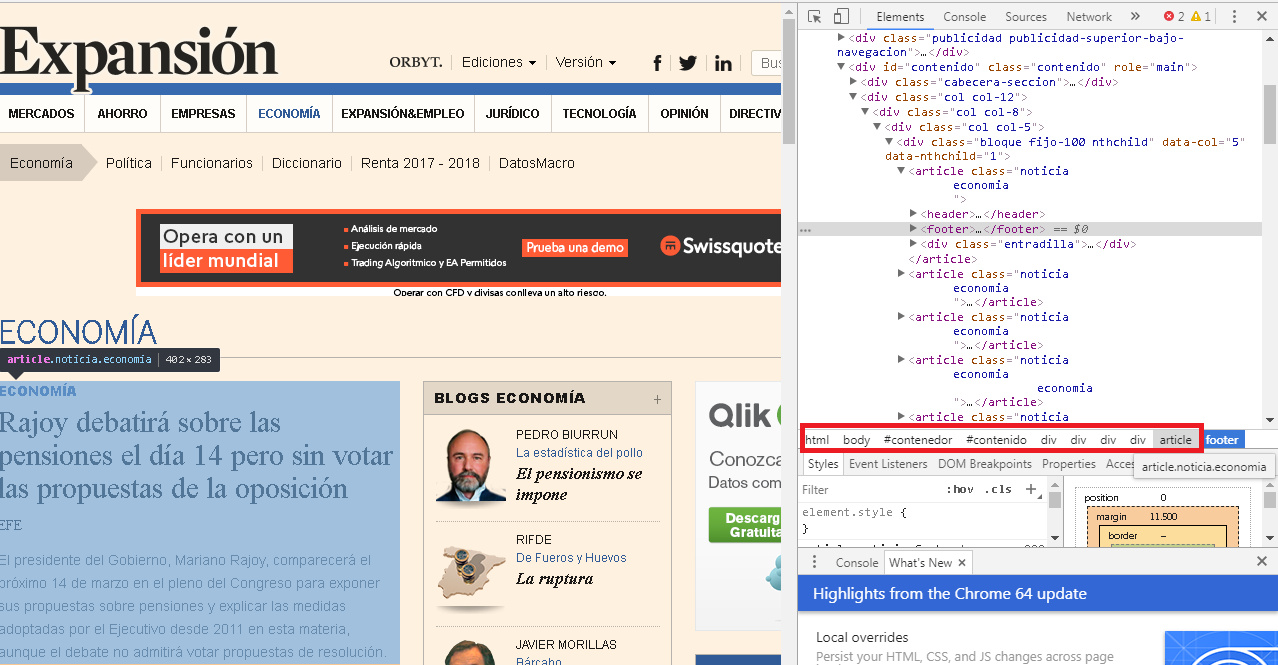
argumento<-as.data.frame(argumento)

argumento<-as.data.frame(argumento[4:(dim(argumento)[1]-1),1])

**¿Necesito usar selector Gadget para encontrar el nodo?**

No. Otra manera de encontrarlo es pulsar el botón derecho del ratón en el apartado de la web que queremos capturar y seleccionar la opción “inspeccionar”.

A la derecha se nos abrirá un menú donde veremos los distintos niveles de nodos, de mayor (izquierda) a menor (derecha). Pasando el ratón por encima de ellos se nos iluminará en la web la información que nos descargaríamos.



# Ejercicio: Captura los hechos relevantes de la CNMV



CNMV <- read\_html("http://www.cnmv.es/portal/HR/HRAldia.aspx")

# Ejercicio: Captura las noticias sobre economía de Expansión



expansion<-read\_html("http://www.expansion.com/economia.html?intcmp=MENUHOM24101&s\_kw=economia")

# Ejercicio: Captura la agenda macroeconómica de investing.com



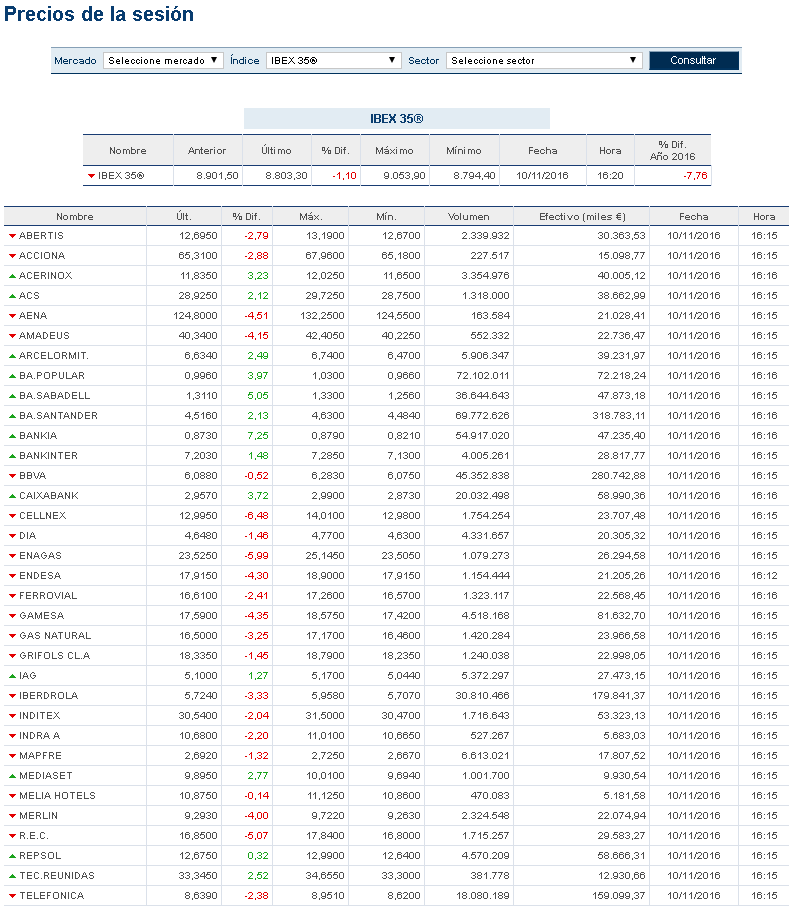
investing<-read\_html("http://es.investing.com/economic-calendar/")

# Ejercicio: Captura las cotizaciones de los commodities de investing.com



investing <- read\_html("https://es.investing.com/commodities/")

# Ejercicio: Captura las cotizaciones del Ibex 35 de Bolsa de Madrid



bolsa <- read\_html("http://www.bolsamadrid.es/esp/aspx/Mercados/Precios.aspx?indice=ESI100000000")

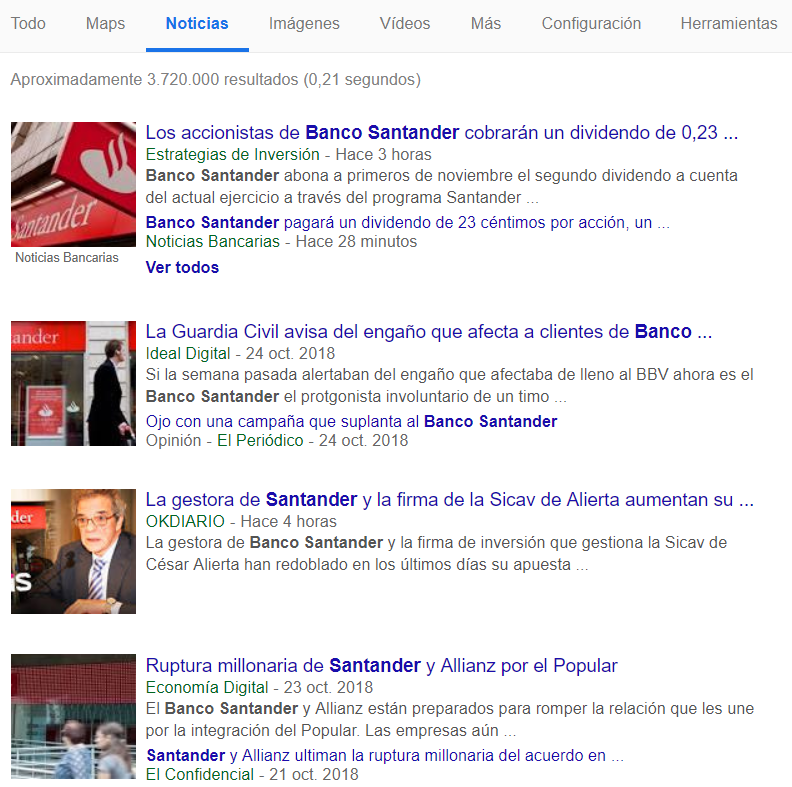
# Ejercicio: Obtén las últimas noticias (expansión), precios objetivo (expansión) y posiciones cortas (CNMV) de: Telefónica, Santander y Siemens Gamesa.

# Noticias: <https://www.expansion.com/mercados/cotizaciones/valores/telefonica_M.TEF.html>

# Precios objetivo: <https://www.expansion.com/mercados/bolsa/recomendaciones/consenso-mercados/t/telefonica_M.TEF.html>

# Posiciones cortas: https://www.cnmv.es/portal/Consultas/EE/PosicionesCortas.aspx?nif=A-28015865

# Ejercicio: Crea una función que descargue noticias de Google, pasándole como parámetros el nombre de la empresa que quieres descargar.

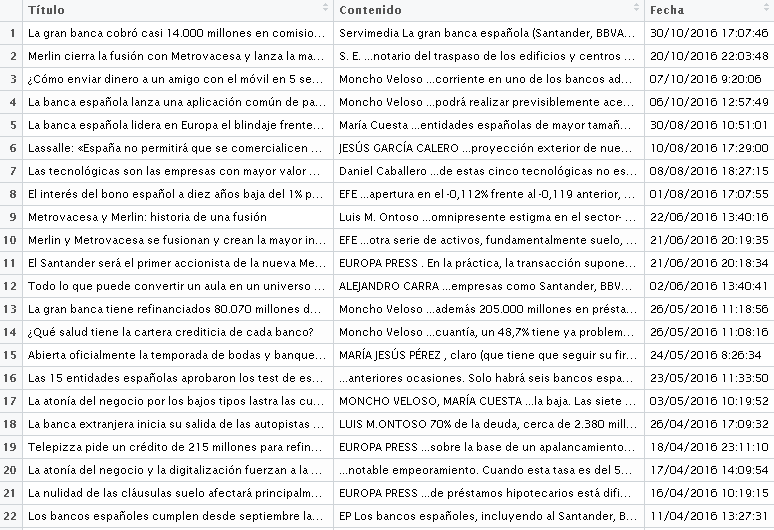


# Obtén las primeras 50 noticias del Banco Santander

# Captura todas las noticias que tengan relación con Telefónica, Santander y BBVA entre enero y noviembre de 2016 de la hemeroteca de ABC.



**Resultado deseado**



# Gráficos

**Código Gráfico resultante**

|  |  |
| --- | --- |
| **4 gráficos en 2 filas y dos columnas.** attach(mtcars) # Cargamos la base de datos. par(mfrow=c(2,2)) # Dividimos en 2 filas y 2 columnas. plot(wt,mpg, main="Scatterplot of wt vs. mpg") plot(wt,disp, main="Scatterplot of wt vs disp") hist(wt, main="Histogram of wt") boxplot(wt, main="Boxplot of wt") |  |
| **3 filas 1 columna** attach(mtcars) par(mfrow=c(3,1))  hist(wt) hist(mpg) hist(disp) |  |
| **Combinaciones** attach(mtcars) layout(matrix(c(1,1,2,3), 2, 2, byrow = TRUE)) hist(wt) hist(mpg) hist(disp)  # layout(matrix(c(1,1,2,3) le estamos indicando que el primer gráfico ocupa las dos primeras posiciones de la matriz (la fila de arriba), el segundo gráfico está en la 3ª posición de la matriz (2ª fila 1ª columna) y el tercer gráfico en el hueco restante (2ª fila 2ª columna). |  |
|  |  |
| **Añadir diagramas de caja a un gráfico de dispersión**  # Creamos el gráfico de dispersión. par(fig=c(0,0.8,0,0.8), new=TRUE)  plot(mtcars$wt, mtcars$mpg, xlab="Car Weight",   ylab="Miles Per Gallon")  # Añadimos el boxplot de arriba.  par(fig=c(0,0.8,0.55,1), new=TRUE)  boxplot(mtcars$wt, horizontal=TRUE, axes=FALSE)  # Añadimos el boxplot de la derecha.  par(fig=c(0.65,1,0,0.8),new=TRUE)  boxplot(mtcars$mpg, axes=FALSE)  # Añadimos el título.  mtext("Enhanced Scatterplot", side=3, outer=TRUE, line=-3) |  |

|  |  |
| --- | --- |
|  | **Gráficos personalizados**  Ejemplo de gráfico personalizado con la librería base de R.  Las velas están generadas a partir de los precios descargados (máximo, mínimo, apertura y cierre).  Las bandas suponen los precios objetivo de compra y venta.  Los triángulos hacia arriba son las compras efectivas y los triángulos hacia abajo las ventas realizadas. |

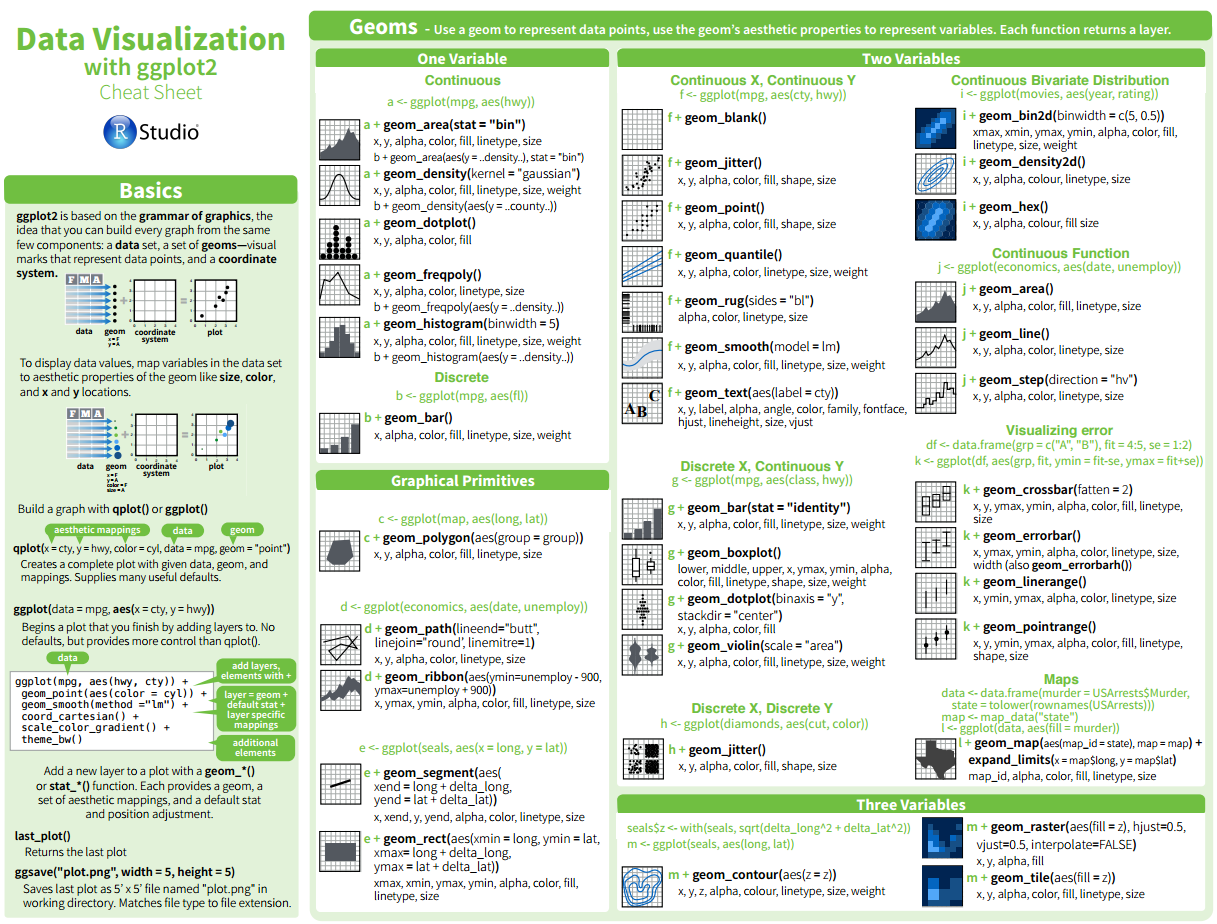
**Ejercicios**

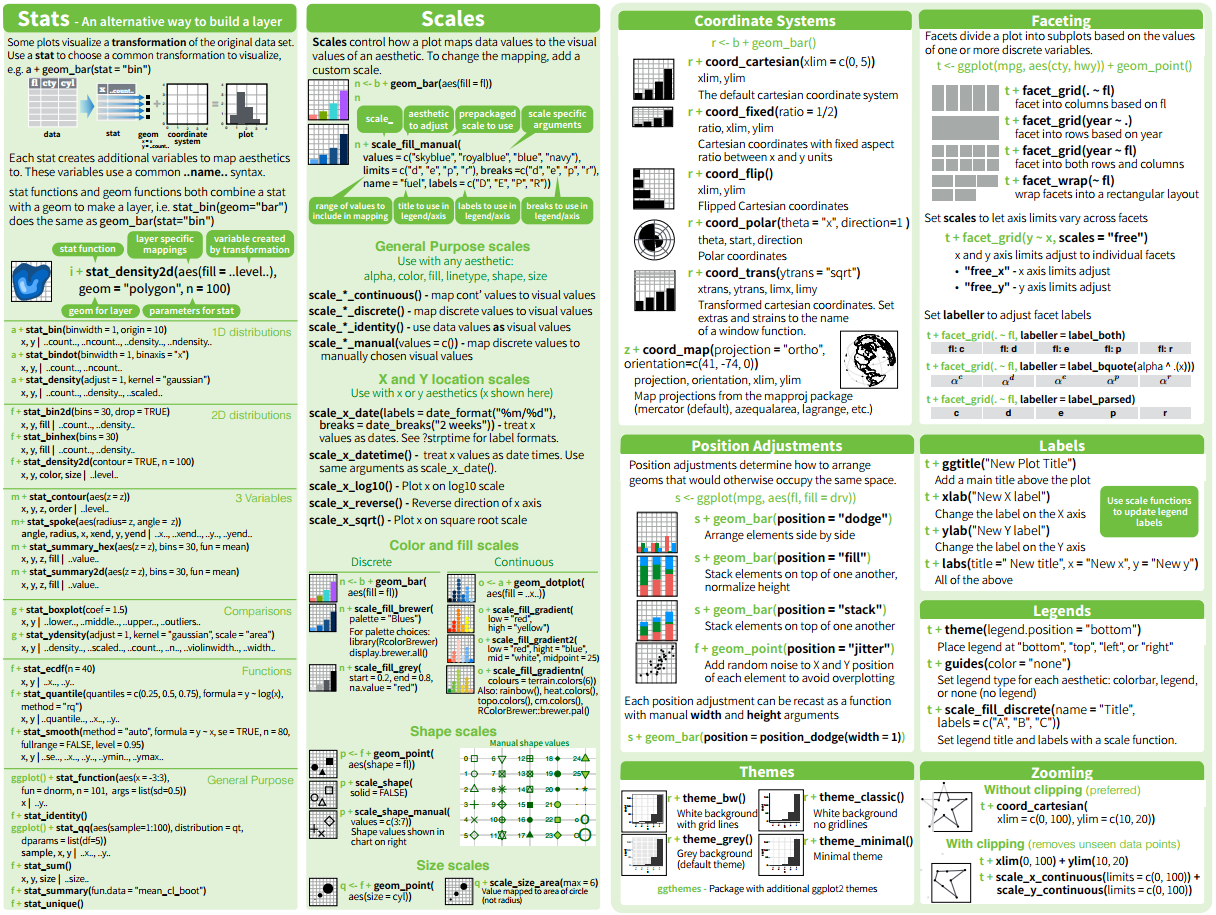
Programa los siguientes gráficos:

|  |  |
| --- | --- |
| **# Definimos un vector con 5 valores**.  Acciona <- c(1, 3, 6, 4, 9)  **# Graficamos el vector con todas las opciones en default.** |  |
| **# Definimos 2 vectores**  Acciona <- c(1, 3, 6, 4, 9)  Telefonica <- c(2, 5, 4, 5, 12)  **# Graficamos Acciona con una línea azul, delimitando el eje Y entre 0 y 12**  **# Graficamos Telefonica con una línea de puntos roja y la añadimos al gráfico anterior.**  **# Ponemos el título en rojo y tamaño 4.** |  |
| **# Definimos 3 vectores**  Acciona<-c(1,3,6,4,9)  Santander<-c(2,5,4,5,12)  BBVA<-c(4,4,6,6,16)  **# Graficamos los vectores usando rainbow colors**  **# Ponemos la leyenda en la parte superior izquierda.** |  |
| **# Definimos 3 vectores**  Acciona<-c(1,3,6,4,9)  Santander<-c(2,5,4,5,12)  BBVA<-c(4,4,6,6,16)  **# Graficamos las acciones usando heat colors,**  **# Ponemos un 10% del espacio entre cada barra y hacemos las etiquetas más pequeñas**  **# Ponemos la leyenda en (6,30) usando heat colors** |  |
| **# Cargamos los datos de mtcars**  **# Graficamos las variables mpg, disp, drat y wt, unas contra otras.** |  |

**Gráficos avanzados (ggplot2)**

<https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf>





# Análisis de rendimiento

Incluso los programadores experimentados tienen dificultades para la identificación de cuellos de botella en su código.

Para poder realizar el análisis instálate profvis en R. install.packages("profvis")

Para realizar el análisis de rendimiento de un programa, lo único que tenemos que hacer es “meter” el código que queremos analizar dentro de la instrucción profvis:

|  |  |
| --- | --- |
| library(profvis)  library(ggplot2)  profvis({  data(diamonds, package = "ggplot2")    plot(price ~ carat, data = diamonds)  m <- lm(price ~ carat, data = diamonds)  abline(m, col = "red")  }) |  |
|  | |

Al ejecutarse se abrirá una nueva pestaña de Data en donde podremos ver el tiempo, en milisegundos, que ha consumido cada línea de código. Justo debajo podremos ver un gráfico “de llamas” interactivo que nos muestra la pila de llamadas que ha realizado cada instrucción y su tiempo consumido.

|  |  |
| --- | --- |
|  | Como podemos ver, en el ejemplo de la izquierda, es muy fácil detectar cuellos de botella dentro del código.  La línea grid.draw.gTree ha consumido 1460 milisegundos del total de 1930 del programa.  Esa línea sola consume el 75% del tiempo de todo el programa. Deberemos pensar, por lo tanto, si hay otra manera de realizar la misma función de un modo más eficiente, o si es un tiempo razonable y, por lo tanto, asumible para nuestro código. |

Puedes consultar toda la información de la librería y ver más ejemplos en el siguiente enlace. <https://rstudio.github.io/profvis/>

**Medición de la duración de un programa**

Una manera rápida de analizar la duración de un código es poner al inicio y final del mismo las siguientes instrucciones.

tiempo <- proc.time() # Inicia el cronómetro

...

proc.time()-tiempo # Detiene el cronómetro (elapsed es el tiempo que ha tardado el programa)

|  |
| --- |
| Ejemplo de análisis de eficiencia: 50.000 combinaciones de lotería.  Objetivo: Análisis de rendimiento del programa. ¿Dónde están los cuellos de botella? |
| library(profvis)  profvis({  tiempo <- proc.time() # Inicia el cronómetro  **# Establecemos una semilla de generación aleatoria determinada para que los rdos sean siempre iguales.**  set.seed(1000)  **# Hacemos una función que saque una bola y compruebe que no ha salido ya en la combinación actual.**  sacar\_bola <- function(combi,nbola){  bola <-sort(sample(seq(1,50,1), 1, replace = TRUE))    **# Comprobamos que este número no ha salido ya en la combinación actual**  for (comprobar\_bola in 1:nbola){  if (bola == combi[comprobar\_bola]){  bola <-sort(sample(seq(1,50,1), 1, replace = TRUE)) # Si la bola ya ha salido sacamos otra.  }  }  return(bola)  }  **# Sacamos la combinación ganadora**  combi\_ganadora<- matrix(0,nrow=1,ncol=5,byrow=T) # Creamos una matriz para la combi ganadora.  for (nbola in 1:5){  combi\_ganadora[nbola]<- sacar\_bola(combi\_ganadora,nbola)  }  **# Sacamos las combinaciones apostadas y comprobamos cuantos aciertos tenemos en cada una de ellas.**  combinaciones <- 50000  apuestas<- matrix(0,nrow=combinaciones,ncol=5,byrow=T) # Creamos una matriz para las apuestas.  aciertos<-matrix(0,nrow=combinaciones,ncol=1,byrow=T) # Creamos un vector para los aciertos.  barra\_progreso <- winProgressBar(title= "Barra de progreso", min = 0, max = combinaciones, width=300)  # width es el nº de pixeles de la barra.  for (combinacion in 1:combinaciones){    **# Obtenemos las apuestas realizadas (sacamos una combinación)**  combi<- matrix(0,nrow=1,ncol=5,byrow=T) # Creamos una matriz donde guardaremos la combi apostada  for (nbola in 1:5){  combi[nbola]<- sacar\_bola(combi,nbola)  }  apuestas[combinacion,]<-combi    **# Comprobamos los aciertos que tenemos entre nuestras apuestas y la combinación ganadora.**  for (bola\_apostada in 1:5){  for (bola\_premiada in 1:5){ # comparamos cada bola\_apostada con cada bola\_premiada  if(apuestas[combinacion,bola\_apostada]==combi\_ganadora[bola\_premiada]){  aciertos[combinacion]<- aciertos[combinacion]+1  }  }  }  setWinProgressBar(barra\_progreso, combinacion, title=paste(round(combinacion/combinaciones\*100,0), "% realizado"))  }  close(barra\_progreso)  **# Calculamos la frecuencia de los aciertos (cuantas veces hemos acertado 1 nº, cuantas veces 2 etc)**  library(plyr)  aciertos<-count(aciertos)  aciertos  print(proc.time()-tiempo) # Detiene el cronómetro (elapsed es el tiempo que ha tardado el programa)  }) |

# Trabajar con distribuciones en R

|  |  |
| --- | --- |
| **Función de densidad**: Probabilidad de que una variable tome un determinado valor. |  |

|  |  |
| --- | --- |
| **Función de distribución**: Indica la probabilidad de que la variable tome un valor igual o menor  que x. Se calcula acumulando el valor de la función de densidad. Es la probabilidad acumulada. Es decir, da la probabilidad de un suceso. |  |

**Función cuantil**: Es la inversa de una función de distribución. Dada una probabilidad indica el valor que tomará la variable.

R denomina de la siguiente manera a las funciones en todas las distribuciones:

dxxx(función de densidad)

pxxx(función de distribución)

qxxx (función cuantil)

rxxx (generación de nº aleatorios)

**Librerías estadísticas útiles**

library(stats) # paquete de funciones estadísticas de R

library(pastecs) # Análisis de series temporales.

# Simulación de variables aleatorias discretas

sample(x, tamaño, replace = FALSE, prob = NULL)

x: vector de más de un elemento del que elegir las ocurrencias.

Tamaño: nº de ocurrencias o extracciones a realizar.

Replace: Indica si la extracción se hace o no con remplazamiento.

Prob: vector de pesos a asignar a cada uno de los posibles valores. Por defecto, todos los valores son equiprobables.

Ejemplo

dadoTrucoNum=sample(c(1:6), 1000, replace = TRUE, prob = c(2,3,1,9,8,5))

Obtenemos un dado trucado, con mayor probabilidad de que salga el 4 y 5, que el resto.

Si ponemos peso 0, ese valor no saldrá nunca.

table(dadoTrucoNum) 🡪 Te pone los resultados en bonito, en una tabla.

# Simulación de variables aleatorias continuas

**Distribución de Bernoulli**

Arroja 1 o 0, cara o cruz. Se suele utilizar para realizar simulaciones de default (con distintas probabilidades al 50%, 5%, 1%...)

|  |  |
| --- | --- |
| dbern(x, prob, log=FALSE)  pbern(q, prob, lower.tail=TRUE, log.p=FALSE)  qbern(p, prob, lower.tail=TRUE, log.p=FALSE)  rbern(n, prob) | X y Q son vectores de cuantiles (ver ayuda para estos parámetros).  p es el vector de probabilidades.  n cuantos números aleatorios queremos.  Prob: probabilidad del suceso a medir. Ej: simulamos default con prob 2%  log o log.p da T o F en lugar de 1 o 0.  Lower.tail: Si es true, las probabilidades son X<=x, sino X>x. |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| library(Rlab)  operaciones <- rbern(1000, 0.02)  table(operaciones) | Resultado   |  |  | | --- | --- | | 0 | 1 | | 979 | 21 | |

**Distribución uniforme**

|  |  |
| --- | --- |
| dunif(x, min = 0, max = 1, log = FALSE) función de densidad (da la probabilidad de un suceso. Ej un dado 1/6)  punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE) Función de distribución (da la probabilidad acumulada en un pto)  qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE) Función cuantil (da un punto a una probabilidad)  runif(n, min = 0, max = 1) # Sacamos números aleatorios con distribución uniforme.  operaciones <- runif(1000, min = 100, max = 200) # Sacamos 1000 nº aleatorios.  table(operaciones) # Esto no vale de nada porque es una tabla enorme, que no es útil de ver.  hist(operaciones) # Para ver la uniformidad de los números, hacemos el histograma. |  |

**Distribución Normal**

|  |
| --- |
| dnorm(x, mean = 0, sd = 1, log = FALSE) ) función de densidad (da la probabilidad de un suceso).  pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE) Probabilidad acumulada en un pto.  qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE) Punto en función de una probabilidad.  rnorm(n, mean = 0, sd = 1) Obtención de números aleatorios. |

|  |  |
| --- | --- |
|  |  |

Ojo, al sacar números aleatorios no suelen surgir por encima o debajo de 5 desviaciones típicas, pero los límites son –inf y +inf, por lo que podría pasar.

Percentiles y desviaciones.

90% 1,6448 En 1,6 desviaciones típicas tenemos el 90% de los datos.

95% 1,9599 En 1,9 desviaciones tenemos el 95% de los datos.

99% 2,5758 En 2,5 desviaciones tenemos el 99% de los datos.

99,99% 4 En 4 desviaciones tenemos el 99,99% de los datos.

El parámetro μ (media) desplaza el centro de la distribución, el parámetro σ (desv) la ensancha o estrecha. (X- μ) /σ = N(0,1)

operaciones <- rnorm(10000, mean = 0, sd = 1)

hist(operaciones, breaks=seq(from=-5,to=5,by=0.2))

plot(operaciones) # Graficarlo así no tiene sentido.

**Ejercicio:** ¿Cómo consigo 1000 nº aleatorios que sigan una distribución normal entre 100 y 200 con una probabilidad del 99%?

**Ejercicio**: Quiero acotar la distribución anterior y que no me de los nº menores de 130, pero me siga dando 1000 nº

|  |  |
| --- | --- |
|  | Es decir, quiero obtener esta distribución |

**Mixturas**: ¿Cómo se puede realizar una mezcla de distribuciones?

**Ejercicio**: Queremos mezclar tres distribuciones normales distintas (como en el gráfico). ¿Cómo lo hacemos?

Para comprobar la muestra resultante hacemos el histograma, donde comprobamos el resultado.



**Distribución Log-Normal**

Los rendimientos siguen una distribución normal, los precios log normales (xq no pueden ser negativos)

|  |  |
| --- | --- |
| dlnorm(x, meanlog = 0, sdlog = 1, log = FALSE)  plnorm(q, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)  qlnorm(p, meanlog = 0, sdlog = 1, lower.tail = TRUE, log.p = FALSE)  rlnorm(n, meanlog = 0, sdlog = 1)  operaciones <- rlnorm(1000, meanlog = 0, sdlog = 1)  hist(operaciones, breaks=seq(from=0,to=30,by=1))  # Ojo, si da error, es porque ha salido un dato por encima del “to 30”  # Habría que aumentar el límite. |  |

**Distribución Binomial**

Es una distribución de Bernuilli N veces (sucesos independientes y con la misma probabilidad)

Se usa en distribuciones de pérdidas.

|  |  |
| --- | --- |
| dbinom(x, size, prob, log = FALSE)  pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)  qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)  rbinom(n, size, prob)  operaciones <- rbinom(1000, 20, 0.5)  hist(operaciones, breaks=seq(from=0,to=30,by=1)) | Donde pone n=20, realmente debería poner size=20 |

**Distribución geométrica**

La distribución geométrica es un modelo adecuado para aquellos procesos en los que se repiten pruebas hasta la consecución del éxito. Por ejemplo, hasta que una empresa con una probabilidad de default p entra en quiebra.

|  |  |
| --- | --- |
| dgeom(x, prob, log = FALSE)  pgeom(q, prob, lower.tail = TRUE, log.p = FALSE)  qgeom(p, prob, lower.tail = TRUE, log.p = FALSE)  rgeom(n, prob)  operaciones<- rgeom(1000, 0.05)  hist(operaciones, breaks=seq(from=0,to=120,by=5)) |  |

**Distribución de Poisson**

Se utiliza para simular nº de incumplimientos (o nº de errores en una página).

|  |  |
| --- | --- |
| dpois(x, lambda, log = FALSE)  ppois(q, lambda, lower.tail = TRUE, log.p = FALSE)  qpois(p, lambda, lower.tail = TRUE, log.p = FALSE)  rpois(n, lambda)  Permite aproximar binomiales λ=n\*p  operaciones <- rpois(1000, 1)  hist(operaciones)  operaciones <- rpois(1000, 50)  hist(operaciones) |  |

**Distribución χ2**

Una distribución χ2 centrada es igual a una distribución normal.

Se usa para contrate de varianzas o hipótesis.

La “chepa” se desplaza a la derecha cuantos más grados pongamos.

|  |  |
| --- | --- |
| dchisq(x, df, ncp = 0, log = FALSE)  pchisq(q, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)  qchisq(p, df, ncp = 0, lower.tail = TRUE, log.p = FALSE)  rchisq(n, df, ncp = 0) # Df son grados de libertad.  operaciones <- rchisq(1000, 5, ncp = 0)  hist(operaciones)  operaciones <- rchisq(1000, 10, ncp = 0)  hist(operaciones) |  |

**Distribución t de Student**

Simétrica respecto del origen, con colas más gruesas que la normal.

Se usa en finanzas para simular pérdidas (por la cola izq), usando grados de libertad bajos.

Cuando los grados de libertad son grandes se asemeja a una N(0,1).

|  |  |
| --- | --- |
| dt(x, df, ncp, log = FALSE)  pt(q, df, ncp, lower.tail = TRUE, log.p = FALSE)  qt(p, df, ncp, lower.tail = TRUE, log.p = FALSE)  rt(n, df, ncp) # ncp es centralidad de la gráfica.  operaciones <- rt(1000, df=5, ncp=0)  hist(operaciones)  operaciones <- rt(1000, df=1000, ncp=0) # Similar a una normal(0,1)  hist(operaciones) |  |

**Distribución exponencial**

Se usa para calcular la probabilidad de default, siendo λ la intensidad de default.

|  |  |
| --- | --- |
| dexp(x, rate = rate, log = FALSE)  pexp(q, rate = rate, lower.tail = TRUE, log.p = FALSE) # Rate es lambda  qexp(p, rate = rate, lower.tail = TRUE, log.p = FALSE)  operaciones <- rexp(1000, rate=1)  hist(operaciones)  operaciones <- rexp(1000, rate=5)  hist(operaciones) |  |

# Intervalos de confianza

Se usa para estimar si un resultado está dentro de un rango de valores en el que el valor del parámetro estará con una probabilidad de 1-α (nivel de confianza).

|  |  |
| --- | --- |
|  | norm.interval = function(datos, varianza = var(datos),nivel.conf = 0.95)  {  z = qnorm((1 - nivel.conf)/2, lower.tail = FALSE)  m = mean(datos)  dt = sqrt(varianza/length(datos))  c(m - z \* dt, m + z \* dt)  }  X = rnorm(50,0,1) # Genero 50 nº aleatorios de media 0 y desviación típica 1.  norm.interval(X) # Quiero comprobar si el 0 está dentro del intervalo con un 95% de probabilidad.  # Da: -0.2566292 0.4148183 # Aceptamos la hipótesis nula porque el valor 0 está dentro del intervalo. |

# Barra de progreso de un programa

En ocasiones el programa que realices tardará mucho en ejecutarse y te entrará la duda de si está “pensando” o si se ha bloqueado. Para evitar estas situaciones, lo mejor es utilizar la típica barra de progreso.

n\_escenarios <- 10000

barra\_progreso <- winProgressBar(title= "Barra de progreso", min = 0, max = n\_escenarios, width=300) # width es el nº de pixeles de la barra.

for (escenario in 1:n\_escenarios){

setWinProgressBar(barra\_progreso, escenario, title=paste(round(escenario/n\_escenarios\*100,0), "% realizado"))

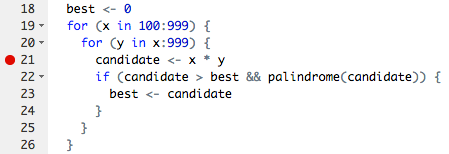
}

close(barra\_progreso)



# Depurar un programa

La forma más común para parar en una línea de código es **establecer un punto de interrupción** en esa línea. Puedes hacer esto en RStudio haciendo clic a la izquierda del número de línea en el editor (ojo, no funciona dentro de una función o de un bucle).



**browser()** detiene la ejecución e invoca un navegador. Se puede poner en cualquier parte del código. Aquí, por ejemplo, se utiliza para detener cuando una función está a punto de devolver TRUE:



A diferencia del punto de interrupción, el navegador es parte del código, por lo que necesita ser ejecutado. El navegador es la herramienta de depuración de nivel más bajo y puede ser usado incluso en las situaciones en las que el punto de interrupción no funciona.

El navegador es útil para crear puntos de interrupción condicionales. Por ejemplo, si quieres iniciar la depuración después de cientos de iteraciones del ciclo puede hacer:

for (i in 1:1024) {

start\_work()

if (i == 512)

browser()

finish\_work()

}

**Detenerse cuando se está ejecutando una función**

Hay que establecer un punto de interrupción antes de la declaración de una función. No cambia la función en sí, sino que hace que el depurador se active inmediatamente cuando se ejecuta la función.

Usa la función debugonce(nombre\_funcion) para establecer un punto de depuración dentro de una función (librería devtools).

debugonce () establece un punto de ruptura, es decir, la función entra en el depurador la próxima vez que se ejecute, pero no todas las veces que entre. Si lo que quieres es depurar una función cada vez que se ejecute, invoca debug(nombre\_funcion) en la función y undebug(nombre\_funcion) cuando ya no quieras seguir depurando.

**Usando debugger**

|  |  |
| --- | --- |
|  | Una vez se detiene el código, el IDE entra automáticamente en modo de depuración, en donde disponemos de varias herramientas para inspeccionar y alterar el estado del programa. |

|  |  |
| --- | --- |
|  | **Environment** (la mayoría de las veces no usarás esto)  R interactúa con el "entorno global" (objetos, valores, funciones y datos).  Los valores en gris son argumentos de la función que no han sido evaluados todavía.  Encima de la lista de objetos locales, hay una lista desplegable que muestra la cadena de herencia para el entorno. Es una lista de lugares en donde se buscará el valor de las variables que utilice la función. |
|  | **Traceback** (Callstack / pila de llamadas)  Muestra cómo se ha alcanzado el punto actual, desde la primera línea ejecutada hasta el punto actual.  Puedes hacer clic en cualquier función de la pila de llamadas para ver el contenido actual de su environment y el punto de ejecución. Ten en cuenta que la selección en la pila de llamadas no cambia el entorno activo en la consola. |
|  | **Ventana de código**  La ventana de código muestra la función que se está ejecutando. La línea que se está a punto de ejecutar se resalta en amarillo. |
|  | **Console**  Mientras se depura, te darás cuenta de dos cambios en la consola.  El primero es que el símbolo es diferente Browse[1]>  Pulsando Enter en la consola, esta ejecutará la instrucción actual y pasará a la siguiente.  La segunda es que hay una nueva barra de herramientas en la parte superior de la consola con los siguientes comandos:  n or Enter F10 Ejecutar siguiente línea  s Shift+F4 Entrar dentro de la función  f Shift+F6 Terminar función / bucle  c Shift+F5 Seguir ejecutando  Q Shift+F8 Salir del depurador |

Adjunto un link donde se explican estas y otras técnicas de depuración de errores con mayor profundidad.

<http://adv-r.had.co.nz/Exceptions-Debugging.html>

# Kaggle

<https://www.kaggle.com/>

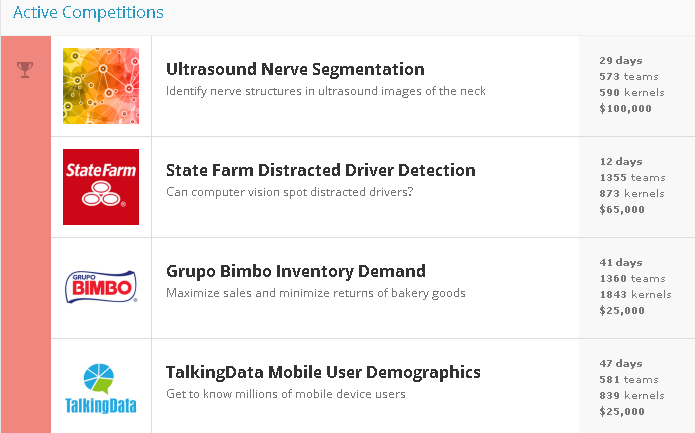
Fundada en 2010, Kaggle es una plataforma de competición en análisis de modelos predictivos donde las empresas publican sus datos y los programadores compiten para generar el mejor modelo predictivo.

Este enfoque se basa en que existen infinidad de maneras de resolver un problema y es “imposible” conocer de antemano qué técnica será la más eficaz.

Por otro lado, Kaggle es un “centro de reclutamiento” de Data Scientist para empresas del calibre de Google, Facebook, Amazon…

**¿Cómo funcionan las competiciones?**

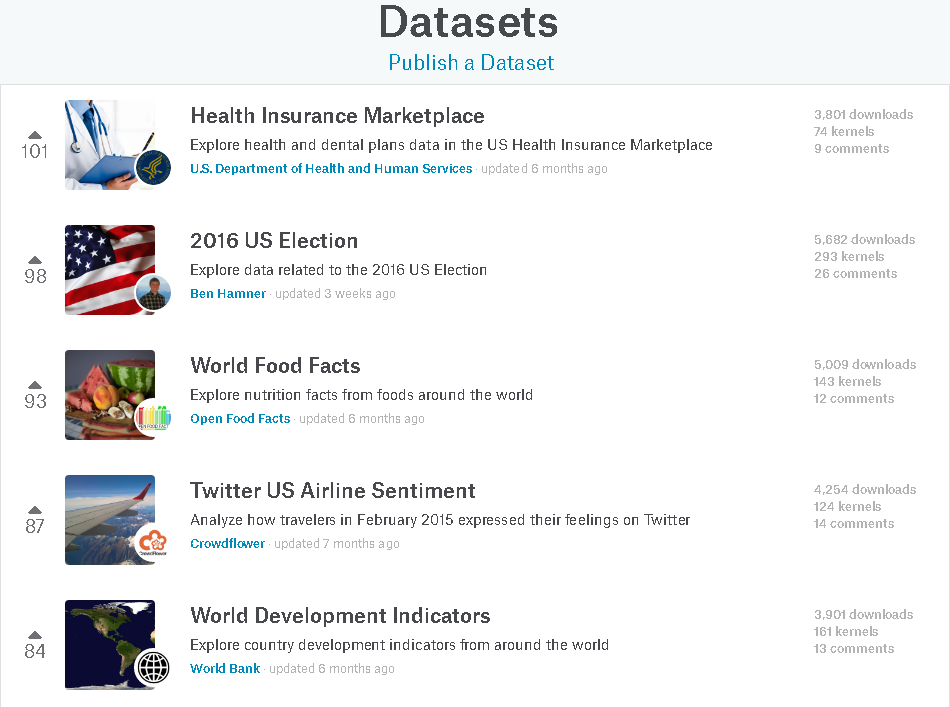
* La empresa prepara y publica un set de datos, indicando el objetivo deseado, el plazo y el premio en metálico.
* Los programadores compiten entre sí para producir los mejores modelos predictivos. El trabajo es compartido públicamente al resto de programadores. Las soluciones se califican inmediatamente en función de su precisión predictica en relación con un archivo de soluciones oculto y se presenta un ranking de programadores.
* Transcurrido el plazo, la empresa paga el premio en metálico a cambio de una “licencia mundial, perpetua, irrevocable y libre de royalties para usar la solución ganadora.



Si no te interesa competir, esta página es muy interesante para descargarse Set de Datos con los que poner a prueba tus conocimientos y habilidades de programación.

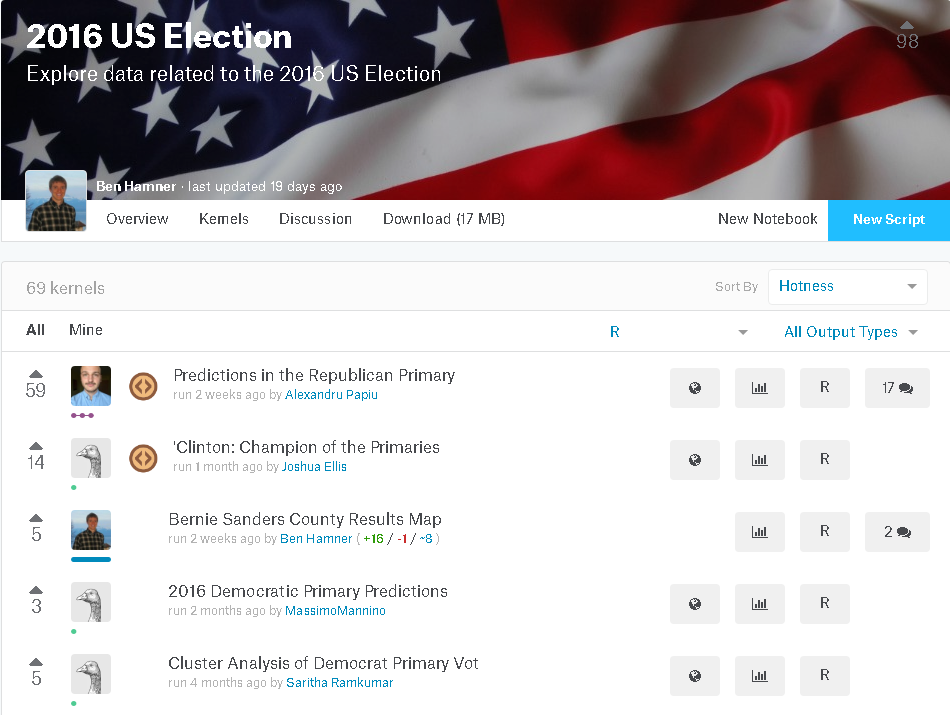
Uno de los principales problemas a la hora de programar es la ausencia de Set de Datos gratuitos.

Estos son algunos de los Datasets que te puedes descargar (échales un vistazo).

****

Por último, y quizás lo más importante cuando estás empezando a programar, es que puedes consultar las soluciones de otros programadores a un problema determinado. El ranking de soluciones te facilita qué es lo que tienes que buscar.

La posibilidad de ver el código de otra persona, que programa mejor que tú, te permitirá avanzar mucho más rápido…

****

****