Interpolación y aproximación

Parte 1

Joaquin Cavieres

Ejemplo 1

```
# Interpolación (ejemplo 1)
linterp <- function (x1 , y1 , x2 , y2) {
  m <- (y2 - y1) / (x2 - x1)
  b <- y2 - m * x2
  return (c(b, m))
}

inter1 <- linterp(1, 2, 0, -2)
inter1
```

```
## [1] -2  4
```

La función "`inter1`" muestra los coeficientes para la pendiente y el intercepto.

Observacion

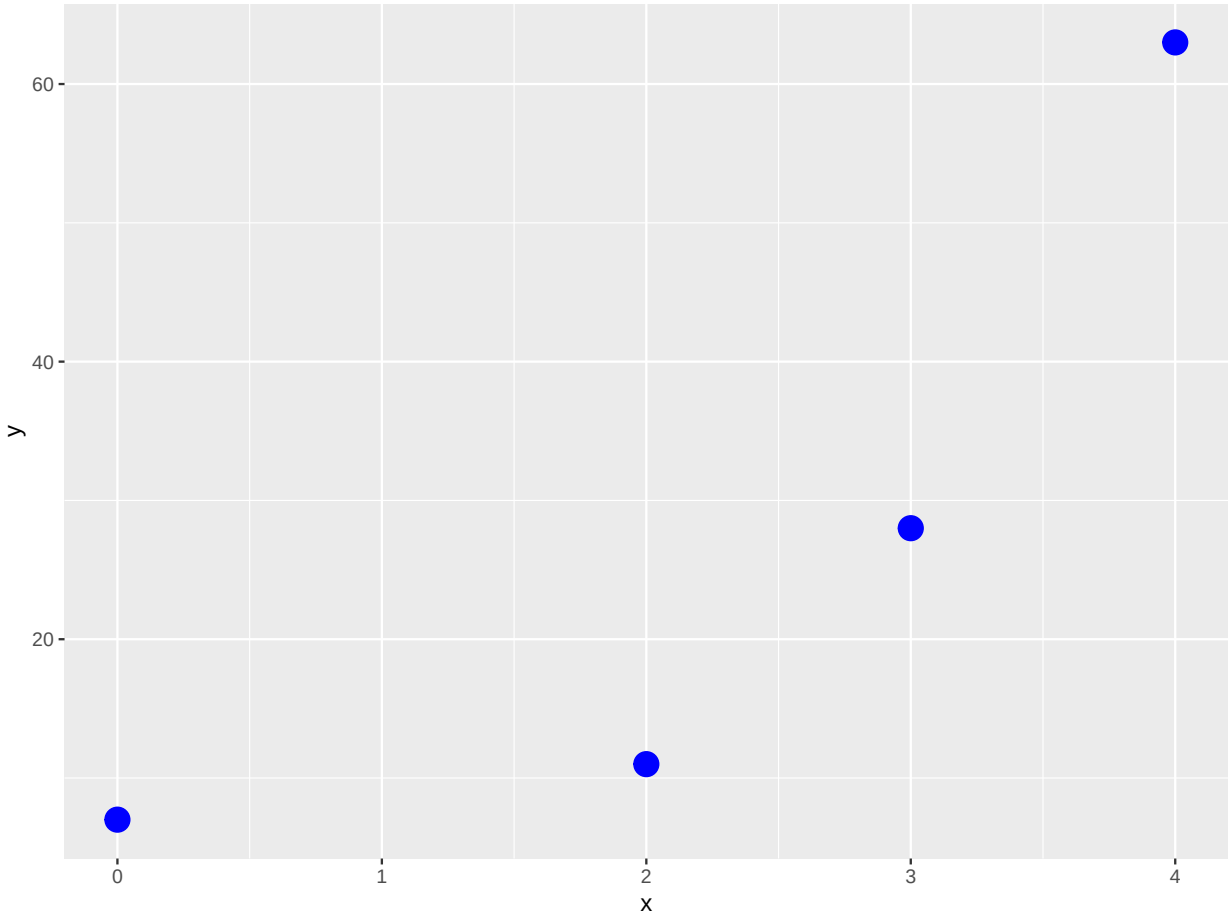
Lo que nos puede preocupar en este cálculo es el error numérico. Hay fuentes potenciales de error numérico aquí si los dos puntos de la muestra están particularmente juntos; entonces las operaciones $y2 - y1$ o $x2 - x1$ pueden resultar en un error falsamente grande. Este puede ser un problema mayor en la segunda resta, ya que se usa como denominador en una fracción.A partir de este método existen otros mucho más convenientes para evitar este tipo de problemas.

Ejemplo 2

```
# Interpolacion de Lagrange (ejemplo 2)
x <- c(0, 2, 3, 4)
y <- c(7, 11, 28, 63)

df <- data.frame(cbind(x, y))
```

```r
library(ggplot2)
ggplot(df, aes(x=x, y=y)) + geom_point(size=5, col='blue')
```



Como queremos encontrar un polinomial que pase (interpole) a través de estos puntos, entonces debemos calcular siguiendo la definición del interpolante de Lagrange:

$$L_1(x) = \frac{(x-2)(x-3)(x-4)}{(0-2)(0-3)(0-4)} = -\frac{1}{24}(x-2)(x-3)(x-4)$$

así luego $L_2(x)$ es:

$$L_2(x) = \frac{(x-0)(x-3)(x-4)}{(2-0)(2-3)(2-4)} = \frac{1}{4}x(x-3)(x-4)$$

y finalmente $L_3(x)$ y $L_4(x)$:

$$L_3(x) = \frac{(x-0)(x-2)(x-4)}{(3-0)(3-2)(3-4)} = -\frac{1}{3}x(x-2)(x-4)$$

$$L_4(x) = \frac{(x-0)(x-2)(x-3)}{(4-0)(4-2)(4-3)} = \frac{1}{8}x(x-2)(x-3)$$

Los polinomiales encontrados son multiplicados por los correspondientes valores de y resultando en el siguiente polinomial:

$$7\left(-\frac{1}{24}(x-2)(x-3)(x-4)\right)+11\left(\frac{1}{4}x(x-3)(x-4)\right)-28\left(-\frac{1}{3}x(x-2)(x-4)\right)+63\left(\frac{1}{8}x(x-2)(x-3)\right)$$

lo que se reduce a la siguiente expresión:

$$x^3 - 2x + 7$$

Ejemplo en R

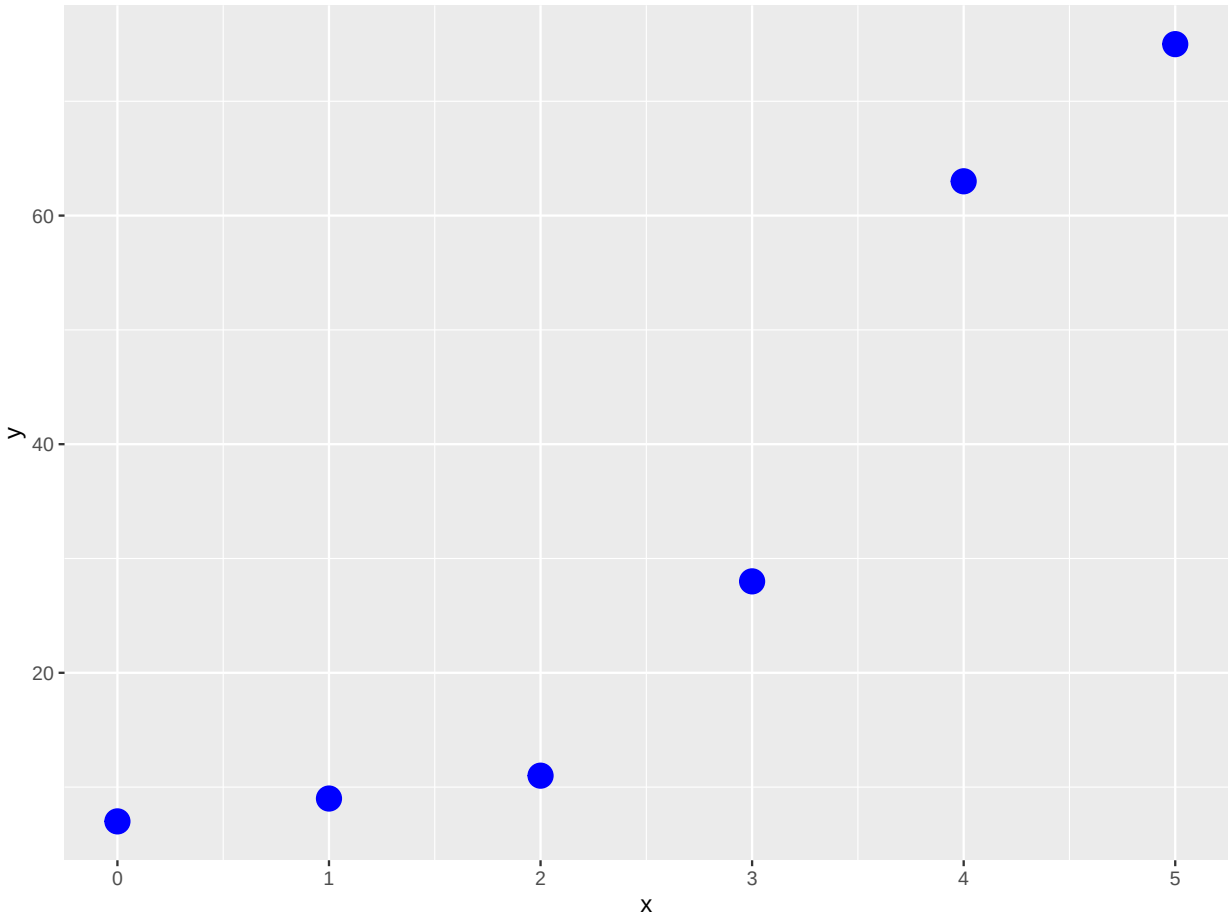
```
poly_lagrange <- function (x, y) {
if( length (x) != length (y))
stop ("El largo de x y de y deben ser el mismo")
n <- length (x) - 1
vandermonde <- rep (1, length (x))
for(i in 1:n) {
xi <- x^i
vandermonde <- cbind ( vandermonde , xi)
}
beta <- solve ( vandermonde , y)
names ( beta ) <- NULL
return ( beta )
}
```

Creamos un conjunto de datos (distinto al anterior)

```r
x <- c(0, 1, 2, 3, 4, 5)
y <- c(7, 9, 11, 28, 63, 75)

df <- data.frame(cbind(x, y))

library(ggplot2)
ggplot(df, aes(x=x, y=y)) + geom_point(size=5, col='blue')
```



Aplicamos la función creada `poly_lagrange`

```r
poly_lagrange(x, y)
```

```
## [1]  7.0000000  3.6000000  0.3333333 -3.8333333  2.1666667 -0.2666667
```

Lo que nos entrega el polinomial:

$$7 + 3{,}6x + 0{,}3333333x^2 - 3{,}833333x^3 + 2{,}166667x^4 - 0{,}2666667x^5$$

Comprobamos nuestros resultados mediante la librería `polynom`:

```
# install.packages("polynom")
library(polynom)
poly.calc(x,y)
```

```
## 7 + 3.6*x + 0.3333333*x^2 - 3.833333*x^3 + 2.166667*x^4 - 0.2666667*x^5
```

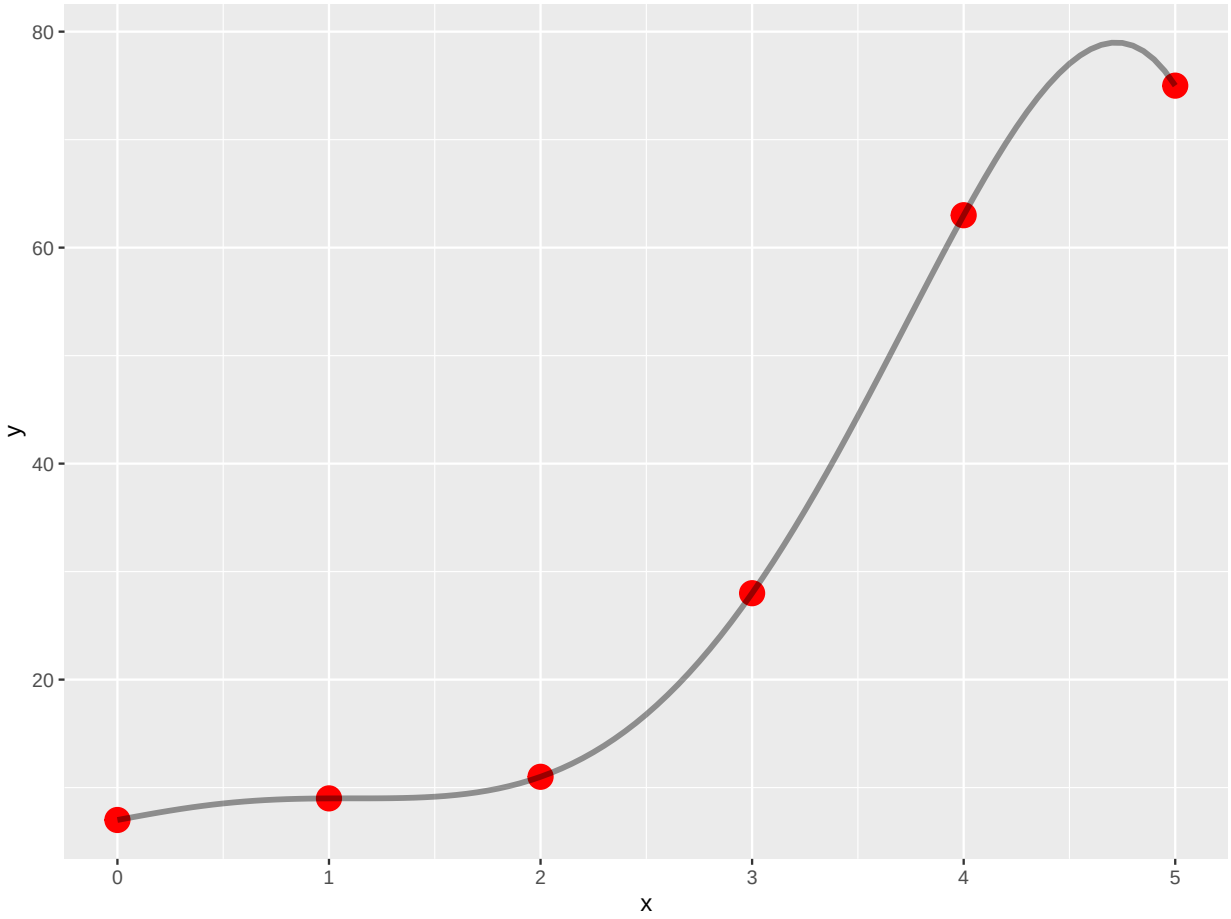
Finalmente, podemos gráficar el polinomial en los puntos dados:

```
x <- c(0, 1, 2, 3, 4, 5)
y <- c(7, 9, 11, 28, 63, 75)

df <- data.frame(cbind(x, y))


g <- function(x) {
     return(7 + 3.6*x + 0.333333*x^2 - 3.83333*x^3 + 2.166667*x^4 - 0.2666667*x^5)
}

ggplot(df, aes(x=x, y=y)) + geom_point(size=5, col='red') +
                            stat_function(fun = g, size=1.25, alpha=0.4)
```



Ejemplo 3

Interpolación polinomial lineal por partes

Un interpolante polinomial por partes tiene un mayor "impacto" cuando la función resultante es continua, ya que una función continua tiene trazos "suaves" (*smoothers*) de una región a otra, y esto tiene un comportamiento mucho más natural. El proceso de interpolación polinomial lineal por partes utiliza el mismo proceso que el modelo de interpolación lineal, excepto que lo repite para cada par consecutivo de puntos de datos a lo largo del eje x. La siguiente función implementada en `R` proporciona ejemplo de este proceso.

```
poly_lin_inter <- function (x, y) {
n <- length (x) - 1
y <- y[ order (x)]
x <- x[ order (x)]
mvec <- bvec <- c()
for(i in 1:n) {
p <- linterp (x[i], y[i], x[i + 1], y[i + 1])
```

```
mvec <- c(mvec , p [2])
bvec <- c(bvec , p [1])
}
return ( list (m = mvec , b = bvec ))
}
```

La función `poly_lin_inter` esta considerando como argumentos a x e y (ambos vectores). Note como dentro de la función estamos utilizando la función `linterp` ya creada previamente. La función entrega dos resultad,s un vector m que contiene los coeficientes asociados a cada elemento de la función, y un vector b que corresponde a los interceptos de cada componente.

```
x <- c(-2, -1, 0, 1, 2, 3, 4)
y <- c(-1, -2, -1, 2, 1, -2, 2, -2)
poly_lin_inter(x, y)
```

```
## $m
## [1] -1  1  3 -1 -3  4
##
## $b
## [1]  -3  -1  -1   3   7 -14
```

Aunque la función `poly_lin_inter`() no hace ninguna supuesto sobre los límites fuera de los puntos datos, existen varias estrategias para hacerlo según el problema a interpolar. Por ejemplo, una opción podría ser asumir que los extremos de la función por partes continúan indefinidamente.

En `R` se puede utilizar la función `approx`() para resolver un problema de interpolación polinomial lineal por partes. Esta función funciona aceptando una lista de puntos para interpolar y una lista de puntos para valores para aproximar (interpolar). Entonces, la función `approx`, en lugar de devolver los parámetros de una función por partes, devuelve valores y asociados con los puntos en cuestión.

```
x <- c(-2, -1, 0, 1, 2, 3, 4)
y <- c(-1, -2, -1, 2, 1, -2, 2)
f <- approx(x, y)
f
```

```
## $x
##  [1] -2.00000000 -1.87755102 -1.75510204 -1.63265306 -1.51020408 -1.38775510
##  [7] -1.26530612 -1.14285714 -1.02040816 -0.89795918 -0.77551020 -0.65306122
## [13] -0.53061224 -0.40816327 -0.28571429 -0.16326531 -0.04081633  0.08163265
## [19]  0.20408163  0.32653061  0.44897959  0.57142857  0.69387755  0.81632653
## [25]  0.93877551  1.06122449  1.18367347  1.30612245  1.42857143  1.55102041
## [31]  1.67346939  1.79591837  1.91836735  2.04081633  2.16326531  2.28571429
## [37]  2.40816327  2.53061224  2.65306122  2.77551020  2.89795918  3.02040816
## [43]  3.14285714  3.26530612  3.38775510  3.51020408  3.63265306  3.75510204
## [49]  3.87755102  4.00000000
##
```

```
## $y
##  [1] -1.00000000 -1.12244898 -1.24489796 -1.36734694 -1.48979592 -1.61224490
##  [7] -1.73469388 -1.85714286 -1.97959184 -1.89795918 -1.77551020 -1.65306122
## [13] -1.53061224 -1.40816327 -1.28571429 -1.16326531 -1.04081633 -0.75510204
## [19] -0.38775510 -0.02040816  0.34693878  0.71428571  1.08163265  1.44897959
## [25]  1.81632653  1.93877551  1.81632653  1.69387755  1.57142857  1.44897959
## [31]  1.32653061  1.20408163  1.08163265  0.87755102  0.51020408  0.14285714
## [37] -0.22448980 -0.59183673 -0.95918367 -1.32653061 -1.69387755 -1.91836735
## [43] -1.42857143 -0.93877551 -0.44897959  0.04081633  0.53061224  1.02040816
## [49]  1.51020408  2.00000000
```

La función `approxfun`() además nos entrega una función ejecutable para evaluar cualquier tipo de valores x dados, por ejemplo

```
x <- c(-2, -1, 0, 1, 2, 3, 4)
y <- c(-1, -2, -1, 2, 1, -2, 2)
f <- approxfun(x, y)
f(1)
```

```
## [1] 2
```

```
f(2)
```

```
## [1] 1
```