Regresión Lineal Simple utilizando técnicas machine learning

Contenido

La Regresión Lineal Simple

Es un modelo de regresión en donde una función lineal representa la relación existente entre una variable dependiente y su respectiva variable independiente. Es decir, la ecuación que describe el modelo adopta la forma $y=ax+b$, en donde y es la variable dependiente, x es la variable independiente, a y b son los coeficientes de la recta (pendiente y punto de corte, respectivamente) que, bajo algún criterio de minimización como el de mínimos cuadrados, ofrece el mejor ajuste a los datos de entrada.

Ingreso de datos

```
# regresion lineal simple con machine learning
# ejemplo: numero de bateos y numero de runs
numero_bateos <- c(5659,5710,5563,5672,5532,5600,5518,5447,5544,5598,
                   5585,5436,5549,5612,5513,5579,5502,5509,5421,5559,
                   5487,5508,5421,5452,5436,5528,5441,5486,5417,5421,
                   5660,5711,5565,5662,5533,5602,5519,5450,5546,5597,
                   5587,5437,5548,5615,5517,5574,5505,5503,5422,5560,
                   5487,5508,5421,5452,5436,5528,5441,5486,5417,5421)
runs <- c(855,875,787,730,762,718,967,721,735,615,708,644,654,735,667,
          713,654,704,731,743,619,625,610,645,708,642,625,575,594,557,
          858,878,789,740,765,728,968,724,736,617,705,646,656,736,668,
          715,658,708,735,746,622,627,612,647,709,644,627,578,598,558)
```

Creando un data.frame

```
datos <- data.frame(numero_bateos, runs)
```

observando datos

```
head(datos)
```

```
##   numero_bateos runs
## 1          5659  855
## 2          5710  875
## 3          5563  787
## 4          5672  730
```

```
## 5           5532  762
## 6           5600  718
```

Solicitando de la función str, podemos explorar la estructura del *dataframe* que contiene ambos conjuntos:

```
str(datos)
```

```
## 'data.frame':    60 obs. of  2 variables:
##  $ numero_bateos: num  5659 5710 5563 5672 5532 ...
##  $ runs         : num  855 875 787 730 762 718 867 721 735 615 ...
```

La data contiene 60 observaciones y dos variables

Utilizando la librería caret para dividir la data en data de entrenamiento (70%) y data de validación del modelo (30%).

```
library(ggplot2)
library(lattice)
library(caret)
set.seed(100)  # Para reproducir los mismos resultados

# dividiendo la data en train y test
grupos <- createDataPartition(y = datos$runs,p = 0.8,list = FALSE)
train <- datos[grupos,]
Test <- datos[-grupos,]
```

Observando el contenido de la data dividida

```
str(train)
```

```
## 'data.frame':    49 obs. of  2 variables:
##  $ numero_bateos: num  5659 5563 5600 5518 5447 ...
```

```
##  $ runs          : num  855 787 718 967 721 735 708 644 654 735 ...
```

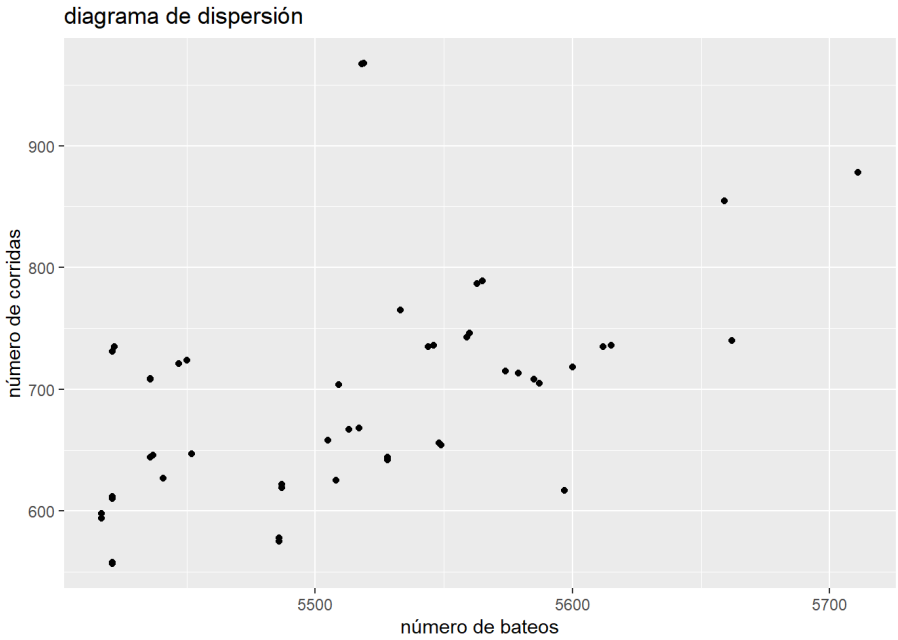
```
str(Test)
```

```
## 'data.frame':    11 obs. of  2 variables:
##  $ numero_bateos: num  5710 5672 5532 5598 5502 ...
##  $ runs          : num  875 730 762 615 654 645 625 858 728 708 ...
```

Como podemos ver, el conjunto *train* contiene 49 observaciones, mientras que el conjunto *test* tiene 11 observaciones. Usaremos los datos de entrenamiento (train) para *entrenar* el modelo de regresión, y luego utilizaremos los datos de validación (test) para probar la calidad del modelo.

A fin de tener una idea más clara de la forma de dispersión de los datos, vamos a graficar el conjunto de entrenamiento. Para ello hagamos uso de la función ggplot de la librería ggplot2:

```
library(ggplot2)
ggplot() + geom_point(data = train, aes(x = numero_bateos, y = runs)) +
  geom_point() +
  xlab("número de bateos") +
  ylab("número de corridas") +
  ggtitle("diagrama de dispersión (data train")
```



Como podemos observar, el conjunto de datos está formado por una serie de puntos que podría tener una dependencia lineal entre ellos, además muestra posibles datos outliers.

Aplicar una regresión sobre estos datos implica obtener la línea recta que mejor ajuste la relación existente entre la variable independiente y la variable dependiente. Para ello, vamos a crear un *regresor* haciendo uso de la función lm del paquete stats:

```
# regresion lineal simple
set.seed(1234)
regresor <- lm(runs ~ numero_bateos, data = train)
regresor
```

```
## Call:
## lm(formula = runs ~ numero_bateos, data = train)
##
## Coefficients:
##   (Intercept)  numero_bateos
##    -2732.7149         0.6217
```

Como primer argumento de lm se coloca la fórmula variable dependiente ~ variable independiente. Como segundo argumento se indica el conjunto de datos que se usará para construir el modelo y un resumen de los estimadores (coeficientes) obtenidos.

Una vez creada la ecuación, podemos explorar los resultados y calidad del ajuste haciendo uso de la función summary:

```
summary(regresor)
```

```
## Call:
## lm(formula = runs ~ numero_bateos, data = train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -129.67  -47.08  -22.48   58.12  269.82
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -2732.7149   847.5196  -3.224 0.002298 **
## numero_bateos     0.6217     0.1537   4.046 0.000193 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 78.93 on 47 degrees of freedom
## Multiple R-squared:  0.2583, Adjusted R-squared:  0.2425
## F-statistic: 16.37 on 1 and 47 DF,  p-value: 0.000193
```

Entre los datos de interés que ofrece la función summary están las estadísticas de los *residuales* (es decir, de las *distancias* entre los valores de la variable *y* del conjunto de datos original y sus proyecciones con el modelo), los valores de los coeficientes obtenidos, sus t-values y el *p-value* (relevancia estadística de la variable independiente como elemento predictivo, y que aparece representado en el reporte como *Pr(>|t|)*). La parte inferior muestra el error estándar de los residuales (78.93), el coeficiente de determinación (calidad del ajuste R^2) que en nuestro caso es de 0.2583(25.83%), el R^2 ajustado y los resultados del Análisis de Varianza del modelo. En este caso es significativa, es decir el modelo en su conjunto es bueno.

Intervalos de confianza para los coeficientes del modelo.

```
confint(regresor)
```

```
##                       2.5 %        97.5 %
## (Intercept)   -4437.7043552 -1027.7254066
## numero_bateos     0.3125241     0.9307792
```

Ya que tenemos el modelo construido, vamos a crear un vector de predicciones basado en el propio conjunto de entrenamiento, y con este vector podremos visualizar la curva de ajuste de los datos. Para obtener las predicciones, hacemos uso de la función predict:

```
y_predict <- predict(regresor, train)
head(y_predict)
```

```
##        1        3        6        7        8        9
## 785.2117 725.5332 748.5343 697.5588 653.4216 713.7218
```

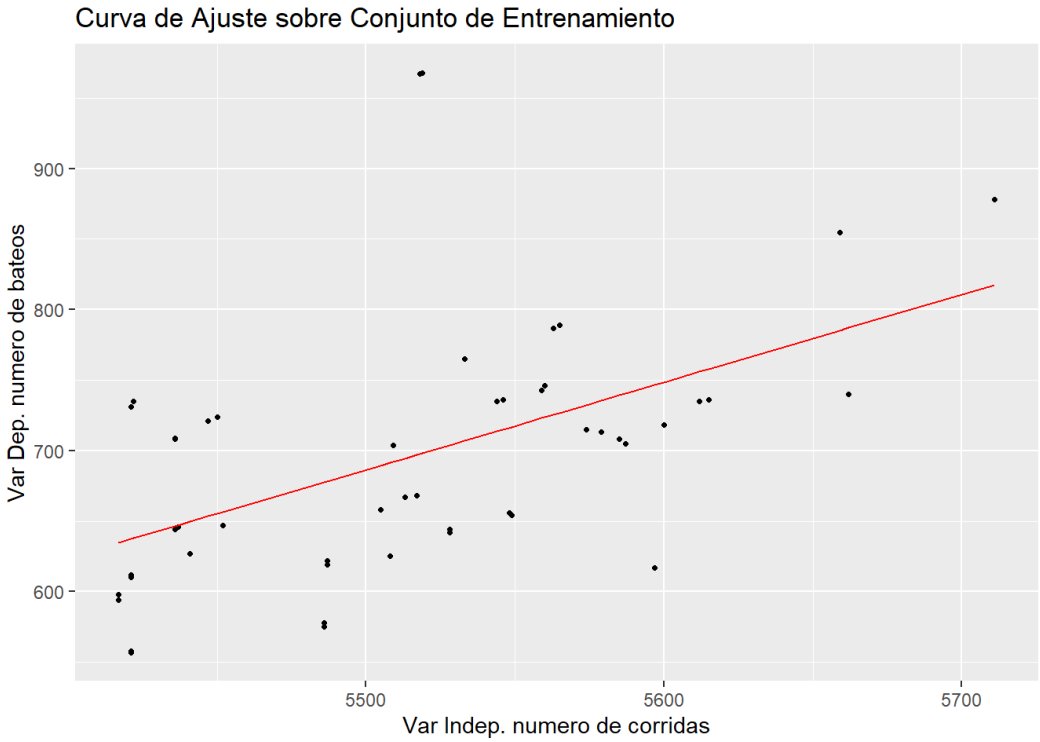
```
Head(cbind(train,y_predict))
```

```
##   numero_bateos runs y_predict
## 1          5659  855  785.2117
## 3          5563  787  725.5332
## 6          5600  718  748.5343
## 7          5518  967  697.5588
## 8          5447  721  653.4216
## 9          5544  735  713.7218
```

Recta de regresión.

```
ggplot() + geom_point(data = train, aes(x = numero_bateos, y = runs), size =
0.9) +
```

```
  geom_line(aes( x = train$numero_bateos, y = y_predict), color = "red") +
  xlab("Var Indep. numero de corridas") +
  ylab("Var Dep. numero de bateos") +
  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento")
```



En efecto, se observa que la curva producida reproduce el comportamiento lineal de los datos de entrenamiento y que la cantidad de puntos tanto por encima como por debajo de esta es semejante, lo que refleja la calidad del ajuste.

Con el modelo *regresor*, vamos a realizar predicciones sobre el conjunto de validación, el cual contiene puntos que no fueron usados durante el entrenamiento:

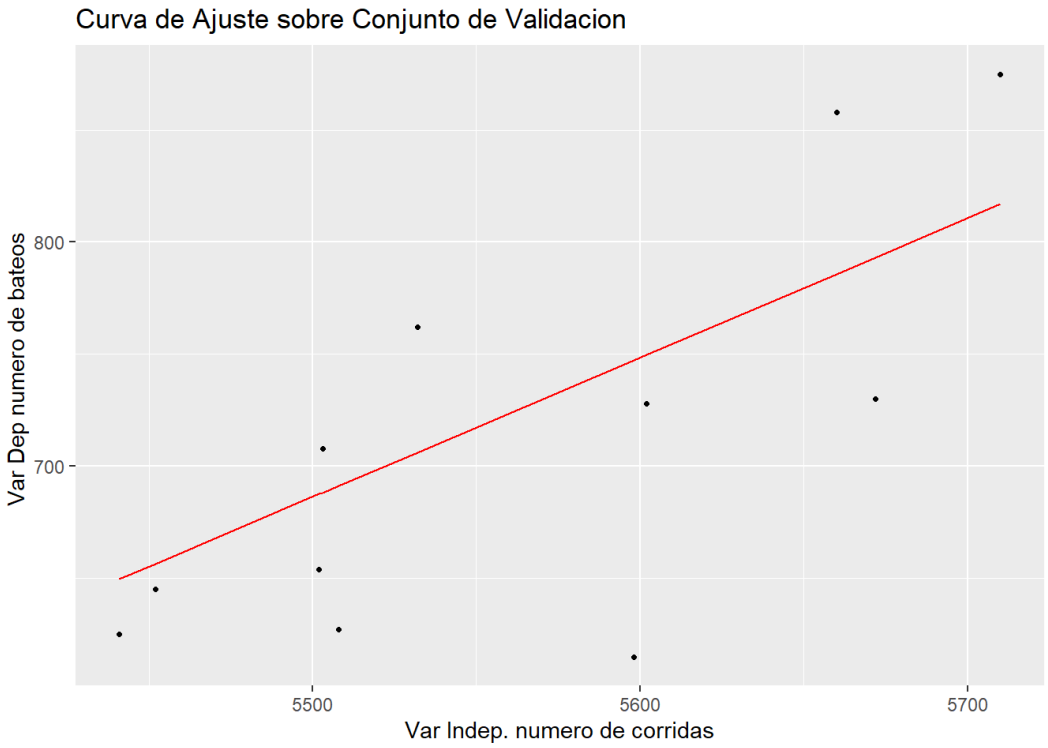
```
y_test_predict <- predict(regresor, Test)
y_test_predict
```

```
##        2        4        5       10       17       24       27       31
## 816.9159 793.2932 706.2620 747.2910 687.6124 656.5298 649.6917 785.8334
##       36       48       52
```

```
## 749.7776 688.2341 691.3423
```

Graficando

```
ggplot() + geom_point(data = Test, aes(x = numero_bateos, y = runs), size = 0
.9) +
  geom_line(aes( x = Test$numero_bateos, y = y_test_predict), color = "red")
+
  xlab("Var Indep. numero de corridas") +
  ylab("Var Dep número de bateos ") +
  ggtitle("Curva de Ajuste sobre Conjunto de Validación")
```




```
# Cálculo de los errores
error = y_test_predict - Test$runs
error
```

```
##         2         4         5        10        17        24        27        31
```

```
## -58.08405  63.29318 -55.73804 132.29096  33.61241  11.52982  24.69166 -72.16664
##        36        48        52
##  21.77757 -19.76594  64.34232
```

```
# Cálculo del error cuadrático medio RMSE:
sqrt(mean(error^2))
```

```
## [1] 60.35707
```

Ahora, veamos la correlación existente entre los valores *y* del conjunto de entrenamiento, y los valores predichos para dicho conjunto. Para ello, usamos la función cor:

```
cor(Test$runs, y_test_predict)
```

```
## [1] 0.7448835
```

El valor de 0.744835 indica una alta correlación y, por lo tanto, un buen ajuste a los datos por parte del modelo obtenido.

Por último, ya que tenemos nuestro modelo regresor listo, si deseáramos tener una predicción particular para un valor cualquiera de *x*, podemos hacerlo de la siguiente manera:

```
predict_value <- predict(regresor, data.frame(numero_bateos= c(5508)))
predict_value
```

```
##        1
## 671.7217
```

En este punto es importante mencionar que la función predict espera como argumento de la variable independiente un *dataframe*.

```
# prediciendo una secuencia
predict_value2 <- predict(regresor, data.frame(numero_bateos=seq(5508,5510)))
predict_value2
```

```
##    1        2        3
## 671.7217 672.4617 673.2017
```

Regresión Polinomial (Polynomial Regression)

Analizando los datos con un modelo polinomial.
Cuando se habla de Regresión Polinomial, se busca producir entonces una ecuación de ajuste de la variable dependiente - independiente que tenga la forma:

$$y = a + bx + cx^2 + \cdots + Nxn$$

en donde n es el grado del polinomio. Es decir, se introducen al modelo términos polinomiales de la variable independiente hasta lograr el mejor ajuste a los datos.

Usaremos los conjuntos de datos de entrenamiento y validación. Seguidamente, vamos a introducir, un término de segundo grado a nuestros datos de entrenamiento a fin de que el regresor pueda ser polinomial:

$$y = a + bx + cx^2$$

Generando el cuadrado de los datos en la data de entrenamiento (train)

```
train$numero_bateos2 <- train$numero_bateos^2
str(train)
```

```
## 'data.frame':    49 obs. of  3 variables:
##  $ numero_bateos : num  5659 5563 5600 5518 5447 ...
##  $ runs          : num  855 787 718 967 721 735 708 644 654 735 ...
##  $ numero_bateos2: num  32024281 30946969 31360000 30448324 29669809 ...
```

Al inspeccionar de nuevo el conjunto de entrenamiento, Vemos que al *dataframe* se le incorporó una columna que representa el cuadrado de la variable independiente x.

Ahora, podemos construir nuestro modelo de regresión polinomial:

```
regresor_poly <- lm(runs ~ numero_bateos + numero_bateos2, data = train)
summary(regresor_poly)
```

Como puede verse, en este caso la fórmula del primer argumento y ~ x + x2 incluye la nueva columna.

Al aplicar la función summary a nuestro nuevo regresor tenemos:

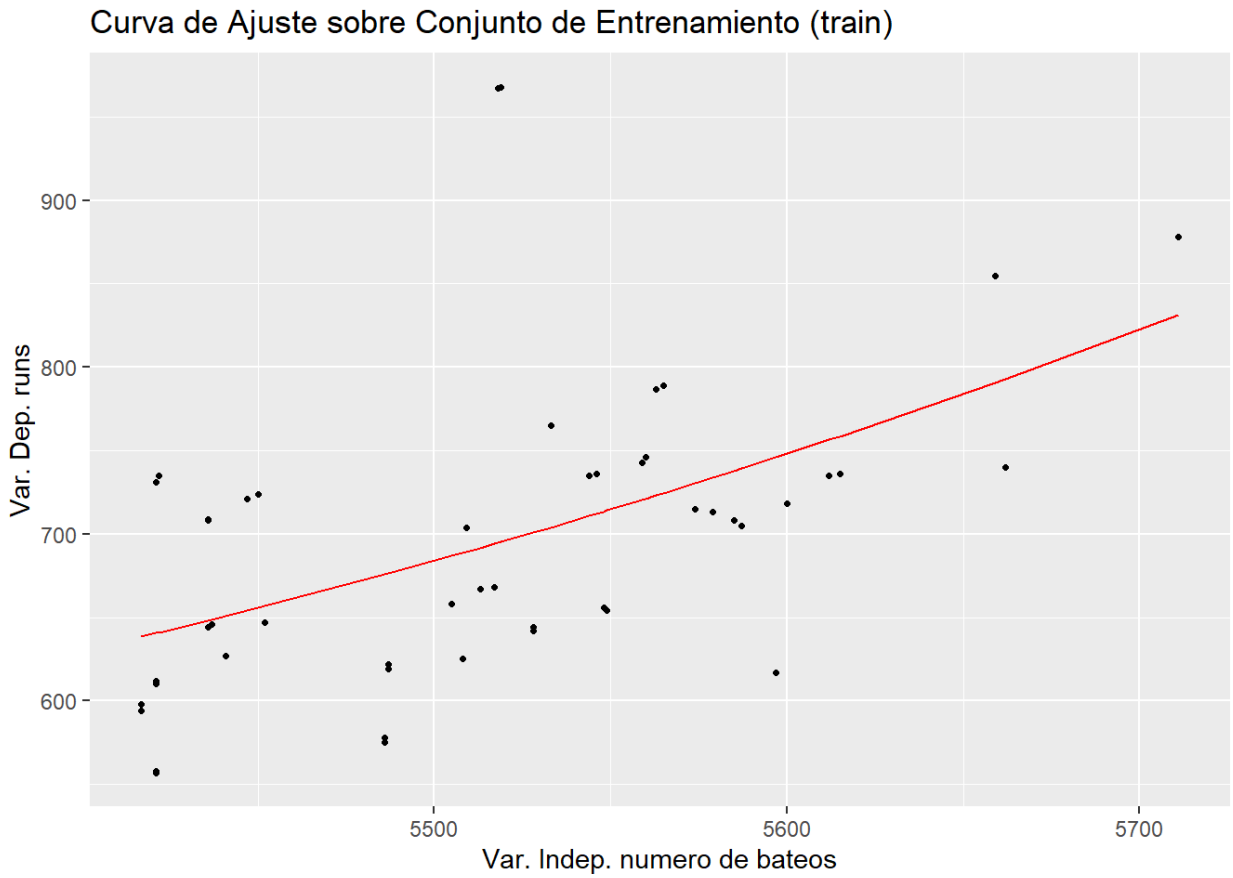
```
## Call:
## lm(formula = runs ~ numero_bateos + numero_bateos2, data = train)
##
## Residuals:
##    in       1Q     Median     3Q     Max
## -129.02  -53.03  -22.60   46.64  272.65
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)     1.303e+04  5.498e+04   0.237    0.814
## numero_bateos  -5.081e+00  1.988e+01  -0.256    0.799
## numero_bateos2  5.155e-04  1.798e-03   0.287    0.776
##
## Residual standard error: 79.71 on 46 degrees of freedom
## Multiple R-squared:  0.2596, Adjusted R-squared:  0.2274
## F-statistic: 8.065 on 2 and 46 DF,  p-value: 0.0009945
```

Como puede verse, tanto la variable x como la variable x^2 no son relevantes a efectos de la predicción (valores mayores del *p-value*) y se obtiene un R^2 de 0.2596 (25.96%). La prueba ANVA muestra significancia en cuanto al modelo en su conjunto $p(0.0009945) < \alpha(0.05)$.

Obtengamos las predicciones para el conjunto de entrenamiento y elaboremos la gráfica del modelo obtenido:

```
y_poly_predict <- predict(regresor_poly, train)
```

```
ggplot() + geom_point(data = train, aes(x = numero_bateos, y = runs), size =
0.9) +
  geom_line(aes( x = train$numero_bateos, y = y_poly_predict), color = "red")
+
  xlab("Var. Indep. numero de bateos") +
  ylab("Var. Dep. runs") +
  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento (train)")
```



En efecto, vemos que la curva de ajuste producida por la regresión sigue la forma no lineal de los datos de entrenamiento.

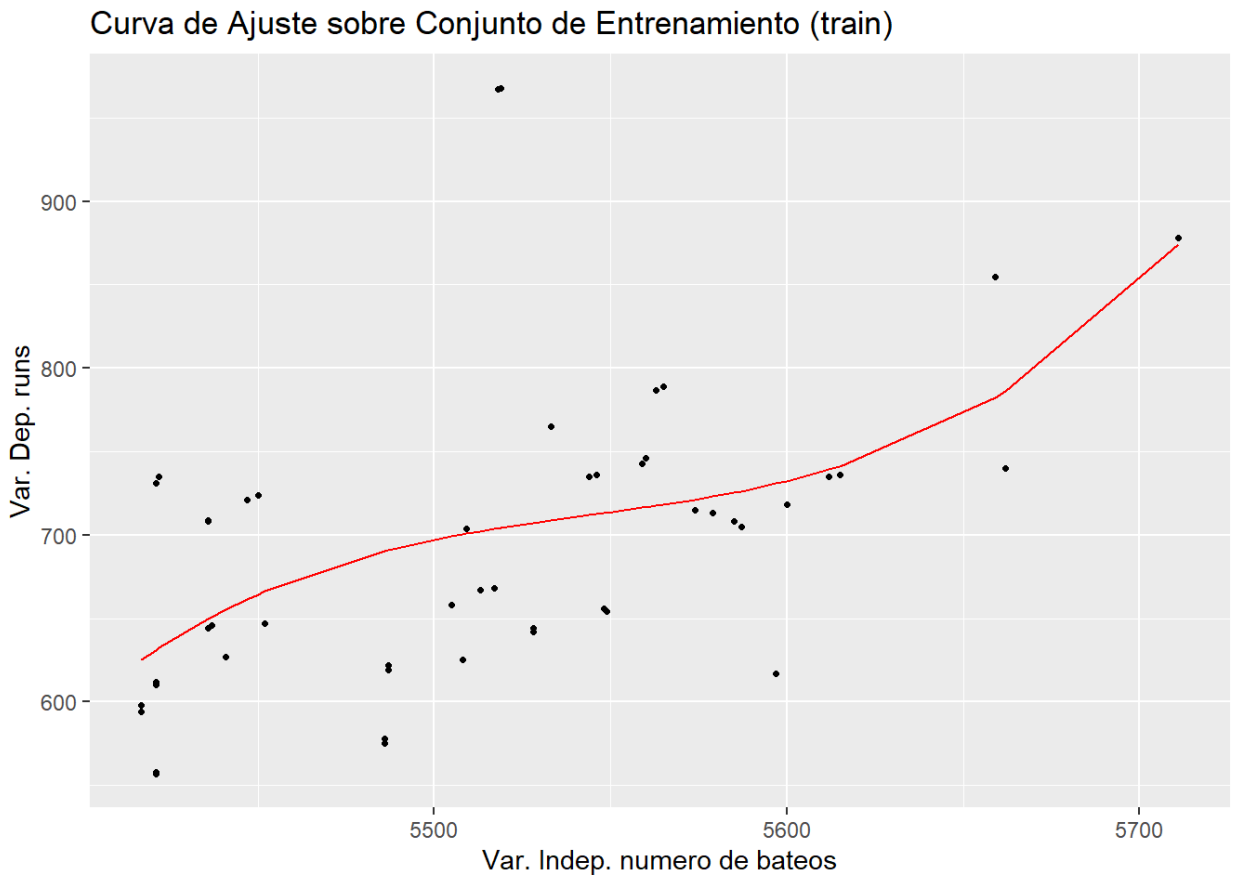
Ahora, ¿qué ocurre si incorporamos más órdenes al polinomio? :

```
train$numero_bateos3 <- train$numero_bateos^3

regresor_poly <- lm(runs ~ numero_bateos + numero_bateos2 + numero_bateos3, d
ata = train)

summary(regresor_poly)
```

```
## Call:
## lm(formula = runs ~ numero_bateos + numero_bateos2 + numero_bateos3,
##     data = train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -115.42  -46.26  -14.40   29.36  263.67
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -4.172e+06  4.119e+06  -1.013    0.317
## numero_bateos   2.257e+03  2.227e+03   1.014    0.316
## numero_bateos2 -4.071e-01  4.012e-01  -1.015    0.316
## numero_bateos3  2.448e-05  2.409e-05   1.016    0.315
##
## Residual standard error: 79.69 on 45 degrees of freedom
## Multiple R-squared:  0.2762, Adjusted R-squared:  0.228
## F-statistic: 5.724 on 3 and 45 DF,  p-value: 0.002089
```

y_poly_predict <- predict(regresor_poly, train))

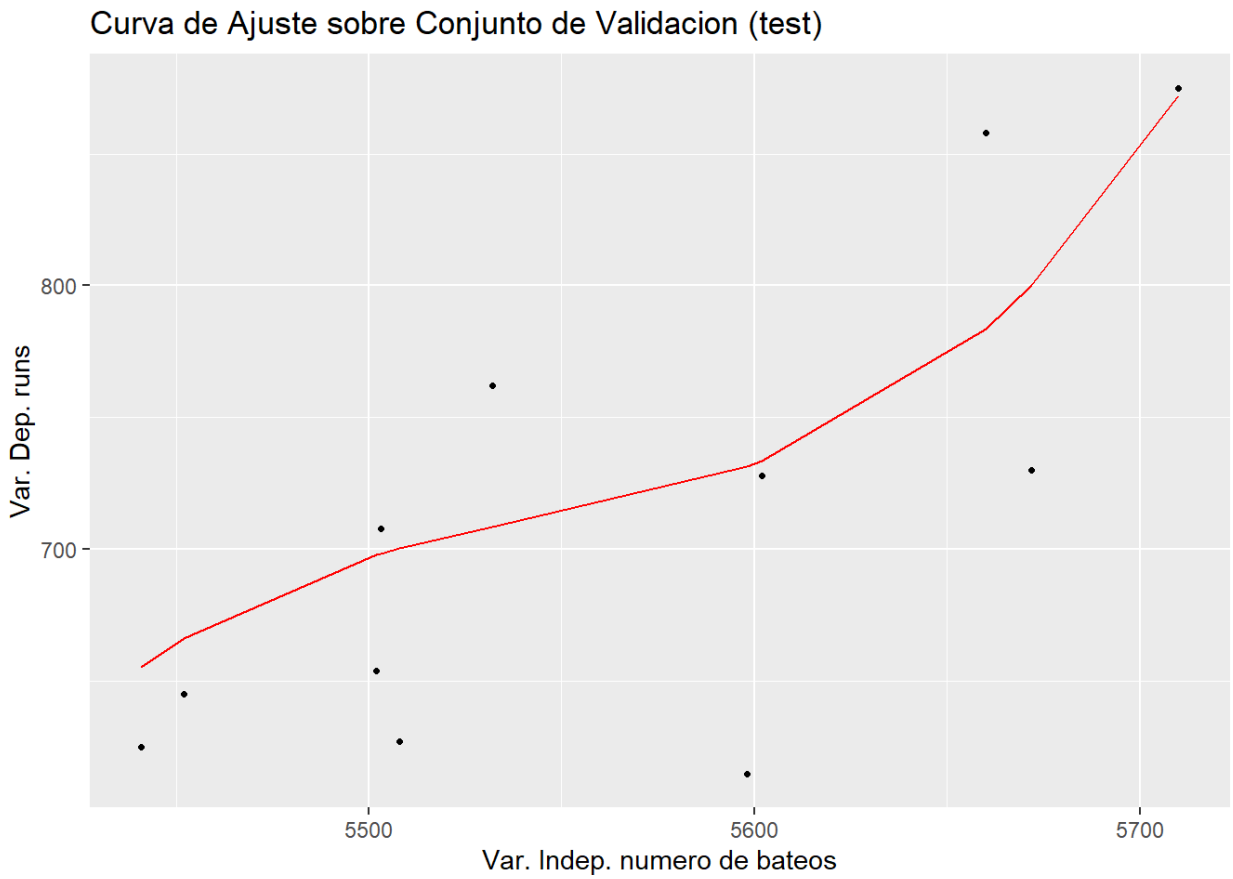
```
y_poly_predict <- predict(regresor_poly, train)


ggplot() + geom_point(data = train, aes(x = numero_bateos, y = runs), size =
0.9) +
  geom_line(aes( x = train$numero_bateos, y = y_poly_predict), color = "red")
+
  xlab("Var. Indep. numero de bateos") +
  ylab("Var. Dep. runs") +
  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento (train)")
```



Ahora, tenemos que la curva de tercer grado. El valor de R^2 es 0.3716.

En función de este resultado, podemos incorporar los elementos polinomiales al conjunto de validación, y aplicar la regresión respectiva:

```
Test$numero_bateos2 <- Test$numero_bateos^2

Test$numero_bateos3 <- Test$numero_bateos^3

y_poly_test_predict <- predict(regresor_poly, Test)


ggplot() + geom_point(data = Test, aes(x = numero_bateos, y = runs), size = 0
.9) +

  geom_line(aes( x = Test$numero_bateos, y = y_poly_test_predict), color = "r
ed") +

  xlab("Var. Indep. numero de bateos") +

  ylab("Var. Dep. runs") +

  ggtitle("Curva de Ajuste sobre Conjunto de Validacion (test)")
```



Las curvas de ajuste producidas, tanto para los datos de entrenamiento como para validación, reproducen entonces el comportamiento no lineal de los datos, y sirven como modelos para predecir cualquier valor nuevo de la variable independiente que se tome, por ejemplo:

```
predict_value_poly <- predict(regresor_poly, data.frame(numero_bateos = 5508,
                                                         numero_bateos2 = 5508
^2,
                                                         numero_bateos3 = 5508
^3))
predict_value_poly
```

```
##        1
## 700.3946
```

Obsérvese que, como parámetro de predicción, se debe colocar el *dataframe* como la variable independiente seleccionada, así como sus respectivas potencias.

```
summary(regresor_poly)
```

```
## Call:
## lm(formula = runs ~ numero_bateos + numero_bateos2 + numero_bateos3,
##     data = train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -115.42  -46.26  -14.40   29.36  263.67
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)    -4.172e+06  4.119e+06  -1.013    0.317
## numero_bateos   2.257e+03  2.227e+03   1.014    0.316
## numero_bateos2 -4.071e-01  4.012e-01  -1.015    0.316
## numero_bateos3  2.448e-05  2.409e-05   1.016    0.315
##
## Residual standard error: 79.69 on 45 degrees of freedom
## Multiple R-squared:  0.2762, Adjusted R-squared:  0.228
## F-statistic: 5.724 on 3 and 45 DF,  p-value: 0.002089
```

```
y_predict2 <- predict(regresor_poly, train)
y_test_predict2 <- predict(regresor_poly, Test)

# Cálculo del error
error = y_test_predict2 - Test$runs
error
```

```
##           2           4           5          10          17          24
27
##  -3.085662  70.267129 -53.582229 116.375997  43.992911  21.337535  30.4830
96
##          31          36          48          52
## -74.334356   5.453844  -9.592263  73.394590
```

```
# Cálculo del error cuadrático medio RMSE:
sqrt(mean(error^2))
```

```
## [1] 56.9812
```

Regresión con Vectores de Soporte o SVR (Support Vector Regression)

La Regresión con Vectores de Soporte o SVR por sus siglas en inglés, es un modelo de regresión basado en las Máquinas de Vectores de Soporte y que, groso modo, son modelos capaces de generar clasificaciones o regresiones de datos no lineales a partir de la transformación de los datos de entrada a otros espacios de mayores dimensiones. En el caso de la regresión, la SVR busca encontrar aquella curva que sea capaz de ajustar los datos garantizando que la separación entre ésta y ciertos valores específicos del conjunto de entrenamiento (los vectores de soporte) sea la mayor posible.

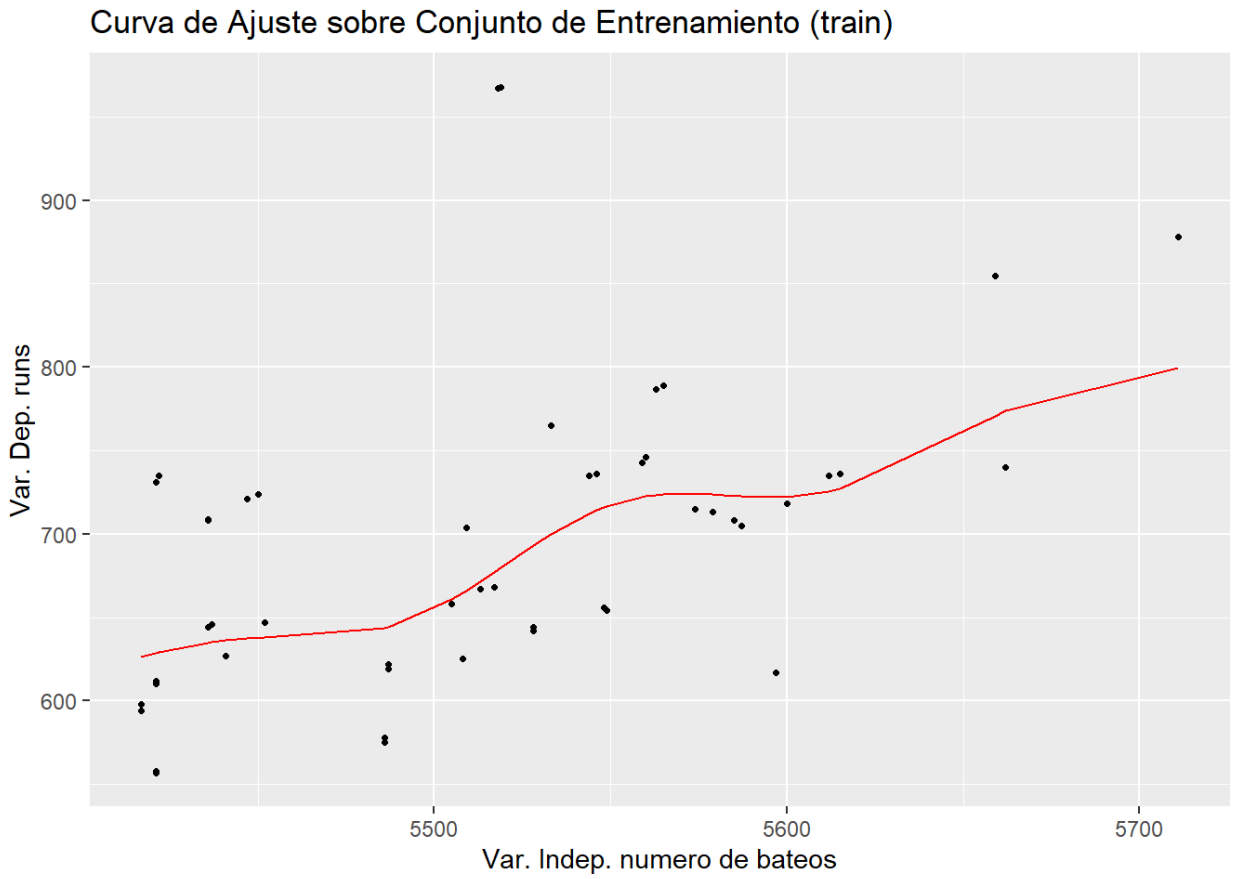
En nuestro caso, haremos uso de la función svm del paquete e1071 para crear el modelo de regresión SVR:

```
library(e1071)
set.seed(1234)
regresor_svr <- svm(runs ~numero_bateos , data = train, type = "eps-regressio
n")
```

Como puede verse, como fórmula se introduce la relación entre y - x (sin considerar los elementos polinomiales que se introdujeron en la regresión anterior). El parámetro eps-regression es específico para regresión, pues con svm se pueden realizar modelos tanto de regresión como clasificación.

Ya que tenemos el regresor SVR, vamos a realizar las predicciones sobre el conjunto de entrenamiento y visualizar la curva:

```
y_svr_predict <- predict(regresor_svr, train)


ggplot() + geom_point(data = train, aes(x = numero_bateos, y = runs), size =
0.9) +
  geom_line(aes( x = train$numero_bateos, y = y_svr_predict), color = "red")
+
  xlab("Var. Indep. numero de bateos") +
  ylab("Var. Dep. runs") +
  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento (train)")
```



Obtengamos una correlación entre datos de entrenamiento y la predicción de:

```
cor(train$runs, y_svr_predict)
```

```
## [1] 0.535423
```

Como puede verse en la curva, el regresor SVR produce un resultado no lineal a pensar de no estar construido con componentes polinomiales. El desempeño del regresor SVR dependerá en gran medida de los *hiperparámetros* del modelo, que en este caso pueden conocerse al usar la función summary:

```
summary(regresor_svr)
```

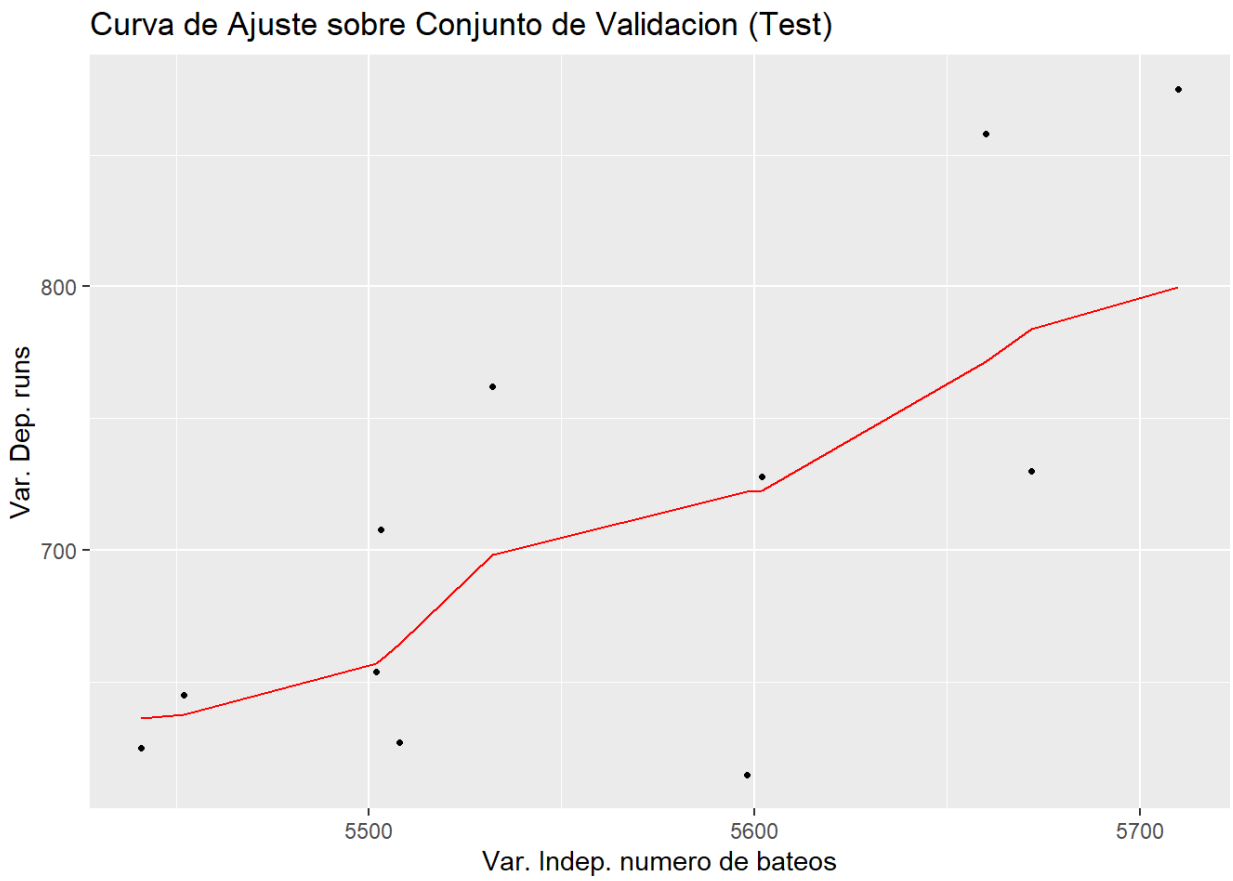
```
## Call:
## svm(formula = runs ~ numero_bateos, data = train, type = "eps-regression")
##
##
## Parameters:
##    SVM-Type:  eps-regression
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  1
##     epsilon:  0.1
##
##
## Number of Support Vectors:  46
```

Por defecto, cost, gamma y epsilon tiene los valores mostrados, pero ajustando dichos parámetros es posible obtener un mejor resultado en la regresión. Sin embargo, en este ejercicio no trataremos el tema de las técnicas para ajustar los hiperparámetros de los modelos.

Para nuestro conjunto de validación tendremos:

```
y_svr_test_predict <- predict(regresor_svr, Test)


ggplot() + geom_point(data = Test, aes(x = numero_bateos, y = runs), size = 0
.9) +
```

```
  geom_line(aes( x = Test$numero_bateos, y = y_svr_test_predict), color = "red") +
  xlab("Var. Indep. numero de bateos") +
  ylab("Var. Dep. runs") +
  ggtitle("Curva de Ajuste sobre Conjunto de Validacion (Test)")
```



Y una correlación de:

```
cor(Test$numero_bateos, y_svr_test_predict)
```

```
## [1] 0.9910936
```

¿Cuál será la predicción del SVR para el valor de 5508 probado en el caso polinomial?

```
predict_value_svr <- predict(regresor_svr, data.frame(numero_bateos = 5508))
predict_value_svr
```

```
##        1
## 664.6071
```

```
# Cálculo del error
error = predict_value_svr - Test$runs
error
```

```
##  [1] -210.39294  -65.39294  -97.39294   49.60706   10.60706   19.60706
##  [7]   39.60706 -193.39294  -63.39294  -43.39294   37.60706
```

```
# Cálculo del error cuadrático medio RMSE:
sqrt(mean(error^2))
```

```
## [1] 98.75133
```

Regresión con Árboles de Decisión (Decision Tree Regression)

Los Árboles de Decisión son modelos de predicción que pueden usarse tanto para clasificación como para regresión, y cuyo funcionamiento se basa en la construcción de reglas lógicas (divisiones de los datos entre rangos o condiciones) a partir de los datos de entrada.

El entrenamiento de los árboles de decisión se centra principalmente en la maximización de la ganancia de información al momento de realizar las reglas lógicas que forman el árbol.

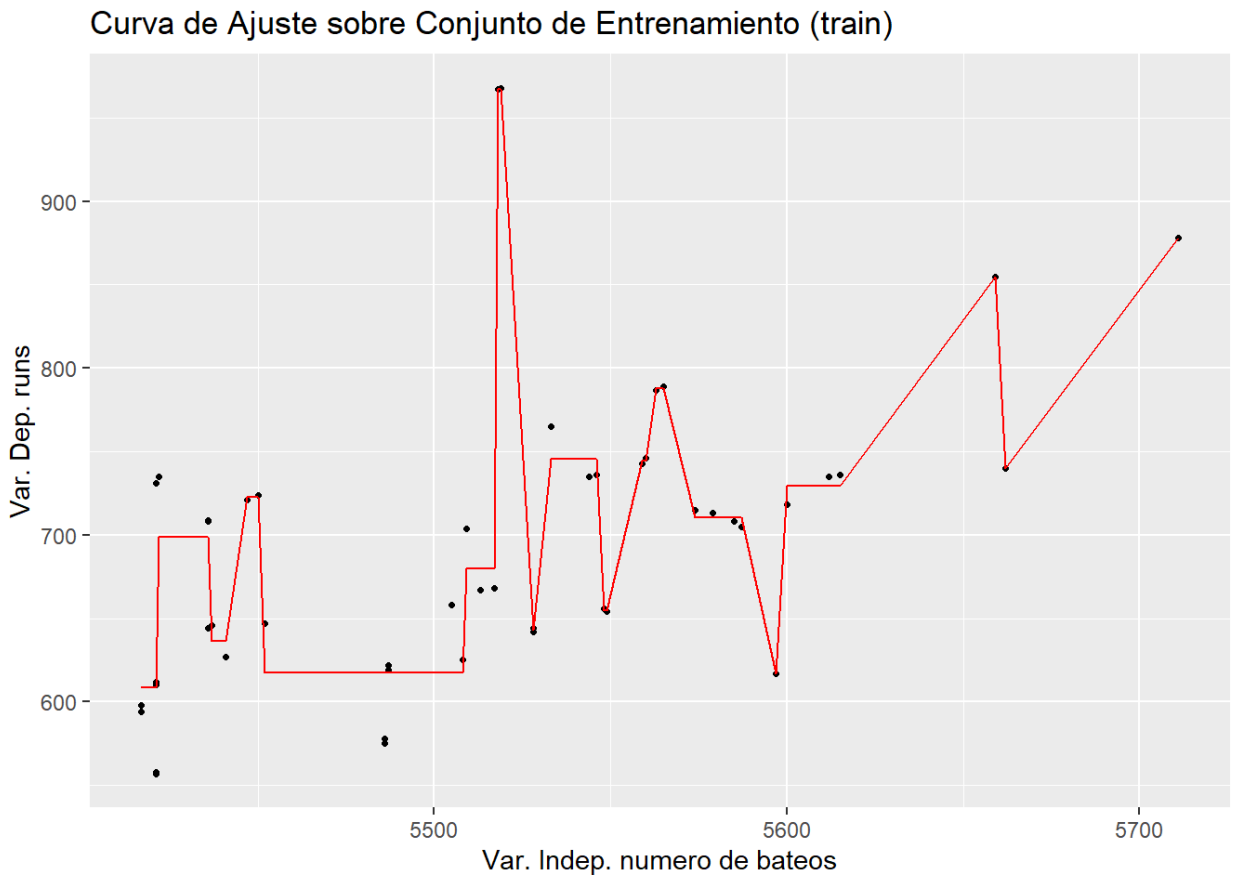
En nuestro ejercicio, usaremos la función rpart del paquete rpart para crear el árbol de regresión:

```
library(rpart)
set.seed(1234)
regresor_decisiontree <- rpart(runs ~ numero_bateos, data = train,
                               control = rpart.control(minsplit = 2))
```

El parámetro control y el valor minsplit igual a 2 establecen que, para el árbol construido, se garantice al menos 2 divisiones de los datos en cada paso del entrenamiento. Esto para asegurarnos que se obtiene al menos una solución adecuada al problema.

Una vez construido el regresor, podemos realizar las predicciones de entrenamiento y visualizar los resultados:

```
y_dt_predict <- predict(regresor_decisiontree, train)


ggplot() + geom_point(data = train, aes(x = numero_bateos, y = runs), size =
0.9) +
  geom_line(aes( x = train$numero_bateos, y = y_dt_predict), color = "red") +
  xlab("Var. Indep. numero de bateos") +
  ylab("Var. Dep. runs") +
  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento (train)")
```



Con una correlación de:

```
cor(train$runs, y_dt_predict)
```

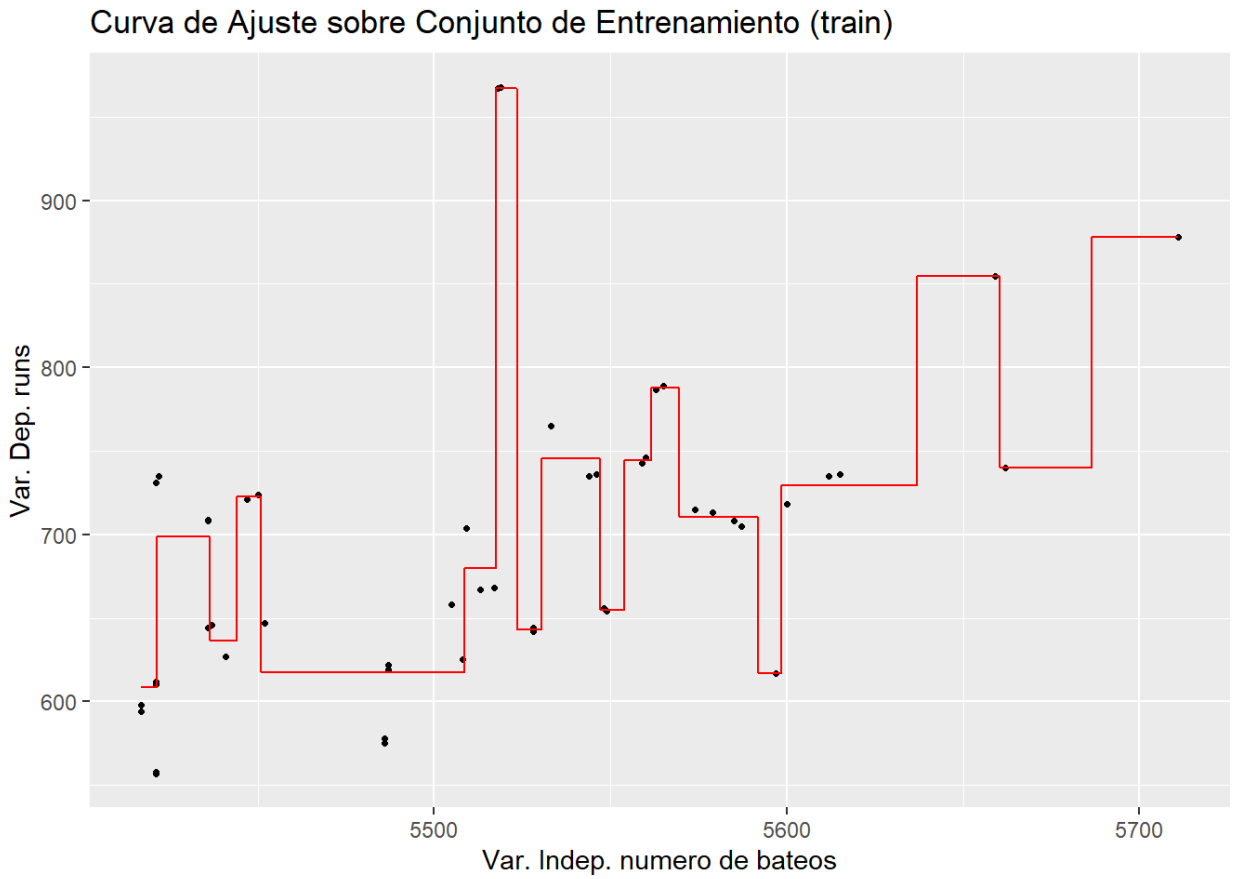
```
## [1] 0.9573812
```

La forma no lineal de la curva de regresión del árbol de decisión generado tiene que ver con el hecho de que, una vez se logran todas las subdivisiones de los datos durante el entrenamiento y se alcanza el árbol definitivo, cada *hoja* del árbol, es decir el extremo en donde se fija el valor final de la regresión (o clasificación), tomará el valor promedio de todos los puntos de los datos de entrada que caen en dicha división. Por ello, la curva resultante es escalonada, y para varios valores de la variable dependiente x se tendrán valores promedio de la dependiente y.

Podemos mejorar el aspecto de la curva si aumentamos la *resolución* de la predicción:

```
x_grid <- seq(min(train$numero_bateos), max(train$numero_bateos), 0.01)
```

```
ggplot() + geom_point(data = train, aes(x = numero_bateos, y = runs), size =
0.9) +

  geom_line(aes(x = x_grid, y = predict(regresor_decisiontree, data.frame(num
ero_bateos = x_grid))),

            color = "red") +

  xlab("Var. Indep. numero de bateos") +

  ylab("Var. Dep. runs") +

  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento (train)")
```

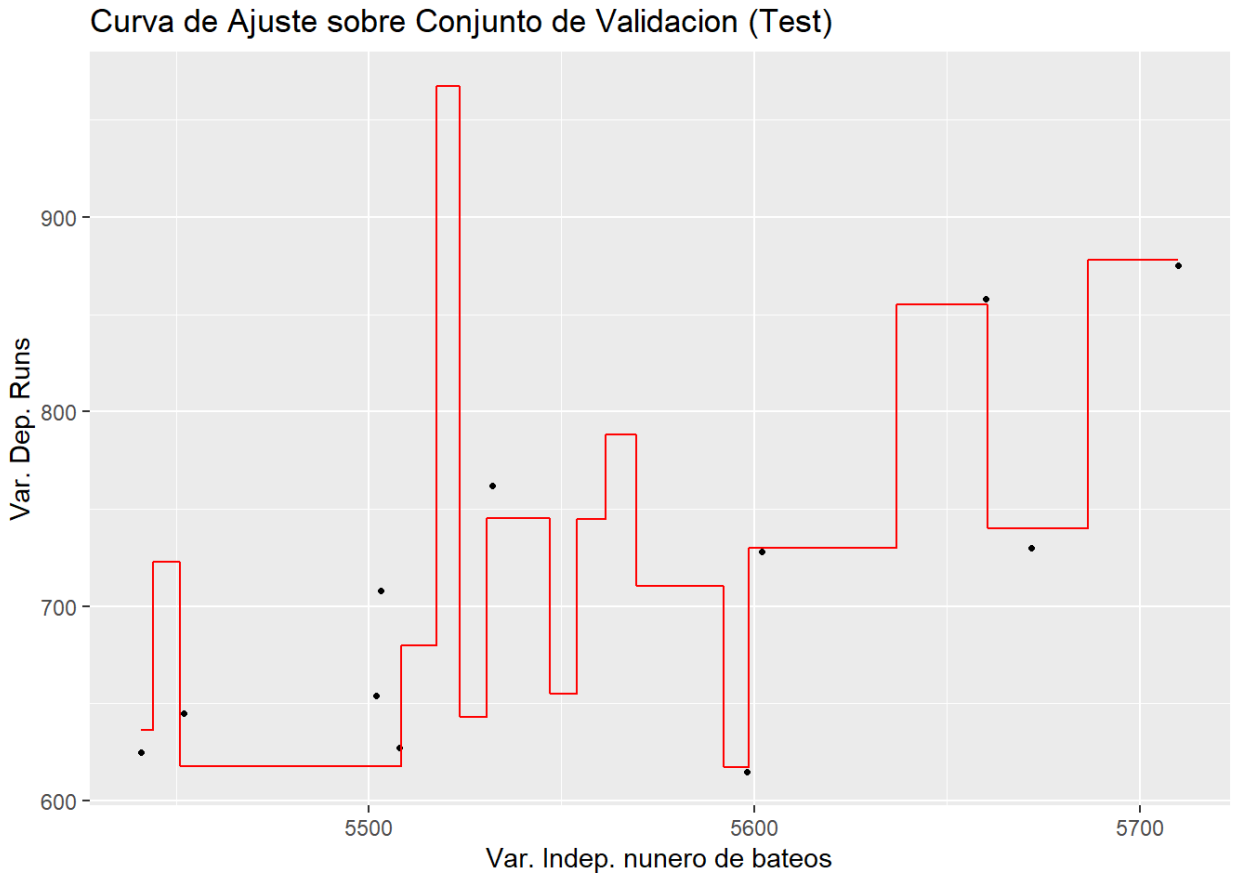


Para el conjunto de validación tendremos entonces:

```
y_dt_test_predict <- predict(regresor_decisiontree, Test)
```

```
x_grid <- seq(min(Test$numero_bateos), max(Test$numero_bateos), 0.01)
```

```
ggplot() + geom_point(data = Test, aes(x = numero_bateos, y = runs), size = 0
.9) +

  geom_line(aes(x = x_grid, y = predict(regresor_decisiontree, data.frame(num
ero_bateos = x_grid))),

            color = "red") +

  xlab("Var. Indep. nunero de bateos") +

  ylab("Var. Dep. Runs") +

  ggtitle("Curva de Ajuste sobre Conjunto de Validacion (Test)")
```



Con correlación de validación de:

```
cor(Test$runs, y_dt_test_predict)
```

```
## [1] 0.9555562
```

Finalmente, la predicción para el valor 5508 será:

```
predict_value_dt <- predict(regresor_decisiontree, data.frame(numero_bateos = 5508))

predict_value_dt
```

```
##        1
## 617.7143
```

```
# Cálculo del error

error = predict_value_dt - Test$runs

error
```

```
##  [1] -257.285714 -112.285714 -144.285714    2.714286  -36.285714  -27.285714
##  [7]   -7.285714 -240.285714 -110.285714  -90.285714   -9.285714
```

```
# Cálculo del error cuadrático medio RMSE:

sqrt(mean(error^2))
```

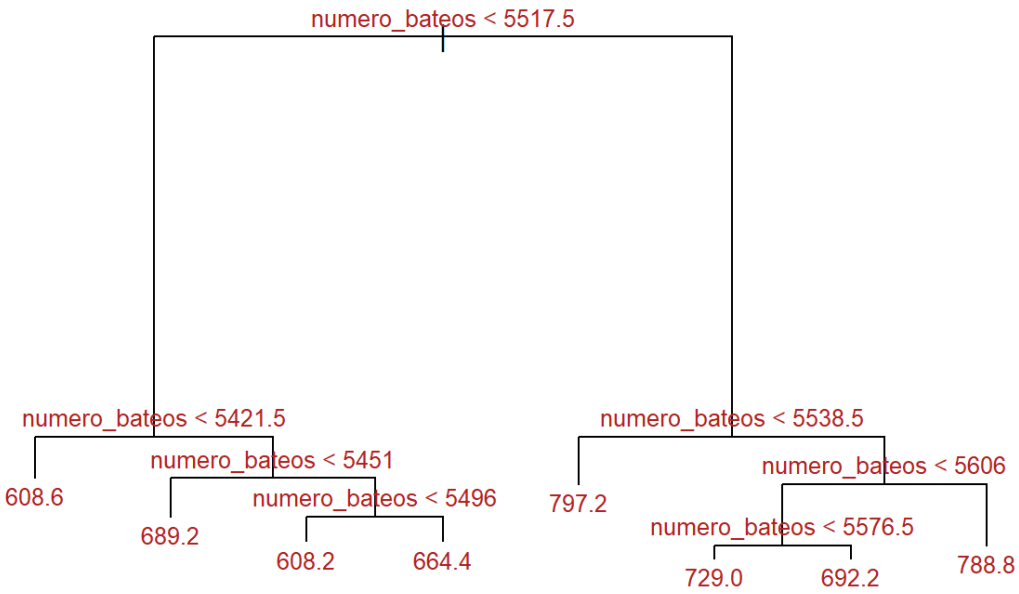
```
## [1] 127.8785
```

```
# estructura del arbol

library(tree)

set.seed(123)

arbol_regresion <- tree::tree(

  formula = runs ~ numero_bateos,

  data    = train,

  split   = "deviance",
```

```
  mincut  = 5,
  minsize = 10
)
summary(arbol_regresion)
```

```
##
## Regression tree:
## tree::tree(formula = runs ~ numero_bateos, data = train, split = "deviance
",
##     mincut = 5, minsize = 10)
## Number of terminal nodes:  8
## Residual mean deviance:  4713 = 193200 / 41
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -155.20  -43.25    3.60    0.00   25.80  170.80
```

```
plot(x =arbol_regresion, type = "proportional")
text(x = arbol_regresion, splits = TRUE, pretty = 0, cex = 0.80, col = "fireb
rick")
```



Regresión con Bosques Aleatorios (Random Forest Regression)

La regresión con Bosques Aleatorios o Random Forest, implica la construcción al azar de una gran cantidad de árboles de decisión sobre un mismo conjunto de datos, y la decisión final de la clasificación o la regresión es tomada a partir de calcular el promedio de las predicciones ofrecidas por cada uno de los árboles que conforman el bosque.

Para implementar el Random Forest sobre nuestros datos se empleará la función randomForest del paquete randomForest:

```
library(randomForest)
set.seed(1234)

regresor_randomForest <- randomForest(runs ~ numero_bateos, data = train, ntr
ee = 10)
```

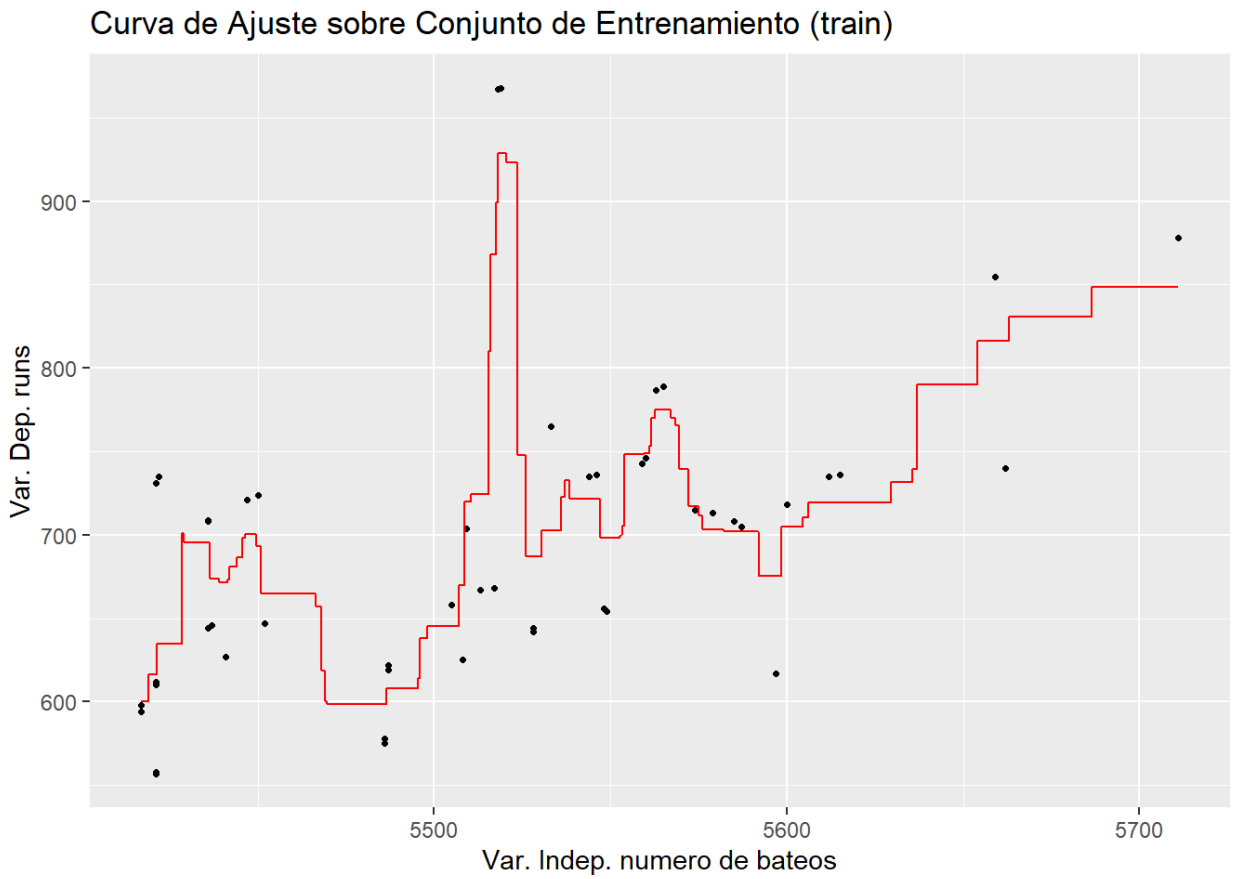
El parámetro ntree establece la cantidad de árboles de decisión que forman el bosque del modelo.

Veamos el resultado:

```
y_rf_predict <- predict(regresor_randomForest, train)


cor(train$runs, y_rf_predict)
```

```
## [1] 0.8436883
```

```
x_grid <- seq(min(train$numero_bateos), max(train$numero_bateos), 0.01)


ggplot() + geom_point(data = train, aes(x = numero_bateos, y =runs), size = 0
.9) +
  geom_line(aes(x = x_grid, y = predict(regresor_randomForest, data.frame(num
ero_bateos = x_grid))),
            color = "red") +
  xlab("Var. Indep. numero de bateos") +
  ylab("Var. Dep. runs") +
  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento (train)")
```



Como se observa, el incorporar muchos árboles a la predicción genera como resultado una curva de ajuste bastante más escalonada que en el caso de un solo árbol de decisión. De manera similar, para la validación:

```
y_rf_test_predict <- predict(regresor_randomForest, Test)
```

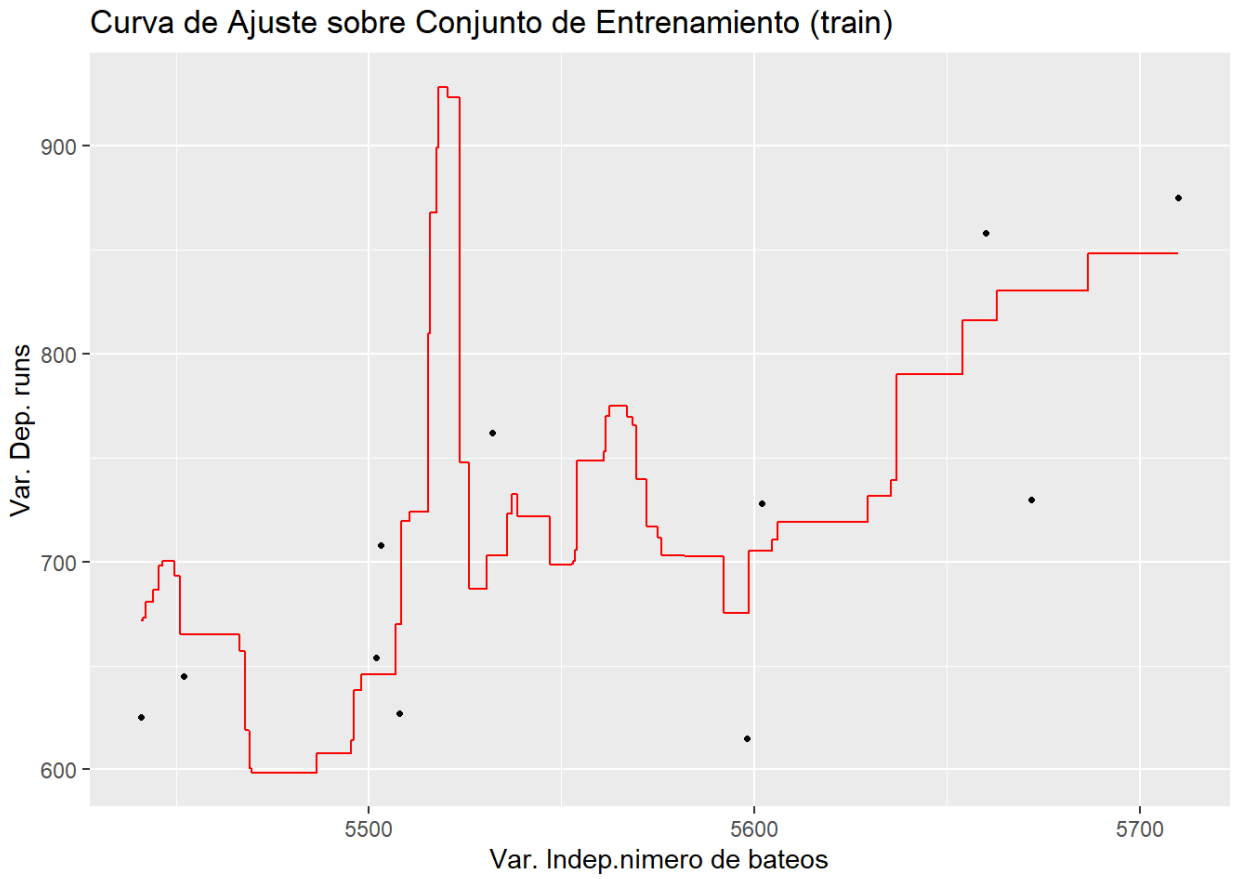
Con correlación de validación de:

```
cor(Test$runs, y_rf_test_predict)
```

```
## [1] 0.8120045
```

Y gráfica:

```
x_grid <- seq(min(Test$numero_bateos), max(Test$numero_bateos), 0.01)

ggplot() + geom_point(data = Test, aes(x = numero_bateos, y = runs), size = 0
.9) +

  geom_line(aes(x = x_grid, y = predict(regresor_randomForest, data.frame(num
ero_bateos = x_grid))),

            color = "red") +

  xlab("Var. Indep.nimero de bateos") +

  ylab("Var. Dep. runs") +

  ggtitle("Curva de Ajuste sobre Conjunto de Entrenamiento (train)")
```



Para el valor 5508, tendremos como predicción con el modelo Random Forest:

```
predict_value_rf <- predict(regresor_randomForest, data.frame(numero_bateos =
5508))

predict_value_rf
```

```
##        1
```

```
## 669.9383
```

```
# Cálculo del error
error = predict_value_rf - Test$runs
error
```

```
##  [1] -205.06167  -60.06167  -92.06167   54.93833   15.93833   24.93833
##  [7]   44.93833 -188.06167  -58.06167  -38.06167   42.93833
```

```
# Cálculo del error cuadrático medio RMSE:
sqrt(mean(error^2))
```

```
## [1] 96.33153
```

```
#########################################################
#Validacion simple - lm
### LOOCV
# Se genera el modelo lineal con GLM, dado que se va a emplear LOOCV no es ne
cesario
# dividir las observaciones en dos grupos
modelo <- glm(runs~numero_bateos, data = datos)
summary(modelo)
```

```
##
## Call:
## glm(formula = runs ~ numero_bateos, data = datos)
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)   -2800.9618   689.7568  -4.061 0.000149 ***
## numero_bateos     0.6336     0.1249   5.074 4.29e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 5732.549)
##
##     Null deviance: 480082  on 59  degrees of freedom
## Residual deviance: 332488  on 58  degrees of freedom
## AIC: 693.47
##
## Number of Fisher Scoring iterations: 2
```

```
MSE <- mean(regresor$residuals^2)
MSE
```

```
## [1] 3360.254
```

```
RMSE <- sqrt(MSE)
RMSE
```

```
## [1] 57.9677
```

```
# Se emplea la función cv.glm() para la validación LOOCV
library(lattice)
library(boot)
```

```
cv_error <- cv.glm(data = datos,glmfit = modelo)
MSE1 <- cv_error$delta
MSE1
```

```
## [1] 5866.650 5863.882
```

```
RMSE1 <-sqrt(MSE1)
RMSE1
```

```
## [1] 76.59406 76.57599
```

```
# K-fold Cross-Validation


# Se genera el modelo lineal con GLM
modelo <- glm(runs~numero_bateos, data = datos)
# Se emplea la función cv.glm() para la validación, empleando en este caso k= 10
set.seed(1)
cv_error <- cv.glm(data = datos, glmfit = modelo, K = 10)
MSE2 <- cv_error$delta
MSE2
```

```
## [1] 5850.656 5834.277
```

```
RMSE2 <-sqrt(MSE2)
RMSE2
```

```
## [1] 76.48958 76.38244
```

```
###Bootstrapping
dim(datos)
```

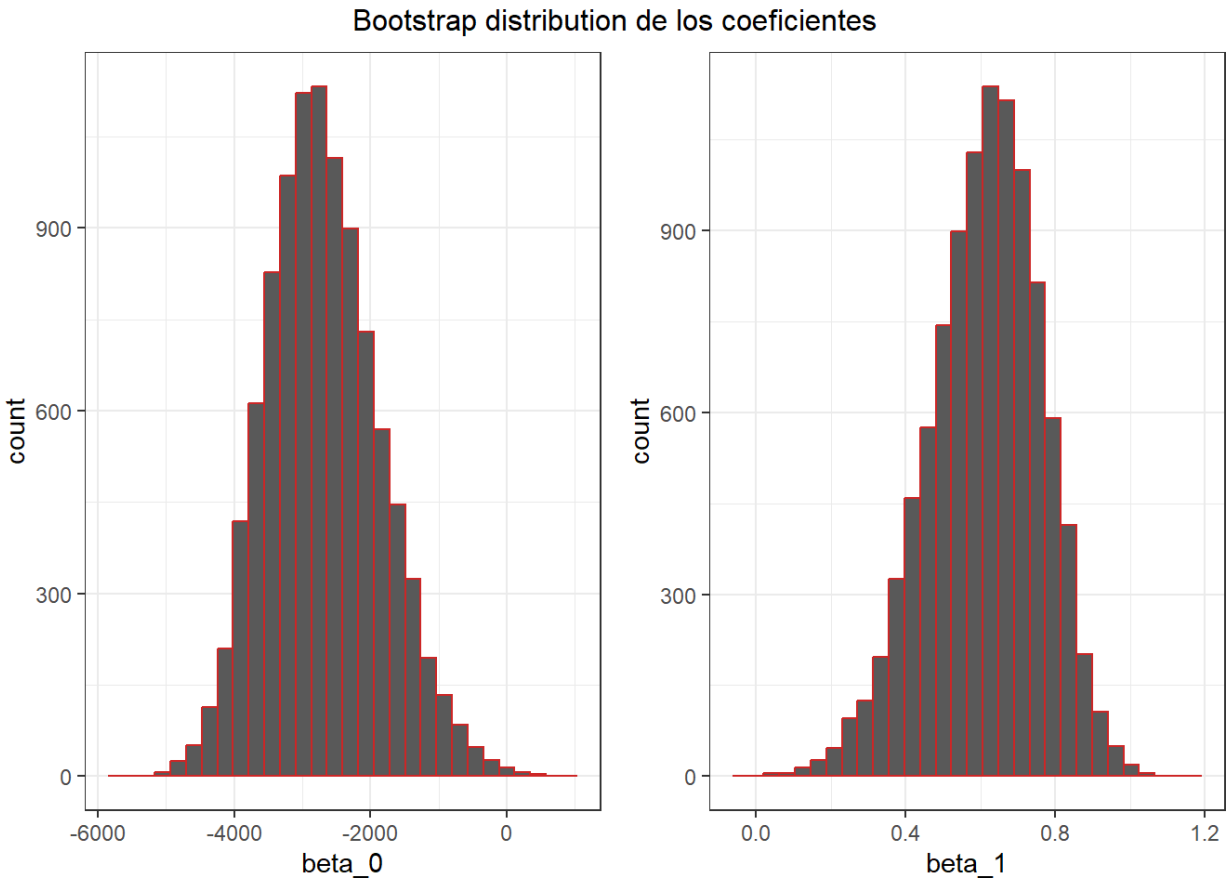
```
## [1] 60  2
```

```
# Se define la función que devuelve el estadístico de interés, los coeficientes
# de regresión
fun_coeficientes <- function(data, index){
  return(coef(lm(runs ~ numero_bateos, data = data, subset = index)))
}

# Se implementa un bucle que genere los modelos de forma iterativa y almacene
# los coeficientes. El data frame Auto tiene 392 observaciones
beta_0 <- rep(NA,9999)
beta_1 <- rep(NA,9999)
for(i in 1:9999) {
  coeficientes <- fun_coeficientes(data = datos,
                                   index = sample(1:30, 30, replace = TRUE))
  beta_0[i] <- coeficientes[1]
  beta_1[i] <- coeficientes[2]
}

coeficientes
```

```
##   (Intercept) numero_bateos
## -2359.5589029     0.5556132
```

```r
# Se muestra la distribución de los coeficientes
p5 <- ggplot(data = data.frame(beta_0 = beta_0), aes(beta_0)) +
  geom_histogram(colour = "firebrick3") +
  theme_bw()
p6 <- ggplot(data = data.frame(beta_1 = beta_1), aes(beta_1)) +
  geom_histogram(colour = "firebrick3") +
  theme_bw()

library(gridExtra)
```

```r
grid.arrange(p5,p6, ncol = 2,
             top = "Bootstrap distribution de los coeficientes")
```



```
# para comparar con lm
summary(lm(runs~numero_bateos,data = datos))$coef
```

```
##                    Estimate  Std. Error   t value     Pr(>|t|)
## (Intercept)   -2800.9618201 689.7567598 -4.060796 1.485135e-04
## numero_bateos     0.6335673   0.1248623  5.074127 4.293144e-06
```

```
## o
boot(data = datos, statistic = fun_coeficientes, R = 9999)
```

```
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = datos, statistic = fun_coeficientes, R = 9999)
##
##
## Bootstrap Statistics :
##          original       bias    std. error
## t1* -2800.9618201 23.356730799 563.1804417
## t2*     0.6335673 -0.004236655   0.1020048
```

Comparación de resultados para caso No Lineal

En los 4 casos anteriores se construyeron modelos de regresión para un conjunto de datos cuyo comportamiento era no lineal, y cada modelo ofreció resultados distintos tanto para las curvas de ajuste como para la predicción del valor de x = 5508, que, de hecho, forma parte del conjunto de entrenamiento. Veamos un cuadro comparativo de los resultados obtenidos para cada modelo:

x	y verdadero	y Polinomial	y SVR	y Decision Tree	y Random Forest
5508	625	676.7475	647.795	648.6667	637.6783

|yver−ypred|

y Polinomial	y SVR	y Decision Tree	y Random Forest
-51.7475	-22.795	-23.6667	- 12.6783

A partir de la diferencia entre el valor verdadero y el valor predicho por cada uno de los modelos, podemos ver que el Random forest ofrece la mayor precisión en la predicción, seguido por SVR y decisión tree. Y, por último, la regresión polinomial.

Sin embargo, ya que al momento de construir los regresores se utilizó la función set.seed para fijar la semilla de generación de los números aleatorios inherentes a cada modelo, los resultados de predicción serán los mismos en cada corrida de los algoritmos. Si no se establece la semilla inicial, es posible entonces tener resultados distintos para cada modelo y las precisiones podrán cambiar según el caso.