Tarea 2

Parte Teórica

- ¿Por qué una matriz equivale a una transformación lineal entre espacios vectoriales?

Sea T: V ->W una transformación lineal, y sean las bases de V y W las formadas por los vectores canónicos, existen escalares únicos aij (i entre [1,m] , j entre [1,n]) pertenecientes a un campo tales que

T(xi) = sigma (aij*yi)

Podemos expresar a T mediante una matriz A de mxn con elementos aij, donde la j-ésima columna de A es T(xj)

- ¿Cuál es el efecto de transformación lineal de una matriz diagonal y el de una matriz ortogonal?

La matriz diagonal expande o contrae los ejes, mientras que la matriz ortogonal rota los ejes

- ¿Qué es la descomposición en valores singulares de una matriz?

Es expresar una matriz de forma equivalente como el producto de tres matrices U*Sigma*V^t donde donde U y V son matrices ortogonales y Sigma es una matriz cuasi diagonal, a los valores de la diagonal de sigma se le llaman valores singulares y cumplen con aparecen de forma descendente, es decir, el primer valor siggular de la diagonal es el mayor de todos, y así sucesivamente

- ¿Qué es diagonalizar una matriz y que representan los eigenvectores?

Diagonalizar equivale a encontrar una base de eigenvectores

- ¿Intuitivamente qué son los eigenvectores?

Los eigenvectores son un sistema de las nuevas coordenadas

- ¿Cómo interpretas la descomposición en valores singulares como una composición de tres tipos de transformaciones lineales simples?

Se rotan los ejes -> se expanden o contraen los ejes ->nuevamente se rotan

- ¿Qué relación hay entre la descomposición en valores singulares y la diagonalización?

- La diagonalización es exclusiva para matrices cuadradas
- La descomposición en valores singulares (svd) siempre existe en número reales, la diagonaliazación siempre existe en número complejos

- La svd permite solucionar sistemas de ecuaciones incluso cuando no existe la inversa de una matriz

- ¿Cómo se usa la descomposición en valores singulares para dar una aproximación de rango menor a una matriz?

Por medio de los valores de la diagonal distintos de cero de la matriz cuasi diagonal

Parte Aplicada

Ejercicio 1

```
In [1]: import numpy as np
   ...: import pandas as pd
   ...: import matplotlib.pyplot as plt
   ...: from PIL import Image
```

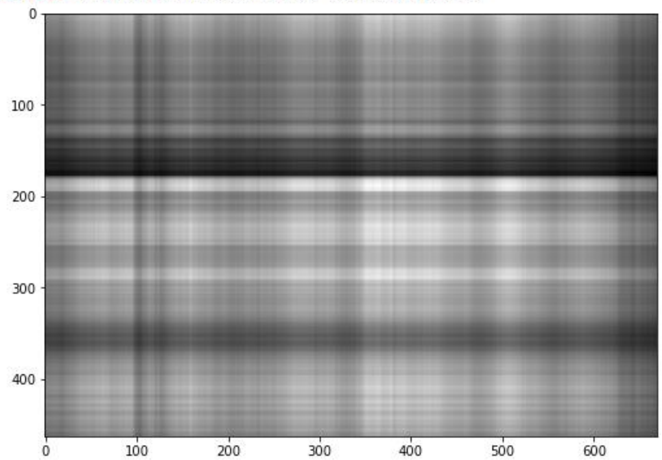
```
######################################################################
## Ejercicio 1

# Importar imagen


imagen = Image.open('C:/Users/Data Mining/Documents/ITAM/Propedeutico/Alumnos/PropedeuticoDataScience2017/Alumnos/Leonardo_Marin/l
imagen_gris = imagen.convert('LA')   ## Convertir a escala de grises


## Convertir la imagen a una matriz

imagen_mat = np.array(list(imagen_gris.getdata(band=0)), float)
imagen_mat.shape = (imagen_gris.size[1], imagen_gris.size[0])
imagen_mat = np.matrix(imagen_mat)

plt.figure(figsize=(9, 6))
plt.imshow(imagen_mat, cmap='gray')
```

```
   ...:
   ...: plt.figure(figsize=(9, 6))
   ...: plt.imshow(imagen_mat, cmap='gray')
Out[3]: <matplotlib.image.AxesImage at 0x2353f42f208>
```

```
## Desomposición singular

U, sigma, V = np.linalg.svd(imagen_mat)
```

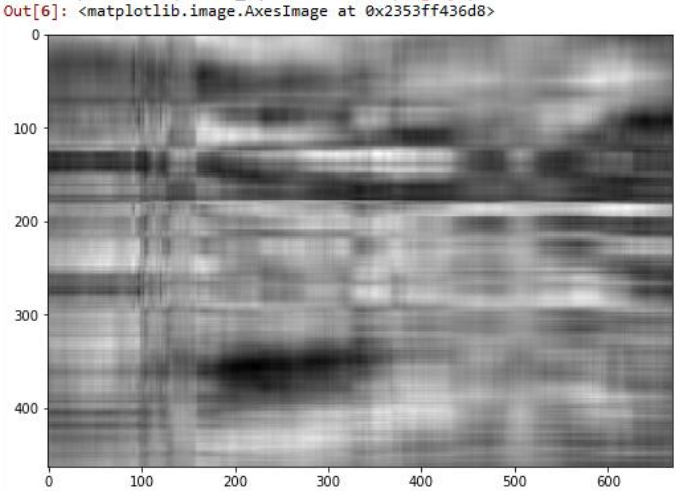
```
## Probar la visualización con los primeris n vectores

# n= 1
j = 1
matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
plt.figure(figsize=(9, 6))
plt.imshow(matriz_equivalente, cmap='gray')
```

```
In [4]: U, sigma, V = np.linalg.svd(imagen_mat)

In [5]: j = 1
   ...: matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
   ...: plt.figure(figsize=(9, 6))
   ...: plt.imshow(matriz_equivalente, cmap='gray')
Out[5]: <matplotlib.image.AxesImage at 0x235401a20f0>
```



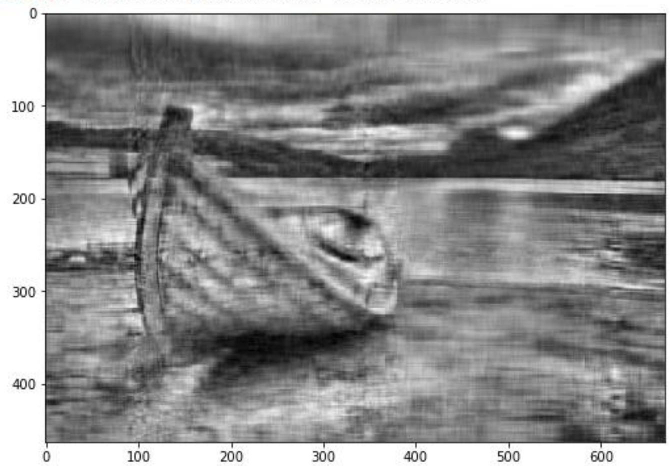
```
# n = 5
j = 5
matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
plt.figure(figsize=(9, 6))
plt.imshow(matriz_equivalente, cmap='gray')
```

```
In [6]: j = 5
   ...: matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
   ...: plt.figure(figsize=(9, 6))
   ...: plt.imshow(matriz_equivalente, cmap='gray')
Out[6]: <matplotlib.image.AxesImage at 0x2353ff436d8>
```



```python
# n = 25
j = 25
matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
plt.figure(figsize=(9, 6))
plt.imshow(matriz_equivalente, cmap='gray')
```

```
In [7]: j = 25
   ...: matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
   ...: plt.figure(figsize=(9, 6))
   ...: plt.imshow(matriz_equivalente, cmap='gray')
Out[7]: <matplotlib.image.AxesImage at 0x2353ff9c828>
```




```python
# n = 50
j = 50
matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
plt.figure(figsize=(9, 6))
plt.imshow(matriz_equivalente, cmap='gray')
```

```
In [8]: j = 50
   ...: matriz_equivalente = np.matrix(U[:, :j]) * np.diag(sigma[:j]) * np.matrix(V[:j, :])
   ...: plt.figure(figsize=(9, 6))
   ...: plt.imshow(matriz_equivalente, cmap='gray')
Out[8]: <matplotlib.image.AxesImage at 0x235400229e8>
```

Podemos ver cómo se puede reconstruir la imagen sin utilizar toda la información de la matriz original,

Ejercicio 2

```
################################################################################
## Ejercicio 2

################################################################################

pseudoinversa1 <- function(A)
{

  # A debe ser matriz
  A.svd <- svd(A)   #Descomposición SVD
  A.svd

  S.inv.fix <- diag(1/A.svd$d)  ## La inversa es 1 / los valores de la diagonal
  S.inv.fix[S.inv.fix == Inf] <- 0

  U <- A.svd$u
  V <- t(A.svd$v)

  ## PseudoInversa
  A.inv <- (t(V))%*%(S.inv.fix)%*%(t(U))

  return(A.inv)
}
```

```
> ########################################
>
> # Probar funcion
> A <- matrix(c(1,1,0,2),ncol=2)
>
> solve(A)
     [,1] [,2]
[1,]  1.0  0.0
[2,] -0.5  0.5
> pseudoinversa1(A)
     [,1]         [,2]
[1,]  1.0 2.775558e-17
[2,] -0.5 5.000000e-01
>
> ########################################
```

```r
## Función que resuleve un  sistema de ecuaciones

####################################################################

sistema_ecuaciones1 <- function(A,b)
{
  ##A: matriz; b: puede ser vectot
  b <- as.matrix(b)
  x <- pseudoinversa1(A)%*%b
  return(x)

}
```

```r
> ## Probar funcion
> A <- matrix(c(1,1,0,2),ncol=2)
> b <- matrix(c(5,3))
>
> # Solución exacta
> solve(A)%*%b
     [,1]
[1,]    5
[2,]   -1
> # solucion Función
> sistema_ecuaciones1(A,b)
     [,1]
[1,]    5
[2,]   -1
>
```

```
> ########################################
>
> # Hacer pruebas con distintas matrices
>
> # Matriz de 2x3
> B = matrix(c(1,1,0,2,8,-1),ncol=3)
> B
     [,1] [,2] [,3]
[1,]    1    0    8
[2,]    1    2   -1
>
> solve(B) # Marca error por no ser cuadrada
Error in solve.default(B) : 'a' (2 x 3) must be square
> pseudoinversa1(B)
           [,1]        [,2]
[1,] 0.03812317  0.21114370
[2,] 0.04105572  0.38123167
[3,] 0.12023460 -0.02639296
>
> ## Validación
>
> ## B.inv * B = In  (Mal)
> ## B * B.inv = In  (Ok)
> pseudoinversa1(B) %*% B
           [,1]        [,2]        [,3]
[1,] 0.24926686  0.42228739  0.09384164
[2,] 0.42228739  0.76246334 -0.05278592
[3,] 0.09384164 -0.05278592  0.98826979
> B %*% pseudoinversa1(B)
             [,1]         [,2]
[1,] 1.000000e+00 3.885781e-16
[2,] 4.163336e-17 1.000000e+00
>
```

```
> ##################################################
> # Solucionar un sistema que incluya la matriz B
>
> c <- matrix(c(2,1))
>
> ##
> sistema_ecuaciones1(B,c)
          [,1]
[1,] 0.2873900
[2,] 0.4633431
[3,] 0.2140762
>
> # Validación
> B%*%sistema_ecuaciones1(B,c)
     [,1]
[1,]    2
[2,]    1
>
```

```
> #Probar la función para matrices aleatorias de nxm  (9x9 max)
>
> #help(sample)
>
> (n <- sample(1:9,1))
[1] 8
> (m <- sample(1:9,1))
[1] 4
> (coeficientes <- sample(-9:9,n*m,replace=TRUE))
 [1] -8  7 -2  5  7 -8 -8 -7 -7 -2  4 -1 -2 -3 -7  8  1 -1 -9 -2  7 -3 -4 -8  9  9  3  7 -9 -8  2  9
> (d <- sample(-9:9,n))
[1]  0 -9  9  3 -6 -7  8  2
>
> (W <- matrix(coeficientes,n,m))
     [,1] [,2] [,3] [,4]
[1,]   -8   -7    1    9
[2,]    7   -2   -1    9
[3,]   -2    4   -9    3
[4,]    5   -1   -2    7
[5,]    7   -2    7   -9
[6,]   -8   -3   -3   -8
[7,]   -8   -7   -4    2
[8,]   -7    8   -8    9
> (z <- matrix(d))
     [,1]
[1,]    0
[2,]   -9
[3,]    9
[4,]    3
[5,]   -6
[6,]   -7
[7,]    8
[8,]    2
>
```

```
> pseudoinversa1(W)
            [,1]        [,2]        [,3]         [,4]         [,5]         [,6]         [,7]        [,8]
[1,] -0.03455444  0.03759717  0.02252755  0.031820842  0.014170856 -0.006930227 -0.000102681 -0.02726798
[2,] -0.01387826 -0.03200185 -0.01069852 -0.026221609 -0.007938257 -0.026611134 -0.051690174  0.04973498
[3,]  0.04649878 -0.03034715 -0.06961227 -0.033485445  0.007986396 -0.040579799 -0.041456617  0.01422474
[4,]  0.03281939  0.01576253 -0.01461022  0.009149458 -0.014910553 -0.031680120 -0.007552111  0.01889061
> sistema_ecuaciones1(W,z)
            [,1]
[1,] -0.13203501
[2,]  0.03292112
[3,] -0.52090569
[4,]  0.04268212
>
> #Validación (W * (resultado sistema) debe dar el vector z)
> W%*%sistema_ecuaciones1(W,z)
            [,1]
[1,]  0.68906563
[2,] -0.08504261
[3,]  5.21195204
[4,]  0.64748999
[5,] -5.02056620
[6,]  2.17877688
[7,]  2.99481924
[8,]  5.73899861
>
```

```
> W%*%pseudoinversa1(W) # (Ok)
            [,1]        [,2]       [,3]        [,4]         [,5]         [,6]        [,7]        [,8]
[1,]  0.71545669  0.03475115 -0.3064350 -0.02215579 -0.184007625 -0.083981123  0.25322705  0.05423927
[2,]  0.03475115  0.49939378  0.1172102  0.39101968 -0.027108867 -0.239830601  0.07614920 -0.13455507
[3,] -0.30643502  0.11721017  0.4948306  0.16028926 -0.176703966  0.177593755  0.14389789  0.18212509
[4,] -0.02215579  0.39101968  0.1602893  0.31634292 -0.041554127 -0.148641242  0.08122522 -0.08229008
[5,] -0.18400763 -0.02710887 -0.1767040 -0.04155413  0.305172255  0.005773159 -0.11956574 -0.36078819
[6,] -0.08398112 -0.23983060  0.1775938 -0.14864124  0.005773159  0.510455575  0.34067871 -0.12486016
[7,]  0.25322705  0.07614920  0.1438979  0.08122522 -0.119565738  0.340678712  0.51337491 -0.14911869
[8,]  0.05423927 -0.13455507  0.1821251 -0.08229008 -0.360788194 -0.124860162 -0.14911869  0.64497332
>
```

```
> ######################################################################################
> ## Pruebas con matrices que tienen filas linealmente dependientes (No son Invertibles)
>
> ## El vector está en la imagen (primer ejemplo)
>
> AA <- matrix(c(1,1,0,0),ncol=2,byrow = TRUE)
> bb <- matrix(c(4,0))
>
> AA
     [,1] [,2]
[1,]    1    1
[2,]    0    0
> bb
     [,1]
[1,]    4
[2,]    0
>
>
> ## AA no tiene inbversa
> solve(AA)
Error in solve.default(AA) :
  Lapack routine dgesv: system is exactly singular: U[2,2] = 0
> ## Psuedoinversa(AA)
> pseudoinversa1(AA)
     [,1] [,2]
[1,]  0.5    0
[2,]  0.5    0
```

```
> # Solución exacta (Error)
> solve(AA)%*%b
Error in solve.default(AA) :
  Lapack routine dgesv: system is exactly singular: U[2,2] = 0
> # solucion Función
> sistema_ecuaciones1(AA,bb)
     [,1]
[1,]    2
[2,]    2
>
>
> ## Validación
> bb
     [,1]
[1,]    4
[2,]    0
> AA%*%sistema_ecuaciones1(AA,bb)
     [,1]
[1,]    4
[2,]    0
>
```

```
> ## Pruebas con matrices que tienen filas linealmente dependientes (No son Invertibles)
>
> ## El vector NO está en la imagen (primer ejemplo)
>
> AA1 <- matrix(c(1,1,0,0),ncol=2,byrow = TRUE)
> bb1 <- matrix(c(1,1))
>
> AA1
     [,1] [,2]
[1,]    1    1
[2,]    0    0
> bb1
     [,1]
[1,]    1
[2,]    1
>
> ## Psuedoinversa(AA1)
> pseudoinversa1(AA1)
     [,1] [,2]
[1,]  0.5    0
[2,]  0.5    0
>
> # solucion Función
> sistema_ecuaciones1(AA1,bb1)
     [,1]
[1,]  0.5
[2,]  0.5
>
>
> ## Validación
> bb1
     [,1]
[1,]    1
[2,]    1
> AA1%*%sistema_ecuaciones1(AA1,bb1)
     [,1]
[1,]    1
[2,]    0
> |
```

Se observa que la validación no se cumple, pero la función calcula correctamente la pseudoinversa

```
> ######################################################################################
> ## Pruebas con matrices que tienen filas linealmente dependientes (No son Invertibles)
>
> ## El vector está en la imagen (segundo ejemplo)
> ## Números muy proximos a cero,
>
> AA2 <- matrix(c(1,1,0,1e-32),ncol=2,byrow = TRUE)
> bb2 <- matrix(c(4,0))
>
> AA2
     [,1]  [,2]
[1,]    1 1e+00
[2,]    0 1e-32
> bb2
     [,1]
[1,]    4
[2,]    0
>
> ## Psuedoinversa(AA2)
> pseudoinversa1(AA2)
              [,1]   [,2]
[1,]  1.000000e+00 -1e+32
[2,] -1.110223e-16  1e+32
>
> # solucion Función
> sistema_ecuaciones1(AA2,bb2)
              [,1]
[1,]  4.000000e+00
[2,] -4.440892e-16
>
>
> ## Validación
> bb2
     [,1]
[1,]    4
[2,]    0
> AA2%*%sistema_ecuaciones1(AA2,bb2)
              [,1]
[1,]  4.000000e+00
[2,] -4.440892e-48
```

Vemos que genera soluciones no correctas por errores de precisión

```
> ## Pruebas con matrices que tienen filas linealmente dependientes (No son Invertibles)
>
> ## El vector NO está en la imagen (segundo ejemplo)
> ## Números muy proximos a cero,
> AA3 <- matrix(c(1,1,0,1e-32),ncol=2,byrow = TRUE)
> bb3 <- matrix(c(1,1))
> AA3
     [,1]  [,2]
[1,]    1 1e+00
[2,]    0 1e-32
> bb3
     [,1]
[1,]    1
[2,]    1
>
> ## Psuedoinversa(AA3)
> pseudoinversa1(AA3)
              [,1]   [,2]
[1,]  1.000000e+00 -1e+32
[2,] -1.110223e-16  1e+32
> AA3*pseudoinversa1(AA3)
     [,1]   [,2]
[1,]    1 -1e+32
[2,]    0  1e+00
>
> # solucion Función
> sistema_ecuaciones1(AA3,bb3)
       [,1]
[1,] -1e+32
[2,]  1e+32
>
> ## Validación
> bb3
     [,1]
[1,]    1
[2,]    1
> AA3%*%sistema_ecuaciones1(AA3,bb3)
     [,1]
[1,]    0
[2,]    1
```

Al igual que el ejemplo anterior, las soluciones no son adecuadas por errores de precisión

Ejercicio 3

```r
## Limpiar area de trabajo

rm(list=ls())
ls()

## Leer datos

datos <- read.table('C:/Users/Data Mining/Documents/ITAM/Propedeutico/Alumnos/PropedeuticoDataScience2017/Alumnos/Leona
datos


## Calcular a mano alpha y beta

## beta
mu.x <- mean(datos$study_hours)
mu.y <- mean(datos$sat_score)


numerador <-  sum((datos$study_hours - mu.x)*(datos$sat_score - mu.y))
denominador <- sum((datos$study_hours - mu.x)^2)

beta <- numerador/denominador


## alpha

alpha <- mu.y - beta*mu.x
```

```
> datos
   study_hours sat_score
1            4       390
2            9       580
3           10       650
4           14       730
5            4       410
6            7       530
7           12       600
8           22       790
9            1       350
10           3       400
11           8       590
12          11       640
13           5       450
14           6       520
15          10       690
16          11       690
17          16       770
18          13       700
19          13       730
20          10       640
```

```
> ##############################################################
> ## Realizar la regresión usando la funciones ya construidas en R
>
> lm.sat <- lm(sat_score~study_hours,data=datos)
> lm.sat

Call:
lm(formula = sat_score ~ study_hours, data = datos)

Coefficients:
(Intercept)  study_hours
     353.16        25.33

>
> summary(lm.sat)

Call:
lm(formula = sat_score ~ study_hours, data = datos)

Residuals:
     Min       1Q   Median       3Q      Max
-120.347  -29.308    9.928   33.734   83.570

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  353.165     24.337   14.51 2.24e-11 ***
study_hours   25.326      2.291   11.05 1.87e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 49.72 on 18 degrees of freedom
Multiple R-squared:  0.8716,	Adjusted R-squared:  0.8645
F-statistic: 122.2 on 1 and 18 DF,  p-value: 1.868e-09
```

```
> ## Estimaciones  yhat = alpha + beta*X
> predict(lm.sat,datos)
       1        2        3        4        5        6        7        8        9       10       11       12       13       14       15       16 
454.4708 581.1031 606.4296 707.7354 454.4708 530.4502 657.0825 910.3472 378.4913 429.1443 555.7766 631.7560 479.7972 505.1237 606.4296 631.7560 
      17       18       19       20 
758.3884 682.4090 682.4090 606.4296 
```

```
##############################################################################
## Definir la matriz X

matriz.x <- matrix(NA, ncol=2,nrow=dim(datos)[1])

matriz.x[,1] <- 1
matriz.x[,2] <- datos$study_hours
print(matriz.x)

## Calcular las estimaciones: sat_score ~ matriz.x * [alpha, beta]
## En R la multiplicación de matrices se hace con el operador %*%

sat_score <- matriz.x %*% c(alpha,beta)
sat_score
```

```
> ## Calcular las estimaciones: sat_score ~ matriz.x * [alpha, beta]
> ## En R la multiplicación de matrices se hace con el operador %*%
>
> sat_score <- matriz.x %*% c(alpha,beta)
> sat_score
          [,1]
 [1,] 454.4708
 [2,] 581.1031
 [3,] 606.4296
 [4,] 707.7354
 [5,] 454.4708
 [6,] 530.4502
 [7,] 657.0825
 [8,] 910.3472
 [9,] 378.4913
[10,] 429.1443
[11,] 555.7766
[12,] 631.7560
[13,] 479.7972
[14,] 505.1237
[15,] 606.4296
[16,] 631.7560
[17,] 758.3884
[18,] 682.4090
[19,] 682.4090
[20,] 606.4296
```

```
##################################################################################
# Calcular la PseudoInversa

X.svd <- svd(matriz.x)


## PseudoInversa = (inversa(transpuesta(V)))*(inversa(S))*(inversa(U))
## Nota: U y V son ortogonales por tanto su inversa es la transpuesta, y la inversa


S.inv <- diag(1/X.svd$d)  ## La inversa es 1 / los valores de la diagonal
S.inv

U <- X.svd$u
U

V <- t(X.svd$v)
V

## PseudoInversa

pseu.inv <- (t(V))%*%(S.inv)%*%(t(U))
pseu.inv

## Calcular alpha y beta

solucion_aprox <- pseu.inv%*%as.matrix(datos$sat_score)
solucion_aprox
```

```
> ## PseudoInversa
>
> pseu.inv <- (t(V))%*%(S.inv)%*%(t(U))
> pseu.inv
            [,1]          [,2]        [,3]         [,4]        [,5]         [,6]         [,7]        [,8]        [,9]       [,10]        [,11]
[1,]  0.15935874  0.0590296210 0.038963797 -0.041299501  0.15935874  0.099161270 -0.001167852 -0.20182610  0.21955622  0.17942457  0.079095445
[2,] -0.01157235 -0.0009555154 0.001167852  0.009661323 -0.01157235 -0.005202251  0.005414588  0.02664826 -0.01794246 -0.01369572 -0.003078883
          [,12]        [,13]        [,14]       [,15]      [,16]       [,17]        [,18]        [,19]       [,20]
[1,] 0.01889797  0.139292919  0.119227094 0.038963797 0.01889797 -0.08143115 -0.021233677 -0.021233677 0.038963797
[2,] 0.00329122 -0.009448986 -0.007325618 0.001167852 0.00329122  0.01390806  0.007537955  0.007537955 0.001167852
>
> ## Calcular alpha y beta
>
> solucion_aprox <- pseu.inv%*%as.matrix(datos$sat_score)
> solucion_aprox
          [,1]
[1,] 353.16488
[2,]  25.32647
>
```

```
##########################################################################################
# Solución Exacta
# (alpha,beta)=(X^t*X)^(-1)*X^t*sat_score.


A <- t(matriz.x)%*%matriz.x

solucion_exacta <- solve(A)%*%t(matriz.x)%*%as.matrix(datos$sat_score)
solucion_exacta


## Comparar con la solución ce la pseudo inversa

# En este caso las soluciones dieron igual

solucion_exacta
solucion_aprox
```

```
> ## Comparar con la solución ce la pseudo inversa
>
> # En este caso las soluciones dieron igual
>
> solucion_exacta
          [,1]
[1,] 353.16488
[2,]  25.32647
> solucion_aprox
          [,1]
[1,] 353.16488
[2,]  25.32647
>
```

```
#################################################################################
# Visualizar las predicciones contra los valores reales

plot(datos$sat_score~datos$study_hours)
# Agregar la linea de la regresión a la grafica de dispersión
abline(lm.sat, col="blue")
```