Análisis numérico

Facultad de Ciencias, UNAM
Mínimos cuadrados lineales

Jorge Zavaleta Sánchez

Semestre 2022-1

Índice

Preliminares

[1]:
```
# Modulos
import numpy as np
import Solsislin as ssl # Modulo propio
import matplotlib.pyplot as plt
import csv # Manejo de archivos csv
import sympy # Simbolico
import ipywidgets as widgets # Controles
from ipywidgets import interact, interact_manual
```

[2]:
```
# Funciones auxiliares
def canonico(n,j = 1):
    """Crea el j-esimo vector canonico e_{j} en R^{n}
    - Entrada >
      n (entero) - Dimension.
      j (entero) - Elemento de la base canonica.
    - Salida >
      x (1D ndarray) - Vector e_{j} en R^{n}. Si j > n entonces regresa e_{1}
    """
    x,indx = np.zeros((max(1,n),)),0
    if j >= 0 and j <= n:
        indx = j-1
    x[indx] = 1.0
    return x
```

Ajuste de datos

Uno de los usos más comunes del método de mínimos cuadrados es el *ajuste de datos*, especialmente cuando los datos tienen algún error asociado a ellos, como la mayoría de las mediciones de laboratorio u otras observaciones de la naturaleza. Dado m datos (t_i, y_i) queremos encontrar un vector $\boldsymbol{x}$ de parámetros de tamaño n que "mejor" ajusten los datos por una *función modelo*, con $f\colon \mathbb{R}^{n+1} \to \mathbb{R}$, donde por el mejor ajuste se refiere en el sentido de mínimos cuadrados

$$\min_{\boldsymbol{x}} \sum_{i=1}^{m} (y_i - f(t_i, \boldsymbol{x}))^2$$

Un problema de ajuste de datos es *lineal* si la función f es lineal en las componentes del vector de parámetros $\boldsymbol{x}$, lo que significa que f es una combinación lineal

$$f(t, \boldsymbol{x}) = x_1\phi_1(t) + x_2\phi_2(t) + \cdots + x_n\phi_n(t)$$

de funciones ϕ_j que dependen solo de t. Por ejemplo, ajuste polinomial, con

$$f(t, \boldsymbol{x}) = x_1 + x_2 t + x_3 t^2 + \cdots + x_n t^{n-1}(t)$$

es un problema lineal de ajuste de datos dado que un polinomio es lineal en sus coeficientes x_j, aunque no lineal con respecto a t. Si definimos una matriz A con entradas $a_{i,j} = \phi_j(t_i)$ y con vector $\boldsymbol{y}$ con las componentes y_i, entonces un problema de mínimos cuadrados lineales toma la forma

$$A\boldsymbol{x} \cong \boldsymbol{y}$$

Por ejemplo, al ajustar un polinomio cuadrático, el cual tiene tres parámetros, a cinco datos $(t_1, y_1), \ldots, (t_5, y_5)$, la matriz A es de tamaño 5×3 y el problema tiene la forma

$$A\boldsymbol{x} = \begin{pmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \\ 1 & t_4 & t_4^2 \\ 1 & t_5 & t_5^2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \cong \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{pmatrix} = \boldsymbol{y}$$

Una matriz de esta forma particular, cuyas columnas (o renglones) son potencias sucesivas de alguna variable independiente, es llamada una **matriz de Vandermonde**.

Ecuaciones Normales

Como un problema de minimización, un problema de minimos cuadrados puede ser tratado usando metodos de calculo multivariado, análogo a hacer la derivada igual a cero en calculo de una variable. Queremos minimizar la norma euclidiana al cuadrado del vector residual $\boldsymbol{r} = \boldsymbol{y} - A\boldsymbol{x}$. Denotando esta función objetivo por $\psi\colon \mathbb{R}^n \to \mathbb{R}$, tenemos

$$\psi(\boldsymbol{x}) = \|\boldsymbol{r}\|_2^2 = \boldsymbol{r}^\intercal \boldsymbol{r} = (\boldsymbol{y} - A\boldsymbol{x})^\intercal (\boldsymbol{y} - A\boldsymbol{x}) = \boldsymbol{y}^\intercal \boldsymbol{y} - 2\boldsymbol{x}^\intercal A^\intercal \boldsymbol{y} + \boldsymbol{x}^\intercal A^\intercal A\boldsymbol{x}$$

Una condición necesaria para encontrar un mínimo es que $\boldsymbol{x}$ sea un punto crítico de ψ, esto es donde el vector gradiente $\nabla\psi(\boldsymbol{x})$, cuya i-ésima componente esta dada por $\partial\psi(\boldsymbol{x})/\partial x_i$, es cero. Así, debemos tener

$$\mathbf{0} = \nabla\psi(\boldsymbol{x}) = 2A^\intercal A\boldsymbol{x} - 2A^\intercal \boldsymbol{y}$$

tal que cualquier mínimo $\boldsymbol{x}$ de ψ debe satisfacer el sistema lineal simétrico de $n \times n$

$$A^{\intercal} A\boldsymbol{x} = A^{\intercal}\boldsymbol{y}.$$

Una condición suficiente para que tal $\boldsymbol{x}$ sea en efecto un mínimo es que la matriz Hessiana de segundas derivadas parciales, la cual está dada en este caso solo por $2A^{\intercal}A$, sea positiva definida. Es fácil ver que $A^T A$ es positiva definida, si y solo si, las columnas de A son linealmente independientes, *i.e.*, $\text{rango}(A) = n$.

El sistema lineal $A^{\intercal} A\boldsymbol{x} = A^{\intercal}\boldsymbol{y}$ es comunmente conocido como sistema de **ecuaciones normales**. Resolviendo este sistema lineal podemos encontra el vector de parámetros $\boldsymbol{x}$ para nuestra función modelo que mejor ajusta los datos, ya que

$$\min_{\boldsymbol{x}} \sum_{i=1}^{m} (y_i - f(t_i, \boldsymbol{x}))^2 = \min_{\boldsymbol{x}} \psi(\boldsymbol{x})$$

Ejemplo Consideramos la tabla de datos

i	t_i	y_i
1	2.6578	-6.4552
2	3.992	-14.9657
3	0.2389	0.2798
4	1.5106	-2.0462
5	3.2851	-10.539

y queremos hacer el ajuste polinomial, para lo cual se considera la función modelo

$$f(t, \boldsymbol{x}) = x_1 + x_2 t + x_3 t^2$$

entonces vamos a buscar el vector de coeficientes $\boldsymbol{x}$ que mejor ajuste f a los datos de la tabla (t_i, y_i) mediante el uso de las ecuaciones normales.

Tomando

$$\boldsymbol{t} = \begin{pmatrix} 2.6578 \\ 3.992 \\ 0.2389 \\ 1.5106 \\ 3.2851 \end{pmatrix} \qquad \text{y} \qquad \boldsymbol{y} = \begin{pmatrix} -6.4552 \\ -14.9657 \\ 0.2798 \\ -2.0462 \\ -10.539 \end{pmatrix}$$

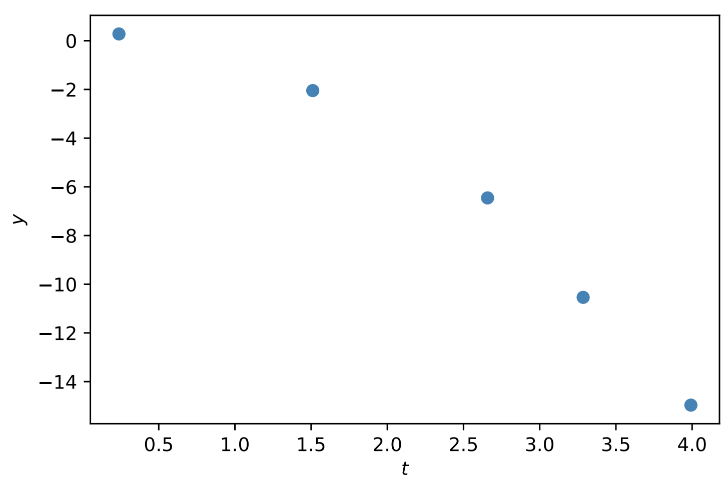
Luego, el problema tiene la forma

$$\begin{pmatrix} 2.6578^2 & 2.6578 & 1 \\ 3.992^2 & 3.992 & 1 \\ 0.2389^2 & 0.2389 & 1 \\ 1.5106^2 & 1.5106 & 1 \\ 3.2851^2 & 3.2851 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} \cong \begin{pmatrix} -6.4552 \\ -14.9657 \\ 0.2798 \\ -2.0462 \\ -10.539 \end{pmatrix}$$

[3]:
```python
t = np.array([2.6578,3.992,0.2389,1.5106,3.2851])
y = np.array([-6.4552,-14.9657,0.2798,-2.0462,-10.539])
A = np.vander(t,3)
with np.printoptions(precision=4, suppress=True):
    print("A = ",A,sep="\n")

# Se grafican los datos
plt.scatter(t,y,color = 'steelblue',label='Datos')
ax = plt.gca()
ax.set_xlabel("$t$")
ax.set_ylabel("$y$")
plt.show()
```

```
A =
[[ 7.0639  2.6578  1.    ]
 [15.9361  3.992   1.    ]
 [ 0.0571  0.2389  1.    ]
 [ 2.2819  1.5106  1.    ]
 [10.7919  3.2851  1.    ]]
```



Se construye el sistema de ecuaciones normales

$$A^{\intercal}A\boldsymbol{x} = \begin{pmatrix} \sum_{i=1}^{5} t_i^4 & \sum_{i=1}^{5} t_i^3 & \sum_{i=1}^{5} t_i^2 \\ \sum_{i=1}^{5} t_i^3 & \sum_{i=1}^{5} t_i^2 & \sum_{i=1}^{5} t_i \\ \sum_{i=1}^{5} t_i^2 & \sum_{i=1}^{5} t_i & 5 \end{pmatrix} \begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{5} y_i t_i^2 \\ \sum_{i=1}^{5} y_i t_i \\ \sum_{i=1}^{5} y_i \end{pmatrix} = A^{\intercal}\boldsymbol{y}$$

y dado que $A^\intercal A$ es una matriz simétrica definida positiva, podemos aplicar la factorización de Cholesky de modo tal que

$$LL^\intercal = A^\intercal A$$

donde L es una matriz triangular inferior.

[4]:
```
L = ssl.chol(A.T@A)
with np.printoptions(precision=4, suppress=True):
    print(L)
```

```
[[20.6284  0.      0.    ]
 [ 5.8804  1.2455  0.    ]
 [ 1.7515  1.1118  0.8343]]
```

De tal manera que se resuelvan los sistemas

$$L\boldsymbol{w} = A^\intercal \boldsymbol{y} \quad \text{y} \quad L^\intercal \boldsymbol{x} = \boldsymbol{w}$$

para obtener el vector de parámetros $\boldsymbol{x}$

[5]:
```
x = ssl.STS(L.T,ssl.STI(L,y@A))
print(x)
```

```
[-0.9123063  -0.2372208   0.40157372]
```

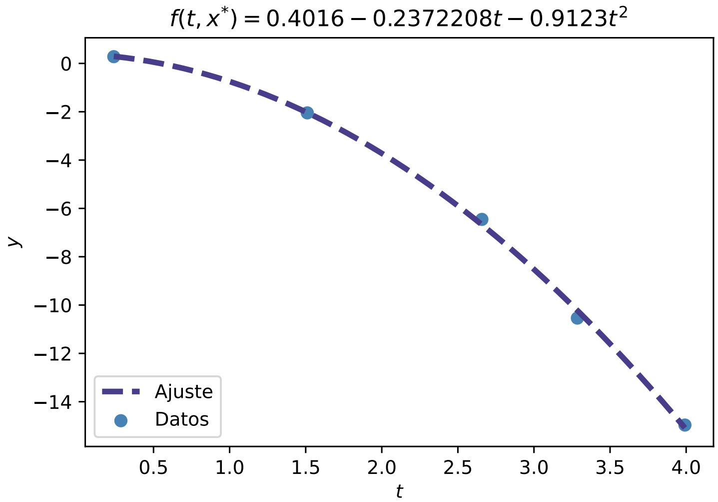
El vector solución es:

$$\boldsymbol{x}^* = \begin{pmatrix} 0.4016 \\ -0.2372208 \\ -0.9123 \end{pmatrix}$$

De esta manera la función modelo que mejor ajusta a los datos en el sentido de mínimos cuadrados es:

$$f(t, \boldsymbol{x}^*) = 0.4016 - 0.2372208t - 0.9123t^2$$

[6]:
```
# Grafica del ajuste
p = np.linspace(min(t),max(t));
ep = np.vander(p,3) @ x;
plt.plot(p,ep,'--',color='darkslateblue',label='Ajuste',linewidth = 3)
plt.scatter(t,y,color = 'steelblue',label='Datos')
ax = plt.gca();
ax.legend(loc = 'lower left')
ax.set_xlabel("$t$")
ax.set_ylabel("$y$")
ax.set_title(f"$f(t,x^{{*}}) = {x[2]:2.4f} {x[1]:+2.4f}t {x[0]:+2.4f}t^{2} $")
plt.show()
```



Factorización QR

La transformación ortogonal a una forma triangular se consigue mediante la factorización QR, la cual, para una matriz A de tamaño $m \times n$ con $m > n$, tiene la forma

$$A = Q \begin{pmatrix} R \\ O \end{pmatrix}$$

donde Q es una matriz ortogonal de $m \times m$, R es una matriz triangular de $n \times n$ y O es una matriz de tamaño $(m-n) \times n$ con todas sus entradas 0. Esta factorización transforma el problema de mínimos cuadrados lineales $A\boldsymbol{x} \cong \boldsymbol{y}$ en un problema de mínimos cuadrados triangular que tiene la misma solución, ya que

$$\|\boldsymbol{r}\|_2^2 = \|\boldsymbol{y} - A\boldsymbol{x}\|_2^2 = \left\| \boldsymbol{y} - Q \begin{pmatrix} R \\ O \end{pmatrix} \boldsymbol{x} \right\|_2^2 = \left\| Q^\intercal \boldsymbol{y} - \begin{pmatrix} R \\ O \end{pmatrix} \boldsymbol{x} \right\|_2^2 = \|\boldsymbol{c}_1 - R\boldsymbol{x}\|_2^2 + \|\boldsymbol{c}_2\|_2^2$$

donde

$$Q^\intercal \boldsymbol{y} = \begin{pmatrix} \boldsymbol{c}_1 \\ \boldsymbol{c}_2 \end{pmatrix}$$

con $\boldsymbol{c}_1 \in \mathbb{R}^n$. Si el vector solución $\boldsymbol{x}$ satisface el sistema triangular

$$R\boldsymbol{x} = \boldsymbol{c}_1$$

la norma residual mínima está dada por $\|\boldsymbol{r}\|_2 = \|\boldsymbol{c}_2\|_2$.

Transformaciones de Householder

Consideraremos el ajuste de curvas utilizando la factorización QR mediante transformaciones de Householder

$$H = I - 2\frac{\boldsymbol{v}\boldsymbol{v}^\intercal}{\boldsymbol{v}^\intercal\boldsymbol{v}}$$

para $\boldsymbol{v} \neq \mathbf{0}$. De esta manera podemos generar un conjunto de transformaciones que aplicados a A que produzcan una matriz triangular superior, esto es,

$$H_n \cdots H_1 A = \begin{pmatrix} R \\ 0 \end{pmatrix}$$

o lo que es lo mismo, una factorización de la forma

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}$$

con $Q = H_1 \cdots H_n$ la cual es ortogonal.

Ejemplo Consideramos nuevamente la tabla de datos

i	t_i	y_i
1	2.6578	-6.4552
2	3.992	-14.9657
3	0.2389	0.2798
4	1.5106	-2.0462
5	3.2851	-10.539

y queremos hacer el ajuste polinomial, para lo cual se considera la función modelo

$$f(t, \boldsymbol{x}) = x_1 + x_2 t + x_3 t^2$$

entonces vamos a buscar el vector de coeficientes $\boldsymbol{x}$ que mejor ajuste f a los datos de la tabla (t_i, y_i) mediante el uso de las ecuaciones normales.

Tomando

$$\boldsymbol{t} = \begin{pmatrix} 2.6578 \\ 3.992 \\ 0.2389 \\ 1.5106 \\ 3.2851 \end{pmatrix} \qquad \text{y} \qquad \boldsymbol{y} = \begin{pmatrix} -6.4552 \\ -14.9657 \\ 0.2798 \\ -2.0462 \\ -10.539 \end{pmatrix}$$

Luego, el problema tiene la forma

$$\begin{pmatrix} 2.6578^2 & 2.6578 & 1 \\ 3.992^2 & 3.992 & 1 \\ 0.2389^2 & 0.2389 & 1 \\ 1.5106^2 & 1.5106 & 1 \\ 3.2851^2 & 3.2851 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} \cong \begin{pmatrix} -6.4552 \\ -14.9657 \\ 0.2798 \\ -2.0462 \\ -10.539 \end{pmatrix}$$

[7]:
```
t = np.array([2.6578,3.992,0.2389,1.5106,3.2851])
y = np.array([-6.4552,-14.9657,0.2798,-2.0462,-10.539])
A = np.vander(t,3)
```

Primero se construye el vector $\boldsymbol{v}$ que hace ceros debajo la diagonal de la entrada $a_{1,1}$ de la matriz A, esto es,

$$\boldsymbol{v}_1 = A^{[1]} - (-\operatorname{sgn}(a_{1,1}))\|A^{[1]}\|_2 e_1$$

con $\alpha_1 = (-\operatorname{sgn}(a_{1,1}))\|A^{[1]}\|_2$ y

$$Q_1 = I_5 - 2\frac{\boldsymbol{v}_1\boldsymbol{v}_1^\intercal}{\boldsymbol{v}_1^\intercal\boldsymbol{v}_1}$$

para obtener

$$Q_1 A = \left(\begin{array}{c|c} \alpha_1 & \star_{1\times 2} \\ \hline \mathbf{0}_{4\times 1} & A^{(2)}_{4\times 2} \end{array}\right)$$

[8]:
```
v = A[:,0] + np.sign(A[0,0])*np.linalg.norm(A[:,0])*canonico(5)
V = np.array([v.tolist()],copy = True)
Q1 = np.eye(5) - 2/(v@v)*(V.T@V)
with np.printoptions(precision=4, suppress=True):
    print(Q1@A)
A2 = Q1@A
```

```
[[-20.6284  -5.8804  -1.7515]
 [ -0.      -0.9215  -0.5834]
 [ -0.       0.2213   0.9943]
 [ -0.       0.807    0.7733]
 [ -0.      -0.0423  -0.0723]]
```

Después, se construye el vector $\boldsymbol{v}$ que hace ceros debajo la diagonal de la entrada $a^{(2)}_{1,1}$ de la matriz $A^{(2)}$, esto es,

$$\boldsymbol{v}_2 = \left(A^{(2)}\right)^{[1]} - \left(-\operatorname{sgn}\left(a^{(2)}_{1,1}\right)\right)\left\|\left(A^{(2)}\right)^{[1]}\right\|_2 e_1$$

con $\alpha_2 = \left(-\operatorname{sgn}\left(a^{(2)}_{1,1}\right)\right)\left\|\left(A^{(2)}\right)^{[1]}\right\|_2$ y

$$Q_2^* = I_4 - 2\frac{\boldsymbol{v}_2\boldsymbol{v}_2^\intercal}{\boldsymbol{v}_2^\intercal\boldsymbol{v}_2}$$

para obtener

$$Q_2^* A^{(2)} = \left(\begin{array}{c|c} \alpha_2 & \star \\ \hline \mathbf{0}_{3\times 1} & A^{(3)}_{3\times 1} \end{array}\right)$$

de tal modo que si tomamos

$$Q_2 = \begin{pmatrix} 1 & \mathbf{0} \\ \mathbf{0} & Q_2^* \end{pmatrix}$$

obtengamos

$$Q_2 Q_1 A = \left(\begin{array}{cc|c} \alpha_1 & \star & \star \\ 0 & \alpha_2 & \star \\ \hline \mathbf{0} & \mathbf{0} & A^{(3)} \end{array}\right)$$

[9]:
```
v = A2[1:,1] + np.sign(A2[1,1])*np.linalg.norm(A2[1:,1])*canonico(4)
V = np.array([v.tolist()],copy = True)
```

```
Q2 = np.eye(5)
Q2[1:,1:] = Q2[1:,1:] - 2/(v@v)*(V.T@V)
with np.printoptions(precision=4, suppress=True):
    print(Q2@A2)
A3 = Q2@A2
```

```
[[-20.6284  -5.8804  -1.7515]
 [  0.       1.2455   1.1118]
 [ -0.       0.       0.8212]
 [ -0.       0.       0.1419]
 [ -0.      -0.      -0.0392]]
```

Por último, se construye el vector $\boldsymbol{v}$ que hace ceros debajo la diagonal de la entrada $a_{1,1}^{(3)}$ de la matriz $A^{(3)}$,

$$\boldsymbol{v}_3 = \left(A^{(3)}\right)^{[1]} - \left(-\text{sgn}\left(a_{1,1}^{(3)}\right)\right)\left\|\left(A^{(3)}\right)^{[1]}\right\|_2 e_1$$

con $\alpha_3 = \left(-\text{sgn}\left(a_{1,1}^{(3)}\right)\right)\left\|\left(A^{(3)}\right)^{[1]}\right\|_2$ y

$$Q_3^* = I_3 - 2\frac{\boldsymbol{v}_3\boldsymbol{v}_3^\intercal}{\boldsymbol{v}_3^\intercal\boldsymbol{v}_3}$$

para obtener

$$Q_3^* A^{(3)} = \begin{pmatrix} \alpha_3 \\ 0 \\ 0 \end{pmatrix}.$$

Así, al tomar

$$Q_3 = \begin{pmatrix} 1 & 0 & \mathbf{0} \\ 0 & 1 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & Q_3^* \end{pmatrix}$$

Se tenga que

$$Q_3Q_2Q_1A = \begin{pmatrix} \alpha_1 & \star & \star \\ 0 & \alpha_2 & \star \\ 0 & 0 & \alpha_3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

[10]:
```
v = A3[2:,2] + np.sign(A3[2,2])*np.linalg.norm(A3[2:,2])*canonico(3)
V = np.array([v.tolist()],copy = True)
Q3 = np.eye(5)
Q3[2:,2:] = Q3[2:,2:] - 2/(v@v)*(V.T@V)
with np.printoptions(precision=4, suppress=True):
    print("R = ",Q3@A3,sep='\n\n')
R = Q3@A3
```

```
R =
```

```
[[-20.6284  -5.8804  -1.7515]
 [  0.       1.2455   1.1118]
 [  0.      -0.      -0.8343]
 [ -0.       0.       0.    ]
 [ -0.      -0.      -0.    ]]
```

[11]:
```
with np.printoptions(precision=4, suppress=True):
    print("Q=",Q1.T@Q2.T@Q3.T,sep="\n\n")
Q = Q1.T@Q2.T@Q3.T
```

```
Q=

[[-0.3424  0.5172  0.2095 -0.5614 -0.5061]
 [-0.7725 -0.4423 -0.1662  0.2851 -0.3142]
 [-0.0028  0.1788 -0.9546 -0.2334  0.0483]
 [-0.1106  0.6906 -0.0461  0.7127 -0.0291]
 [-0.5232  0.1676  0.123  -0.2029  0.8012]]
```

Una vez calculada R, procedemos a resolver el sistema

$$R\boldsymbol{x} = \boldsymbol{c}_1$$

para obtener el vector de parámetros $\boldsymbol{x}$. El vector $\boldsymbol{c}_1$ se puede obtener al particionar $Q = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix}$, donde Q_1 tiene las primeras n columnas de Q y Q_2 tiene las $m-n$ restantes. De esta forma, $\boldsymbol{c}_1 = Q_1^{\intercal}\boldsymbol{y}$.

[12]:
```
x = ssl.STS(R[:3,:3],y@Q[:,:3])
print(x)
```

```
[-0.9123063  -0.2372208   0.40157372]
```

El vector solución es:

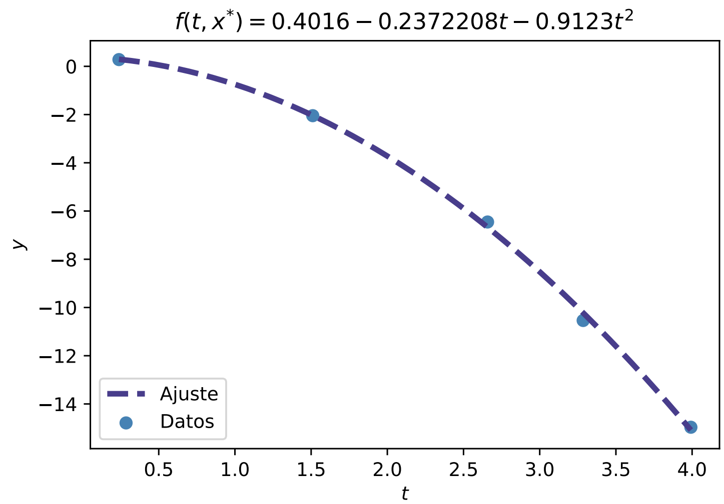
$$\boldsymbol{x}^{*} = \begin{pmatrix} 0.4016 \\ -0.2372208 \\ -0.9123 \end{pmatrix}$$

De esta manera la función modelo que mejor ajusta a los datos en el sentido de mínimos cuadrados es:

$$f(t, \boldsymbol{x}^{*}) = 0.4016 - 0.2372t - 0.9123t^2$$

[13]:
```
# Grafica del ajuste
p = np.linspace(min(t),max(t));
ep = np.vander(p,3) @ x;
plt.plot(p,ep,'--',color='darkslateblue',label='Ajuste',linewidth = 3)
plt.scatter(t,y,color = 'steelblue',label='Datos')
ax = plt.gca();
ax.legend(loc = 'lower left')
ax.set_xlabel("$t$")
ax.set_ylabel("$y$")
ax.set_title(f"$f(t,x^{{*}}) = {x[2]:2.4f} {x[1]:+2.4f}t {x[0]:+2.4f}t^{2}$")
```

```
plt.show()
```



En `numpy` contamos con la factorización QR mediante la función `qr` dentro del submódulo `linalg`. Si la función es llamada mediante un sólo argumento, la función regresará la matriz Q_1 y la matriz R, que sería la forma *reducida* de la factorización QR. Tambien se puede obtener la factorización completa al pasar la cadena `'complete'` como segundo argumento a la función. Utilizaremos el ejemplo anterior para obtener el vector de parámetros $\boldsymbol{x}$.

[14]:
```
# Se calcula la factorización QR reducida
Q,R = np.linalg.qr(A)#,'complete') # sintaxis factorizacion completa
with np.printoptions(precision=4, suppress=True):
    print("A = ",Q@R,sep="\n")
    print("Q = ",Q,sep="\n")
    print("R = ",R,sep="\n")
```

```
A =
[[ 7.0639  2.6578  1.    ]
 [15.9361  3.992   1.    ]
 [ 0.0571  0.2389  1.    ]
 [ 2.2819  1.5106  1.    ]
 [10.7919  3.2851  1.    ]]
Q =
[[-0.3424  0.5172  0.2095]
 [-0.7725 -0.4423 -0.1662]
 [-0.0028  0.1788 -0.9546]
 [-0.1106  0.6906 -0.0461]
 [-0.5232  0.1676  0.123 ]]
R =
[[-20.6284  -5.8804  -1.7515]
 [  0.        1.2455   1.1118]
```

```
 [ 0.        0.       -0.8343]]
```

[15]:
```
# Se resuelve el sistema triangular y se muestra la solucion
x = ssl.STS(R,y@Q)
print(x)
```

```
[-0.9123063  -0.2372208   0.40157372]
```

Algoritmo (Transformaciones de Householder)

Algoritmo 1: Factorización QR - Transformaciones de Householder

---

Entrada: $A \in \mathbb{R}^{m\times n}$ con $m > n$.
Salida: $Q \in \mathbb{R}^{m\times m}$ y $R \in \mathbb{R}^{m\times n}$ tales que $A = QR$.

$Q = I_m$
$R = A$
Para $k = 1 \to n$ **hacer**
  $\boldsymbol{v} = R_{[k:m]}^{[k]}$
  $\alpha = -\mathrm{sgn}(v_1)\|\boldsymbol{v}\|_2$
  $\boldsymbol{v} = \boldsymbol{v} - \alpha\boldsymbol{e}_1$
  $\boldsymbol{v} = \dfrac{\boldsymbol{v}}{\|\boldsymbol{v}\|_2}$
  $Q^* = I_{m-k-1} - 2\boldsymbol{v}\boldsymbol{v}^\intercal$
  $R_{[k:m]}^{[k:n]} = Q^* R_{[k:m]}^{[k:n]}$
  $Q_{[k:m]}^{[k:m]} = Q_{[k:m]}^{[k:m]} Q^*$

---

[16]:
```
def qr_Householder(A):
    m,n = A.shape
    Q = np.eye(m)
    R = np.array(A,dtype = "float64")
    for k in range(n):
        v = np.copy(R[k:,k])
        alfa = -np.sign(v[0])*np.linalg.norm(v)
        v -= alfa*canonico(v.size)
        v /= np.linalg.norm(v)
        Q_hh = np.eye(v.size) - 2*(v.reshape(v.size,1)@v.reshape(1,v.size))
        R[k:,k:] = Q_hh@R[k:,k:]
        Q[:,k:] = Q[:,k:]@Q_hh
    return Q,R
```

Prueba

Usamos la matriz A de antes para probar la rutina de factorización QR con transformaciones de Householder

[17]:
```python
Q_hh,R_hh = qr_Householder(A)
with np.printoptions(precision=4, suppress=True):
    print(f"Q = \n{Q_hh}",f"R = \n{R_hh}",sep='\n\n')
    print(f"\nA = \n{A}",f"QR = \n{Q_hh@R_hh}",f"Son iguales? = {np.
 ↪allclose(A,Q_hh@R_hh)}",sep='\n\n')
```

```
Q =
[[-0.3424  0.5172  0.2095 -0.5614 -0.5061]
 [-0.7725 -0.4423 -0.1662  0.2851 -0.3142]
 [-0.0028  0.1788 -0.9546 -0.2334  0.0483]
 [-0.1106  0.6906 -0.0461  0.7127 -0.0291]
 [-0.5232  0.1676  0.123  -0.2029  0.8012]]

R =
[[-20.6284  -5.8804  -1.7515]
 [ -0.       1.2455   1.1118]
 [ -0.       0.      -0.8343]
 [ -0.      -0.       0.    ]
 [ -0.       0.       0.    ]]

A =
[[ 7.0639  2.6578  1.    ]
 [15.9361  3.992   1.    ]
 [ 0.0571  0.2389  1.    ]
 [ 2.2819  1.5106  1.    ]
 [10.7919  3.2851  1.    ]]

QR =
[[ 7.0639  2.6578  1.    ]
 [15.9361  3.992   1.    ]
 [ 0.0571  0.2389  1.    ]
 [ 2.2819  1.5106  1.    ]
 [10.7919  3.2851  1.    ]]

Son iguales? = True
```

Ortogonalización de Gram-Schmidt

Teorema 1. *Sea V un espacio vectorial con producto interior y sea $S = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n\}$ un conjunto linealmente independiente de V. Se define $S' = \{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$, donde $\boldsymbol{u}_1 = \boldsymbol{w}_1$ y*

$$\boldsymbol{u}_k = \boldsymbol{w}_k - \sum_{j=1}^{k-1} \frac{\langle \boldsymbol{w}_k, \boldsymbol{u}_j \rangle}{\|\boldsymbol{u}_j\|^2} \boldsymbol{u}_j \quad \textit{para } 2 \leq k \leq n \tag{1}$$

entonces S' es un conjunto ortogonal de vectores no cero tal que $\operatorname{span}(S') = \operatorname{span}(S)$.

Observación 2. La construcción de $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n\}$ mediante (1) es conocido como el *proceso de ortogonalización de Gram-Schmidt*. A partir de este conjunto de vectores, se puede generar un conjunto *ortonormal* de vectores $\{\boldsymbol{v}_1, \dots, \boldsymbol{v}_n\}$ al tomar

$$\boldsymbol{v}_k = \frac{\boldsymbol{u}_k}{\|\boldsymbol{u}_k\|} \quad \forall\, k = 1, \dots, n. \tag{2}$$

Si se despeja $\boldsymbol{w}_k$ de (1) y se escribe en términos de $\boldsymbol{v_k}$ dados en (2), tendremos que

$$\boldsymbol{w}_k = \|\boldsymbol{u}_k\|\boldsymbol{v}_k + \sum_{j=1}^{k-1} \langle \boldsymbol{w}_k, \boldsymbol{v}_j \rangle \boldsymbol{v}_j \quad 1 \leq k \leq n$$

De esta manera, si se define a A, Q como las matrices cuya k-ésima columna es $\boldsymbol{w}_k$ y $\boldsymbol{v}_k$ respectivamente y se define R como una matriz de tamaño $n \times n$, con entradas,

$$R_{jk} = \begin{cases} \|\boldsymbol{u}_j\| & j = k \\ \langle \boldsymbol{w}_k, \boldsymbol{v}_j \rangle & j < k \\ 0 & j > k \end{cases}$$

se tiene que $A = QR$.

Ejemplo Consideramos nuevamente la tabla de datos

i	t_i	y_i
1	2.6578	-6.4552
2	3.992	-14.9657
3	0.2389	0.2798
4	1.5106	-2.0462
5	3.2851	-10.539

y queremos hacer el ajuste polinomial, para lo cual se considera la función modelo

$$f(t, \boldsymbol{x}) = x_1 + x_2 t + x_3 t^2$$

entonces vamos a buscar el vector de coeficientes $\boldsymbol{x}$ que mejor ajuste f a los datos de la tabla (t_i, y_i) mediante el uso de las ecuaciones normales.

Tomando

$$\boldsymbol{t} = \begin{pmatrix} 2.6578 \\ 3.992 \\ 0.2389 \\ 1.5106 \\ 3.2851 \end{pmatrix} \qquad \text{y} \qquad \boldsymbol{y} = \begin{pmatrix} -6.4552 \\ -14.9657 \\ 0.2798 \\ -2.0462 \\ -10.539 \end{pmatrix}$$

Luego, el problema tiene la forma

$$\begin{pmatrix} 2.6578^2 & 2.6578 & 1 \\ 3.992^2 & 3.992 & 1 \\ 0.2389^2 & 0.2389 & 1 \\ 1.5106^2 & 1.5106 & 1 \\ 3.2851^2 & 3.2851 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} \cong \begin{pmatrix} -6.4552 \\ -14.9657 \\ 0.2798 \\ -2.0462 \\ -10.539 \end{pmatrix}$$

[18]:
```
t = np.array([2.6578,3.992,0.2389,1.5106,3.2851])
y = np.array([-6.4552,-14.9657,0.2798,-2.0462,-10.539])
A = np.vander(t,3)
Q = np.empty(A.shape)
R = np.zeros((A.shape[1],A.shape[1]))
```

Primero se construye la primer columna de Q a partir de la primera columna de A, esto es,

$$\tilde{Q}^{[1]} = A^{[1]}$$

luego se calcula $R_{1,1}$

$$R_{1,1} = \|\tilde{Q}^{[1]}\|$$

y por último se actualiza $Q^{[1]}$ para que sea normal,

$$Q^{[1]} = \frac{\tilde{Q}^{[1]}}{R_{1,1}} = \frac{\tilde{Q}^{[1]}}{\|\tilde{Q}^{[1]}\|}$$

[19]:
```
Q[:,0] = A[:,0]
R[0,0] = np.linalg.norm(Q[:,0])
Q[:,0] = Q[:,0]/R[0,0]
with np.printoptions(precision=4, suppress=True):
    print(f"Q = \n{Q}",f"R = \n{R}",sep='\n\n')
```

```
Q =
[[0.3424 2.6578 1.    ]
 [0.7725 3.992  1.    ]
 [0.0028 0.2389 1.    ]
 [0.1106 1.5106 1.    ]
 [0.5232 3.2851 1.    ]]

R =
[[20.6284  0.      0.    ]
 [ 0.      0.      0.    ]
 [ 0.      0.      0.    ]]
```

Luego se construye la segunda columna de Q y R. Para construir la segunda columna de Q utilizando Gram-Schmidt es necesario calcular $R_{1,2}$. Primero se hará

$$\tilde{Q}^{[2]} = A^{[2]}$$

luego se calcula $R_{1,2}$

$$R_{1,2} = \langle Q^{[1]}, A^{[2]} \rangle$$

y actualizamos $\tilde{Q}^{[2]}$

$$\tilde{Q}^{[2]} = \tilde{Q}^{[2]} - R_{1,2}Q^{[1]} = A^{[2]} - \frac{\langle \tilde{Q}^{[1]}, A^{[2]} \rangle}{\|\tilde{Q}^{[1]}\|^2}\tilde{Q}^{[1]}$$

Una vez calculada el vector ortogonal a $Q^{[1]}$ mediante Gram-Schmidt, se procede como antes al calcular $R_{2,2}$ y normalizar $\tilde{Q}^{[2]}$, esto es

$$R_{2,2} = \|\tilde{Q}^{[2]}\|$$

y por último se actualiza $Q^{[2]}$ para que sea normal,

$$Q^{[2]} = \frac{\tilde{Q}^{[2]}}{R_{2,2}} = \frac{\tilde{Q}^{[2]}}{\|\tilde{Q}^{[2]}\|}$$

[20]:
```
Q[:,1] = A[:,1]
R[0,1] = Q[:,0].T@A[:,1]
Q[:,1] = Q[:,1] - R[0,1]*Q[:,0]
R[1,1] = np.linalg.norm(Q[:,1])
Q[:,1] = Q[:,1]/R[1,1]
with np.printoptions(precision=4, suppress=True):
    print(f"Q = \n{Q}",f"R = \n{R}",sep='\n\n')
```

```
Q =
[[ 0.3424  0.5172  1.    ]
 [ 0.7725 -0.4423  1.    ]
 [ 0.0028  0.1788  1.    ]
 [ 0.1106  0.6906  1.    ]
 [ 0.5232  0.1676  1.    ]]

R =
[[20.6284  5.8804  0.    ]
 [ 0.      1.2455  0.    ]
 [ 0.      0.      0.    ]]
```

Por último se construye la tercera columna de Q y R. Para construir la tercera columna de Q utilizando Gram-Schmidt es necesario calcular $R_{1,3}$ y $R_{2,3}$. Se toma de forma análoga al paso anaterior

$$\tilde{Q}^{[3]} = A^{[3]}$$

luego se calcula $R_{1,3}$

$$R_{1,3} = \langle Q^{[1]}, A^{[3]} \rangle$$

y actualizamos $\tilde{Q}^{[3]}$

$$\tilde{Q}^{[3]} = \tilde{Q}^{[3]} - R_{1,3}Q^{[1]}$$

Después se calcula $R_{2,3}$

$$R_{2,3} = \langle Q^{[2]}, A^{[3]} \rangle$$

y actualizamos $\tilde{Q}^{[3]}$

$$\tilde{Q}^{[3]} = \tilde{Q}^{[3]} - R_{2,3}Q^{[2]}$$

de tal manera que,

$$\tilde{Q}^{[3]} = A^{[3]} - \sum_{j=1}^{2} \frac{\langle \tilde{Q}^{[j]}, A^{[3]} \rangle}{\|\tilde{Q}^{[j]}\|^2} \tilde{Q}^{[j]}$$

Una vez calculada el vector ortogonal a $Q^{[1]}$ y $Q^{[2]}$ mediante Gram-Schmidt, se procede como antes al calcular $R_{3,3}$ y normalizar $\tilde{Q}^{[3]}$, esto es

$$R_{3,3} = \|\tilde{Q}^{[3]}\|$$

y por último se actualiza $Q^{[3]}$ para que sea normal,

$$Q^{[3]} = \frac{\tilde{Q}^{[3]}}{R_{3,3}} = \frac{\tilde{Q}^{[3]}}{\|\tilde{Q}^{[3]}\|}$$

[21]:
```
Q[:,2] = A[:,2]
R[0,2] = Q[:,0].T@A[:,2]
R[1,2] = Q[:,1].T@A[:,2]
Q[:,2] = Q[:,2] - R[0,2]*Q[:,0] - R[1,2]*Q[:,1]
R[2,2] = np.linalg.norm(Q[:,2])
Q[:,2] = Q[:,2]/R[2,2]
with np.printoptions(precision=4, suppress=True):
    print(f"Q = \n{Q}",f"R = \n{R}",sep='\n\n')
```

```
Q =
[[ 0.3424  0.5172 -0.2095]
 [ 0.7725 -0.4423  0.1662]
 [ 0.0028  0.1788  0.9546]
 [ 0.1106  0.6906  0.0461]
 [ 0.5232  0.1676 -0.123 ]]

R =
[[20.6284  5.8804  1.7515]
 [ 0.      1.2455  1.1118]
 [ 0.      0.      0.8343]]
```

[22]:
```
# Comprobamos que A = QR
with np.printoptions(precision=4, suppress=True):
    print(f"A = \n{A}",f"QR = \n{Q@R}",f"Son iguales? = {np.
 ↪allclose(A,Q@R)}",sep='\n\n')
```

```
A =
[[ 7.0639  2.6578  1.    ]
 [15.9361  3.992   1.    ]
 [ 0.0571  0.2389  1.    ]
 [ 2.2819  1.5106  1.    ]
 [10.7919  3.2851  1.    ]]
```

```
QR =
[[ 7.0639  2.6578  1.     ]
 [15.9361  3.992   1.     ]
 [ 0.0571  0.2389  1.     ]
 [ 2.2819  1.5106  1.     ]
 [10.7919  3.2851  1.     ]]

Son iguales? = True
```

Una vez calculada R, procedemos a resolver el sistema

$$R\boldsymbol{x} = \boldsymbol{c}$$

donde $\boldsymbol{c} = \boldsymbol{Q}^\intercal \boldsymbol{y}$ para obtener el vector de parámetros $\boldsymbol{x}$, ya que obtenemos la forma reducida.

[23]:
```
x = ssl.STS(R,y@Q)
print(x)
```

```
[-0.9123063  -0.2372208   0.40157372]
```

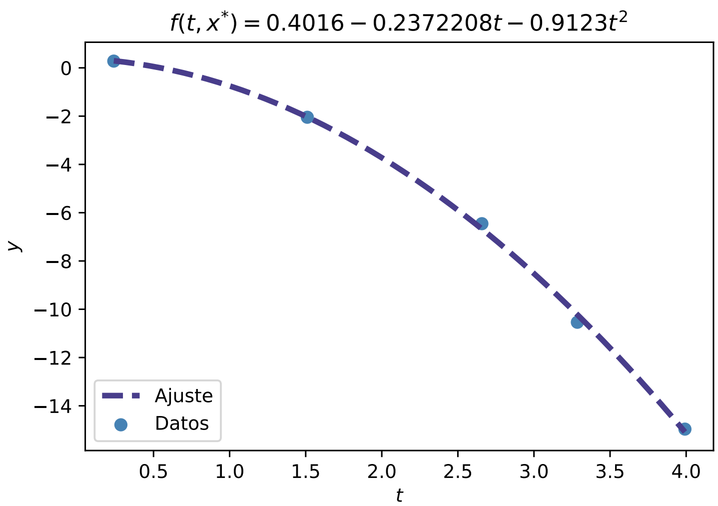
El vector solución es:

$$\boldsymbol{x}^* = \begin{pmatrix} 0.4016 \\ -0.2372208 \\ -0.9123 \end{pmatrix}$$

De esta manera la función modelo que mejor ajusta a los datos en el sentido de mínimos cuadrados es:

$$f(t, \boldsymbol{x}^*) = 0.4016 - 0.2372t - 0.9123t^2$$

[24]:
```
# Grafica del ajuste
p = np.linspace(min(t),max(t));
ep = np.vander(p,3) @ x;
plt.plot(p,ep,'--',color='darkslateblue',label='Ajuste',linewidth = 3)
plt.scatter(t,y,color = 'steelblue',label='Datos')
ax = plt.gca();
ax.legend(loc = 'lower left')
ax.set_xlabel("$t$")
ax.set_ylabel("$y$")
ax.set_title(f"$f(t,x^{{*}}) = {x[2]:2.4f} {x[1]:+2.4f}t {x[0]:+2.4f}t^{2}$")
plt.show()
```



Algoritmo (Ortogonalización de Gram-Schmidt)

Algoritmo 2: Factorización QR - Ortogonalización de Gram-Schmidt

Entrada: $A \in \mathbb{R}^{m\times n}$ con $m > n$.
Salida: $Q \in \mathbb{R}^{m\times n}$ y $R \in \mathbb{R}^{n\times n}$ tales que $A = QR$.

Para $k = 1 \to n$ **hacer**
  $Q^{[k]} = A^{[k]}$
  Para $j = 1 \to k-1$ **hacer**
    $R_{jk} = Q^{[j]\intercal} A^{[k]}$ // $\langle Q^{[j]}, A^{[k]} \rangle$
    $Q^{[k]} = Q^{[k]} - R_{jk} Q^{[j]}$
  $R_{kk} = \|Q^{[k]}\|_2$
  $Q^{(k)} = Q^{[k]}/R_{kk}$

[25]:
```
def qr_GramSchmidt(A):
    m,n = A.shape
    Q = np.empty((m,n),dtype = "float64")
    R = np.zeros((n,n))
    for k in range(n):
        Q[:,k] = np.copy(A[:,k])
        for j in range(k):
            R[j,k] = Q[:,j].T@A[:,k]
            Q[:,k] -= R[j,k]*Q[:,j]
        R[k,k] = np.linalg.norm(Q[:,k])
        Q[:,k] /= R[k,k]
    return Q,R
```

Prueba

Usamos la matriz A de antes para probar la rutina de factorización QR con ortogonalización de Gram-Schmidt.

[26]:
```
Q_gs,R_gs = qr_GramSchmidt(A)
with np.printoptions(precision=4, suppress=True):
    print(f"Q = \n{Q_gs}",f"R = \n{R_gs}",sep='\n\n')
    print(f"\nA = \n{A}",f"QR = \n{Q_gs@R_gs}",f"Son iguales? = {np.
 ↪allclose(A,Q_gs@R_gs)}",sep='\n\n')
```

```
Q =
[[ 0.3424  0.5172 -0.2095]
 [ 0.7725 -0.4423  0.1662]
 [ 0.0028  0.1788  0.9546]
 [ 0.1106  0.6906  0.0461]
 [ 0.5232  0.1676 -0.123 ]]

R =
[[20.6284  5.8804  1.7515]
 [ 0.      1.2455  1.1118]
 [ 0.      0.      0.8343]]

A =
[[ 7.0639  2.6578  1.    ]
 [15.9361  3.992   1.    ]
 [ 0.0571  0.2389  1.    ]
 [ 2.2819  1.5106  1.    ]
 [10.7919  3.2851  1.    ]]

QR =
[[ 7.0639  2.6578  1.    ]
 [15.9361  3.992   1.    ]
 [ 0.0571  0.2389  1.    ]
 [ 2.2819  1.5106  1.    ]
 [10.7919  3.2851  1.    ]]

Son iguales? = True
```