Linear Regression

May 30, 2023

1 Least squares approximation

In our project, we are rewriting the notebook on "Least squares approximation" into an R script. We then extend this model by generating a random multidimensional dataset using Python and performing a multiple linear regression with a neural network.

[7]:
```
library(Matrix)
library(MASS)
library(rgl)
library(tidyverse)
# install.packages('plot3D')
library(plot3D)
```

In our project, we are rewriting the notebook on "Least squares approximation" into an R script. We then extend this model by generating a random multidimensional dataset using Python and performing a multiple linear regression with a neural network.

While in Sage, these functions are used to create a matrix:

A = matrix(QQ, [[1,1],[2,1],[3,1],[4,1],[5,1],[6,1],[7,1]])

Y = matrix(QQ, [[25],[15],[9],[24],[37],[50],[51]])

we define the matrices in R as follows.

[9]:
```
# Define the matrices A and Y
A <- matrix(c(1, 2, 3, 4, 5, 6, 7, rep(1, times = 7)), ncol = 2)
A
Y <- matrix(c(25, 15, 9, 24, 37, 50, 51), ncol = 1)
Y
```

[9]:

A matrix: 7 × 2 of type dbl

1	1
2	1
3	1
4	1
5	1
6	1
7	1

[9]:

A matrix: 7 × 1 of type dbl	25 15 9 24 37 50 51

In the following we want to solve the least squares minimisation problem.

[10]:
```
# M = (A.transpose()*A).inverse()*A.transpose()
M <- ginv(A)
M
# P is the projection matrix in R^7
P <- A %*% M
P
# X is the solution to the least squares problem
X <- M %*% Y
X
```

[10]:

A matrix: 2 × 7 of type dbl	-0.1071429	-0.07142857	-0.03571429	2.595216e-17	3.571429e-02	0.07142857
	0.5714286	0.42857143	0.28571429	1.428571e-01	1.161685e-17	-0.14285714

[10]:

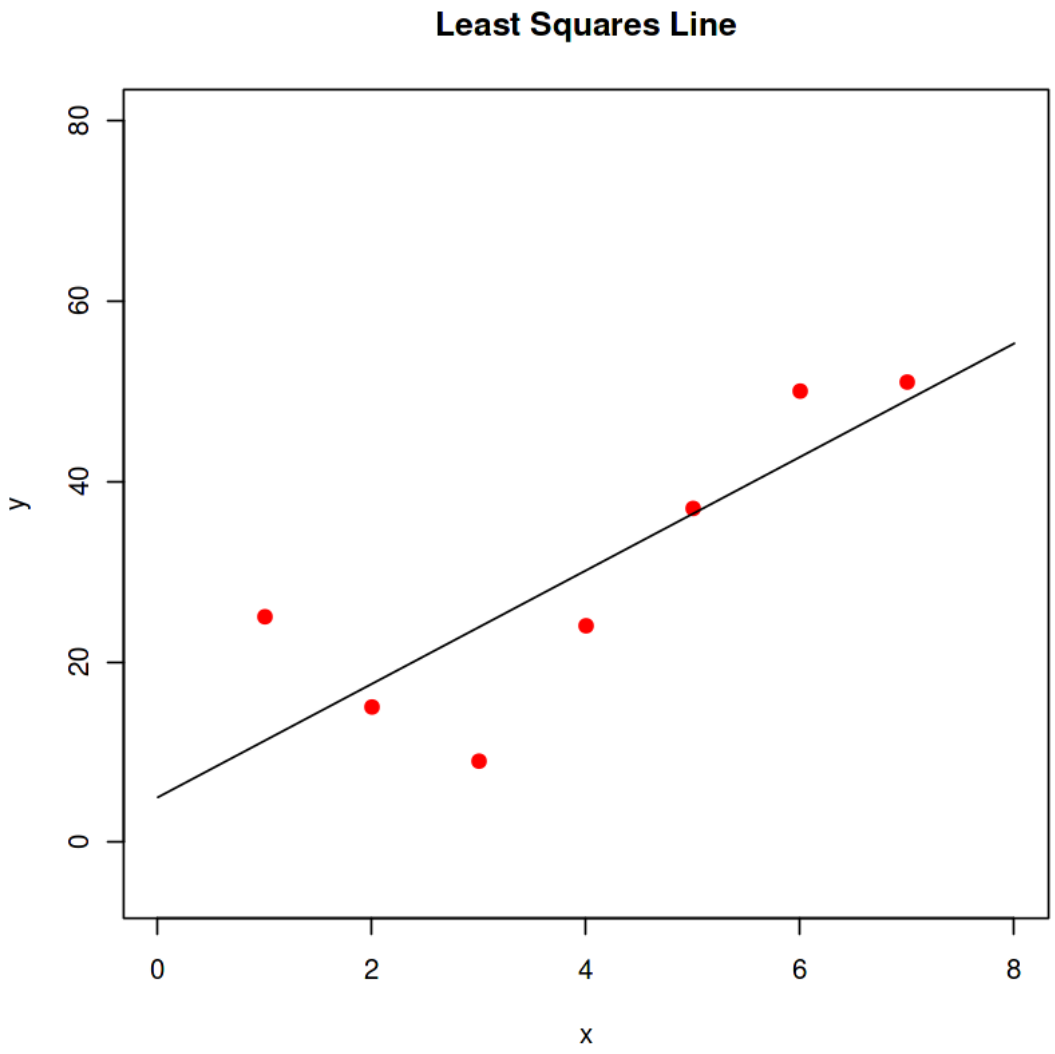
A matrix: 7 × 7 of type dbl	0.46428571	3.571429e-01	0.25000000	0.1428571	0.03571429	-7.142857e-02
	0.35714286	2.857143e-01	0.21428571	0.1428571	0.07142857	1.387779e-16
	0.25000000	2.142857e-01	0.17857143	0.1428571	0.10714286	7.142857e-02
	0.14285714	1.428571e-01	0.14285714	0.1428571	0.14285714	1.428571e-01
	0.03571429	7.142857e-02	0.10714286	0.1428571	0.17857143	2.142857e-01
	-0.07142857	1.110223e-16	0.07142857	0.1428571	0.21428571	2.857143e-01
	-0.17857143	-7.142857e-02	0.03571429	0.1428571	0.25000000	3.571429e-01

[10]:

A matrix: 2 × 1 of type dbl	6.285714 5.000000

[11]:
```
# Initialize the plot figure with axis labels and a title
plot(x = NULL, y = NULL, xlim = c(0, 8), ylim = c(-5, 80), xlab = "x", ylab =␣
 ↪"y", main = "Least Squares Line")

# Plot the data points with larger size
points(x = c(1, 2, 3, 4, 5, 6, 7), y = c(25, 15, 9, 24, 37, 50, 51), col =␣
 ↪"red", pch = 20, cex = 1.5)

# Define the function corresponding to the least squares line
f <- function(x) { X[1] * x + X[2] }

# Plot the least squares line
curve(f, from = 0, to = 8, add = TRUE)
```

[11]:



In the following we will set up the least square parabola.

[13]:
```
# Create a new matrix A with three columns
A <- matrix(c(1, 4, 9, 16, 25, 36, 49, 1, 2, 3, 4, 5, 6, 7, rep(1, times=7)),␣
 ↪ncol = 3)

# Recalculate M, P, and X for the new matrix A
M <- ginv(A) # Moore-Penrose-Pseudoinverse
M
P <- A %*% M
P
X <- M %*% Y
X
```

[13]:

A matrix: 3 × 7 of type dbl

0.05952381	1.417212e-16	-0.03571429	-0.04761905	-0.03571429	-4.554694e-1
-0.58333333	-7.142857e-02	0.25000000	0.38095238	0.32142857	7.142857e-02
1.28571429	4.285714e-01	-0.14285714	-0.42857143	-0.42857143	-1.428571e-0

[13]:

A matrix: 7 × 7 of type dbl

0.76190476	0.35714286	0.07142857	-0.0952381	-0.14285714	-7.142857e-02
0.35714286	0.28571429	0.21428571	0.1428571	0.07142857	-5.551115e-17
0.07142857	0.21428571	0.28571429	0.2857143	0.21428571	7.142857e-02
-0.09523810	0.14285714	0.28571429	0.3333333	0.28571429	1.428571e-01
-0.14285714	0.07142857	0.21428571	0.2857143	0.28571429	2.142857e-01
-0.07142857	0.00000000	0.07142857	0.1428571	0.21428571	2.857143e-01
0.11904762	-0.07142857	-0.14285714	-0.0952381	0.07142857	3.571429e-01

[13]:

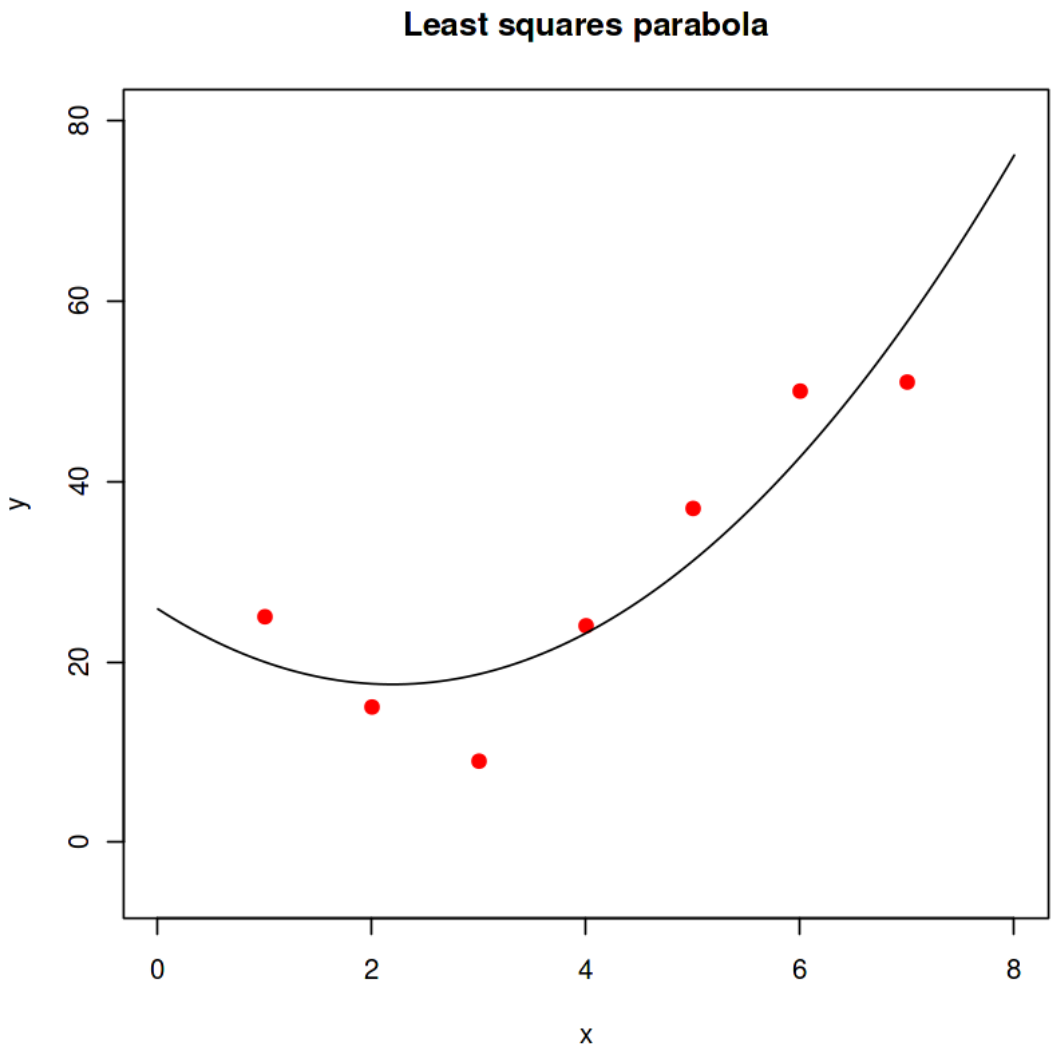
A matrix: 3 × 1 of type dbl

1.738095
-7.619048
25.857143

[14]:

```
# Initialize the plot figure
plot(x = NULL, y = NULL, xlim = c(0, 8), ylim = c(-5, 80), xlab = "x", ylab =␣
 ↪"y", main = "Least squares parabola")
# Replot the data points
points(x = c(1, 2, 3, 4, 5, 6, 7), y = c(25, 15, 9, 24, 37, 50, 51), col =␣
 ↪"red", pch = 20, cex = 1.5)
# Define the function corresponding to the least squares parabola
g <- function(x) { X[1] * x^2 + X[2] * x + X[3] }
# Plot the least squares parabola
curve(g, from = 0, to = 8, add = TRUE)
```

[14]:



In the notebook on least square approximation, the next step involves comparing the "predicted value for y8" with the "decimal approximation of y8".

y8 = f.substitute(x == 8)

show(y8)

RR(y8)

However, this comparison does not make sense in the given R code. In symbolic computation systems like Sage, mathematical expressions are often represented symbolically, meaning they are represented as symbolic variables and equations. The decimal approximation is obtained by converting the symbolic expression to a decimal number using a function like RR().

If for example the the rank of the matrix A is not full, we cannot use the model we set up earlier but we can ask R to find the linear regression coefficients, just like in Sage.

data = [(1,25),(2,15),(3,9),(4,24),(5,37),(6,50),(7,51)]

var('a, b, x')

model(x) = a*x + b

sol = find_fit(data,model)

show(sol)

fnum = model(a=sol[0].rhs(),b=sol[1].rhs())

show(fnum)

var('a, b, c, x')

model2(x) = a*x^2 + b*x +c

sol2 = find_fit(data,model2)

gnum = model2(a=sol2[0].rhs(),b=sol2[1].rhs(),c=sol2[2].rhs())

show(gnum)

[15]:
```r
data <- data.frame(x = c(1, 2, 3, 4, 5, 6, 7), y = c(25, 15, 9, 24, 37, 50, 51))

# Fit a linear model to the data
lm_fit <- lm(y ~ x, data = data)
print(lm_fit)

# Fit a quadratic model to the data
quad_fit <- lm(y ~ poly(x, 2, raw = TRUE), data = data)
print(quad_fit)
```

```
Call:
lm(formula = y ~ x, data = data)

Coefficients:
(Intercept)            x
      5.000        6.286


Call:
lm(formula = y ~ poly(x, 2, raw = TRUE), data = data)

Coefficients:
            (Intercept)  poly(x, 2, raw = TRUE)1  poly(x, 2, raw = TRUE)2
                 25.857                   -7.619                    1.738
```

[16]:
```r
# Compare the constant coefficients of the two quadratic polynomials
c1 <- coef(quad_fit)[1]
```

```
c2 <- g(0)
c1 - c2
c1 == c2
```

[16]: **(Intercept):** -1.4210854715202e-14

[16]: **(Intercept):** FALSE

2 Least squares using gradient methods

2.1 Generate random Data

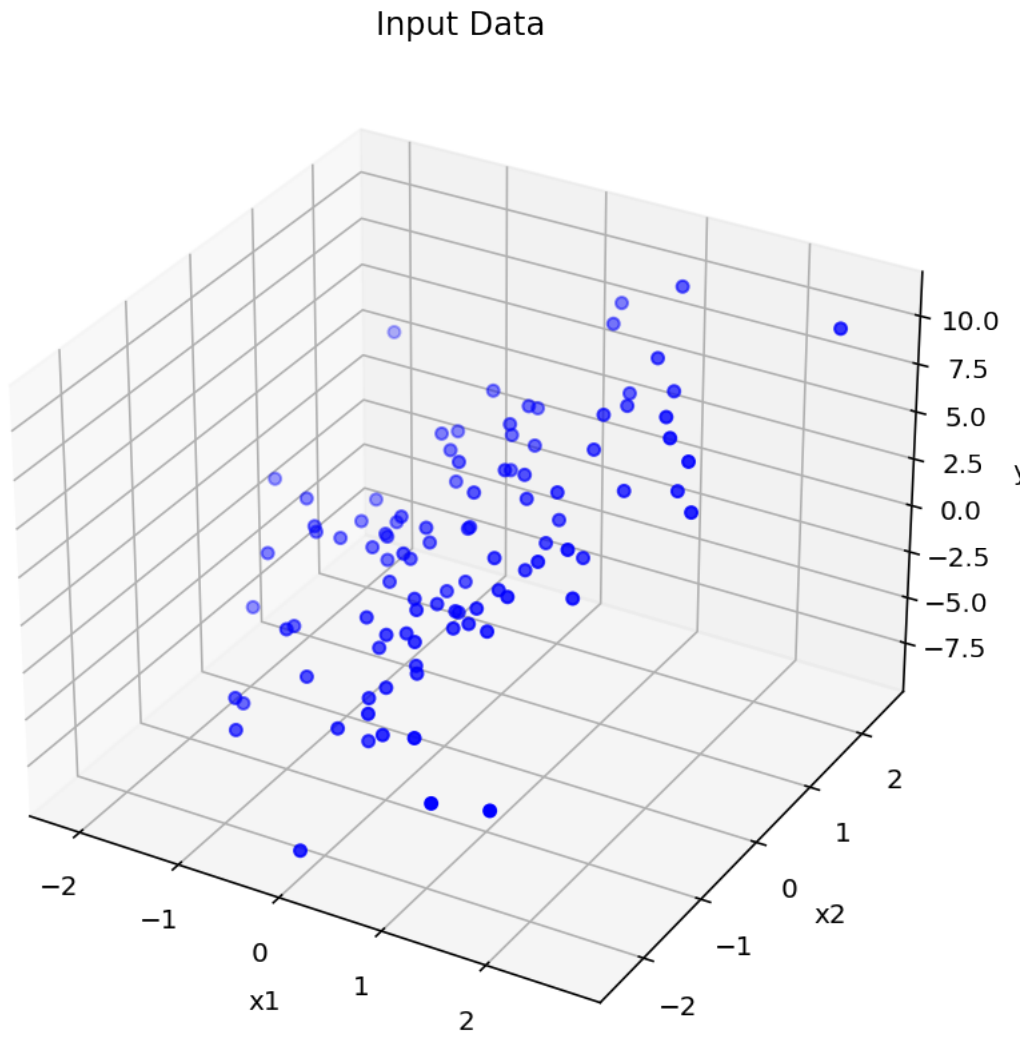
We generate data with a standard deviation of one sigma of (2,3) times the x = (x1,x2) value.

[1]:
```
import numpy as np
# Generate random input data
x = np.random.randn(100, 2)
y = np.dot(x, np.array([[2], [3]])) + np.random.normal(0, 1, size=(100, 1))
```

2.2 Visualize Data

[2]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Extract the x1, x2, and y coordinates for plotting
x1 = x[:, 0]
x2 = x[:, 1]

# Create a 3D figure
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot the data points
ax.scatter(x1, x2, y, c='b', marker='o')

# Set labels for the axes
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')

# Set title for the plot
ax.set_title('Input Data')

# Show the 3D plot
plt.show()
```

[2]:



2.3 Perform Linear Regression (least squares with gradient descent)

What this does, is that we minimize the squared error (loss) by taking small steps in the 'right' direction. This direction can be computed via the gradient of the squared error.

[3]:
```
def linear_regression(x, y, learning_rate=0.0005, num_iterations=3000):
    # Initialize parameters
    As = list()
    bs = list()
    A = np.zeros((x.shape[1], y.shape[1]))  # Slope matrix
    b = np.zeros((1, y.shape[1]))  # Intercept vector

    # Number of training examples
    m = len(x)
```

```
    for i in range(num_iterations):
        # Calculate predicted values
        y_pred = np.dot(x, A) + b

        # Calculate gradients - least squares
        gradient_A = (2/m) * np.dot(x.T, y_pred - y)
        gradient_b = (2/m) * np.sum(y_pred - y, axis=0, keepdims=True)

        # Update parameters using gradient descent
        A -= learning_rate * gradient_A
        b -= learning_rate * gradient_b

        # Print the loss every 100 iterations
        if i % 300 == 0:
            loss = np.mean((y_pred - y) ** 2)
            print(f"Iteration {i}: Loss = {loss}")
            As.append(A.copy())
            bs.append(b.copy())

    return A, b, As, bs



# Perform linear regression
A, b, As, bs = linear_regression(x, y)

# Print the parameters
print("Slope Matrix A:")
print(A)
print("\nIntercept Vector b:")
print(b)
del A,b
```

```
Iteration 0: Loss = 15.690504717008762
Iteration 300: Loss = 8.451442424753633
Iteration 600: Loss = 4.802829677710483
Iteration 900: Loss = 2.9540744892729354
Iteration 1200: Loss = 2.011833551907254
Iteration 1500: Loss = 1.5285586923553813
Iteration 1800: Loss = 1.278995220440838
Iteration 2100: Loss = 1.149186117488844
Iteration 2400: Loss = 1.081153227537608
Iteration 2700: Loss = 1.0452166355683454
Slope Matrix A:
[[1.80515507]
 [2.97398609]]
```

```
Intercept Vector b:
[[0.19688433]]
```

2.4 Plot the results

[4]:
```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Extract the x1, x2, and y coordinates for plotting
x1 = x[:, 0]
x2 = x[:, 1]

# Determine the grid size
num_plots = len(As)
grid_size = int(np.ceil(np.sqrt(num_plots)))

# Create a grid of subplots
fig, axs = plt.subplots(grid_size, grid_size, figsize=(12, 12),␣
 ↪subplot_kw={'projection': '3d'})

# Iterate over the parameters and create subplots
for i, (A, b) in enumerate(zip(As, bs)):
    # Plot the data points
    ax = axs[i // grid_size, i % grid_size]
    ax.scatter(x[:, 0], x[:, 1], y.flatten(), c='b', marker='o')

    # Plot the plane defined by A and b
    x1_plane, x2_plane = np.meshgrid(np.linspace(min(x[:, 0]), max(x[:, 0]),␣
 ↪10), np.linspace(min(x[:, 1]), max(x[:, 1]), 10))
    y_plane = A[0] * x1_plane + A[1] * x2_plane + b
    ax.plot_surface(x1_plane, x2_plane, y_plane, alpha=0.3, color='r')

    # Set labels for the axes
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('y')

    # Set title for the subplot
    ax.set_title(f'Subplot {i+1}')

    # Set axis limits
    ax.set_xlim(min(x[:, 0]), max(x[:, 0]))
    ax.set_ylim(min(x[:, 1]), max(x[:, 1]))
    ax.set_zlim(min(y.flatten()), max(y.flatten()))
```

```
    # Enable grid lines
    ax.grid(True)

# Remove any unused subplots
for i in range(num_plots, grid_size ** 2):
    fig.delaxes(axs[i // grid_size, i % grid_size])

# Adjust spacing between subplots
fig.tight_layout()

# Show the plot grid
plt.show()
```

[4]: