Aprendizagem não supervisionada

Equações normais

O normal mesmo. Reta que melhor se ajusta aos dados por meio da minimização da distância entre a reta e os pontos (mínimos quadrados).

Resolvido na forma matriz. Onde há uma matriz de n linhas que corresponde aos dados de entrada A = [x__o^3 x_0^2 x_0 1, x__1^3 x_1^2 x_1 1, ..., x__n^3 x_n^2 x_n 1], multiplicado por C = [a b c d], coeficientes de uma polinomial de grau 3. O resultado da multiplicação é igual à phi = [phi_0, phi_1, phi_2, ..., phi_n]. Como multiplica nx4 por 4x1 têm-se um vetor como resultado (nx1).

A ideia é que o módulo da diferenta entre phi e y ao quadrado seja mínimo.

|phi - y|² = (AC-y)^T (AC - y), onde T corresponde à transposta = C^t(A^T A)C-2C^T A^T y + y^T y derivando, = 2A^t A C - 2 A^T y

como quer o mínimo, iguala à 0

= 2A^T A C - 2 A^T y = 0 = A^T A C = A^T y

Para deixar A^T A igual a 1, tem que multiplicar pela inversa para obter a identidade.

= (A^T A)^-1 (A^T A) C = (A^T A)^-1 A^t y, onde ^-1 corresponde à inversa

=> C = (A^T A)^-1 A^t y

[ ] *6 células ocultas*

Algoritmo com descida de gradiente

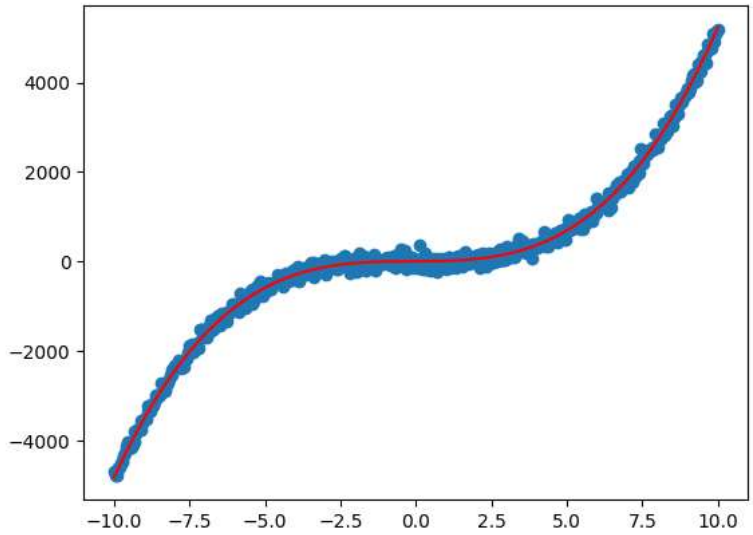
Feito por iteração.

Calcula a derivada num ponto e então anda com o ponto para onde a derivada aponta para um baixo. Isso é feito até que o ponto começa a convergir. O passo da função corresponde à alpha, onde este deve ser definido com cuidado para não aumentar o peso computacional e também para não fazer a função divergir.

É possível que caia num mínimo local.

```
1 import numpy as np
2 from matplotlib import pyplot as plt
```

```
1 x = np.linspace(-10,10,550)
2 y = 5*x**3 + 2*x**2 + 3*x
3 ruido = 1e2*np.random.normal(size=len(x))
4 y2 = y + ruido
5 plt.plot(x,y,'r');
6 plt.scatter(x,y2);
```




```
1 def compute_cost(a,b,c,d,points):
2   total_cost = 0
3   n = float(len(points))
4
5   # calcular soma dos erros ao quadrado
6
7   for i in range(0,len(points)):
```

```
  for i in range(0,len(points)):
    x = points[i,0]
    y = points[i,1]
    yp = a * x**3 + b*x**2 + c*x + d
    total_cost += (y-yp)**2 # soma dos quadrados dos erros
  return total_cost/2*n
```

```
def gradient_descent(points, a, b, c, d, learning_rate, num_iterations):
  cost_graph = []

  #Para cada iteração, otimizar a, b, c e d e computar custo

  for i in range(num_iterations):
    cost_graph.append(compute_cost(a,b,c,d,points))
    a, b, c, d = step_gradient(a, b, c, d, np.array(points), learning_rate)

  return [a, b, c, d, cost_graph]
```

```
def step_gradient(a_atual, b_atual, c_atual, d_atual, points, learning_rate):

  a_gradient = 0
  b_gradient = 0
  c_gradient = 0
  d_gradient = 0
  n = float(len(points))

  # Calcular o gradiente
  for i in range(0,len(points)):
    x = points[i,0]
    y = points[i,1]
    yp = a_atual * x**3 + b_atual*x**2 + c_atual*x + d_atual
    a_gradient += -(1/n)* x**3 * (y - yp)
    b_gradient += -(1/n)* x**2 * (y - yp)
    c_gradient += -(1/n)* x * (y - yp)
    d_gradient += -(1/n) * (y - yp)

  # Atualizar o a, b, c e d_atual

  a_updated = a_atual - learning_rate * a_gradient
  b_updated = b_atual - learning_rate * b_gradient
  c_updated = c_atual - learning_rate * c_gradient
  d_updated = d_atual - learning_rate * d_gradient

  return a_updated, b_updated, c_updated, d_updated
```

```
#Hiperparâmetros
learning_rate = 0.00001 # tamanho do passo (alpha)
ini_a = 0
ini_b = 0
ini_c = 0
ini_d = 0
num_iterations = 50
```

```
points = np.array([x,y2]).T

a,b,c,d,cost_graph = gradient_descent(points, ini_a, ini_b, ini_c, ini_d, learning_rate, num_iterations)

print(a,b,c,d)
```

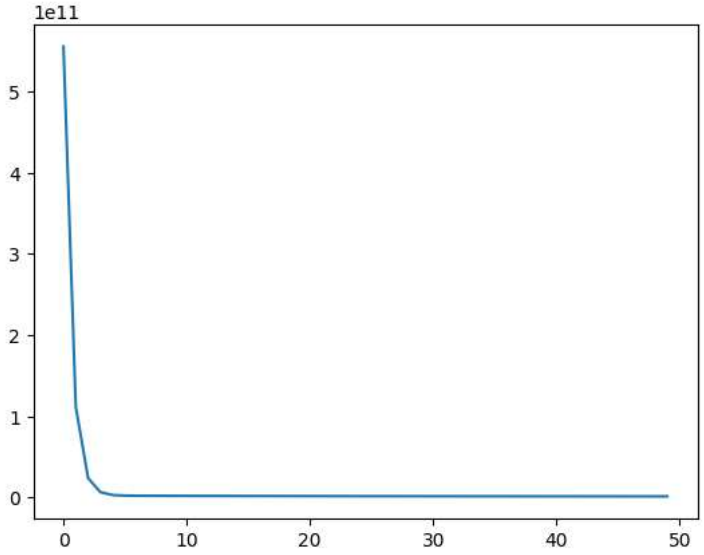
```
5.02877148255201 1.2263001503289954 0.07589829545131518 0.019500099715624667
```

```
plt.plot(x,a*x**3+b*x**2+c*x+d,'r')
plt.scatter(x,y2)
plt.plot(x,y,'k')
```

```
[<matplotlib.lines.Line2D at 0x7d73cfdb59f0>]
```


```
plt.plot(cost_graph)
```

```
[<matplotlib.lines.Line2D at 0x7d73cf549a80>]
```




```
1
```